Energy Analysis and Optimisation Techniques for Automatically Synthesised Coprocessors

Paul Morgan

A thesis submitted to
the Universities of

Edinburgh
Glasgow
Heriot-Watt
Strathclyde

For the Degree of
Doctor of Engineering in System Level Integration

Executive summary

The primary outcome of this research project is the development of a methodology enabling fast automated early-stage power and energy analysis of configurable processors for system-on-chip platforms. Such capability is essential to the process of selecting energy efficient processors during design-space exploration, when potential savings are highest. This has been achieved by developing dynamic and static energy consumption models for the constituent blocks within the processors.

Several optimisations have been identified, specifically targeting the most significant blocks in terms of energy consumption. Instruction encoding mechanism reduces both the energy and area requirements of the instruction cache; modifications to the multiplier unit reduce energy consumption during inactive cycles. Both techniques are demonstrated to offer substantial energy savings.

The aforementioned techniques have undergone detailed evaluation and, based on the positive outcomes obtained, have been incorporated into Cascade, a system-on-chip coprocessor synthesis tool developed by Critical Blue, to provide automated analysis and optimisation of processor energy requirements. This thesis details the process of identifying and examining each method, along with the results obtained. Finally, a case study demonstrates the benefits of the developed functionality, from the perspective of someone using Cascade to automate the creation of an energy-efficient configurable processor for system-on-chip platforms.

Contents

1 Introduction **1**
1.1 Project aims . . . 1
1.2 Project timeline . . . 2
1.3 Thesis organisation and project outcomes . . . 3

2 Industrial and Academic Context **6**
2.1 Industrial relevance . . . 6
2.2 Literature review . . . 14

3 Coprocessor power evaluation tool flow **22**
3.1 Overview of the power analysis flow . . . 23
3.2 RTL Synthesis (Design Compiler) . . . 25
3.3 Netlist Simulation (VCS) . . . 30
3.4 Power analysis (Power Compiler) . . . 34
3.5 Summary . . . 39

4 Evaluation of open-source processor cores **40**
4.1 TestCore processor . . . 41
4.1.1 RTL synthesis . . . 41
4.1.2 Code compilation . . . 41
4.1.3 Simulation and power analysis . . . 42
4.1.4 Comparison of process technologies . . . 43
4.2 LEON2 processor . . . 47
4.2.1 Configuring and simulating LEON2 using ModelSim . . . 47
4.2.2 The SOCks project and simulation using NC-Sim . . . 49
4.2.3 Monitoring switching activity . . . 52
4.2.4 Configuring the software build environment . . . 54
4.2.5 RTL synthesis . . . 55
4.2.6 Netlist simulation and power analysis . . . 57
4.3 OpenRISC 1200 processor . . . 61
4.3.1 Building the OpenRISC tool chain . . . 61
4.3.2 Cross-compiling applications . . . 63
4.3.3 Synthesis . . . 64
4.3.4 Simulation and power analysis . . . 66
4.4 Summary . . . 68

5 Accelerating MediaBench using Cascade **69**
5.1 Cross-compiling MediaBench for ARM . . . 70
5.2 Offloading functions to Cascade coprocessors . . . 74

5.3 Summary . . . 79

6 Creating functional unit models **80**
6.1 Multiplier unit . . . 81
6.2 Other functional units . . . 88
6.3 Output banks . . . 91
6.4 Summary . . . 96

7 Characterising memories and register files **98**
7.1 Hard macro memory blocks . . . 99
7.2 Register files and tag RAM . . . 102
7.3 Summary . . . 106

8 Clock tree power and clock gating **107**
8.1 Clock tree power . . . 107
8.1.1 SPICE and Nanosim analysis . . . 108
8.1.2 Design Compiler topographical mode analysis . . . 108
8.1.3 Integrating clock tree power analysis into Cascade . . . 115
8.2 Clock gating . . . 116
8.2.1 Clock gating methods applicable to Cascade . . . 117
8.2.2 Automated clock gating using Power Compiler . . . 121
8.3 Summary . . . 123

9 Leakage power issues **125**
9.1 Sources of leakage power . . . 125
9.2 Calculating coprocessor leakage power . . . 127
9.3 Implementing leakage calculation in Cascade . . . 133
9.4 Summary . . . 134

10 Power and energy optimisations **137**
10.1 Multiplier optimisation . . . 137
10.1.1 Reducing multiplier idle power . . . 138
10.1.2 Preventing wasteful input latch toggling . . . 140
10.2 Instruction cache width reduction . . . 141
10.2.1 Existing approaches . . . 141
10.2.2 Leveraging the application-specific instruction word . . . 143
10.2.3 Instruction word encoding algorithm . . . 146
10.2.4 Experimental analysis . . . 149
10.3 Idle and sleep modes . . . 155
10.4 Summary . . . 158

11 Physical layout and place & route **160**
11.1 Physical layout using Synopsys Astro . . . 160
11.1.1 Library creation . . . 161
11.1.2 Floorplanning . . . 162
11.2 Physical layout using Cadence Encounter . . . 163
11.2.1 Initial configuration . . . 163
11.2.2 Floorplanning . . . 167

11.2.3 Power planning . . . 167
11.2.4 Macro placement and clock tree synthesis . . . 168
11.2.5 Post-layout analysis . . . 169
11.3 Summary . . . 174

12 Case study **175**
12.1 Cascade energy analysis flow overview . . . 175
12.2 Cascade design flow . . . 179
12.2.1 Initial configuration . . . 179
12.2.2 Architectural synthesis . . . 180
12.2.3 Function offloading . . . 181
12.2.4 Functional simulation . . . 182
12.2.5 Data cache configuration . . . 182
12.2.6 Candidate architecture generation . . . 185
12.3 Energy analysis within Cascade . . . 186
12.3.1 Obtaining coprocessor energy results . . . 186
12.3.2 Analysis of results produced by Cascade . . . 187
12.4 Summary . . . 191

13 Conclusion **192**
13.1 Project summary . . . 192
13.2 Future work . . . 196

Appendices

A Top-level Tool Flow Script **198**
A.1 Integrated synthesis, simulation and power script . . . 198
A.2 Independent power and energy analysis script . . . 207

B Open-source cores support files **209**
B.1 LEON processor device.vhd . . . 209
B.2 LEON processor config.h . . . 212
B.3 LEON processor synthesis script . . . 215
B.4 OpenRISC 1200 toolchain build script . . . 217

C MediaBench build/test scripts **221**
C.1 Sample MediaBench build.tcl . . . 221
C.2 Sample MediaBench test.tcl . . . 223
C.3 MediaBench configuration file—default.xml . . . 224

D Functional unit analysis files **225**
D.1 Output bank simulation script . . . 225
D.2 Output bank testbench . . . 230
D.3 Functional unit active cycle energy script . . . 233

E Memory energy analysis code **238**
E.1 Memory library creation script . . . 238
E.2 Memory analysis Java source code . . . 242
E.3 Memory library CSV file (130 nm) . . . 248

E.4 Memory library CSV file (90 nm) 251

F Leakage power analysis script **254**

G Technology library power comparison script **256**

H Physical layout and place & route scripts **258**
H.1 Milkyway library creation script 258
H.2 JupiterXT floorplanning script 260
H.3 First Encounter physical layout script 261

I Case study supporting files **263**
I.1 TSMC 90 nm technology.xml energy entries 263
I.2 TSMC 130 nm technology.xml energy entries 270
I.3 Sample analysis summary 276

J Embedded Systems Conference 2005 Paper **283**

K CODES-ISSS 2005 Paper **290**

L Design Automation Conference 2007 Paper **296**

References **300**

List of Figures

1.1 Overview of project timeline . . . 3

2.1 Layout of typical power-efficient SoC . . . 8
2.2 Simplified Cascade design flow . . . 10
2.3 Typical system integrating coprocessor . . . 11
2.4 Increases in SoC functionality against battery energy density . . . 12
2.5 Requirements of high-level SoC estimations and reprogrammability . . . 12
2.6 Sources of power dissipation in CMOS devices . . . 15

3.1 Overview of power analysis tool flow . . . 23
3.2 Example SAIF file output . . . 33
3.3 Switching activity annotation report . . . 35
3.4 Power summary and worst cells report . . . 37
3.5 Top-level cells energy report . . . 38

4.1 TestCore power summary report for 130 nm process . . . 42
4.2 TestCore power summary report for Vendor B 130 nm process . . . 44
4.3 LEON2 processor architecture . . . 47
4.4 SOCks project design flow . . . 50
4.5 Artisan SRAM read cycle timing . . . 56
4.6 LEON2 SRAM read cycle timing . . . 56
4.7 LEON2 processor core power report . . . 58
4.8 Cascade coprocessor Fibonacci power report . . . 59
4.9 OpenRISC 1200 processor architecture . . . 61
4.10 OpenRISC 1200 core power summary report . . . 67

5.1 Flat function profile for MPEG2 decode benchmark . . . 76

6.1 Functional unit block diagram . . . 81
6.2 Excerpt from multiplier testbench . . . 84
6.3 Example multiplier testbench stimulus file . . . 85
6.4 Multiplier input signal masking . . . 86
6.5 Multiplier input mask implementation . . . 87
6.6 Output bank layout . . . 91
6.7 Output bank best case stimulus file . . . 92
6.8 Output bank worst case stimulus file . . . 93

7.1 Example Artisan memory data file . . . 101
7.2 Excerpt from DesignWare SRAM IP testbench . . . 104

8.1 Design Compiler topographical mode inputs and outputs . . . 109

8.2 Clock tree power as a function of total area . . . 112
8.3 Clock tree power as a function of logic area . . . 114
8.4 D-type register with load-enable signal . . . 118
8.5 Latch based clock gated register . . . 118
8.6 TSMC 90 nm register schematics . . . 119
8.7 Implementation of both register types . . . 119

9.1 Excerpt of power analysis report for ADPCM encode test . . . 132

10.1 Pipelined multiplier stages . . . 140
10.2 Multiplier input latch masking on enable signal . . . 140
10.3 Approaches to code decompression at run-time . . . 142
10.4 Encoded instruction word layout . . . 145
10.5 Comparison of instruction word formats . . . 146
10.6 Opcode assignment algorithm flow . . . 147
10.7 Instruction decode VHDL code excerpt . . . 150
10.8 Coprocessor area using encoded instructions . . . 152
10.9 Coprocessor cycle counts using encoded instructions . . . 153
10.10 Coprocessor energy use using encoded instructions . . . 154
10.11 Coprocessor with sleep controller to reduce idle energy . . . 157

11.1 Synopsys Astro platform Milkyway library creation . . . 162
11.2 First Encounter physical layout flow . . . 164
11.3 First Encounter input configuration file . . . 166
11.4 Post-layout power analysis report . . . 170
11.5 Gate level power analysis report . . . 172

12.1 Cascade design flow incorporating energy analysis . . . 176
12.2 Functional simulation and instrumented binary flow . . . 183

List of Tables

3.1 VCSi command options 31
3.2 Key to SAIF file entries 33

4.1 Vendor A 130 nm NAND2X1 cell parameters 44
4.2 LEON2 cache ram cell sizes 55

5.1 MediaBench suite offloaded functions 77
5.2 MediaBench suite coprocessor evaluation (TSMC 130 nm) 78

6.1 Functional units available to Cascade 80
6.2 Multiplier operating mode corner cases 83
6.3 32-bit multiplier power usage under various operating conditions 83
6.4 64-bit multiplier power usage under various operating conditions 86
6.5 32-bit multiplier power usage with input mask 87
6.6 Energy per inactive cycle of functional units 89
6.7 Energy per active cycle of functional units 90
6.8 Output bank energy per cycle (8-bit width) 94
6.9 Output bank energy per cycle (16-bit width) 95
6.10 Output bank energy per cycle (32-bit width) 95

7.1 Artisan memory blocks used by Cascade 99
7.2 DesignWare IP memory blocks used by Cascade 103
7.3 Tag RAM power consumption (100 MHz operation) 105

8.1 Area and clock tree power using topographical and wire load models . . . 111
8.2 Clock tree power estimate calculated from total area 113
8.3 Total area and logic area compared with clock tree power 114
8.4 Clock tree power estimate calculated from logic area 115
8.5 Area, delay and power figures for TSMC 90 nm cells 120
8.6 Cell power comparison between gated and ungated clock designs 122

9.1 Average functional unit leakage current in 130 nm & 90 nm technology . . 128
9.2 Standard deviation across all leakage tests in 130 nm & 90 nm technology 129
9.3 Output bank leakage statistics in 130 nm & 90 nm technology 132
9.4 Functional unit leakage power excluding memory blocks and output banks 136

10.1 Coprocessor synthesis templates provided by Cascade 138
10.2 Multiplier power savings 139
10.3 Idle power comparison 155
10.4 Proportion of energy consumed in active and idle states 156

11.1 Floorplanning parameters for coprocessor in 130 nm technology 167
11.2 Floorplan and placement area report 171

12.1 Coprocessor architectural synthesis required units templates 180
12.2 Average power consumption estimates (TSMC 90 nm technology) 189
12.3 Average power consumption estimates (TSMC 130 nm technology) 190

Acknowledgements

I would like to offer my thanks to everyone who has supported me throughout this project.

I owe a huge gratitude to Critical Blue Ltd., who sponsored me throughout this project, for both financial support and the knowledge and expertise offered by my colleagues there. Particular thanks go to Richard Taylor, my industrial sponsor within Critical Blue, who has mentored me during much of the project, and implemented much of the functionality required within Critical Blue's products to support my project. I would also like to thank the Engineering and Physical Sciences Research Council (EPSRC) for financial support, both personal and for training and travel, throughout the project.

My academic supervisors, Prof. Tughrul Arslan and Dr. Ahmet Erdogan in the first half of the project, and Prof. Nigel Topham in the latter half, have all been an excellent source of expertise, knowledge and advice, both in technical terms and in helping to plan and document the project.

Thanks to all the staff at the Institute for System Level Integration, particularly Sandie Buchanan who supported me through the first few years of my project, and recently Siân Williams who has offered a prompt and friendly response to many issues throughout my research period.

Finally, I would like to thank friends, family and fellow students who have all provided moral support, particularly through the more stressful periods of the project.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Paul Morgan*)

List of Abbreviations

ALU	Arithmetic Logic Unit
ASIP	Application Specific Instruction (set) Processor
CISC	Complex Instruction Set Computer
CMOS	Complimentary Metal Oxide Semiconductor
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DRAM	Dynamic Random Access Memory
DSE	Design Space Exploration
DSP	Digital Signal Processor
EDA	Electronic Design Automation
FU	Functional Unit
IDCT	Inverse Discrete Cosine Transform
IP	Intellectual Property
ISS	Instruction-Set Simulator
ITRS	International Technology Roadmap for Semiconductors
NRE	Non-Recurring Engineering
PDA	Personal Digital Assistant
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SoC	System-on-Chip
SAIF	Switching Activity Interchange Format
SRAM	Static Random Access Memory
Tcl	Tool command language
VCD	Value Change Dump
VLIW	Very Long Instruction Word

1. Introduction

This project is sponsored by Critical Blue Limited, who are based in Edinburgh, Scotland. Critical Blue is an Electronic Design Automation (EDA) company, developing software to enable performance acceleration in embedded devices with minimal design time. The flagship product at the initiation of this project is *Cascade*, an EDA tool that performs automatic coprocessor synthesis, around which most of the project is centred.

1.1 Project aims

Areas of interest during the conception phase of the project consisted of:

- Power, area and timing awareness for coprocessor synthesis
- Optimal interconnect fabric architectures for specific data flows
- Optimal cache configuration for specific memory access patterns and system bus loads
- Stream processing execution model for coprocessors
- Automatic partitioning across multiple coprocessors to minimise bus traffic and system power

As the project has developed, power and energy considerations have become the overriding topics of interest, mainly due to a high level of commercial importance attributed to those capabilities within an EDA tool. Therefore research has focused on areas that contribute toward a high-performance automated power and energy analysis capability that can be integrated within Cascade, along with a derived optimisation capability, with consideration given to the area and timing effects of any applied optimisation.

1.2 Project timeline

This section highlights key events that occurred at various points throughout the project, giving a general coarse-grain overview of how the project developed over time.

The initial nine months of the programme were spent undertaking the required 120 technical credits on a full-time basis at ISLI. The project was not defined until around month 6 of this period, therefore during the latter four months some time was spent on preliminary research, such as literature searching, in parallel to full-time technical modules. All modules were successfully completed at the first sitting, and the required 120 credits obtained, with the examination results attained averaging above MSc distinction level.

Around June 2004 full-time research on the project commenced at Critical Blue's offices in Edinburgh. This arrangement continued for the rest of the project, with business modules being undertaken on a part-time evening class basis to allow full-time research to continue uninterrupted. 45 of the required business credits were undertaken at the Hunter Centre for Entrepreneurship at the University of Strathclyde, between September 2005 and September 2006. This consisted of 3 distinct modules, each taking around 3–4 months. The remaining 15 business credits were undertaken on a distance learning basis, to be worked on in spare time to a flexible schedule, with guidance from the Hunter Centre for Entrepreneurship. The distance learning module consists of some market research (in this case on the Cell Broadband Engine), undertaken on behalf of the sponsoring company, along with an analysis of the entrepreneurial behaviour within the company, both at the early start-up stage and once the company has become more mature. The various stages of the module were completed between autumn 2006 and summer 2007. With the conclusion of the distance learning module, all 60 business credits were successfully attained, completing the 180 non-research credits required as part of the Engineering Doctorate programme.

Academic supervision and guidance of the project was initially provided by Prof. Tughrul Arslan and Dr. Ahmet Erdogan. However, due to commercial reasons not directly related to the project, it was decided that a change of academic supervisor would be necessary. Therefore in October 2005 supervision by Prof. Arslan and Dr. Erdogan ended, and in December 2005 Prof. Nigel Topham was appointed as academic supervisor. This change had little direct effect on the overall direction and key objectives of the project, which were guided primarily by commercial interests and the business requirements of the sponsoring company, although

differences in the academic guidance offered inevitably influenced some details of how the project progressed after December 2005. Figure 1.1 shows a coarse-grain overview of the programme time line. More detailed examinations of individual work done throughout the project period will be considered in the following section.

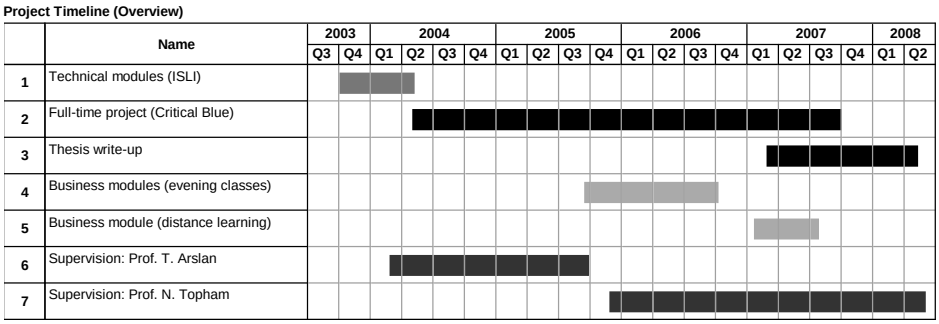


Figure 1.1: Overview of project timeline

1.3 Thesis organisation and project outcomes

The work detailed in this thesis is organised as follows. The relevance of the work in the industrial and academic context is discussed in chapter 2. A coprocessor power evaluation tool flow is developed in chapter 3, with much of the framework for which having been developed during evaluation of open-source processor cores in chapter 4. The MediaBench benchmarking suite is ported to ARM, and subsequently accelerated using coprocessors synthesised by Cascade in chapter 5. This work provides a consistent and relevant analysis framework for work in the proceeding chapters.

Models for the functional units used by Cascade are created in chapter 6, and similar models for memory blocks and register files are characterised in chapter 7. Analysis of clock tree power, and implementation of clock gating to reduce this power, is examined in chapter 8. The final component of the analysis model, leakage power, is tackled in chapter 9.

Optimisations to reduce the power and energy usage of coprocessors, while minimising the impact on performance and area, are examined in chapter 10. A back-end flow comprising physical place & route is undertaken in chapter 11, with the aim of comparing the results of high-level estimates with high accuracy results available at a low level of abstraction.

Finally, a case study is carried out in chapter 12, detailing the complete Cascade flow along with the newly-integrated energy analysis functionality. The performance of this functionality, in terms of both speed and accuracy, is compared with the traditional gate-level analysis method detailed in chapter 3.

The key achievements of the project can be briefly summarised as:

- Implementation of a fast, detailed, early-stage coprocessor energy analysis model, comprising of dynamic, static and clock tree energy components.
- Examination of energy optimisations, focused on units with high energy consumption, with consideration given to impact on area and timing; examples include instruction cache compression, multiplier idle mode and coprocessor sleep mode.
- Evaluation of the efficacy, and analysis of the optimal configuration, of clock tree gating as applied to coprocessors synthesised by Cascade.
- Verification of the accuracy of the developed functionality against existing low-level analysis flows.

In the course of this research, several assumptions were made and limitations identified, to enable feasible, high-performance analysis and optimisation techniques to be developed within the available time frame. These limitations may be re-examined in future research tasked with expanding the scope and accuracy of the work carried out in this project. Briefly, the identified issues are:

- Cascade supports multiple hosts and/or multiple coprocessors within a platform. For simplicity, a single host and single coprocessor are assumed throughout this work; however the techniques developed are extensible to multiple host/coprocessor platforms.
- Adjustments to the optimisation methods that may be beneficial with the availability of external architectural optimisations (dual Vt libraries, or dynamic voltage/frequency scaling) are largely unexplored in this work. This is partly due to non-availability of suitable libraries to test with, and partly due to the increased complexity that would result.
- Energy optimisation of coprocessor blocks is carried out in a static manner and stored in Cascade's internal library, rather than being performed dynamically at run time.

- Cascade's function offload identification mechanism targets cycle count reductions, which may not be optimal when instead clock frequency reduction is desired (as may be the case when dynamic voltage and frequency is used).

In a broader context, this project has shown that high-level power modelling of configurable processors is feasible, despite the complexity inherent in performing such analysis at a high level of abstraction. The accuracy obtained is well within the bounds of what could be considered as useful for early stage analysis, taking into account the large speed-up offered. Such a development is relevant to other configurable processor architectures, as the techniques could be suitably adapted to offer similar benefits to those conferred to Cascade.

Throughout the project, although most of the work is focused on developing energy analysis and optimisation functionality that can be deployed within Cascade, efforts have been made to ensure a level of general applicability of the research in a broader scope. Thus, much of the work undertaken could be adapted for use in other types of configurable processors typical in system-on-chip platforms, and this extensibility of the research is discussed as part of the summary of work carried out at the end of each chapter.

A more detailed examination of the project outcomes, along with a discussion of the limitations of the project and suggestions for future work, are summarised in chapter 13.

2. Industrial and Academic Context

This chapter sets the context for the research project, both in terms of industrial and commercial relevance, and how it relates to existing work in the academic field.

2.1 Industrial relevance

Continual advancement in silicon technology has seen (and continues to show) exponential increases in the number of transistors available on a given silicon area. Since the late 1990s the huge level of functionality available on a single die has driven a move away from multiple chips performing different functions on a printed circuit board, and toward integration of an entire system onto a single silicon chip—the system-on-chip (SoC) era.

Modern SoC projects typically combine embedded microprocessors, memories, dedicated hardware processing blocks, and analogue or mixed-signal blocks. The complexity of such projects results in high costs both in the design stage, and during the initial stage of manufacture such as mask and tooling costs. A typical mask cost for a 90 nm wafer can exceed $1 million [1]. These one-off costs of bringing a product to market are known as non-recurring engineering (NRE) costs. As of 2003, NRE costs accounted for 62% of the total cost of a typical SoC project [2].

It is imperative to the cost-effectiveness of a project that the device gets to market as early as possible, and has the longest life-cycle possible to ensure that commercial income from the project is sufficient to cover the high NRE costs. These factors are driving the trend for an increasing proportion of functionality being implemented in software rather than fixed hardware, as shorter development times and the ability to reprogram the device make software implementations highly attractive.

There are two key disadvantages to software implementations running on standard embedded processors: poor performance, and energy inefficiency. There will always be a significant performance penalty compared with running on dedicated custom hardware, which can offer exactly the required resources for the target application while being tuned to minimise latency. With regard to power and energy consumption, specialised hardware can often be an order of magnitude more efficient than a general-purpose processor when implementing the same functionality. These drawbacks mean it is often necessary to resort to implementing some of the functionality of a system in custom hardware blocks to overcome the limitations of a general-purpose processor, revisiting the issues of longer design time and an inability to reprogram the device once it is implemented in silicon.

Combining many of the benefits of both fixed custom hardware and general-purpose processors is the Application Specific Instruction-set Processor (ASIP). This is a type of software processor that has an architecture and instruction set customised to be a better fit for the target application. Thus ASIPs frequently offer better performance, lower area, and lower power and energy consumption compared to an equivalent general-purpose processor. As an example, an ASIP deployed in a set-top box performing video decoding and programme guide functions will often match the performance of a leading edge desktop processor, but with one quarter of the silicon area and running at one quarter of the clock frequency [3], in turn resulting in a corresponding reduction in power and energy consumption.

Being software driven, ASIPs offer flexibility via reprogramming, helping to reduce design time and increase longevity in the market compared to fixed hardware. One potential pitfall of deploying an ASIP over a general-purpose processor is that performance will be reduced if the application changes significantly from the original target application, although many embedded applications undergo only minor changes, such as bug fixes and minor functionality improvements, over their lifespan. Additionally, difficulty designing efficient ASIPs within time and cost budgets has historically been a barrier to their adoption in many projects. An analysis and summary of the commercial and technical motivations behind the move away from fixed hardware and toward ASIPs is published by Keutzer, Malik and Newton in [4]. The 2005 edition of the *International Technology Roadmap for Semiconductors* (ITRS) [5] predicts that ASIPs, in the form of processing engines (PEs) will play an increasing role in future SoC design, particularly where power-efficient design is a key criterion, with a typical design being similar to the form shown in Figure 2.1.

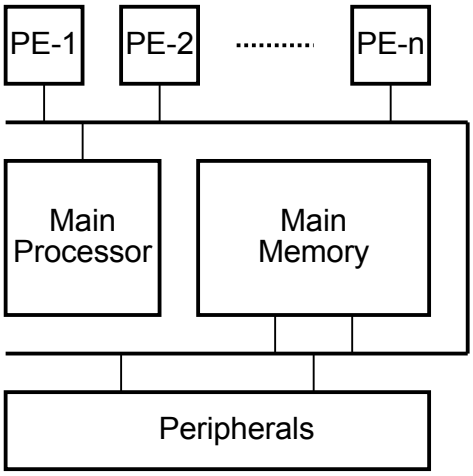


Figure 2.1: Layout of typical power-efficient SoC [5]

Critical Blue has developed a solution, in the form of an EDA tool called *Cascade*, that effectively and efficiently explores the ASIP design space. Coprocessor synthesis is a technology that allows software functions to be offloaded from the main processor of a system directly onto an automatically generated coprocessor in order to improve overall application performance, while retaining programmability. Generated coprocessors are micro-coded VLIW ASIPs consisting of an array of functional units and register files communicating over a fast interconnection fabric, along with independent instruction and data caches. Each coprocessor is designed specifically to provide significantly accelerated performance for the offloaded functions.

There are several commercial vendors who offer competing technologies, with varying degrees of customisation and power awareness. Examples include ARC, ARM OptimoDE, Mimosys, and Tensilica. The key difference between these technologies, and the solution offered by Cascade, is that Cascade creates pure application-specific coprocessors and the associated software to run on the coprocessor. Other solutions are typically customisable or extensible processors, that allow hardware blocks to be selected or added for certain specialised functions, and/or are manually designed under the user's control. Cascade's key strengths are that the entire process is highly automated, and the coprocessor runs alongside a standard host processor, allowing the coprocessor to be very specialised, and therefore efficient, for running specific, computationally intensive code kernels. The suitability of one or more of these potential solutions is highly dependent on the specifics of the target platform and application.

Cascade analyses a binary executable targeted at the host processor of the system, for example an ARM 7, ARM 9, MIPS or PowerPC processor. This executable is typically compiled using the conventional software development flow for that platform, with no changes being required to the code itself or the build environment. An execution profile generated by the standard tools can be loaded into Cascade to highlight hot spots in the code and thus aid in the selection of one or more functions to offload to a coprocessor. Normally any child functions of offloaded functions will also be offloaded, ensuring that the entire execution cycle of offloaded functions remain on the coprocessor. Such child functions are automatically determined by Cascade (except in the case where function pointers are present, the destination of which cannot be statically determined).

Once the functions to be offloaded have been identified, Cascade generates a functional simulation in the form of a new executable binary for the target host processor. This contains hooks to monitor the both the execution behaviour and the memory access activity of the offloaded functions. The executable is run on a standard instruction set simulator (ISS) such as ARM's *Armulator* or the free GNU ARM ISS. The modifications to the binary result in the creation of execution and memory trace files, which can then be read into Cascade to be used in the design space exploration phase (DSE) for both the execution logic and the cache memories.

With knowledge of both the execution and memory access behaviour of the offloaded functions, Cascade can extract parallelism inherent in the functionality of the code and implement an ASIP coprocessor and corresponding instruction set with the aim of finding the optimal solution within the constraints set by the user—typically the highest possible performance within an area limit. The extent of DSE undertaken is dependent upon the effort level selected by the user, higher effort will explore more candidates thus potentially obtaining a more favourable result, at the cost of longer run-time.

When DSE has completed, Cascade presents a graph plotting the area and performance of each generated coprocessor candidate. This allows the user to select the best suited candidate for their project goals. Cascade will then proceed to generate the coprocessor hardware (in synthesisable VHDL or Verilog, or a cycle-accurate SystemC model), along with the microcode that implements the offloaded functionality in the coprocessor's custom instruction format. Testbenches are also generated for verification purposes. The host processor executable is modified by Cascade to perform the necessary communication between the host and coprocessor, and the coprocessor hardware has an integrated bus interface (typi-

cally AMBA AHB master or slave, depending on system requirements). Cascade greatly simplifies and expedites the process of generating ASIPs to accelerate, or reduce the power consumption of, embedded applications. Figure 2.2 gives an overview of the Cascade coprocessor design flow, while Figure 2.3 shows a typical system block diagram integrating a Critical Blue coprocessor implemented by Cascade.

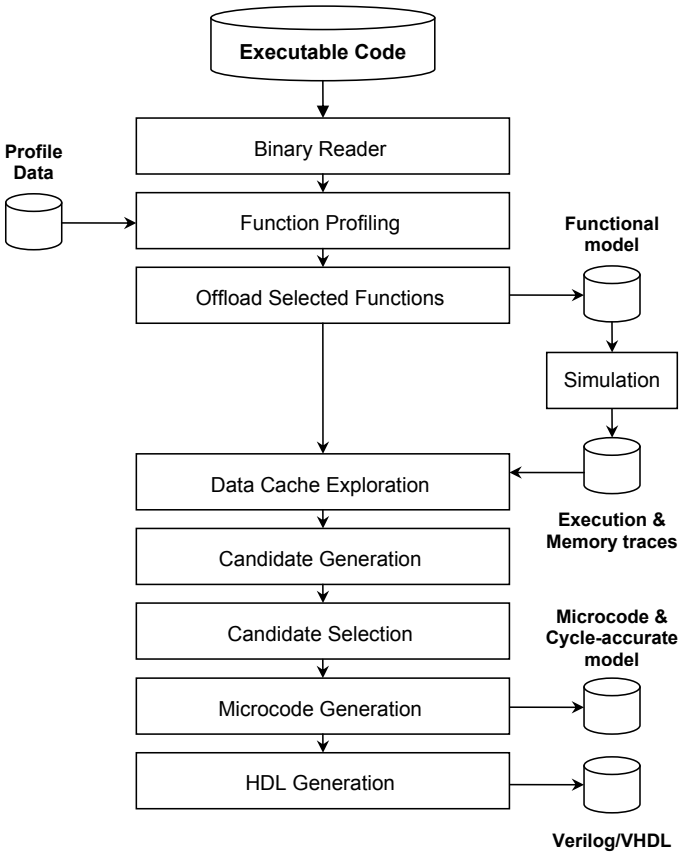


Figure 2.2: Simplified Cascade design flow

Power and energy considerations are becoming increasingly important in the embedded market alongside traditional key considerations of area and performance [6]. Lower energy consumption requirements, along with design time and lifespan pressures have driven a surge in the adoption of application-specific processors [7]. To maximise the energy benefit of deploying an ASIP rather than a general-purpose processor, the ASIP must be designed using an energy aware approach, which requires that energy estimates be determined at an early stage of design space exploration to allow an appropriate architecture to be selected. The need for both reprogrammability and accurate energy estimates is highlighted in Figure 2.5.

The terms power and energy are often used incorrectly or interchanged, particularly with regards to “low power” or “energy efficient” devices. Power is an instantaneous measurement

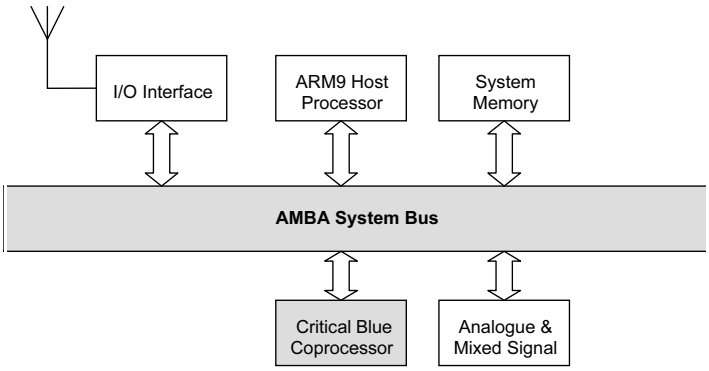


Figure 2.3: Typical system integrating coprocessor

of the amount of work done per unit of time; whereas energy is the capacity for a system to do work. The relationship between the two values can be seen in Equation 2.1.

$$E = P_{average} \times t \tag{2.1}$$

where E is energy measured in Joules (J), P is average power measured in Watts (W), and t is time measured in seconds (s). Therefore, over a known time period T, the total energy can be calculated by integrating the power over that time:

$$E = \int_{t=0}^{T} P(t) \tag{2.2}$$

It is clear from Equations 2.1 and 2.2 that power and energy are very closely related, but optimising for one does not necessarily improve the other. For example, reducing the clock frequency of a CMOS hardware block will reduce power consumption through a reduction in switching activity, but will also proportionally increase the length of time required to process the same amount of data. Assuming no other factors are changed, the energy consumed over that run will stay the same even though average power consumption has reduced. When other factors are taken into account, such as leakage current, the energy consumed may actually increase with a reduction in clock frequency, if the device is powered down at the end of its run time. On the other hand, a lower clock frequency may allow a lower supply voltage to be used, which can significantly reduce both power and energy consumption. Factors like these are taken into consideration throughout this project.

There are good reasons behind the desire to reduce both power and energy consumption. Instantaneous power peaks place more demanding requirements on power rails and interconnects within a chip, as well as external provision for dissipating heat from the device. However for typical embedded and portable systems it is the total energy consumed over a particular application run that is of prime concern, as these devices are often battery powered, meaning that they have a finite source of energy from which to operate. Therefore the amount of energy consumed during normal operation has a direct influence on the length of time the device can operate before the battery is depleted and needs to be recharged—a significant driver of desirability in the market place.

Battery technology development has lagged far behind the pace of semiconductor technology advancement, meaning that the huge increases in available functionality within a chip have not been matched by improvements in battery capacity. The 2003 edition of the ITRS [8] predicts that battery energy density will double from 200 Wh/Kg[1] in 2006, to 400 Wh/Kg in 2012—far short of the expected increase in functionality during that period, as can be seen in Figure 2.4. In reality the available energy capacity is likely to stay near constant despite these small improvements in energy density, due to the continuing trend of decreasing size and weight resulting in diminishing physical dimensions of the battery. Thus the 2003 ITRS predicts that the required average power remains constant through to 2018 despite predicted functionality increases of several orders of magnitude.

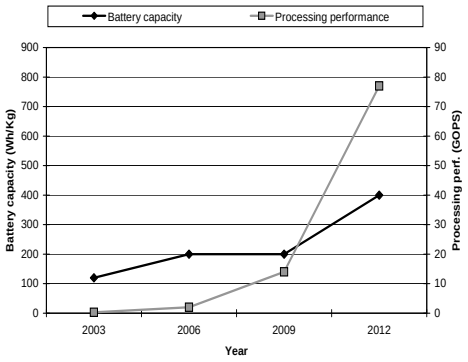


Figure 2.4: Increases in functionality against battery energy density [8]

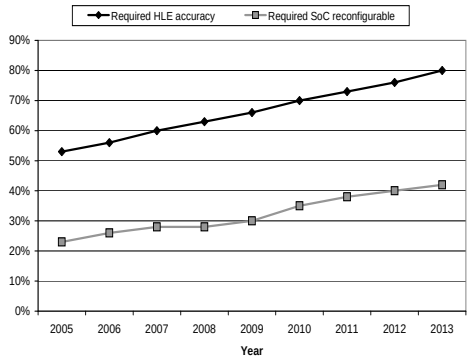


Figure 2.5: Increases in % of required high-level SoC estimation accuracy and reprogrammability [8]

Analysis and optimisation of energy consumption is usually tackled by considering the power consumption at different stages of device operation, averaging those, and taking into account execution time. Although this document will often discuss techniques to reduce power con-

[1]Wh = 1 Watt for a period of 1 hour = 3600 J. Wh/Kg = 3600 J energy per Kg physical weight.

sumption, in most cases this will be toward a goal of an overall energy consumption reduction, therefore power reduction techniques will take into account any effect on execution time to ensure that energy consumption is effectively reduced.

A variety of techniques for analysing power and energy have emerged over the last decade. Analysis for hardware blocks is typically performed at the RTL or gate level, requiring simulation at that level to characterise the switching activity of internal nodes, in order that accurate results may be obtained. Such simulation is very time consuming for any realistic application, rendering such an approach infeasible at the design space exploration stage of an ASIP. Analysing power at the instruction-level is a higher level, and therefore much faster, approach that can be applied to software processors [9]. However this approach relies on pre-characterisation of each instruction used by the processor being analysed, making it more suited to fixed processor implementations running different software, rather than the design stage of an ASIP where hundreds of potential architectures may be considered. System-level analysis taking into account both hardware and software influences is essential to effective selection of an appropriate architecture as early as possible in the design process [10] [11]. Although Cascade generates both the hardware and software for the coprocessor, it is not true HW/SW co-design, as the source software functionality was pre-designed before analysis by Cascade rather than partitioned between hardware and software implementations [12].

This research investigates ways in which the generation of coprocessors may be optimised to take into account the requirements of SoC implementation. Key topics of interests include power and energy awareness, analysis and optimisation for application-specific coprocessor synthesis; of particular interest is a high-level modelling scheme that allows power and energy to be estimated at an early stage of the design process, quickly enough to be performed on a large number of potential candidates.

Although the research presented in this thesis is focused on particular applications to be employed within Cascade, much of the work on high-level energy analysis and optimisation is applicable in a more general context. The most obvious external candidates that could benefit from this work are other types of application-specific processors, however the applicability of the underlying methodology is much wider—development of configurable general purpose processors could employ a suitably modified approach toward implementing high speed energy analysis techniques similar to those developed as part of this project.

2.2 Literature review

Minimising power and energy consumption has become the key criterion in many designs, ranging from portable computing devices to embedded systems. As a result, a substantial volume of research has been undertaken on this topic. This section undertakes a review of previous work in the field of power and energy analysis and optimisation, initially in the broad scope of digital CMOS circuits, then later paying particular attention to work that considers these issues specifically in relation to ASIPs.

There are three distinct sources of power consumption in CMOS devices, as illustrated in Figure 2.6 and summarised in Equation 2.3:

$$P_{total} = P_{dynamic} + P_{short-circuit} + P_{static} \tag{2.3}$$

where dynamic power is due to charging and discharging the node capacitance when a circuit switches, short-circuit power is dissipated when both NMOS and PMOS gates are momentarily conducting during switching, and finally static power is continuous dissipation of leakage current while the device is powered up, regardless of any activity taking place. Occasionally dynamic power and short-circuit power are combined as just dynamic power, as both are dependent upon gate switching.

Equation 2.3 can be expanded into its dynamic, short-circuit and static power components as shown in Equations 2.4, 2.5 [13] and 2.6 [13], respectively.

$$P_{dynamic} = KC_L V_{dd}^2 f \tag{2.4}$$

where K is average switching activity in one clock cycle; C_L is load capacitance; V_{dd} is supply voltage; and f is the clock frequency.

$$P_{short-circuit} = K\frac{\beta}{12}(V_{dd} - 2V_T)^3 f\tau \tag{2.5}$$

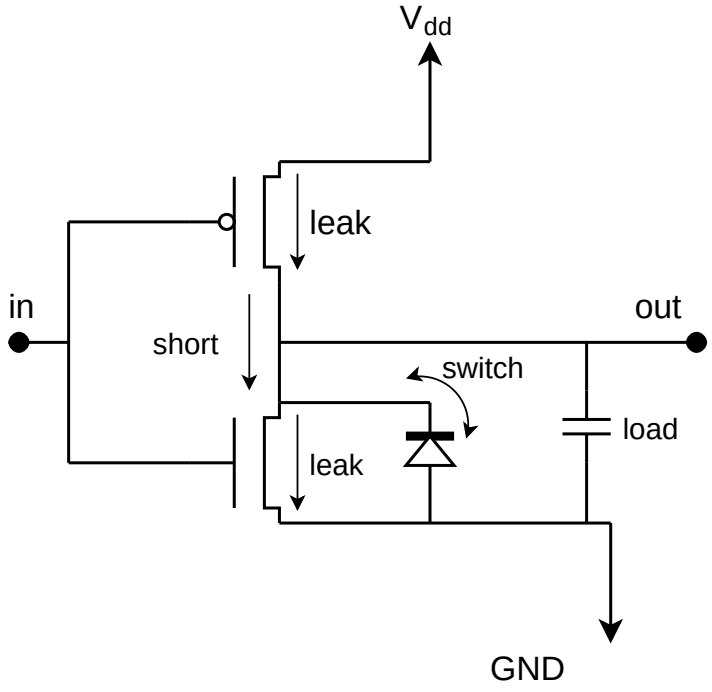


Figure 2.6: Sources of power dissipation in CMOS devices; switch = load switching power (active), short = short-circuit power (active), leak = leakage power (static)

where K is average switching activity in one clock cycle; β is the gain factor (measured in $\mu A/V^2$) of a MOS device; V_{dd} is supply voltage; V_T is threshold voltage; f is the clock frequency; and τ is the transition time between on/off states.

$$P_{static} = VI_{leakage} \tag{2.6}$$

Static power is comparatively the simpler of the three elements to analyse, as it is independent of level of the activity within a hardware component (although steady-state input vectors do influence static power dissipation). Therefore in the context of ASIPs static power depends primarily on the hardware of the ASIP rather than the software being executed. Although static power accounts for an increasing proportion of overall power consumption with every process technology generation (up to 45% of overall power worst-case in 90 nm process technology [14]), leakage power is dominated by physical design factors whereas dynamic power can be tackled more readily at the system level. Therefore, although static power is an important factor that will be considered in analysis, dynamic and short-circuit power receives more attention due to a higher level of analysis complexity, and the opportunities present to optimise for dynamic power at the system level. Static power will be

considered later in this section; in the proceeding paragraphs the focus is on dynamic power.

Often in the literature, dynamic power is a term used to describe both short-circuit and dynamic power consumption combined, since both these elements are strongly correlated with switching activity within a circuit. Therefore the remainder of this section will use the term "dynamic power" or "dynamic energy" to mean combined dynamic and short-circuit power or energy, unless otherwise noted.

Examining dynamic power in the context of Equation 2.4, the most influential element in the equation is voltage, which has a quadratic relationship to dynamic power. However a lower voltage results in slower switching gates, therefore to enable a lower supply voltage it is often necessary to simultaneously lower the clock frequency which provides an additional power saving (although not an energy saving in itself, as the execution time is proportionally increased). Doing so reduces throughput which is often undesirable in embedded systems, as there may be processing deadlines, for example in real-time multimedia applications.

Architectural techniques for lowering power and energy consumption in digital CMOS circuits have been successfully applied to application-specific signal processing devices for over a decade [15] [16] [17]. There are additional savings to be made at a higher level, such as at the system design stage. Relocating some of the functionality of a software application from a general-purpose processor to an application-specific coprocessor allows the processing hardware to exploit concurrency within the algorithm, allowing the operating frequency to be reduced while maintaining throughput. The resultant timing slack allows supply voltage to be lowered in many cases, further reducing power consumption [18]. For cases where it is desirable to retain software programmability, moving away from general-purpose devices toward more specific architectures with a tailored instruction set provides an opportunity for more efficient use of the hardware resources, resulting in reduced power and energy consumption through lower levels of control and data path switching activity when implementing the same algorithm [19]. The compiler can be optimised to reduce energy consumption of the software component [20].

Previous work in the field of application-specific instruction set design with power constraints dates back to 1993. Alomary *et al.* [21] describe a method of selecting an ASIP instruction set that maximises the chip performance under the constraints of chip area and power consumption. The same authors demonstrate a hardware/software co-design tool, *PEAS-I*, that aids the designer in developing an ASIP from the target application source

code written in C using a formal method [22]. Although power is mentioned as a design constraint, it is paid very little attention in the paper, and is completely ignored for the experimental results. The field of ASIP design has also moved on significantly since the paper was published, with much more complex designs and tools having since become commonplace.

Several other authors offer different methods for implementing ASIPs under power or energy constraints. Binh *et al*. [23] describe a partitioning algorithm as part of a process for synthesising high-performance ASIPs. Their algorithm uses an approach similar to that used by PEAS-I, generating application-specific processors with multiple identical functional units. Once again, although power consumption is mentioned as a constraint, it does not feature in the experimental results, which concentrate on the area/performance trade-off. The same authors also present a method of creating ASIPs with the lowest gate counts under execution cycle and power constraints [24].

The importance of carefully selecting the instruction set when designing an ASIP, specifically referring to power and energy consumption, is demonstrated by Dougherty *et al*. [25]. The paper focuses on demonstrating and proving the theory, rather than providing a generally applicable analysis method.

A different approach is to create macro-models of hardware blocks that can be characterised for power depending on input and output sequence, such as that described in [26] and improved upon in [27]; the latter particularly targeting behavioural synthesis. Both works utilise three-dimensional look-up tables that reference average input probability, average input data switching activity, and average output data switching activity. Once the table is constructed, power can be analysed quickly and with high accuracy if the number of samples is reasonably high, but completely populating the look-up table is a slow and complex process which does not particularly lend itself to early-stage ASIP design with varying instruction sets and data streams to deal with.

A comparison of the energy consumption of a range of inverse discrete cosine transform (IDCT) implementations is offered by Vermeulen *et al*. [28]. Their work shows that fixed custom hardware is unsurprisingly the most energy efficient implementation, but that a carefully designed ASIP can perform the same computation with around double the energy consumption of the fixed hardware—often a very worthwhile trade-off to obtain a degree of programmability. By comparison, a more general purpose processor such as an ARM uses more than 8x more energy than the ASIP. A high performance DSP produced similar energy

results to that of the ARM. The authors go on to demonstrate a novel hybrid processor architecture that allows minor pseudo-programmability in fixed hardware by means of transparently re-mapping any changed parts of the application onto the host processor. This approach appears to be effective for very small code changes (the authors suggest a limit of less than 10%), but clearly offers less flexibility than an ASIP. An article by Yang *et al*. [29] offers an analysis of both power and performance of optimised ASIP and fixed custom hardware engines to implement a motion estimation algorithm for video compression. They focus on selecting the most efficient algorithm for each implementation, rather than making a direct power or performance comparison between ASIP and fixed hardware implementations.

Research by Jeng *et al*. [30] shows that the memory hierarchy within an ASIP dominate both energy cost and performance. A detailed article by Wehmeyer *et al*. [31] on the influence of register file size on ASIP energy consumption and execution time appears to agree that memory issues dominate ASIP performance. Further examination of the influence of register file selection is undertaken in [32]. Karuri *et al*. [33] presents a novel memory access profiling technique specifically targeted at ASIP design.

Individual constituent components of ASIPs and their effect on power and energy performance have also been covered in previous work. Kalyanaraman *et al*. [34] consider the effect of the arithmetic logic unit (ALU) on power consumption. They compare four different types of ALU within a digital filter ASIP, and conclude that a more complex ALU performing fewer operations is more efficient than a simpler resource-sharing ALU that requires more accesses to achieve the same throughput. Middha *et al*. [35] present a framework for exploring the design space of ASIPs using custom coarse-grain functional units for performing more complex calculations, alongside conventional fine-grain functional units.

Examining the issue of developing an entire ASIP along with the corresponding software development and verification tools, there are several research works that have produced automation tools in this area. *PEAS-III* is a development of the aforementioned *PEAS-I* tool, developed at Osaka University in Japan [36] [37]. The designer determines the processor architecture, selects the resources the processor should have, and finally the instruction format and interrupts are set. A simulation model and synthesisable VHDL hardware description are generated.

Researchers at the department of Integrated Signal Processing Systems, Aachen University of Technology in Germany developed the Language for Instruction Set Architectures (LISA)

and a framework to help designers and engineers accelerate the process of ASIP hardware and tool set development [38] [39]. LISA is described as a machine description language, where a designer describes the desired resources and operations for a custom processor, allowing the tool to automatically generate both the RTL hardware description of the ASIP, and the associated software development tools including a compiler, linker, assembler and simulator. The technology behind LISA was spun out into an independent company, LISATek, to commercialise LISA. LISATek were in turn acquired by CoWare Inc. in January 2003. Mostafizur *et al*. demonstrate a case study showing the development of an ASIP for network processors using LISA [40].

Neither of the two aforementioned tools appear to offer any form of power or energy analysis or optimisation. Although LISA shows implementation results for a low-power ASIP for DVB-T acquisition and tracking [39], it seems that the low-power aspect is simply a property of the designed processor being application-specific rather than general purpose. It should also be noted that using either of the above tools requires a degree of expertise in processor architecture creation to obtain optimal results.

A more recent overview of the trade-offs between general purpose processors, ASIPs, and fixed custom hardware is written by Shekhar *et al*. [7], taking into consideration the energy performance of all three implementations. Ascia *et al*. [41] offer a framework for exploring and evaluating the design space when developing an ASIP, known as *EPIC-Explorer*, which is freely available for download.

Moving on to static power analysis and optimisation techniques with relation to ASIPs and VLIW processors, a very different approach is required to that taken for dynamic power and energy. Unlike dynamic energy, which is consumed only during switching transitions, static energy is dissipated continually while power is applied to a device. Although this makes the analysis simpler, optimising for static power and energy needs to be performed using more fundamental techniques, typically operating at a lower level.

Perhaps unsurprisingly, cache leakage receives a lot of attention in existing literature. Analysis techniques targeted at embedded applications are examined in [42]. Three simple techniques for reducing static power consumption in microprocessor caches are presented in [43]. Mamidipaka *et al*. [44] developed an analytical model for leakage power estimation in memory arrays such as caches and register files, with some interesting results regarding dual-threshold voltage memory arrays. Zhou *et al*. [45] considered the performance effects of

actively controlling sleep mode for various parts of the cache. Further developing the sleep mode technique, several studies have proposed placing parts of the cache into a "drowsy" mode to reduce leakage while minimising the effect on performance [46][47][48]. Li and Hwang [49] offer a development of sleep mode, known as "Snug" caches, that reduce leakage power while actually improving performance for the benchmarks demonstrated. Guo *et al*. [50] offer a method of characterising both dynamic and leakage energy usage of cache pre-fetch mechanisms, as commonly used in high-performance embedded processors.

Taking logic blocks into consideration, a simple but effective technique of reducing standby leakage current by applying the most efficient input vector is demonstrated by Halter and Najm [51], offering savings of up to 54% on ISCAS-89 benchmarks. A comprehensive review of input vector control techniques and their effect on leakage power is presented by Abdollahi *et al*. [52]. A fast algorithm, based on signal probability, to determine the optimal input pattern for minimal leakage is developed in [53]. An alternative technique, first presented in [54], involves temporarily cutting the supply voltage to unused blocks, and has since become a widely used approach. Zhang *et al*. [55] suggests utilising schedule slack in VLIW architectures to reduce both active and dynamic energy consumption. A similar proposal offers a method of compilation for VLIW architectures that disables unused functional units to reduce leakage power [56].

System level optimisation of leakage power and energy has also been considered for SoCs. Cao and Yasuura propose a technique for adjusting the data path width to minimise both dynamic and leakage power, claiming leakage power reductions of up to 66% [57]. Liao *et al*. [58] devote a section to leakage power reduction at the system-level for VLIW processors, with a focus on leakage power in the level 2 cache. A comprehensive summary of the issues and potential solutions concerning leakage power in CMOS technologies is presented by Elgharbawy and Bayoumi in [59].

It is clear that a significant volume of research has been carried out in the area of ASIP design, and more specifically power and energy analysis of individual ASIPs. Several of the aforementioned works detail interesting approaches that may be referenced later in this research.

Much of the work detailed in this section does not place a great significance on the power and energy performance of the employed ASIP implementation methods, often simply analysing the power consumption of a single ASIP, or a small number of ASIPs, using conventional

analysis such as gate or transistor-level analysis tools. Although some research offered power estimates for the generated architectures, few opportunities are presented to effectively search the design space with low energy as a key goal alongside existing area and performance targets.

There appears to be no previous work that offers a fully automated energy analysis method for application-specific processors, designed to be integrated within an automated ASIP design tool. Such an approach is highly desirable as it allows for fast design space exploration resulting in a list of candidate architectures and their corresponding area, performance and energy statistics, which can be traded against each other depending on the overall requirements. Achieving this requires both the ASIP hardware itself to be generated, and also the software mapped onto that hardware. This must be done for each iteration, and information harvested about the activity the software is likely to generate, for each candidate ASIP/software combination. Doing such is a complex and unique problem that has yet to be tackled—a problem that forms a key part of this research project.

3. Coprocessor power evaluation tool flow

This chapter details development of a tool flow for analysing coprocessors generated by Cascade, using Synopsys tools for synthesis, simulation and power analysis. Much of the knowledge and understanding of the tools used in this chapter was developed during the analysis of open-source processor cores, described in more detail later in chapter 4.

The information gathered during this chapter of the project serves two main purposes:

- Create a framework for automated yet detailed gate-level power/energy analysis
- Identify the importance of individual components to the overall power/energy picture

The framework created in this chapter will be used throughout the project. Creation of power/energy models, exploration of high-level analysis techniques and testing of optimisations will all require to be validated or compared against a detailed gate-level result; hence the requirement for a fully automated flow.

Similarly, the identification of components worthy of a more detailed examination will allow a higher level of accuracy to be implemented in models representing those components, improving overall modelling accuracy. It would be inefficient to assign the same amount of resource to all components within the coprocessor design space, due to the large number of components and their huge variance in power and energy performance, therefore prioritisation at an early stage of the project is paramount.

3.1 Overview of the power analysis flow

Power Compiler is an automated power analysis and optimisation tool that is integrated with Synopsys' synthesis tool, Design Compiler. Power Compiler can operate at either the RTL or gate level of abstraction, although for accurate results gate level analysis is preferred where possible. Both dynamic and static power are considered; dynamic power requires the annotation of switching activity whereas static power can be analysed using just the power information for ASIC cells provided in the technology library.

An overview of the tool flow using Synopsys tools is shown in Figure 3.1.

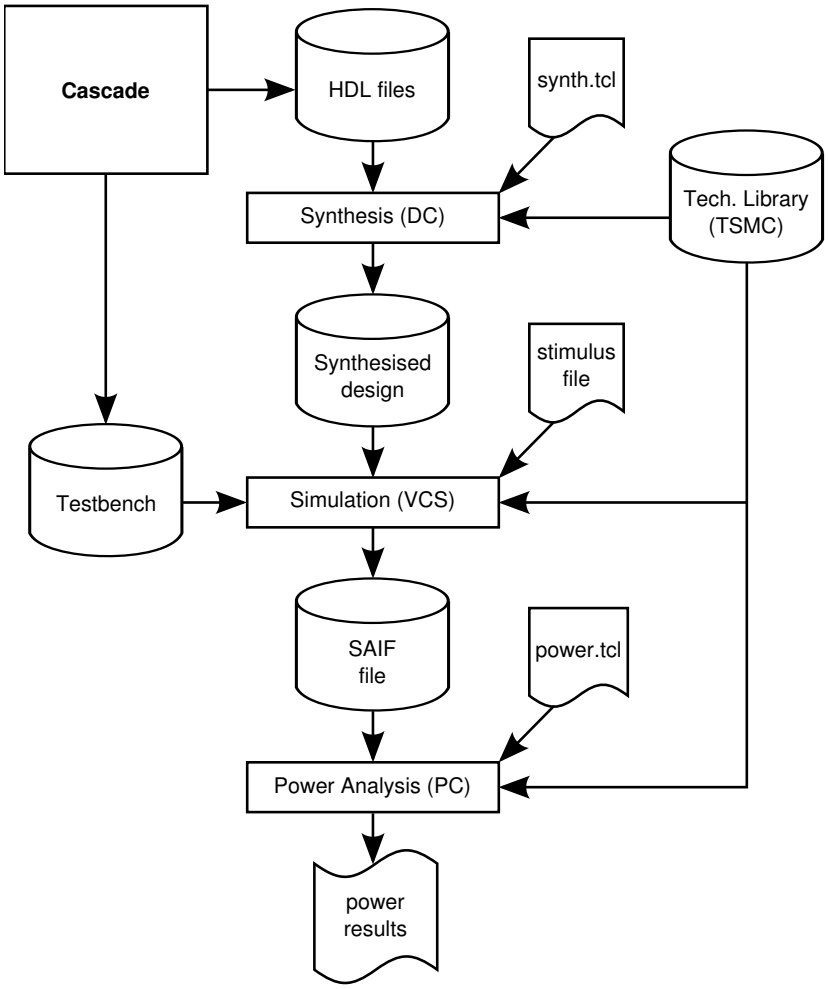


Figure 3.1: Overview of power analysis tool flow

Power Compiler uses data provided by the standard cell technology library vendor, which in the case of this project is provided by Artisan Components (now a subsidiary of ARM) on behalf of the ASIC foundry Taiwan Semiconductor Manufacturing Company (TSMC).

Most of the first half of this project utilises TSMC's 130 nm process technology (this library being called TSMC13). There are also references to 90 nm and 180 nm processes (TSMC90 and TSMC18 respectively), with the 90 nm libraries becoming increasingly prevalent, and the 180 nm libraries being phased out, in the latter half of the project, in line with customer demand. The TSMC13 datasheet [60] states that power is calculated as in Equation 3.1.

$$P_{avg} = \sum_{n=1}^{x} (E_{in} \times f_{in}) + \sum_{n=1}^{y} (C_{on} \times V_{dd}^2 \times \frac{1}{2} f_{on}) + E_{os} \times f_{o1} + P_{static} \tag{3.1}$$

where:

	P_{avg}	=	average power (μW)
	x	=	number of input pins
	E_{in}	=	energy associated with the nth input pin (μW/MHz)
	f_{in}	=	frequency at which the nth input pin changes state (MHz)
	y	=	number of output pins
	C_{on}	=	external capacitive loading on the nth output pin
	V_{dd}	=	operating voltage (1.2V for typical libraries)
	f_{on}	=	frequency at which the nth output pin changes state
	E_{os}	=	energy associated with the sequential cells output pin (μW/MHz)
	P_{static}	=	static power dissipated through leakage currents (μW)

All values in Equation 3.1 are required to be annotated in order that Power Compiler can calculate the average power of the design (or average power of any sub-blocks within the design). Most of the variables can be determined quickly with little effort: E_{in}, E_{os} and P_{static} are available in the technology library, therefore Power Compiler looks up the appropriate values for each cell; x and y can be easily determined by examining the design; C_{on} can be approximated by examining the design connectivity with an appropriate wire-load model; and V_{dd} can either assume the default value provided by the library, as is the typical approach, or it can be explicitly defined during analysis.

The remaining variables reflecting switching frequency, f_{in} and f_{out}, are somewhat more complex to determine accurately as they must be representative of the likely switching activity during real-world use of the design, in order that the power and energy consumption estimations are accurate. Typically these variables are determined by netlist simulation using appropriate input stimulus, with a simulator that can monitor switching activity and gener-

ate either a SAIF file (switching activity interchange format) or a VCD file (value change dump). For calculating either average power or total energy, SAIF is the preferred format as it summarises the switching activity over the entire simulation run for each node. VCD on the other hand annotates every change on each node, leading to a file that can quickly grow very large, particularly on longer simulation runs. The advantage of VCD is that it is possible to calculate power variance (due to switching variance) over time with the appropriate tools. However doing so carries a significant performance penalty in terms of analysis time compared with average power analysis, and is therefore unsuited to this project due to the large data sets typically generated during coprocessor simulation.

3.2 RTL Synthesis (Design Compiler)

In electronic design, synthesis is the process of taking a design from a higher level of abstraction to a lower one. Cascade provides coprocessor synthesis, where an RTL hardware description and associated microcode are generated from a purely software representation of some functionality. In this section, the synthesis described is the process of mapping a technology-independent RTL hardware description (typically using a coprocessor produced by Cascade as input) down to a technology-mapped gate-level netlist. The netlist represents an implementation of the original RTL functionality, but using standard cells such as NAND, NOR and XOR gates, multiplexers and flip-flops that are present in the target ASIC library. Synthesising to a netlist exposes more detail about how the design will actually be implemented in silicon, allowing more accurate power, area and timing estimates to be made compared with an RTL design. However this comes at a cost of much higher complexity, resulting in longer run times when performing analyses.

Synopsys' Design Compiler is the main RTL synthesis tool used throughout this project. Design Compiler was selected as it is currently the market leader in RTL synthesis, being a mature tool with proven results across all types of hardware designs. Critical Blue, Edinburgh University and ISLI all have licences for Design Compiler, ensuring a high availability throughout the project period. Occasionally during the project, Cadence RTL Compiler is used for RTL synthesis—this is used as part of the Cadence Encounter back-end flow, which is detailed in section 11.2.

Design Compiler has two distinct input methods, the original proprietary dcsh mode and a more advanced method based on the industry-standard Tcl language. Tcl mode is recommended because, in addition to its more powerful scripting capabilities, it enables the use of XG mode, which is a newer and more efficient internal data storage method used by Design Compiler, providing capacity and performance improvements. The use of dcsh mode is effectively deprecated at the time of writing, and is likely to eventually be dropped in a future version of Design Compiler. Initially the early project scripts were written in dcsh mode for legacy reasons, but due to the overwhelming advantages of Tcl mode all old scripts were hand converted to the latter, and new scripts written in Tcl. Further information on Synopsys' implementation of Tcl, and help on converting existing legacy scripts from dcsh to Tcl can be found in [61].

There is a graphical interface to Design Compiler, known as Design Vision. This shows a schematic representation of the design after synthesis, in addition to providing menus and shortcuts to shell commands. Except for very small designs, however, the schematic interface is slow and cumbersome, providing little benefit over command-line version of Design Compiler—particularly where all commands to be executed have been scripted. Therefore this project relies entirely on the command-line interface to Design Compiler.

Design Compiler is started with the command:

```
dc_shell -tcl_mode -xg_mode
```

or alternatively abbreviated to,

```
dc_shell-xg-t
```

both of which achieve the same goal of starting Design Compiler in Tcl mode, using the XG internal storage format.

A configuration file must be provided to Design Compiler, specifying details such as the target technology library and the location of technology and synthetic libraries. Many standard Design Compiler Tcl commands can be called automatically from the configuration file; variables and flags can also be set. An example configuration file, with some irrelevant parts removed, is shown below.

```
# dc_shell Tcl setup file
  set designer "Paul Morgan"
  set company  "CriticalBlue Ltd"
# Search path looks in the following directories in order:
# current directory, synthetic library(sldb), technology library(db),
# Artisan compiled memories
  set search_path {  .                                        \
                     /opt/EDA/DesignCompiler/libraries/syn    \
                     /opt/SynthLibs/Synopsys/TSMC_130         \
                     /opt/Artisan/CompiledMemories/TSMC_130   \
                  }

  set hdlin_translate_off_skip_text TRUE

# Define libraries to be used; typical.db is the target technology library,
# sldb are synthetic libraries, the final two are Artisan memory macros.
  set link_library {"*" typical.db                          \
                    dw01.sldb dw02.sldb dw03.sldb dw04.sldb \
                    dw05.sldb dw06.sldb dw07.sldb dw08.sldb \
                    dw_foundation.sldb                      \
                    sp_rw_s_instrmax.db rw_s_bw_4096x32.db
                   }

  set target_library {typical.db}
  set symbol_library {tsmc13.sdb}
  define_design_lib work -path work
  set default_schematic_options "-size infinite"

# Site Specific Variables
  set synthetic_library {dw01.sldb dw02.sldb dw03.sldb dw04.sldb \
                         dw05.sldb dw06.sldb dw07.sldb dw08.sldb \
                         dw_foundation.sldb}

# Define naming style to ensure synthesised entity names are not too long
  set template_naming_style "%s_%p"
  set template_parameter_style "%d"

# Enable command-line editing mode
  set sh_enable_line_editing "true"
```

The commands `set template_naming_style` and `set template_parameter_style` are of particular interest; they define how synthesised modules are declared by Design Compiler in relation to the original RTL module from which the synthesised module is derived. Often an RTL module will expand to multiple distinct modules during synthesis (for example, using the VHDL `generate` statement) so for this reason synthesised modules are named differently from their RTL equivalents as part of the uniquifying process. The default approach used by Design Compiler is to add the names of generics or parameters, along with their cor-

responding values, to the end of the module name. Usually parameters are different between each instantiation of a module, therefore each synthesised module will have a unique name using this approach.

During synthesis, and with some simulators, this approach works well. However, building a simulation using VCS highlights a problem with this approach due to filename lengths. VCS builds each module into a separate file, the file inheriting its name from the corresponding module. Typical Linux/UNIX file systems, such as ext2, ext3, ReiserFS and UFS, have a filename limit of 255 characters, while modules that have been synthesised with a lot of parameters can have names of over 1000 characters in length. This obviously breaches the file system limits, meaning that a file cannot be created for any module with a name longer than 251 characters (allowing for the four filename extension characters); the simulation build subsequently fails.

Editing the netlist by hand to rename problematic modules is both time consuming and error prone. Using the aforementioned commands modifies Design Compiler's naming convention for synthesised modules, solving the name length problem and avoiding the need to edit the netlist. The "template_naming_style" variable defines that synthesised modules should be labelled with the source design name (%s) followed by the parameter list (%p). The "template_parameter_style" variable defines how parameters are declared in the parameter list; %d means that only the value of the parameter should be used [62]. This option produces much shorter module names, particularly for modules with many parameters, compared to the default parameter template of %s%d which includes both the parameter name and value in the instantiated module name.

The only disadvantage of this modification is that it can be more difficult to determine the parameters used to instantiate a module when examining a netlist, which may be desirable when investigating an unexpected response or error during gate-level simulation. This can be easily overcome by consulting the synthesis log, which details the parameter values used for each module instantiated from the RTL.

Design Compiler is launched with a synthesis script similar to that shown below. On successful completion of the script, Design Compiler outputs a Verilog gate-level netlist representing a synthesised equivalent of the original RTL coprocessor design. The netlist is then carried forward to be used in gate-level simulation and, ultimately, power analysis.

```
# dc_shell Tcl script file for synthesis. Written by Paul Morgan, 2004-2005

  remove_design -all
  if {![file isdirectory work]} {file mkdir work}

# Configure paths containing source files to be analyzed
  set mem_lib_path       "."
  set hdl_com_path       "./Common"
  set artisan_path       "./DesignWare_Artisan"

# Analyze all files within the target directories
  foreach dirlist [list $mem_lib_path $hdl_com_path $artisan_path] {
     foreach hdlfile [glob -nocomplain -directory $dirlist -- *.v] {
   analyze -format verilog -library work $hdlfile}
  }

# Preserve RTL hierarchy names for SAIF file annotation
  set power_preserve_rtl_hier_names "TRUE"

  elaborate test_copro -lib WORK
  link
  current_design test_copro

# Set wire load model and operating conditions
  set_wire_load_model -name "tsmc13_wl10" -library "typical"
  set_operating_conditions -library "typical" "typical"
  create_clock -period 10 clk_i
  set_input_delay 0 -clock clk_i [all_inputs]
  set_output_delay 0 -clock clk_i [all_outputs]
  set_drive 0 { clk_i }
  set_dont_use {typical/CLK*}

  uniquify
  compile

# Generate reports then check design and timing
  if {![file isdirectory $report_dir]} {file mkdir $report_dir}
  report_area              > $report_dir/area.txt
  check_design             > $report_dir/design_check.txt
  report_timing -path full -delay max -max_paths 3 -nworst 1 \
     > $report_dir/timing_check.txt

# Change names to be compatible with Verilog netlist
  change_names -rule verilog -hierarchy

# Write out the Verilog gate-level netlist
  if {![file isdirectory synth]} {file mkdir synth}
  write -format verilog -hierarchy -output ./synth/$current_test\.v

  quit
```

3.3 Netlist Simulation (VCS)

Obtaining accurate switching activity information for power and energy analysis requires that the netlist be simulated while monitoring for switching activity. Synopsys VCS is used; it is a compiled simulator therefore significantly faster than interpreted simulators, supports both VHDL and Verilog languages including mixed-mode, and is capable of monitoring and annotating switching activity into a SAIF file—an important attribute for larger designs or longer simulations as explained previously in this chapter.

The accuracy of switching activity information obtained through simulation depends on how closely the input stimulus to the simulation realistically reflects typical real-world operation of the device. In addition to the content of the stimulus, accurate simulation also requires a sufficient number of stimuli to allow any temporal fluctuations in switching to average out toward a representative value. The complexity of such an input pattern necessitates the use of an automated approach; in the case of this project the automatic testbench and test vector stimulus generation capabilities of Cascade are utilised.

During the coprocessor and microcode generation phase of the Cascade flow, RTL and SystemC testbenches are generated to verify the functionality of the coprocessor RTL. Also generated is a text file, `SimInput.txt`, containing a hexadecimal representation of the inputs into the coprocessor derived from the generated microcode. The testbenches are designed to read and decode this input file, and apply the stimulus to the coprocessor's input ports—mimicking the behaviour of a hardware coprocessor executing microcode—while simultaneously monitoring the output ports checking for any deviation from expected output. Although this approach is primarily designed as a verification engine, the fact that it generates a complete simulation, including the entire data set, of the executable software offloaded to the coprocessor, makes it an ideal mechanism for harvesting representative switching activity information.

Before a simulation can be run using VCS, it has to be built into a compiled executable to run natively on the target platform. With VHDL designs, the analysis of source code is carried out in a separate stage from elaboration into the executable. Verilog designs are analysed and elaborated in a single step, and the simulation can immediately commence after the build process has completed.

The small example script below demonstrates a typical method of performing Verilog netlist simulation using VCS. After ensuring that the simulation directory is configured correctly, *vcsi* is called with the necessary options to build the design.

```
# Delete any data files left over from previous simulation
  rm -rf simv simv.daidir csrc

# Ensure work directory exists and is writeable
  if ! [ -d work ] || ! [ -w work ]; then
     mkdir work
  fi

 # Build and run simulation
 vcsi -R +v2k +cli+1 ../Testbench/Verilog_Testbench/copro_testbench.v \
   -v synth/$current_test\.v                                          \
   -v /opt/SynthLibs/Synopsys/TSMC_130/tsmc13.v                       \
   -y "/opt/Artisan/CompiledMemories/TSMC_130/MemoryModels/*.v"       \
      >> $report_dir/sim.txt
```

A summary of the options used with *vcsi* is given in Table 3.1.

+v2k	enables Verilog-2001 mode
+cli+1	provides additional detail for debugging
-v	provides a Verilog file containing instantiated modules
-y	provides a Verilog library containing instantiated modules
-R	informs VCS to start the simulation after build completes

Table 3.1: VCSi command options

Generating a SAIF file using a Verilog simulation requires insertion of PLI commands into the testbench, instructing the simulator to monitor the desired nodes, and also controlling the times during which toggles will be monitored. The Verilog code below shows an initial block that will be inserted into the top-level testbench. It enables gate-level monitoring, allowing for the highest level of detail, then sets the toggle region to the top level of the design, as instantiated from the testbench. The instruction to start toggling during this initial block will ensure that switching will be monitored from the start of the simulation run.

```
initial
begin
   $set_gate_level_monitoring("on");
   $set_toggle_region(copro_testbench.copro);
   $toggle_start();
   $display("Starting toggle.");
end
```

It is necessary to have a corresponding set of commands to be called before simulation completes, to signal to the simulator that switching activity monitoring should stop and the SAIF file written to disk. If it is desired to monitor the entire simulation run, a method of enabling the testbench to detect when the simulation has completed should be implemented, triggering the SAIF generation commands before exiting the simulator. In the case of coprocessors generated by Cascade, the testbench detects such a signal from the simulated coprocessor indicating the test case has completed; alternatively a time-based mechanism can be used if the simulation has a known run-time. The commands used to generate the SAIF file are shown below. Note that during the `$toggle_report` command, the `1e-9` option indicates SAIF file time resolution—in this example 1×10^{-9} s or 1 nanosecond.

```
$toggle_stop();
$display("Stopping toggle, generating SAIF file.");
$toggle_report("backward.saif",1e-9,"copro_testbench");
//finish simulation after SAIF file has been written
$finish;
```

The process of inserting both sets of SAIF file generation commands has been automated as part of the top-level power and energy analysis script, listed in Appendix A.1.

Successful completion of the simulation results in the creation of two files: `SimResults.txt` containing the values of coprocessor outputs for the purpose of hardware verification, and `backward.saif` containing switching activity information for all nodes within the coprocessor. An abbreviated example of such a SAIF file is shown in Figure 3.2. This example shows switching activity for four nets; in reality even a relatively small coprocessor will have tens of thousands of net instance entries in the SAIF file.

This type of SAIF file is known as a backward SAIF as it back-annotates from simulation to the gate-level design. A forward SAIF file can be generated for an RTL design using Design Compiler, the purpose being to indicate to the simulator which nodes are synthesis-invariant. Use of a forward SAIF file is not required (nor does it offer any advantage) for complete monitoring of a gate-level simulation, therefore this flow does not use forward SAIF files.

Each monitored instance in the hierarchy is listed in the SAIF file, taken from the perspective of the top-level during simulation (i.e. the testbench). Therefore in the example shown above, all nodes within *copro_testbench/copro* appear beneath the `INSTANCE copro` entry. A summary of the labels used for each individual entry in the SAIF file is given in Table 3.2.

```
/** The set_gate_level_monitoring command explicitly turns **/
/** ON the internal nets monitoring **/
(SAIFILE
(SAIFVERSION "2.0")
(DIRECTION "backward")
(DATE "Mon Jul  3 09:08:34 2006")
(VENDOR "Synopsys, Inc")
(PROGRAM_NAME "VCS-Scirocco-MX Power Compiler")
(TIMESCALE 1 ns)
(DURATION 1244300300.00)
(INSTANCE copro_testbench
   (INSTANCE copro
      (NET
         (m_hresp_i\[1\]
            (T0 1231744099) (T1 12548600) (TX 7601)
            (TC 125486) (IG 0)
         )
         (m_hresp_i\[0\]
            (T0 1237993699) (T1 6299000) (TX 7601)
            (TC 62990) (IG 0)
         )
         (m_hrdata_i\[31\]
            (T0 121749400) (T1 1122550899) (TX 1)
            (TC 865634) (IG 0)
         )
         (m_hrdata_i\[30\]
            (T0 124317387) (T1 1119982912) (TX 1)
            (TC 963584) (IG 0)
         )
      )
   )
)
```

Figure 3.2: Example SAIF file output (all times ns)

T0	total time node has value 0
T1	total time node has value 1
TX	total time node has value X (unknown or don't care)
TC	toggle count (number of toggles over simulation run)
IG	instances of glitching (requires event driven simulation)

Table 3.2: Key to SAIF file entries

3.4 Power analysis (Power Compiler)

Power Compiler is an analysis tool that is integrated with Design Compiler, therefore the initial setup and configuration of Power Compiler is very similar to that described in section 3.2. The technology libraries used for power analysis are the same ones used for RTL synthesis, so no change to the configuration is required. Due to licensing issues (specifically, Power Compiler being licensed to run on a different machine from the machine on which VCS runs), it is necessary to transfer netlist and SAIF files between machines at this stage. Archiving, compression and subsequent decompression is performed automatically by the analysis scripts, requiring intervention only for performing the transfer over SFTP/SCP—this cannot be fully automated for security reasons concerning storage of login credentials. Public-key cryptography could be used along with a key agent to cache the private key unencrypted on the client machine, allowing password-free login to the remote machine. However automating such a process poses a potential security risk to the remote network should the client machine be compromised, and as such may be in contravention of university computing regulations.

Design Compiler is started as normal, and the previously synthesised gate-level netlist is read. The design can then be annotated with switching activity from the SAIF file generated during simulation, using the command:

```
read_saif -input backward.saif -instance copro_testbench/copro
```

To ensure successful annotation, the environmental variable `find_ignore_case` should be set to `TRUE` within Design Compiler. VCS tends to change the case of instance names in the SAIF file, meaning that they no longer match the corresponding case-sensitive names in the design loaded into Design Compiler. Setting the variable to ignore case differences solves this issue.

Once the SAIF file has completely loaded, the command

```
propagate_switching_activity -effort high
```

will initiate an internal zero-delay simulation that calculates appropriate switching values for any non-annotated nodes. This approach only works for nodes where the value has a direct relationship to visible input values; primary inputs, black-box outputs and glitching

information cannot be determined using the zero delay simulation. If netlist simulation was performed correctly with the appropriate SAIF generation commands, the majority of nodes should be annotated. This can be checked using the command

```
report_saif -hier
```

that will produce a report similar to that shown in Figure 3.3. The results of this report should be taken into account when considering the accuracy of power analysis results. *User Annotated* objects (derived from the SAIF file) tend to be the most accurate, subject to the quality of simulation; *Propagated Activity* provides similar accuracy except for glitching (although it is dependent on the accuracy of User Annotated values); *Default Activity* is used where neither User Annotated information is provided, nor can Propagated Activity be calculated, resulting in switching information that seldom reflects actual behaviour of the object in question. Therefore it is important to ensure that the number of nodes assigned Default Activity values remain low by annotating the design as completely as is practical.

Nodes that are not annotated and cannot be calculated using zero-delay simulation, such as black-box outputs, can have switching activity statistics manually entered into Power Compiler. Obtaining and entering switching activity information is a very time consuming process making it suitable for only a small number of nodes.

```
****************************************
Report : saif
         -hier
Design : test_copro
Version: W-2004.12
Date   : Wed May 10 12:36:32 2006
****************************************

--------------------------------------------------------------------
             User            Default        Propagated
Object type  Annotated (%)   Activity (%)   Activity (%)   Total
--------------------------------------------------------------------

 Nets        26320(74.10%)   2960(8.33%)    6240(17.57%)   35520
 Ports       97(100.00%)     0(0.00%)       0(0.00%)       97
 Pins        96154(69.46%)   19588(14.15%)  22693(16.39%)  138435
--------------------------------------------------------------------
```

Figure 3.3: Switching activity annotation report

When the quality of switching activity annotation has been deemed satisfactory, the power analysis functionality of Power Compiler can be invoked. This is achieved simply using the command `report_power`, although there are several options that should be considered to improve the usefulness of information gleaned from the analysis. During this project, the most commonly used options are those below:

```
report_power -nosplit                              >  reports/power.txt
report_power -nosplit -hierarchy -hier_level 2     >> reports/power.txt
report_power -nosplit -cell -nworst 20             >> reports/power.txt
report_power -nosplit -hierarchy -hier_level 1     >  reports/power_h1.txt
```

On its own, the `report_power` command summarises the average power used over the time duration specified in the SAIF file, calculating both dynamic and static (leakage) power for cells and nets. The addition of the `-hierarchy -hier_level 2` options provide a detailed breakdown of all elements within the design hierarchy to the depth specified. Similarly, the use of the option `-hier_level 1` produces a report of only the top level units; this report is directed to a different file to be used for top-level energy calculations. Finally, the `-cell -nworst 20` options list the 20 cells with the highest power consumption—useful for highlighting which cells should receive most effort during optimisation. All reports have the `-nosplit` option to ensure each entry occupies only one line regardless of length, enabling the results to be accurately parsed by an automatic processing algorithm at a later stage if desired. An example of the standard summary power report appended with a report of the 20 worst cells is shown in Figure 3.4.

Examining the worst cells section of Figure 3.4 shows that the first four entries in the list (`CBNative_Slave_Generic`, `fu_mult64_0`, `fu_Cache0`, `fu_registerfile_0`) consume 65% of the total dynamic energy consumed by the coprocessor. As an example of the importance of prioritising more significant units, a 10% reduction in the power of the aforementioned four units would have a greater effect than the complete elimination of power consumption in the bottom four units in the worst cells list.

The total energy for each top-level unit over the simulation run is calculated using the values provided in the level 1 hierarchical report, along with the run duration extracted from the SAIF file. The calculation is automated as part of the shell script shown in Appendix A.2, and an example output is shown in Figure 3.5.

```
****************************************
Report : power
Design : test_copro
Version: W-2004.12
Date   : Wed May 10 12:36:48 2006
****************************************
Design        Wire Load Model            Library
------------------------------------------------
test_copro             tsmc13_wl10       typical

Global Operating Voltage = 1.2
Power-specific unit information :
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW

  Cell Internal Power  =   3.4719 mW   (87%)
  Net Switching Power  = 541.3599 uW   (13%)
                         ---------
Total Dynamic Power    =   4.0133 mW  (100%)

Cell Leakage Power     = 692.3362 uW

                        Cell      Driven Net  Tot Dynamic      Cell
                        Internal  Switching   Power            Leakage
Cell                    Power     Power       (% Cell/Tot)     Power
------------------------------------------------------------------------------
CBNative_Slave_Generic     1.0810       N/A        N/A (N/A)   95576832.0000
fu_mult64_0                0.8022       N/A        N/A (N/A)   35984552.0000
fu_Cache0                  0.4340       N/A        N/A (N/A)   512340032.0000
fu_registerfile_0          0.3516       N/A        N/A (N/A)   16859114.0000
fu_arithmetic_Z            0.1864       N/A        N/A (N/A)   5102002.5000
fu_arithmetic_Y            0.1111       N/A        N/A (N/A)   4386454.5000
fu_bitshift_0              0.1041       N/A        N/A (N/A)   4191139.2500
fu_copy_0                  0.0928       N/A        N/A (N/A)   3619795.5000
fu_immediate32_0           0.0787       N/A        N/A (N/A)   2359040.7500
fu_immediate8_0            0.0641       N/A        N/A (N/A)   2040246.3750
fu_select_0                0.0490       N/A        N/A (N/A)   1822646.5000
fu_logical_0               0.0322       N/A        N/A (N/A)   1523074.6250
fu_select_1                0.0284       N/A        N/A (N/A)   1454848.6250
fu_addrlink_0              0.0148       N/A        N/A (N/A)   847660.0625
fu_sat_arithmetic_0        0.0132       N/A        N/A (N/A)   2258325.5000
fu_squash_0              9.624e-03      N/A        N/A (N/A)   743620.8750
fu_predicate_0           7.638e-03      N/A        N/A (N/A)   358125.5938
fu_branch_0              5.351e-03      N/A        N/A (N/A)   563452.0000
fu_combine_0             5.347e-03      N/A        N/A (N/A)   278494.0625
U18                      1.529e-05 1.837e-04   1.99e-04 (8%)    1748.9520
------------------------------------------------------------------------------
Totals (20 cells)          3.472mW     N/A         N/A (N/A)    692.311uW
```

Figure 3.4: Power summary and worst cells report

```
test_copro                                133741.57270 nJ
fu_squash_0                                 286.93337 nJ
fu_select_0                                2112.30595 nJ
fu_sat_arithmetic_0                         491.35922 nJ
fu_registerfile_0                         11259.01601 nJ
fu_predicate_0                              419.34235 nJ
fu_multiplier64_0                         24882.68056 nJ
fu_logical_0                               2665.19140 nJ
fu_immediate8_0                            3008.26391 nJ
fu_immediate32_0                           2559.71787 nJ
fu_coreregfile_0                          30173.36902 nJ
fu_combine_0                                130.99132 nJ
fu_branch_0                                 168.70094 nJ
fu_bitshift_1                               995.47734 nJ
fu_bitshift_0                              2458.21377 nJ
fu_arithmetic_1                            2480.89625 nJ
fu_arithmetic_0                            5806.71488 nJ
fu_addrlink_0                               431.25065 nJ
fu_access_st_1r_0                         13306.10983 nJ
AMBA_AHB_Slave_Generic                    29750.90783 nJ
```

Figure 3.5: Top-level cells energy report

Enquiring further into `CBNative_Slave_Generic` reveals that the instruction cache within the hierarchy of that cell is responsible for the largest part of its power consumption. Taking that into consideration, of the four most power-hungry components, three are memories and the fourth is a complex pipelined multiplication unit. It is perhaps unsurprising that memories and multipliers dominate the power consumption, such is a common occurrence in SoC processors.

Similar results to those above have been observed with a number of different coprocessor configurations running a range of applications. An occasional exception to this is observed in a small number of applications, particularly those that have been targeted at lower end systems, that do not utilise the multiplier unit even when it is present, resulting in a lower power and energy figure for the multiplier under that particular application. In most other scenarios multipliers and memories are they key consumers within the coprocessor boundaries, highlighting the need to focus on those units, both for accurate analysis and during the optimisation phase.

3.5 Summary

At the start of this chapter, two main goals were set out; these are reiterated below:

- Create a framework for automated yet detailed gate-level power/energy analysis
- Identify the importance of individual components to the overall power/energy picture

The framework for an automated power/energy analysis has been successfully created, enabling further development to improve the coprocessor energy models during forthcoming work as part of this project.

In addition, the relative importance of individual components within a coprocessor has been identified, allowing a more detailed examination of the more significant components to be undertaken at a later stage of the project, with a view to obtaining a higher level of accuracy for the models representing those components.

The analysis framework developed in this chapter could easily be extended to cover other types of configurable processors in addition to coprocessors generated by Cascade, offering a useful generic flow for analysing such processors within system-on-chip platforms.

4. Evaluation of open-source processor cores

Comparing the power and energy performance of coprocessors generated by Cascade against commercial processors is a challenging task due to the lack of soft IP cores available from commercial vendors that could be analysed with target application code. Some vendors will supply such soft IP, but at a high cost and usually with tight restrictions on how it can be used, making such an approach infeasible for this project.

On the other hand, several processor cores are freely available to the public, with full visibility of the design in its native hardware description language, along with associated scripts. Often additional resources are also provided, such as testbenches, sample test applications and software build environments.

In this chapter, a selection of open-source cores will be considered for power and energy analysis. This serves three goals:

- Provide a loose comparison platform for Cascade-generated coprocessors
- Allow for further familiarisation with the synthesis and power analysis tools
- Determine the variance of power consumption between process technology vendors

Much of the tool flow detailed in chapter 3 was developed during the analysis of the open source cores as detailed in this chapter, and adapted as appropriate for use in analysing coprocessors generated by Cascade.

4.1 TestCore processor

TestCore is an open-source processor core written in synthesisable Verilog. The real name of this core has been changed for reasons of commercial sensitivity, which came to light after the work in this section had been completed. The core is considered to be of experimental and educational value rather than a realistic alternative to other established open-source or commercial cores.

Due to the issues mentioned above, some parts of this section have been pruned to remove details that may identify the core being examined. Although this results in some areas of the report being quite concise, care has been taken to ensure that no important information has been left out. A more detailed report is provided for the other two processor cores, in the latter sections of this chapter, which have no such restrictions.

4.1.1 RTL synthesis

The processor is synthesised using Design Compiler in a similar manner to that described in section 3.2. Some modifications are required to the Verilog code to resolve errors in the synthesis process. In particular, several modules contain some duplicate wire definitions that must be removed. The synthesis script requires that only the top level file be analysed directly; all other required modules are referenced from the top level source by the use of Verilog `` `include `` directives.

4.1.2 Code compilation

Code can be compiled for TestCore using standard supplied compilation tools, both commercial and open-source, although it may be necessary to modify the resulting assembly code to ensure compatibility with the limited instruction set support of TestCore. The resulting assembly code can be built into an executable binary using freely-available tools.

4.1.3 Simulation and power analysis

Compiled binaries can be executed on the simulated hardware by first converting the binary to a hexadecimal representation stored within an ASCII text file that can be read by the Verilog testbench. This conversion is automatically performed using a freely-available tool written by the author of TestCore. The instruction memory within the processor can then be loaded from the resulting ASCII text file named `asc` by the command shown below:

```
$readmemh("asc",inst_TestCore.inst_MemoryController_WB_Bhv.Memory);
```

In this manner, both RTL and synthesised netlist versions of the TestCore can be simulated while running the desired applications for analysis. Toggle commands can be added to the testbench, in the same manner as that shown in section 3.3, to monitor switching activity for power analysis.

Initial power analysis was undertaken using several of the provided assembly code test applications, such as StrCmp. Results for StrCmp running at 100 MHz on a 0.13 μm process are shown in Figure 4.1.

```
****************************************
Report : power
        -analysis_effort high
Design : TestCore
****************************************


Operating Conditions: TYPICAL
Wire Load Model Mode: top

Global Operating Voltage = 1.2
Power-specific unit information :
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1uW


  Cell Internal Power  =   2.9059 mW   (98%)
  Net Switching Power  =  49.4004 uW    (2%)
                         ---------
Total Dynamic Power    =   2.9553 mW  (100%)

Cell Leakage Power     = 152.5208 uW
```

Figure 4.1: TestCore power summary report for 130 nm process

Although the assembly code nature of the test applications provided with TestCore make running the same code on a Cascade coprocessor more complex, this particular example is simple enough to be re-implemented in C. The key string compare function is then offloaded to a coprocessor. A quick analysis indicates that the coprocessor consumes an average of around 0.5 mW of dynamic power when executing this test at 100 MHz. In addition to the lower power of the coprocessor, it can also complete the same length of input stimulus in far fewer cycles.

It is clear from this information that TestCore is poorly optimised from an energy efficiency standpoint, which is not unexpected given its experimental status. Therefore it is not considered worthwhile to undertake a detailed comparison of the power and energy consumption of this core with that of a Cascade coprocessor. Instead, the remainder of this section concentrates on undertaking a process technology comparison using TestCore as a basis.

4.1.4 Comparison of process technologies

TestCore is re-analysed using a different set of 130 nm process technology libraries from an alternative vendor, to determine how much of a difference the choice of process technology vendor makes to power consumption. Due to licensing restrictions, it is not permissible to publish named comparisons between the vendors used, therefore they are simply referred to "Vendor A" and "Vendor B" for the remainder of this section. The previous analysis steps of synthesis, simulation (including generation of switching activity information) and power analysis are carried out with Vendor A library references (the results of which were listed in Figure 4.1) replaced by Vendor B libraries. The results of this power analysis are shown in Figure 4.2.

Comparison of Figures 4.1 and 4.2 clearly show that the power figures generated by Power Compiler are much higher when Vendor B is the target technology, compared with those for Vendor A. Both cases target a 130 nm technology, and both cases use the "typical" library and operating conditions for the analysis. In order to examine this further, the data sheets for each technology reveal typical area, timing and power consumption figures for each of the components present in the standard cell libraries. For example, Table 4.1 lists the parameters of a 2-input NAND cell with drive strength and fan-out of one, for Vendor A's 130 nm technology.

```
****************************************
Report : power
        -analysis_effort high
Design : TestCore
****************************************


Operating Conditions: typical   Library: typical
Wire Load Model Mode: top


Global Operating Voltage = 1.2
Power-specific unit information :
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  =   5.8260 mW   (85%)
  Net Switching Power  = 989.6710 uW   (15%)
                         ---------
Total Dynamic Power    =   6.8157 mW  (100%)

Cell Leakage Power     =  37.8025 uW
```

Figure 4.2: TestCore power summary report for Vendor B 130 nm process

The corresponding Vendor B 130 nm data sheet does not list directly comparable figures to those provided by the Vendor A data sheet. Rather they have to be calculated from intrinsic delay and load values. The following four equations list the calculations and results for propagation delay.

$$\begin{aligned} \text{PinA} \uparrow \quad t_{typical} &= t_{intrinsic} + K_{load} \times C_{load} \\ &= 0.0132 + (3.579 \times 0.003) \\ &= 0.0239 \text{ ns} \end{aligned} \tag{4.1}$$

Transition	**Propagation delay (ns)**	**Energy dissipation (nJ)**
A1 rise	0.051	0.007
A1 fall	0.023	0.002
A2 rise	0.061	0.009
A2 fall	0.027	0.002

Table 4.1: Vendor A 130 nm NAND2X1 cell parameters

$$\begin{aligned} \text{PinA} \downarrow \quad t_{typical} &= t_{intrinsic} + K_{load} \times C_{load} \\ &= 0.0167 + (5.0022 \times 0.003) \\ &= 0.0317 \text{ ns} \end{aligned} \tag{4.2}$$

$$\begin{aligned} \text{PinB} \uparrow \quad t_{typical} &= t_{intrinsic} + K_{load} \times C_{load} \\ &= 0.0145 + (3.5807 \times 0.003) \\ &= 0.0252 \text{ ns} \end{aligned} \tag{4.3}$$

$$\begin{aligned} \text{PinB} \downarrow \quad t_{typical} &= t_{intrinsic} + K_{load} \times C_{load} \\ &= 0.0177 + (4.9995 \times 0.003) \\ &= 0.0327 \text{ ns} \end{aligned} \tag{4.4}$$

Comparing the results of Equations 4.1, 4.2, 4.3 and 4.4 with the values listed in Table 4.1 for the propagation delays in Vendor A cells, it is notable that the Vendor B cell shows lower propagation delay for rising transitions, but conversely Vendor A shows lower delay for falling transitions. Assuming a roughly equal number of rising and falling transitions, the Vendor B cell would have a lower average propagation delay.

Moving on to a comparison of power/energy consumption between each vendor's standard cells, the values for input transitions on the Vendor A cells can again be referenced from Table 4.1. The Vendor B data sheet lists only one value for each pin, rather than a separate value for rising and falling transitions. For pin A the energy per transition is listed as 0.0020 nJ and for pin B it is 0.0024 nJ. Assuming that these values represent the average of energy consumed for both rising and falling transition, then both pins show a lower per-transition energy cost than the Vendor A NAND2 cell. If the energy performance of the NAND2 cell is representative of the entire standard-cell library for each process technology, this result appears to be contrary to the results observed for the TestCore processor using each vendor's technology files, as reported in Figures 4.2 and 4.2.

However, a closer examination of the operating conditions for which the results are calculated reveals some significant differences between the two process technologies. Vendor B's energy values are calculated with zero loading on the outputs, and with an input slew of

0.018 ns. On the other hand, Vendor A's values represent an output load of 0.003 pF and an input slew of 0.08 ns. Thus, it is expected that the values in the data sheet for Vendor A will be significantly worse than those listed for Vendor B, even though the results of the full TestCore processor analysis returned by Power Compiler indicate significantly less energy being consumed when Vendor A's technology libraries are used.

Due to the limited level of detail provided in the data sheets for both process technologies, it is not possible to examine further the reasons for the difference in power and energy performance, without careful analysis of the synthesised netlist to determine the cell sizing used for each technology. The high level of complexity and sheer size of a netlist representing even a simple processor like TestCore makes such a task very time consuming and error prone. It is not a necessity of the overall project to determine the reasons for the power differences between each vendor's technology libraries, since all coprocessors generated by Cascade will be targeted to TSMC libraries for commercial reasons, and therefore any analysis should be made using such libraries to ensure consistency. For this reason, no further analysis using non-TSMC libraries will be undertaken as part of the project beyond this chapter, and as such the vendor power comparison examination concludes here.

It was originally intended to undertake more detailed examinations and comparisons of the power and energy performance of the TestCore processor. However several issues came to light during the early stages of the analysis. First, the lack of complete compatibility with available compilation tools means that many target applications compiled using these tools will not run on TestCore without performing assembly-level modifications to the code. This can be a time-consuming process depending on the complexity of the application being built. Second, it quickly became apparent that TestCore is not a particularly well implemented core in terms of power and energy efficiency – it is far less optimised than comparable open source cores, therefore the results are unlikely to be particularly relevant. Finally, and most importantly, commercial sensitivity requires that the identity and details of "TestCore" be obscured, preventing any details of the real name or implementation details of the core from being revealed. It was therefore decided to discontinue work involving TestCore. No further updates regarding TestCore have appeared since the work in this section was undertaken, therefore it is assumed that work on the project has been abandoned.

4.2 LEON2 processor

LEON2 is a synthesisable processor originally developed as a fault-tolerant processor for the European Space Agency by Gaisler Research [63]. The non fault-tolerant version is licensed under the GNU Lesser General Public Library (LGPL) licence, which makes it freely available for both commercial and non-commercial purposes. LEON2 supports the AMBA AHB bus [64], and implements the SPARC V8 instruction set [65], which has been ratified as IEEE standard 1754. The LEON2 architecture is shown in Figure 4.3. Throughout this section, the *LEON2-1.0.32-xst* version of the processor and accompanying support files are used.

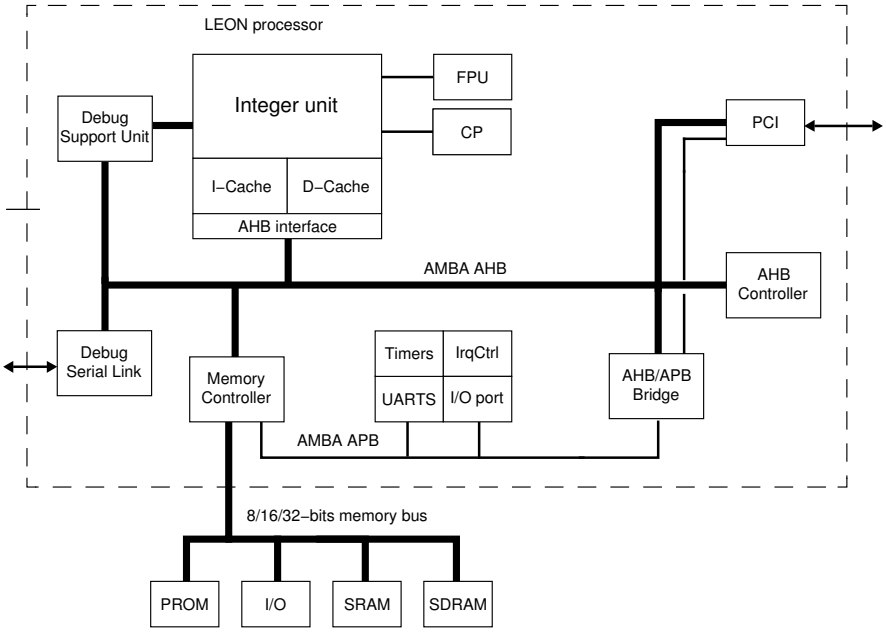


Figure 4.3: LEON2 processor architecture [66]

4.2.1 Configuring and simulating LEON2 using ModelSim

Included with the LEON2 sources is a graphical configuration application written in Tcl/Tk. It is invoked by running `make xconfig` from the root LEON2 directory. From within this tool, various processor parameters can be configured such as the target process technology, memory configuration, the AMBA AHB bus configuration, debug options and boot options. The preferred target technologies for this project, TSMC 0.18 µm and 0.13 µm, are not supported in this version; however TSMC 0.25 µm can be substituted as the target technology,

and the required modifications performed later. The procedure for targeting a new process technology is detailed in section 13 of *The LEON-2 Processor User's Manual* [66].

Once LEON2 has been configured, a simulation model can be built. Originally it was intended that all simulation would be done using Synopsys VCS, as detailed in section 3.3. However, attempting to build the simulation using the included scripts results in errors; it appears that LEON2 targets an older version of VCS than the one used in this project, indicated by use of the `-interp` flag, which is no longer supported by recent versions. Attempts to work around the problems were unsuccessful, therefore a decision was made to use Mentor Graphics' *ModelSim SE v6.1e* instead.

The build process for ModelSim can be invoked by calling the included Makefile using the command `make vsim`. Once this completes, the included testbench can be used to check the configuration and build process has resulted in a compliant LEON2 processor—the testbench is initialised by running `make test`. If the short test completes successfully, the output from the simulator should be similar to that shown below.

```
# run -all
# LEON-2 generic testbench (leon2-1.0.31-xst)
# Bug reports to Jiri Gaisler, jiri@gaisler.com
#
# Testbench configuration:
# 32 kbyte 32-bit rom, 0-ws
# 2x128 kbyte 32-bit ram, 2x64 Mbyte SDRAM
#
# *** Starting LEON system test ***
# Register file
# Multiplier (SMUL/UMUL/MULSCC)
# Divider (SDIV/UDIV)
# Watchpoint registers
# Cache controllers
# Interrupt controller
# UARTs
# Timers, watchdog and power-down
# Parallel I/O port
# Test completed OK, halting with failure
# ** Failure: TEST COMPLETED OK, ending with FAILURE
#    Time: 375082 ns  Iteration: 0  Process: /tbleon/tb/testmod0/rep
#    File: /home/pmorgan/leon2-1.0.32-xst/tbench/testmod.vhd
# Break at /home/pmorgan/leon2-1.0.32-xst/tbench/testmod.vhd line 118
# Stopped at /home/pmorgan/leon2-1.0.32-xst/tbench/testmod.vhd line 118
```

In addition to the automated test mode described above, ModelSim can be used in graphical mode by starting it with the command `vsim -gui`. Once the tool has loaded, standard text commands can be entered into the GUI window, such as `vsim -c tbfunc_32` to initialise the LEON2 testbench simulation, which is then started with `run -all`. If desired, signals within the processor can be added to the waveform window before starting the test, allowing the behaviour of those signals to be monitored—this is done with the `add wave` command.

Frequently used commands can be automated in a ModelSim "do" file, which is a script containing ModelSim commands. This approach is often used to add a list of wave signals to be monitored before each simulation run. The example below shows a small excerpt from the `wave.do` file included with LEON2:

```
add wave -format Logic /tbleon/tb/p0/leon0/resetn
add wave -format Logic /tbleon/tb/p0/leon0/clk
add wave -format Logic /tbleon/tb/p0/leon0/errorn
add wave -format Literal -radix hexadecimal /tbleon/tb/p0/leon0/address
add wave -format Literal -radix hexadecimal /tbleon/tb/p0/leon0/data
```

4.2.2 The SOCks project and simulation using NC-Sim

As an alternative to the standard build and simulation environment provided by Gaisler Research, the SOCks project developed by Johannes Grad at the Illinois Institute of Technology [67] provides a complete design flow for a system on chip incorporating the LEON2 processor. It combines the LEON2 with some custom logic connected on an AMBA bus, along with a software build environment based on GNU tools. Figure 4.4 provides a basic overview of the SOCks design flow.

One particularly useful feature of SOCks is the ability to generate output from within the embedded application, that will be relayed either to the display or to a text file via the testbench. This is achieved by the use of several custom functions, such as `print_txt()` and `print_int()`, that can be called from within the embedded C program, with the result being the output character or integer is written to a memory address that is mapped to a hardware location monitored by the testbench. Although the provided functions can output only single characters or integers, extra functions can be built upon the provided functions to allow strings or larger numbers to be output. Details of the implementation and usage of these functions is available in the SOCks documentation [68].

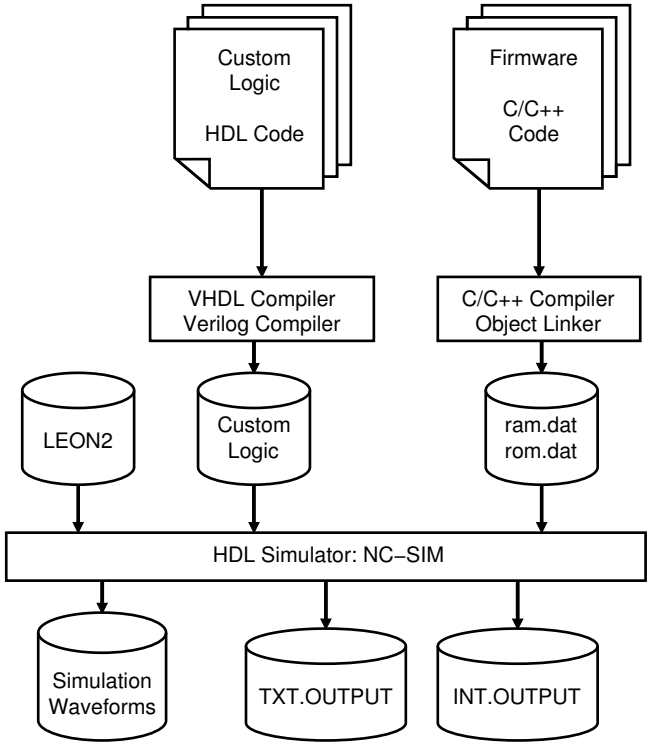


Figure 4.4: SOCks project design flow [67]

Example applications along with suitable Makefiles are provided in the `firmware` directory. Custom applications can be built in a similar manner to the example scripts, although care must be taken to avoid the use of any functions, such as `printf()`, that are not supported by the SOCks build environment. The entry function in SOCks applications is called `leon_test()` rather than the conventional `main()` found in most C applications. It is also necessary to modify the `locore1.S` assembly language file by removing the following code section:

```
#ifdef __leon__
  call leon_test    ! call test routine
#else
  call _leon_test   ! call test routine
#endif
```

and simply replacing it with:

```
  call leon_test    ! call test routine
```

to bypass the `#ifdef` construct, which causes problems with the linker.

A program that generates and displays a list of numbers that form the Fibonacci sequence is chosen to test LEON2 under SOCks. This was chosen both because it's a simple program that runs relatively quickly, and it's one of the standard Cascade coprocessor test suite applications. The Fibonacci sequence is defined by the relationship shown in Equation 4.5.

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases} \tag{4.5}$$

This sequence is implemented using the C function listed below, where `size` represents the number of elements in the sequence, and a pointer to the array that will be used to store the sequence is passed as `sequence`:

```
void fib(unsigned* sequence, unsigned size) {
  int i;
  if (size > 0) sequence[0] = 1;
  else return;
  if (size > 1) sequence[1] = 1;
  else return;
  for(i = 2; i < size; ++i) {
    sequence[i] = sequence[i - 1] + sequence[i - 2];
  }
}
```

This function is incorporated into a wrapper application that calls the function with a `size` value of 60 000, and outputs the sequence onto the screen during simulation via the testbench.

The SOCks project has been designed to be simulated on Cadence NC-Sim, and as such includes some auto-configuration files that are specific to NC-Sim. Therefore it was decided that, rather than spend time modifying the simulation scripts and HDL files to run with VCS or ModelSim, NC-Sim should be used with the provided scripts. Running a pre-built application is done simply by entering the `socks/sim` directory and executing the command:

```
../exe/socks_sim <application_name>
```

where `<application_name>` should be a build directory present within the `firmware` directory. The script will initialise the testbench to load the appropriate `ram.dat` file for the test application. When NC-Sim loads it starts executing commands in the `socks.tcl` file

within the `testbench/Tcl` directory—any desired simulation parameters or directives, such as maximum run time or data probes, can be placed in this file.

4.2.3 Monitoring switching activity

To perform power and energy analysis on the LEON2 processor, it is necessary to monitor switching activity within ModelSim and NC-Sim, in a similar manner to the procedure used for VCS in section 3.3. ModelSim's native switching activity format is value change dump (VCD). Although VCD output can later be converted to SAIF for use with Power Compiler using the `vcd2saif` command, the VCD file size is often very large making it a cumbersome format to use for longer or more complex simulations, even as an intermediate format.

As an alternative to generating VCD files from within ModelSim and later converting them to SAIF, Synopsys provides a library that can be integrated with ModelSim to allow direct generation of SAIF from VHDL simulations within ModelSim. This is known as the DPFLI interface, and it allows a subset of commands that are normally used within VCS to generate SAIF output, to be used from within ModelSim. Thus, many of the simulation commands used in section 3.3 can be applied to the ModelSim simulation. Further details on the DPFLI interface can be found in chapter 4, *Generating Switching Activity Information*, in the Power Compiler User Guide [69].

Shown below is a ModelSim do file used to run the standard LEON2 testbench while monitoring switching activity, and dumping a backward SAIF file once simulation completes.

```
vsim -c tb_func32 -foreign \
   "dpfli_init $SYNOPSYS/auxx/syn/power/dpfli/lib-linux/dpfli.so"
set_toggle_region /tbleon/tb/p0/leon0
toggle_start
run -all
toggle_stop
toggle_report backward_rtl.saif 1e-9 /tbleon/tb/p0/leon0
```

After the `vsim` command has been initialised, it is important to check for the following line in the output console; this confirms that the Synopsys power interface has been successfully initialised, ensuring that subsequent commands will be recognised. The indicator of a successful initialisation will be similar to that below.

```
# Loading /apps/Synopsys/syn_vX-2005.09-SP3/
              auxx/syn/power/dpfli/lib-linux/dpfli.so
# Synopsys power code initialized and linked successfully
```

Successful completion of the simulation results in the creation of `backward_rtl.saif`, which contains all the switching activity information for the RTL description of the LEON2 processor.

Generating switching activity from within NC-Sim requires a slightly different approach, as there is no Synopsys interface available to directly generate a SAIF file. Similarly to ModelSim, NC-Sim can generate VCD using either the standard Verilog PLI, or by issuing simulation directives. Since both LEON2 and the SOCks testbench are written in VHDL, the latter approach is used.

The `socks/testbench/Tcl/socks.tcl` file is modified to include VCD simulation directives as follows:

```
database -vcd -open backward -default
probe -create -vcd KS_top_inst.CoreInst.leon1 -depth all
run -timepoint 500 ms -absolute
finish
```

This instructs NC-Sim to monitor switching activity within the entire LEON2 processor and dump it to a file named `backward.vcd`.

As previously mentioned, the VCD file format produces very large, verbose output that quickly becomes cumbersome for long or complex simulations. Although there is no direct Synopsys interface allowing NC-Sim to create SAIF files, a VCD to SAIF conversion utility is provided by Synopsys that can use a UNIX pipe to convert VCD to SAIF from any simulator, while the simulation is running. Starting the `vcd2saif` utility with the command shown below creates a named pipe for the VCD output before launching the simulator using the supplied command.

```
vcd2saif -input backward.vcd -output backward_rtl.saif \
   -format VHDL -pipe "../exe/socks_sim fibonacci"
```

Once simulation is complete the specified output SAIF file (in this example `backward.saif`) is created, containing the switching activity information for the LEON2 processor.

4.2.4 Configuring the software build environment

The ability to run arbitrary software on the LEON2 testbench is required before meaningful power and energy analysis can be performed. Therefore it is necessary to set up a cross-compilation build environment targeted at the SPARC V8 architecture. A version of the GNU build environment is available free from Gaisler Research for this purpose, with the prefix `sparc-rtems`; e.g. the C compiler is `sparc-rtems-gcc`.

A Makefile is provided in the `tsource` directory for the purpose of building test programs, which results in the creation of a `ram.dat` file that is accessed by the VHDL testbench. The target build environment for the Makefile is `sparc-elf`, therefore it is necessary to modify it to point to `sparc-rtems` instead. This is done by running the command:

```
sed s/sparc-elf/sparc-rtems/g < Makefile > Makefile.rtems
```

The new Makefile is then referenced directly by using the `-f` option when running `make`.

Gaisler Research offers a software LEON2 simulator called TSIM, which allows software compiled for LEON2 to be verified, analysed and debugged much more quickly than is possible doing such tasks under RTL hardware simulation. Unfortunately this simulator is not available under the free licence that covers the LEON2 processor itself. An evaluation version is available at no cost, but no full licence was available for use during this project.

The recently compiled application can be run on the testbench from within ModelSim, ensuring that the configuration (as defined in `tbench/tbleon.vhd`) points to the correct `ram.dat` file. The configuration used is shown below—note that the `DISASS` option controls whether or not the simulator outputs a disassembly of all executed instructions to the display.

```
configuration tb_custom of tbleon is
  for behav
    for all:
      tbgen use entity work.tbgen(behav) generic map (
        msg2 => "2x128 kbyte 32-bit ram, 2x64 Mbyte SDRAM",
        DISASS => 0, ramfile => "tsource_new/ram.dat" );
    end for;
  end for;
end tb_custom;
```

4.2.5 RTL synthesis

LEON2 can be synthesised in Design Compiler using a similar technique to that described in section 3.2. A synthesis script is provided with the support files, although it requires some modification to point to the correct technology libraries, to set the desired operating frequency, and to create VHDL and Verilog netlists for gate-level simulation, along with their corresponding SDF files. Netlist and SDF generation is done by adding the following lines to the synthesis script:

```
change_names -rule vhdl -hierarchy
write -format vhdl -hierarchy -output ./leon_synth.vhd
write_sdf ./leon_vhdl.sdf

change_names -rule verilog -hierarchy
write -format verilog -hierarchy -output ./leon_synth.v
write_sdf ./leon_verilog.sdf
```

It is also necessary to ensure that the correct memory macro blocks are available for instantiation during synthesis. These are created using the Artisan Memory Generator, with the settings based on those shown in Table 4.2.

Cache set size	Words/line	Tag ram	Data ram
1 kbyte	8	32x30	256x32
1 kbyte	4	64x26	256x32
2 kbyte	8	64x29	512x32
2 kbyte	4	128x25	512x32
4 kbyte	8	128x28	1024x32
4 kbyte	4	256x24	1024x32
8 kbyte	8	256x27	2048x32
8 kbyte	4	512x23	2048x32
16 kbyte	8	512x26	4096x32
16 kbyte	4	1024x22	4096x32

Table 4.2: LEON2 cache ram cell sizes [66]

Interfacing LEON2 with memory blocks generated using the Artisan Memory Generator raises an issue with timing differences between the two blocks. The Artisan SRAM block requires the read/write address to be loaded and stable before the rising clock edge, as shown in Figure 4.5. However, LEON2 places the desired memory address onto the bus on the rising clock edge as shown in Figure 4.6, meaning that the address value is not stable during

the period required by the Artisan SRAM specification. To resolve this, a simple wrapper is inserted between the Artisan SRAM memory blocks and LEON2 processor. The wrapper delays the requested address provided by LEON2 by a single cycle, allowing the value on the SRAM's address input to be held steady during the following rising clock edge. The SRAM will provide the requested data on the same cycle after a delay of t_a; as this is the same cycle that LEON2 expects to receive the data with zero wait states, there is no performance loss introduced by the wrapper. The wrapper also interfaces the *RAMOEN* signal of LEON2 with the equivalent *CEN* signal of the SRAM block to assert the chip enable signal as appropriate.

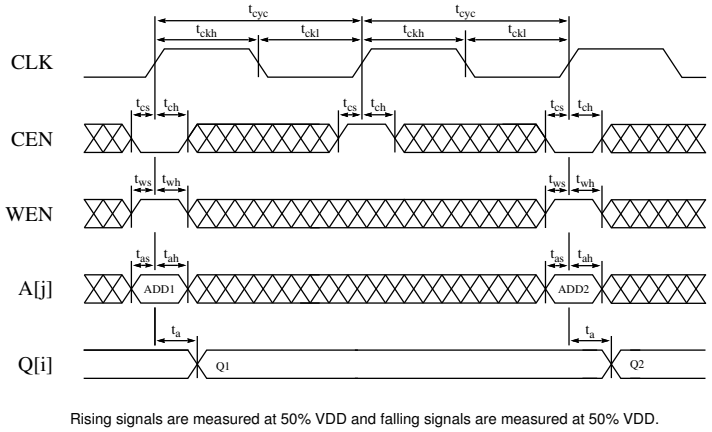


Figure 4.5: Artisan SRAM read cycle timing [70]

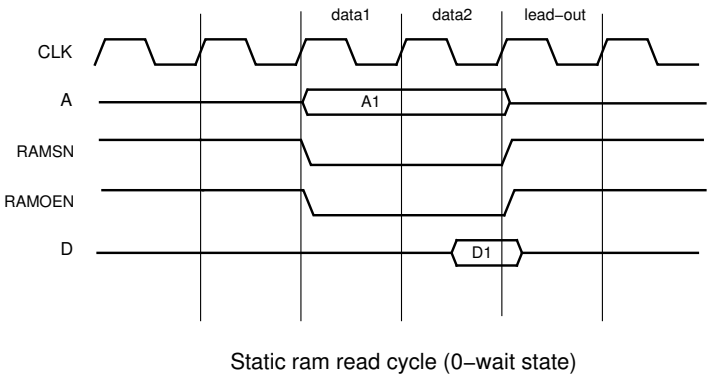


Figure 4.6: LEON2 SRAM read cycle timing [66]

As previously stated, configuration files for TSMC 0.18 µm and 0.13 µm are not included with LEON2, therefore it is necessary to modify the TSMC 0.25 µm configuration file `tech_tsmc25.vhd`. This file contains component declarations for instantiated memories (generated with Artisan Memory Generator) and pads. It also contains simulation models for the memories, which do not need to be modified as the Artisan Memory Generator can automatically create simulation models of the desired memory blocks.

After initial hand modification of the file had begun, it was discovered that Daniel Mok had already made the required modifications and published a new TSMC 0.13 µm file, `tech_tsmc13.vhd`, on the LEON mailing list board. The file is licensed under the same terms as the original `tech_tsmc25.vhd` configuration file, and can be downloaded at the following URL (free membership required to view messages or download files):

`http://tech.groups.yahoo.com/group/leon_sparc/files/`

The technology file is simply added to the `leon` directory, and the scripts modified as necessary to point to the new technology file. The Tcl synthesis script, based on the original dcsh-format script provided as part of the LEON2 package, is listed in Appendix B.3. Synthesis is started by issuing the command:

```
dc_shell-xg-t -f synth.tcl > synth.txt
```

Once synthesis completes, the generated netlist files `leon_synth.v` and `leon_synth.vhd` can be used for gate-level simulation, allowing the creation of more detailed switching activity files.

4.2.6 Netlist simulation and power analysis

Once synthesised, the generated VHDL netlist file `leon_synth.vhd` along with the LEON2 testbench can be run in a similar manner to that for RTL-level simulation, with the VHDL modules representing the LEON2 processor replaced by a single netlist file. As expected, gate-level simulation is a lot slower, and it is no longer possible to have disassembly of the currently executing instruction output to the screen (the `DISASS` configuration option has no effect).

Other than the extended run-time, netlist simulation is a similar process to RTL simulation as detailed in subsection 4.2.5, with a much larger SAIF file being generated due to the much greater level of detail being recorded. A script very similar to that listed in Appendix A.2 is used, with the `read_saif` command modified to point to the correct instance, as shown below:

```
read_saif -input backward.saif \
  -instance KS_top_tb/KS_top_inst/CoreInst/leon1
```

The power report for LEON2 is created for the processor core without cache memories, due to inaccuracies inherent in calculating power consumption for black-box components. Since determining absolute power consumption of the LEON2 processor is not the key goal of this section, omitting black-box power is not a significant issue—comparison of the core power can be undertaken for LEON2 and coprocessors generated by Cascade, omitting black-box power for both. The hierarchical power report showing the average consumption of non-black box components is shown in Figure 4.7. A breakdown of the power consumption of blocks within the processor core is not available due to the design having been flattened prior to power analysis.

```
------------------------------------------------------------------
                            Switch   Int      Leak     Total
Hierarchy                   Power    Power    Power    Power
------------------------------------------------------------------
leon
  mcore0 (mcore)
    proc0 (proc)
      c0 (cache)            8.46e-06    1.138 2.53e+06    1.140
      iu0 (iu)              3.91e-05    4.814 1.37e+07    4.828
```

Figure 4.7: LEON2 processor core power report

Summing the power of the two components listed in Figure 4.7 gives a total average power consumption of 5.968 mW. It should be noted that the component `c0 (cache)` is not a cache unit; rather it is a cache controller, which contains the cache block. Therefore the cache controller is included in the analysis as a synthesised block that forms part of the processor core, whereas the cache itself is excluded as a black-box component.

For comparison, the same software is run through Cascade to be offloaded to an automatically synthesised coprocessor, the power performance of which is analysed in a similar manner to that of the LEON2 processor core. The results of this analysis are shown in Figure 4.8, which contains an excerpt of the overall power report, highlighting both the overall coprocessor power, and that consumed by the memory macro blocks.

To enable a proper comparison between the two results, the memory macro blocks must be excluded from the coprocessor analysis. Thus the dynamic power works out to be 4.9166 - (1.146 + 0.339) = 3.4316 mW. Clearly this instantaneous power figure is somewhat lower than that determined for the LEON2 processor, which was 5.968 mW. However, the figure of most interest is that of the energy required to complete the entire test, and to determine that the number of cycles required by each implementation needs to be determined.

```
****************************************
Report : power
        -analysis_effort high
Design : test_copro
****************************************

                Cell Internal Power  =   4.5995 mW   (94%)
                Net Switching Power  = 317.0794 uW    (6%)
                                       ---------
              Total Dynamic Power    =   4.9166 mW  (100%)

              Cell Leakage Power     = 801.9362 uW


                        Cell      Driven Net  Tot Dynamic      Cell
                        Internal  Switching   Power (mW)       Leakage
Cell                    Power     Power       (% Cell/Tot)     Power (pW)
------------------------------------------------------------------------
fu_access_st_1r_0/ex_access_st_1r_0/cache_mem/ram_rsws_rsws_bw_4kx32
                           1.1439 1.627e-03      1.146 (100%)  600000000

CBNative_Slave_Generic/Inst_cu_direct_inst_cache/inst_instr_ram/mem
                           0.3283    0.0103      0.339 (97%)    80000000
```

Figure 4.8: Cascade coprocessor Fibonacci power report

For the coprocessor, determining the cycle count is trivial, since Cascade includes this statistic in its report for a coprocessor executing the software and data set used in the generation of the coprocessor. In this particular example, the coprocessor architectural simulation determines that it will take 615788 cycles to complete the Fibonacci test. At a clock speed of 100 MHz, that equates to 6.15788 ms. Thus the energy used during the test can be calculated as in Equation 4.6.

$$\begin{aligned} E &= P \times t \\ &= 3.4316 \times 10^{-3} \times 6.15788 \times 10^{-3} \\ &= 2.113138 \times 10^{-5} \\ &= 21.131 \mu J \end{aligned} \tag{4.6}$$

For LEON2, cycle count can be determined through either the use of a cycle-accurate processor simulator, or by monitoring the number of cycles taken to complete an HDL simulation. As the former approach requires the use of TSim, which in turn requires to be licensed from Gaisler Research, the latter approach is used. Conveniently, the SAIF file generated during simulation for the purposes of power analysis includes a `DURATION` entry, which, along with the `TIMESCALE` entry, indicates the length of simulation time for which switching activity was monitored. In this particular case, monitoring was active for the entire test run, giving the length of time taken to complete the test. The SAIF file value is listed as 12390845000 with the timescale in picoseconds, which is more conveniently written as 12.390845 ms. Thus, similarly to the case for the Cascade-generated coprocessor, the energy used during execution of the Fibonacci test can be calculated as in Equation 4.7.

$$\begin{aligned} E &= P \times t \\ &= 5.968 \times 10^{-3} \times 12.390845 \times 10^{-3} \\ &= 7.394856296 \times 10^{-5} \\ &= 79.395 \mu J \end{aligned} \tag{4.7}$$

By comparing the results of Equations 4.6 and 4.7, it can be seen that for this particular example an application-specific coprocessor generated by Cascade uses just over a quarter of the energy consumed by a general-purpose LEON2 processor.

Although this result is not unexpected (due to the efficient nature of a well-implemented application-specific processor running its target application), it is important not to read too much into this specific example. Neither LEON2 nor the coprocessor synthesised by Cascade are particularly optimised in terms of their configurations, and the test application is a very simple one. However it does provide a good basis for expanding into more complex, commercially-relevant comparisons in future cases.

4.3 OpenRISC 1200 processor

The OpenRISC project was started by Damjan Lampret with the aim of creating a free and open-source computing platform containing both RISC CPU/DSP architectures, and the software tools to support development on the platform. OpenRISC 1000 [71] is the current platform within the project at the time of writing, and the OpenRISC 1200 processor [72] is at the core of this platform.

OpenRISC 1200 is a 32-bit scalar RISC with Harvard architecture, 5 stage integer pipeline, virtual memory support (MMU) and basic DSP capabilities. A block diagram of the processor architecture is shown in Figure 4.9. OpenRISC 1200 is designed to interface with a WISHBONE rev.B3 SoC bus [73], with the required interfacing hardware being on-board in the standard configuration.

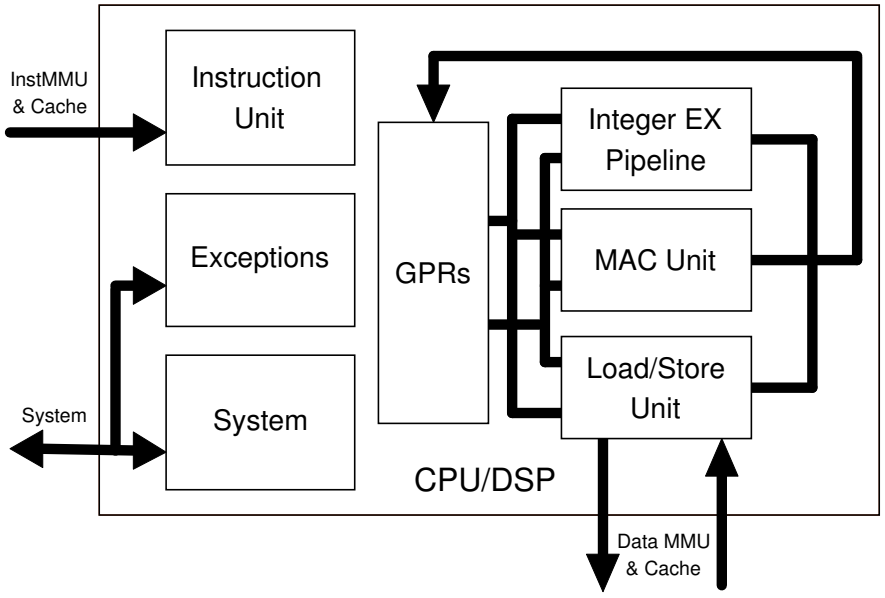


Figure 4.9: OpenRISC 1200 processor architecture [71]

4.3.1 Building the OpenRISC tool chain

The OpenRISC 1000 platform including OpenRISC 1200 processor and supporting tools can be checked out from the OpenCores CVS server by running the following commands:

```
CVSROOT=:pserver:anonymous@cvs.opencores.org:/cvsroot/anonymous
cvs -z9 checkout or1k
```

Once the check out has completed, the `or1k` directory will contain a number of directories. Among these is the `or1200` directory containing the OpenRISC 1200 processor RTL, testbench, synthesis scripts and documentation. Also of interest is the `or1ksim` directory, containing a functional simulator of the OpenRISC 1200 processor, which can be used to verify the correct operation of target applications before running a much slower hardware simulation.

Before any applications can be compiled for OpenRISC, the build environment needs to be configured. First, ensure that at least version 1.7 of the GNU tools `aclocal`, `autoconf` and `automake` are available. Some older Linux environments do not have the required versions of these tools pre-installed, and in that case updated versions will need to be downloaded and installed before the build process commences, otherwise problems will be encountered at a later stage in the process.

The OpenRISC binary utilities can then be built. This is done by entering the `or1k` directory and running the commands:

```
mkdir b-b
cd b-b

../binutils-2.16.1/configure --target=or32-elf --prefix=$HOME/or32-elf

make -w all install

cd ..
```

Once the binary utilities have been built, the gcc cross-compiler can be built. This is performed using the following commands:

```
mkdir b-gcc
cd b-gcc

../gcc-3.4.4/configure --target=or32-elf --prefix=$HOME/or32-elf   \
--enable-languages=c,c++ --with-gnu-as --with-gnu-ld --with-newlib \
--with-gxx-include-dir=$HOME/or32-elf/or32-elf/include -v          \

make -w all install

cd ..
```

Finally, the gdb debugger is built using the commands shown below:

```
mkdir b-gdb
cd b-gdb

../gdb-5.3/configure --target=or32-elf --prefix=$HOME/or32-elf

make all

cd ..
```

When all tools have been successfully built, it is necessary to add the path where the binaries are located to the system path. This is done with the following command, on the assumption that the home directory allows executables to be run. Otherwise, it will be necessary to install the binaries to a system directory, which requires superuser privileges.

```
export PATH=$HOME/or32-elf/bin:$PATH
```

Occasionally problems can occur with the CVS server used to download the aforementioned tools, resulting that many of the required build files are missing. For example, when checking out `binutils`, the `configure` file may not be present, resulting in the build failing immediately. The missing files do not reappear even if an old version is selected for download using the `-D <date>` flag on the CVS command line.

To overcome this problem, a fall-back script has been written that downloads the standard GNU toolchain utilities directly from their original sources, patches them, and builds each component automatically. The script also logs the output from each build to enable any error conditions to be analysed. The full script is listed in Appendix B.4.

4.3.2 Cross-compiling applications

Applications can be cross-compiled for the OpenRISC processor by defining `or32-elf-*` or `or32-uclinux-*` tools as the target compiler, linker, assembler, etc. within the Makefile for the application. As an example, an excerpt from the modified Makefile for the Dhrystone benchmarking tool is shown below.

```
cases = dhry-nocache-O0 dhry-nocache-O2 dhry-icdc-O2
common = ../support/libsupport.a

all: $(cases)

dhry-nocache-O0: dhry-O0.o ../support/reset-nocache.o $(common)
        or32-elf-ld -T ../support/orp.ld $? -o $@.or32
        or32-elf-objcopy -O binary $@.or32 $@.bin
        ../utils/bin2hex $@.bin > $@.hex
        cp $@.hex ../../sim/src/
```

For an initial test run using OpenRISC, the Fibonacci sequence program listed in subsection 4.2.2 is built using a Makefile similar to that above. The resulting hex file can be used with the or32-sim functional simulator, to ensure that the test completes correctly. In this case, as the test is quite small it will be run directly on the simulated hardware.

4.3.3 Synthesis

A synthesis script located at `syn/synopsys/top.scr` is provided with the OpenRISC 1200 processor. This script is written in the now deprecated "dcsh" language, so for consistency with other scripts used in the project, and to enable the use of Design Compiler's XG mode, it must be converted to Tcl. Synopsys provides a tool with Design Compiler, `dc-transcript`, that can perform the conversion automatically. The Tcl script output by the tool requires a little clean-up to maintain legibility, but overall the automated conversion process is clean and effective.

Synthesis of OpenRISC 1200 is very similar to that carried out in section 3.2. The script file includes a number of variable definitions that are declared before reading in the design. These can be modified to control a number of parameters relating to the synthesis process, such as the target technology, the target clock frequency at which the processor will be run (subject to critical path timing limitations), and the target area constraint that the synthesis tool should aim to meet. The variables are listed below:

```
set TOPLEVEL or1200_top
set TECH umc13          # vs_umc18, art_umc18, umc13
set CLK clk_i
set RST rst_i
set CLK_PERIOD 10       # 100 MHz
set MAX_AREA 0          # Push hard
set DO_UNGROUP no       # yes, no
set DO_VERIFY no        # yes, no
set CLK_UNCERTAINTY 0.1 # 100 ps
set DFF_CKQ 0.2         # Clk to Q in technology time units
set DFF_SETUP 0.1       # Setup time in technology time units
```

The only items that require to be changed from the defaults are `TECH`, which must be set to target the UMC 130 nm technology, and `CLK_PERIOD`, which is set to 100 MHz.

The supplied script does not have the required definitions for UMC 130 nm technology libraries. Therefore a few lines must be added to set the appropriate variables to reflect the requirements of the aforementioned technology library. The additional lines are added to the `TECH` conditional construct, and are listed below:

```
else if (TECH == "umc13") {
        HDDFFPQ2  = HDDFFPQ2
        LIB_DFF_D = TYPICAL/HDDFFPQ2/D
        TYPICAL   = TYPICAL
}
```

Finally, the `.synopsys_dc.setup` file must be modified to ensure that the correct libraries are being referenced. This involves modifying three variable declarations as below:

```
set target_library {"umcl13u210t3_typ.db"}
set link_library   {"*" "umcl13u210t3_typ.db"}
set symbol_library {"umcl13u210t3.sdb" "generic.sdb"}
```

Synthesis results in the creation of the Verilog netlist `out/final_or1200_top.v` upon completion of the script. The netlist can then be simulated to generate switching activity information for power analysis.

4.3.4 Simulation and power analysis

There are some issues that arise when attempting to simulate the OpenRISC 1200 netlist using the testbench provided as part of the OpenRISC Reference Platform. Rather than spending time resolving these issues, a testbench from an alternative project was sourced. The testbench used is included as part of the OpenRISC Infrastructure Tutorial, written by Tushar Kumar at Georgia Institute of Technology's Department of Electronic and Computer Engineering, and modified as necessary to work with the synthesised processor. The required modifications consist of changes to port names and widths to ensure consistency between modules, and component instantiation changes to reflect the configuration of the synthesised OpenRISC 1200 processor.

Netlist simulation of the OpenRISC 1200 processor is undertaken using ModelSim, since the supplied simulation scripts are targeted at this simulator. The approach taken is similar to that described in subsection 4.2.1. After compiling the Verilog files, the Synopsys DPFLI interface is initialised to allow output of switching activity information in SAIF format, using the commands shown below.

```
vsim work.testbench_top -foreign \
   "dpfli_init $SYNOPSYS/auxx/syn/power/dpfli/lib-linux/dpfli.so"
set_toggle_region /testbench_top/or1200_top/or1200_cpu
toggle_start
run -all
toggle_stop
toggle_report backward.saif 1e-9 /testbench_top/or1200_top/or1200_cpu
```

Once simulation completes, the SAIF file is used for power analysis with Power Compiler, in a process similar to that carried out in previous sections of this chapter. The `.synopsys_dc.setup` file from subsection 4.3.3 is re-used for power analysis.

Results of power analysis for OpenRISC 1200, running the Fibonacci sequence test on a UMC 130 nm process technology, are shown in Figure 4.10.

Examining only the CPU core itself, OpenRISC 1200 consumes 6.626 mW at a clock speed of 100 MHz. Examination of the SAIF file reveals that the processor takes 10.00013 ms to complete the test. Therefore the energy consumed during the test can be calculated as in Equation 4.8.

```
****************************************
Report : power
        -cell
        -nworst 20
Design : or1200_top
****************************************

Operating Conditions: TYPICAL   Library: umcl13u210t3_wc
Wire Load Model Mode: top

Global Operating Voltage = 1.08
Power-specific unit information :
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1uW

                      Cell         Cell
                      Internal     Leakage
Cell                  Power        Power
------------------------------------------------
or1200_cpu               6.6260     188.9228
dwb_biu                  0.3342       2.0221
or1200_immu_top          0.3063       1.4216
or1200_tt                0.1907       2.7752
or1200_dc_top            0.1778       2.4562
or1200_ic_top            0.1760       2.3995
or1200_pic               0.1229       1.4402
or1200_du                0.0990       1.1388
or1200_pm                0.0207       0.2327
------------------------------------------------
Totals                   8.054mW    202.809uW
```

Figure 4.10: OpenRISC 1200 core power summary report

$$\begin{aligned} E &= P \times t \\ &= 6.6260 \times 10^{-3} \times 10.00013 \times 10^{-3} \\ &= 6.626086 \times 10^{-5} \\ &= 66.26\mu J \end{aligned} \tag{4.8}$$

Although not directly comparable with the energy consumption determined for a Cascade generated coprocessor on the same test (due to the process technology being different), examination of the result in Equation 4.6 shows that the coprocessor using TSMC 130 nm technology uses around a third of the energy of OpenRISC 1200 using UMC 130 nm run-

ning the same test. Comparing the LEON2 processor energy calculation in Equation 4.7, the OpenRISC 1200 processor is slightly more efficient than the LEON2. It must again be emphasised that the difference in process technology reduces the usefulness of a direct comparison of these figures however.

4.4 Summary

The initial plan when starting work on open-source processor cores was to analyse the power and energy consumption of those cores, such that they could later be compared with the power and energy consumption of a Cascade coprocessor running the same applications. As the analysis progressed, it became clear that such a direct comparison is unlikely to be particularly useful to the project, particularly since the available open-source processors are not often used in the target markets for Cascade. The time taken to get a scientifically valid comparison would be quite substantial, as the process technology would have to be consistent across all processors, which would require sourcing memory blocks or writing wrappers to allow designs to target a different technology to what the included scripts and memory blocks are targeted to. The work undertaken in subsection 4.1.4 highlights the large variance in power and energy results that occur when the process technology vendor is changed, therefore making comparisons across vendors, even at the same process technology node such as 0.13 μm, is meaningless.

As a result, it was decided to undertake a direct analysis of each processor core, rather than a comparative analysis against a Cascade coprocessor. Doing so has proved to be particularly useful in building the knowledge of the tools used for analysis, as well as contributing to the development of a tool flow that is used in various other parts of this project—particularly the generic coprocessor power evaluation detailed in chapter 3. The three goals set at the start of this chapter have been met, albeit with some modification of the exact utility, and therefore interpretation, of those goals.

5. Accelerating MediaBench using Cascade

Benchmark suites are often used as a fixed, usually impartial, means of comparing different devices for a desired set of criteria. Typically for processors this will be performance, although power and energy consumption are increasingly compared using benchmarks. A benchmark suite should closely reflect the target applications of the devices that it is intended to be used on, to ensure the results are a meaningful indicator of the real world performance of those devices. For example, there are benchmark suites that specialise on integer or floating-point operations, networking operations or I/O operations.

A widely-used, commercial benchmark suite is SPEC, produced by Standard Performance Evaluation Corporation. SPEC is intended for use with general purpose 32-bit desktop and server computing systems and as such is not particularly suited to evaluating embedded processor performance. As a result of this limitation of SPEC, the EDN Embedded Microprocessor Benchmark Consortium created the EEMBC benchmark suite, which is actually composed of a selection of benchmark suites targeted at different applications. Unfortunately EEMBC is typically licensed only to consortium members, and as such it is seldom used in academia.

There are several free, fully-open alternative benchmark suites that have been developed by academic researchers with an interest in embedded software and devices. One of these is MediaBench [74], a collection of open-source C applications and reference data sets that are suitable for cross-platform compilation. The applications are mostly multimedia orientated, although two are cryptography applications. Another suite is MiBench [75], which more closely resembles EEMBC in that it contains a collection of suites each with different target applications, such as automotive, networking and telecommunications. Similarly to MediaBench, all applications are written in C, making them portable to any platform with compiler support.

After consideration, MediaBench was selected as the benchmark suite to use in this project. A key driver is its acceptance in academic literature, meaning that it provides a solid and well-understood foundation for any academic publications, and also provides a basis for any comparisons with previous work. MiBench is also respected in academia, but to a lesser degree at present. Another consideration is the size of each suite; although the larger set of applications in MiBench provides greater diversity, time restrictions mean that it may not be possible to use all applications within the suite. Dropping arbitrary applications from the suite is one solution, but doing so may create unintentional bias in the results. As such, the smaller MediaBench suite is more suited to the requirements of this project.

The primary purpose of undertaking this work is to provide a relevant, consistent and impartial platform for development and subsequent analysis of the functionality that will be added to Cascade as part of this project. The MediaBench suite is considered to be highly representative of typical applications targeted by Cascade, and as such it provides an ideal platform to fulfil the aforementioned requirements.

5.1 Cross-compiling MediaBench for ARM

The first stage in creating coprocessors for accelerating applications in the MediaBench suite is to compile the applications for a supported host processor. The ARM9 processor has been selected as it is the most commonly used host processor at the time of writing.

There are several toolchains available for building applications for the ARM architecture, one of which is the freely-available GNU toolchain port from CodeSourcery [76]. From the 2005-Q1 version, CodeSourcery's tools are fully compatible with the ARM Application Binary Interface (ABI) standard [77], meaning that the output from a CodeSourcery tool can be used with one of ARM's own tools, such as RealView Debugger.

Many of the build scripts included with MediaBench are targeted at the GNU toolchain, therefore using the CodeSourcery tools minimises the amount of modification required to port the suite to the ARM processor.

The MediaBench suite consists of the following applications (descriptions taken from [74]):

ADPCM Adaptive differential pulse code modulation is one of the simplest and oldest forms of audio coding

EPIC An experimental image compression utility. The compression algorithms are based on a bi-orthogonal critically sampled dyadic wavelet decomposition and a combined run-length/Huffman entropy coder. The filters have been designed to allow extremely fast decoding without floating-point hardware

G.721 Reference implementations of the CCITT (International Telegraph and Telephone Consultative Committee) G.711, G.721 and G.723 voice compressions

Ghostscript A PostScript language interpreter. The single application for Ghostscript is gs, which does file I/O but no graphical display

GSM European GSM 06.10 provisional standard for full rate speech transcoding, prI-ETS 300 036, which uses residual pulse excitation/long term prediction coding at 13 kbit/s. GSM 06.10 compresses frames of 160 13-bit samples (8 kHz sampling rate, i.e. a frame rate of 50 Hz) into 260 bits

JPEG JPEG is a standardised compression method for full colour and greyscale images. JPEG is lossy, meaning that the output image is not exactly identical to the input image. Two applications are derived from the JPEG source code; cjpeg does image compression and djpeg, which does decompression

Mesa Mesa is a 3-D graphics library clone of OpenGL. All display output functions were removed from the library and demo programs included in the package. Three applications are used: mipmap executes fast texture mapping using precomputed filter results, osdemo executes a standard rendering pipeline, and texgen generates a texture mapped version of the Utah teapot

MPEG MPEG2 is the current dominant standard for high quality digital video transmission. The important computing kernel is a discrete cosine transform for coding and the inverse transform for decoding. The two applications used are mpeg2enc and mpeg2dec for encoding and decoding respectively

Pegwit A program for public key encryption and authentication. It uses an elliptic curve over GF(2255), SHA1 for hashing, and the symmetric block cipher square

PGP PGP uses "message digests" to form signatures. A message digest is a 128-bit cryptographically strong one-way hash function of the message (MD5). To encrypt data, it uses a block-cipher IDEA, RSA for key management and digital signatures

RASTA A program for speech recognition that supports the following techniques: PLP, RASTA, and Jah-RASTA. The technique handles additive noise and spectral distortion simultaneously, by filtering the temporal trajectories of a non-linearly transformed critical band spectrum

Most of the applications in the suite come with a GNU Makefile to automate the build process. In such cases, the Makefile is used to attempt the build, initially with only minor modifications to target cross-compilation for ARM. This would typically involve modifying the following two variables to that shown:

```
CC = /home/paulm/arm_tools/bin/arm-none-eabi-gcc
CFLAGS = -O -mcpu=arm7tdmi
```

This approach provides successful compilation of tests ADPCM, EPIC, G.721, JPEG, MPEG and Pegwit (with the addition of `-DLITTLE_ENDIAN` to the `CFLAGS`). All tests successfully complete the provided benchmark run, the only modification being required is to the MPEG test; the file `data/options.par` has to be modified to point to the correct path containing stimulus files.

Some of the other tests require modification to either build files or source files. GSM requires changes to the Makefile, to reflect its use of `CCFLAGS` rather than `CFLAGS`. In addition, the `toast.c` file has to be modified to remove calls to OS provided functions—`chmod()`,

`chown()` and `utime()`—as these functions are not available when the application is not running under an operating system. Their removal does not affect application functionality.

PGP requires some similar modifications to those of GSM to remove operating system calls—in this case `getch()` is replaced by `getchar()`. In addition, PGP depends on the RSAREF package, which is included with the PGP source; RSAREF must be built before PGP otherwise the PGP build will fail. To ensure username and passphrase consistency, the PGP test is run with the command:

```
pgp "-es data/pgptest.plain paulm -zpaulm -u paulm"
```

which encrypts and signs the data in `pgptest.plain` with username and passphrase both `paulm`, writing the output ciphertext to `pgptest.plain.pgp`.

The RASTA benchmark requires several Sphere library files—`libsp.a` and `libutil.a`—to be re-built into an ARM compatible format. Similarly to GSM and PGP, it also requires the removal of OS-dependent calls from the source code. These changes allow RASTA to be successfully built for ARM. However upon performing a test execution, RASTA complains that it cannot open the input file, even though the file is in the correct location and is readable. Considerable time spent investigating the problem did not provide a solution, with the most likely cause being an incompatibility with the ARM gcc tools. Therefore the RASTA benchmark is excluded from the MediaBench suite for the purposes of this project.

Mesa also proved to be a problematic benchmark to build—the top-level Makefile attempts to call a second Makefile within the `demos` directory, but no Makefile exists there. There is no `README` file included, nor are there any `exec` scripts typical of MediaBench suite demos. It is likely that there are files missing from this benchmark, preventing a successful build.

Finally, the ghostscript benchmark also had to be excluded due to build problems. It has a complex Makefile that attempts to compile and run small build tools as part of the build script. Normally this would happen transparently, but when cross-compiling for a different target architecture (as is the case here), it is not possible to run the compiled tools on the host, except via a simulator. After some attempts to work around the issue, by either introducing a target architecture simulator to the build script, or compiling the component parts manually, it was decided that the time required would be substantial—too long for the benefit it would provide the project.

5.2 Offloading functions to Cascade coprocessors

Once successfully built, each benchmark is analysed to determine suitable functions for offloading to a Cascade coprocessor. This is done with the help of a profiling tool, `gprof`, to determine the proportion of execution time spent in each function. To ensure each application has the necessary hooks for profiling, the `-pg` flag must be passed to both the compiler and linker. When the application is run it will generate a file, `gmon.out`, containing profiling information that can be analysed by `gprof`. Dynamic functions (those which are dependent on the result of a conditional statement) can produce highly variable profiling results based on the test application. Cascade treats such functions similarly to static functions, therefore it is important to ensure the test run is highly representative of the target application.

Each benchmark is split into two runs—one for the encode operation and another for the decode operation. The MPEG benchmark is an exception in that the decode function is split into a further two operations: one using fast Fourier transform (FFT), and the other using a reference integer calculation. In some cases both encode and decode operations are carried out by a single binary (with the operation selected by passing a command-line flag). In such cases each operation may call a different core function, meaning the execution pattern of the binary can vary significantly between encode and decode operations. In addition, regardless of the functional behaviour, each operation requires a separate run cycle, therefore the best option from a coprocessor acceleration viewpoint is to treat both operations within a benchmark as a separate test.

Some of the benchmarks within MediaBench have short run-times. Ideally the granularity of profile monitoring could be increased, but there does not appear to be a simple way to achieve this. In the absence of granularity control, better profiling results can be generated by running the benchmark several times and aggregating the profiling information. This is done by performing the following steps:

1. Run the program to be profiled—this generates `gmon.out`.
2. Rename `gmon.out` to `gmon.sum`.
3. Run the program again to generate a new `gmon.out`.
4. Run `gprof -s <program> gmon.out gmon.sum`. This combines the information from `gmon.out` into `gmon.sum`.

5. Repeat steps 3 and 4 as many times as desired to get the combined profiling information from all runs into `gmon.sum`.
6. Run `gprof <program> gmon.sum` to get the summarised profile for all runs.

An abbreviated sample output from `gprof` for the `mpeg2_decode.ref` test is shown in Figure 5.1. It can be seen from the results in the highlighted case that `Reference_IDCT` is an ideal candidate for offload, since 63.33% of the execution time is spent within that function (including time spent in its child functions).

The desired functions are selected for offloading using the following procedure within the `test.tcl` file called by Cascade (the entire file can be found in Appendix C.2):

```
proc Map {} {
        copro_map_function_group ENTRY function_name
}
```

Table 5.1 lists the functions offloaded for each benchmark within the MediaBench suite. It is possible to offload multiple function groups to a coprocessor, with varying degrees of functional overlap, however, for the sake of simplicity and consistency, only a single function group is offloaded for each benchmark. It should be noted that, for the `epic_encode` benchmark, the function `reflect1` is explicitly offloaded as a local function in addition to the parent function listed in Table 5.1. This is due to the function being called indirectly via a pointer, resulting in it not being implicitly offloaded as part of the function group as statically determined by Cascade. For all of the other tests, the function group is determined automatically by analysing the call graph from the top-level function.

Each of the benchmarks are individually run through Cascade's automated test suite, which offloads the selected function group and creates a coprocessor optimised to the task of executing the offloaded function group. Cascade allows the user to specify several preferences, including effort level (which determines how many candididates will be considered, and how quickly they will be pruned during DSE) and area/performance trade-off. Default settings are used in all cases, to ensure consistency for any future comparisons. The configuration used by Cascade is determined by the `default.xml` file, listed in Appendix C.3.With effort level set to low, design space exploration typically takes around 10 minutes on a Pentium 4 PC, although this can vary greatly depending on the complexity of the target application. Increasing the effort level causes the run time to increase rapidly, as the number of candidates examined increases.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self
 time   seconds   seconds    calls  name
 63.33      0.19     0.19    39600  Reference_IDCT
 10.00      0.22     0.03    41190  form_component_prediction
 10.00      0.25     0.03           __mcount_internal
  6.67      0.27     0.02    39600  Add_Block
  3.33      0.28     0.01  2534400  putbyte
  3.33      0.29     0.01    11335  Decode_MPEG1_Non_Intra_Bl
  3.33      0.30     0.01       60  store_yuv1
  0.00      0.30     0.00   517570  Show_Bits
  0.00      0.30     0.00   510605  Flush_Buffer
  0.00      0.30     0.00   249235  Get_Bits
  0.00      0.30     0.00    39600  Clear_Block
  0.00      0.30     0.00    21600  Get_Bits1
  0.00      0.30     0.00    13730  form_prediction
  0.00      0.30     0.00    13240  Get_motion_code
  0.00      0.30     0.00    13240  decode_motion_vector
  0.00      0.30     0.00    10740  Decode_MPEG1_Intra_Block
  0.00      0.30     0.00     7160  Get_Luma_DC_dct_diff
  0.00      0.30     0.00     6620  motion_vector
  0.00      0.30     0.00     6600  motion_compensation
  0.00      0.30     0.00     6445  Get_macroblock_address_in
  0.00      0.30     0.00     6445  Get_macroblock_type
  0.00      0.30     0.00     6445  decode_macroblock
  0.00      0.30     0.00     6445  macroblock_modes
  0.00      0.30     0.00     4810  form_predictions
  0.00      0.30     0.00     3640  Get_coded_block_pattern
  0.00      0.30     0.00     3580  Get_Chroma_DC_dct_diff
  0.00      0.30     0.00     3250  Get_B_macroblock_type
  0.00      0.30     0.00     1650  Get_I_macroblock_type
  0.00      0.30     0.00     1545  Get_P_macroblock_type
  0.00      0.30     0.00      375  next_start_code
  0.00      0.30     0.00      340  Flush_Buffer32
  0.00      0.30     0.00      300  slice
  0.00      0.30     0.00      300  slice_header
  0.00      0.30     0.00      300  start_of_slice
  0.00      0.30     0.00      155  skipped_macroblock
  0.00      0.30     0.00      100  Fill_Buffer
  0.00      0.30     0.00       40  Get_Bits32
  0.00      0.30     0.00       30  Get_Hdr
```

Figure 5.1: Flat function profile for MPEG2 decode benchmark

Benchmark	Offloaded function
adpcm_decode	adpcm_decoder
adpcm_encode	adpcm_coder
epic_decode	collapse_pyr
epic_encode	internal_filter
g721_decode	predictor_zero
g721_encode	predictor_zero
gsm_decode	gsm_asl
gsm_encode	gsm_asl
jpeg_decode	jpeg_idct_islow
jpeg_encode	encode_mcu_AC_first
mpeg2_decode.fft	Fast_IDCT
mpeg2_decode.ref	Reference_IDCT
mpeg2_encode	idct
pegwit_decode	SHA1Transform
pegwit_encode	SHA1Transform
pgp_decode	ideaCfbDecrypt
pgp_encode	ideaCfbEncrypt

Table 5.1: MediaBench suite offloaded functions

The instructions passed to Cascade to indicate what functions should be offloaded are contained within a `test.tcl` file, of which there is one for each test. This file also allows the inclusion of other directives to control the characteristics of the coprocessor, such as memory configuration and base architecture selection. As before, the use of additional directives is avoided, for consistency reasons.

Once the scripts have been put in place for the desired functions to be offloaded from each test, it is necessary to ensure that the results generated by the accelerated code running on the coprocessor are identical to those generated by the original code running on an ARM processor simulator. This is done by comparing both the standard output, and any files generated, from test runs before and after function offload to a coprocessor.

Code modifications are required to some of the tests to allow this automated verification to take place. Specifically, tests which do not direct their output to standard out, or which do not create files, must be modified to do so in order that consistency of operation can be automatically checked between test runs. Additionally, any command line options that require to be passed to test must be hard-coded into the main function of the program, as there is no connection to standard input in the test environment.

Cryptographic tests—Pegwit and PGP—have more complex issues preventing consistency between tests. When using encryption, most cryptographic algorithms create a unique session key to encrypt the source data. The cryptographic key is then used to encrypt the session key, allowing later recovery of the session key, and subsequent decryption of the ciphertext. As the session key is changed each time the encryption function is called, the consistency check on the ciphertext fails.

Randomisation of the session key can be prevented by modifying the source code to prevent any random seed from being generated to create the source key. Although this introduces a serious weakness into the cryptographic strength of the algorithm, it should not significantly affect the computational performance of the benchmark, and therefore is a suitable solution for test and analysis purposes.

With all offloaded functions now able to run through Cascade's automated test system, the coprocessor for each test can be created and evaluated using the existing analysis flow developed in chapter 3. Table 5.2 lists the results of this analysis for each individual coprocessor created to accelerate each of the MediaBench benchmarks. All coprocessors were targeted at TSMC 130 nm process technology, and no coprocessor optimisations were enabled.

Benchmark	**Execution cycles**	**Total area (mm^2)**	**Average power (mW)**
ADPCM Decode	6103597	2.781	5.70
ADPCM Encode	4343782	3.463	5.01
Epic Decode	8950780	6.084	6.16
Epic Encode	850436567	3.007	5.10
g721 Decode	29262397	4.834	4.61
g721 Encode	26502367	4.951	4.55
GSM Decode	1424608	1.923	2.39
GSM Encode	1506469	1.951	2.51
JPEG Decode	3069928	3.515	3.12
JPEG Encode	8674122	2.440	3.12
MPEG2 Decode (fft)	14337639	3.395	5.29
MPEG2 Decode (ref)	204618480	3.477	4.77
MPEG2 Encode	16508100	6.231	5.34
Pegwit Decode	70873	4.393	4.51
Pegwit Encode	2513105	4.331	4.94
PGP Decode	2076627	2.579	5.32
PGP Encode	1695210	3.019	6.35

Table 5.2: MediaBench suite coprocessor evaluation (TSMC 130 nm)

It is clear from the results that the coprocessors generated have a wide range of sizes and power consumption, even using the standard base template for all coprocessors. This spread ensures that any functionality being tested using the MediaBench suite will be exercised over a range of real-world conditions, reducing the possibility of false results due to, for example, coprocessor size bias.

5.3 Summary

In this chapter, MediaBench was selected as a benchmark suite representative of the target applications typically accelerated using Cascade coprocessors. The applications and their build environments within MediaBench were adapted as necessary to cross-compile them for the ARM processor.

After compilation, each benchmark was split into two portions—encode and decode—and analysed to determine suitable functions for offloading to a coprocessor. The offload and coprocessor generation process was then automated to allow a test run of the entire MediaBench suite to be run without intervention. Additionally, a verification system was put in place to ensure consistency with the original results; this required modification to some of the benchmarks to remove randomisation elements within the code.

The coprocessors generated for each benchmark were then analysed to determine their area requirements, power consumption, and the number of cycles taken to complete processing of the data supplied with each benchmark.

The work undertaken in this chapter will be used in subsequent work to develop and validate new power analysis and optimisation functionality to be integrated into Cascade. MediaBench offers an an ideal target for such work, as it provides a good representation of typical target applications, while offering enough variety between those applications to thoroughly test the performance of newly implemented functionality.

6. Creating functional unit models

This chapter details the creation of energy models for each of the functional units present in the library available to Cascade. Functional units represent the basic building blocks that are used to synthesise a coprocessor, for example adder, shifter, multiplier and branch units; a complete list of the available units is shown in Table 6.1. Wrappers around the memory blocks that are used for the data cache as also implemented as functional units, known as access units, of which there are fourteen different types. For clarity, Table 6.1 does not list each access unit type individually. Each functional unit represents a complete instruction-level operation, issued from the part of the VLIW instruction stream decoded by the coprocessor.

access_*	logical
arithmetic	multiplier32
bitshift	multiplier64
branch	single_cycle_multiplier64
combine	predicate
coreregfile	registerfile
immediate32	select
immediate8	squash

Table 6.1: Functional units available to Cascade

Functional unit energy models are developed with reference to the tool flow described in chapter 3. Figure 3.4 on page 37 lists the 20 worst-case cells, in terms of average power consumption, for a typical coprocessor design. Although the order of cells in such a table will vary between target applications, such a list serves as a valid basis for an initial prioritisation of analysis resources.

A typical functional unit is made up of a number of blocks as shown in Figure 6.1. The execution unit is the key difference between the various functional units available to Cascade,

although the exact blocks and layout of a functional unit depends on both its general function and the chosen configuration for each particular instantiation.

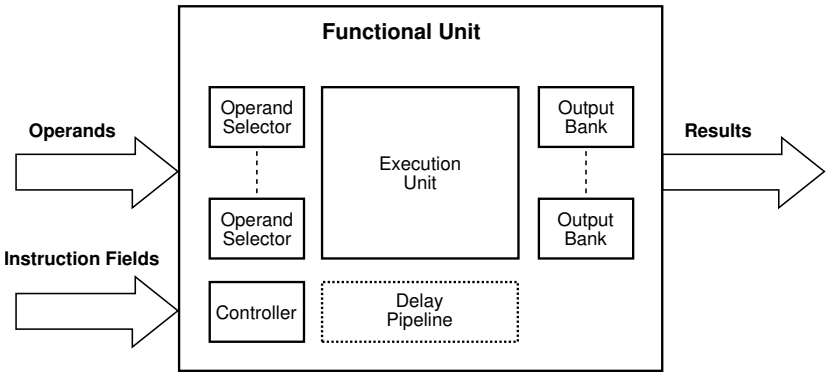


Figure 6.1: Functional unit block diagram

As determined in chapter 3, the most significant units in terms of overall power and energy consumption within a Cascade coprocessor are memory blocks and the multiplier unit. Due to the complexity and variety of memory blocks, a separate chapter is dedicated to their analysis—see chapter 7. In section 6.1, the multiplier unit is analysed in detail. All other functional units are analysed in a more coarse-grain manner in section 6.2, reflective of their smaller influence on the overall power and energy performance of the coprocessor.

Hierarchical analysis of the functional units over a range of tests revealed that the power consumption of the execution unit within a particular functional unit (as shown on Figure 6.1) usually does not vary by a large degree across different tests. Rather, it is the output banks (and the number of output banks present in each instantiation) that show the largest variance within many functional units. For that reason, a detailed analysis of output bank power and energy consumption is undertaken in section 6.3.

6.1 Multiplier unit

Multipliers are often the largest area consuming logic unit in embedded devices, and depending on utilisation, may also be the most energy hungry. The example power figures in Figure 3.4 show the significance of the multiplier unit in one Cascade generated coprocessor; research on other coprocessors has indicated that in cases where a multiplier unit is utilised, it will usually be the most significant logic block in terms of both energy and area utilisation.

Therefore in this section, a more detailed examination of the multiplier units and their energy consumption is undertaken.

Several multiplier units are used by Cascade, the key ones being a 64-bit stallable pipelined multiplier, and a similar 32-bit multiplier. The multiplier units used by Cascade are built around a Synopsys DesignWare IP block, specifically the stallable pipelined multiplier block *DW_mult_pipe* [78]. Using the IP block is more efficient than implementing the multipliers by hand, as it is specifically optimised at synthesis time for the particular parameters of the individual instantiation, such as input widths and pipeline length. The multiplier IP block is synthesised to standard cells in the netlist, giving full visibility for area and energy analysis; there is no issue of black-box component restrictions.

There are some subtle differences between the 64-bit and 32-bit multipliers, aside from the obvious input and output width differences. The 32-bit multiplier unit does not have an enable signal input, which means it cannot be independently stalled like the 64-bit multiplier can. In addition, the 64-bit multiplier has a signed/unsigned mode, whereas the 32-bit multiplier defaults to unsigned.

For the purpose of analysing the average power consumed by the aforementioned multiplier block under varying operating conditions, seven corner cases have been devised, listed in Table 6.2. These provide best and worst case results, along with the special case of the clock signal being halted (as may occur in a clock gated implementation); they are intended to highlight the potential variance in average power depending on activity within the multiplier.

To account for the lack of an enable signal, and the lack of a signed multiplication mode, a slightly simpler analysis is used for the 32-bit multiplier compared with that used for the 64-bit multiplier. As a result, the tests *stalled*, *stalled (inputs toggling)* and *signed worst case*, as listed in Table 6.2, are omitted from the tests performed on the 32-bit multiplier.

A testbench is implemented to run each of the cases described in Table 6.2, using the flow described in chapter 3 to obtain average power figures for each case. The approach taken to allow any particular use case to be implemented with minimal effort, was to design the testbench to read a simple text stimulus file containing the input vectors to the multiplier, along with a method control signal. An excerpt of the testbench, showing the stimulus file reading and input vector applying loop, is listed in Figure 6.2, with a short sample stimulus file for worst-case switching with unsigned inputs shown in Figure 6.3. The enable and clock

Operating mode	Description
idle	inputs steady, clock running, enabled
stalled	inputs steady, clock running, disabled
stalled (inputs toggling)	all inputs toggling, clock running, disabled
clock disabled	inputs steady, clock halted, disabled
clock disabled (inputs toggling)	all inputs toggling, clock halted, disabled
signed worst case	all inputs toggling, signed mode
unsigned worst case	all inputs toggling, unsigned mode

Table 6.2: Multiplier operating mode corner cases

signals, which are present elsewhere in the testbench, are changed as part of the simulation script using the `sed` tool.

The results of running the cases described in Table 6.2 are listed in Table 6.3 for the 32-bit multiplier, and Table 6.4 for the 64-bit multiplier. These results are somewhat surprising; disabling the clock input to the 32-bit multiplier has very little effect on the power consumption, compared to an equivalent case with the clock enabled. This indicates that the internals of the multiplier continue to toggle in line with the inputs toggling, regardless of whether the clock signal is toggling. In the case of the 64-bit multiplier, disabling the clock input results in a reduction in power consumption of around 88% compared to worst-case toggling in signed mode with the clock running. However, this is still an increase of 600% compared to holding the inputs steady, regardless of whether the clock is disabled or not.

Operating condition	**Power (μW)**	
	130 nm technology	**90 nm technology**
idle	9.63	3.76
clock disabled	9.63	3.76
clock disabled (inputs toggling)	248.58	20.52
worst case	288.24	22.50

Table 6.3: 32-bit multiplier power usage under various operating conditions

Setting the enable signal low with the inputs toggling in the case of the 64-bit multiplier actually increases power consumption compared to worst-case toggling with the multiplier enabled. Clearly stalling the multiplier does not halt registers internal to the multiplier, rather it simply enables a feedback loop allowing the registers to continue storing the same values.

```
--initial configuration
n_wait_flag_i     <= '0';
left_i            <= (others => '0');
right_i           <= (others => '0');
wait for 105 ns;

--activate unit and cycle through input stimulus
n_wait_flag_i     <= '1';
wait for 10 ns;

while not endfile(stimulus_file) loop

  readline(stimulus_file, stimulus_line);
  if (stimulus_line(1) /= '#') then

    hread(stimulus_line, left_i_stim, read_check);
    assert read_check
      report "File read error reading left_i." severity error;

    hread(stimulus_line, right_i_stim, read_check);
    assert read_check
      report "File read error reading right_i." severity error;

    hread(stimulus_line, method_i_stim, read_check);
    assert read_check
      report "File read error reading method_i." severity error;

    -- method_i is being read as a hexadecimal value, but only
    -- lower two bits are used. Check upper two bits are zero.
    assert (method_i_stim(3 downto 2) = "00")
      report "Invalid input to method_i (value > 3)." severity error;

    left_i         <= left_i_stim;
    right_i        <= right_i_stim;
    method_i       <= method_i_stim(1 downto 0);

    wait for 10 ns;
  end if;

end loop;
```

Figure 6.2: Excerpt from multiplier testbench

```
# Stimulus file for ex_multiplier64_b_tb.vhd
#
# First input to left_i in hexadecimal format:
# width dependent upon op_width generic
# (default 32)
#
# Second input to right_i in hexadecimal format:
# width dependent upon op_width generic
# (default 32)
#
# Third input to method_i in hexadecimal format:
# Range 0-3, other values are invalid and will result
# in an error being generated from simulation.
#
# method_i values:
# 0 = Unsigned multiply, 64-bit result
# 1 = Signed Multiply, 64-bit result
# 2 = Unsigned multiply, 32-bit result
# 3 = Signed multiply, 32-bit result

00000000 00000000 0
FFFFFFFF FFFFFFFF 0
00000000 00000000 0
FFFFFFFF FFFFFFFF 0
00000000 00000000 0
FFFFFFFF FFFFFFFF 0
```

Figure 6.3: Example multiplier testbench stimulus file

Operating condition	**Power (μW)**	
	130 nm technology	**90 nm technology**
idle	13.60	5.37
stalled	13.60	5.37
stalled (inputs toggling)	1330.10	746.85
clock disabled	13.60	5.37
clock disabled (inputs toggling)	81.14	27.07
worst case (signed mode)	582.64	281.88
worst case (unsigned mode)	577.30	279.43

Table 6.4: 64-bit multiplier power usage under various operating conditions

The only approach that significantly reduces the average power dissipated by both multipliers is masking the input signals to prevent the input latches within the multipliers from toggling. Such an input mask can be implemented very easily using basic logic blocks like that shown in Figure 6.4, but the trade-off is increased logic area and increased active power consumption due to the additional gates that the input signal must pass through. A slight increase in signal delay will also result, although this will be minimal due to the simple gates used with a very short signal path and single fan-out.

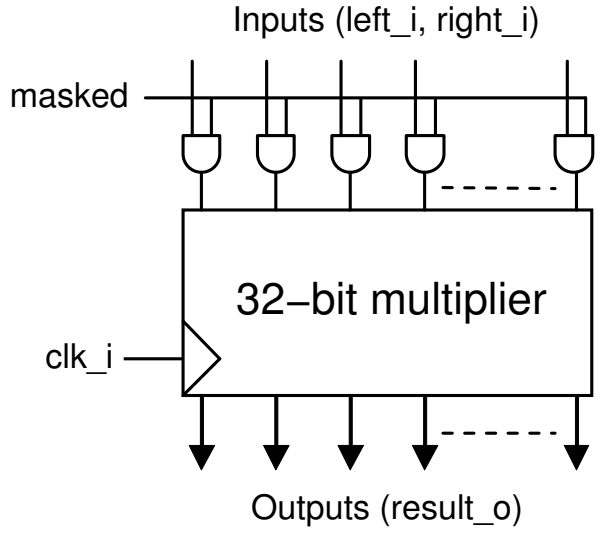


Figure 6.4: Multiplier input signal masking

To test the effect of such a mask, the 32-bit multiplier VHDL file is modified with the additional code shown in Figure 6.5. The inputs to the DesignWare pipelined multiplier are changed to `left_i_masked` and `right_i_masked`, and the `masked` signal is fed up to the top level testbench, allowing it to be easily enabled and disabled as desired.

```
mask_inputs: process (masked, left_i, right_i)
begin

   if (masked = '1') then
      left_i_masked  <= X"00000000";
      right_i_masked <= X"00000000";
   else
      left_i_masked  <= left_i;
      right_i_masked <= right_i;
   end if;
end process;
```

Figure 6.5: Multiplier input mask implementation

The results of this experiment are quite interesting. When the mask is active, the power consumption drops significantly even though all inputs are toggling on every cycle. On the other hand, when the mask is inactive, power consumption increases significantly compared to the previous worst case power consumption, due to the power consumed by the mask gates. The power consumption results are shown in Table 6.5.

Operating condition	**Power (μW)**	
	130 nm technology	**90 nm technology**
previous worst case	288.24	22.50
worst case (mask enabled)	12.20	5.01
worst case (mask disabled)	2048.90	260.66

Table 6.5: 32-bit multiplier power usage with input mask

Timing results were also considered after implementation of the input mask, to determine the effect on the maximum frequency at which the multiplier can operate. Critical path timing slack in 130 nm technology fell from 7.95 ns to 7.44 ns—a drop of just under 6.95%. For 90 nm process technology, timing slack drops from 8.38 ns to 8.28 ns—a reduction of just over 1.1%. Therefore the addition of an input mask has little effect on the timing performance of the multiplier unit using 90 nm technology, although depending on where the critical path is elsewhere in the coprocessor it may have an effect using 130 nm technology.

The potential optimisations considered in this section are simple modifications. Further consideration is given to optimising power and energy consumption of the multiplier unit in section 10.1.

The results derived from analysis of the multiplier units in this section will be incorporated into Cascade's energy analysis algorithm, to allow a more accurate approximation of the likely energy consumed by multiplier units used within a coprocessor.

6.2 Other functional units

The remaining functional units individually contribute a small proportion of the overall power and energy consumption of a typical coprocessor. Therefore a much simpler analysis method is applied to these units, giving a more coarse-grained calculation.

Initial observation of the energy consumption of functional units during execution of a typical application execution indicates that during any particular unit's "inactive" cycles, where the unit is not performing any useful work, a significant amount of dynamic energy is still being consumed due to switching related to control logic within the unit. Therefore, to take account of this dynamic energy consumption during inactive cycles, the typical inactive energy values for each functional unit can be characterised, allowing dynamic energy during inactive cycles to be included in overall energy calculations.

Inactive cycle energy can be determined for each functional unit by running the coprocessor for a number of cycles in a stalled state, meaning that all functional units will be inactive but not sleeping—that is, the control logic will still be operative. By monitoring switching activity during this period, and subsequently undertaking power analysis as described in section 3.4, the average energy per inactive cycle can be determined for all the functional units present in the coprocessor being analysed. Power Compiler reports average power consumption for each unit, therefore determining energy per inactive cycle is simply a case of multiplying the dynamic power (both switching and internal power) with the clock period.

Any memory blocks present within the functional units are excluded from the inactive cycle energy figures presented here, as the energy per cycle for memory blocks will be determined for each of their various states of operation in chapter 7. Due to the black-box nature of memory blocks, their energy will be calculated separately from the containing functional units by Cascade. Table 6.6 lists the key functional units used by Cascade, along with their dynamic energy consumption per inactive cycle for both 130 nm and 90 nm process technologies.

Functional unit	**Energy per inactive cycle (nJ)**	
	130 nm technology	**90 nm technology**
access_st_1	0.00342	0.000325
access_st_1r	0.00480	0.000456
access_1x	0.00495	0.000471
access_1	0.00479	0.000455
access_1r	0.00604	0.000574
access_2	0.00656	0.000623
access_assoc_1	0.00510	0.000484
access_assoc_1r	0.00653	0.000620
access_stream_1	0.00262	0.000249
access_stream_1r	0.00349	0.000332
access_stream_1x	0.00301	0.000286
access_stream_st_1	0.00035	0.000033
access_stream_st_1r	0.00095	0.000090
access_remap_1	0.00179	0.000170
arithmetic	0.00024	0.000120
bitshift	0.00334	0.000168
branch	0.00023	0.000114
combine	0.00027	0.000114
coreregfile	0.01750	0.005050
immediate32	0.00385	0.000754
immediate8	0.00322	0.000168
logical	0.00239	0.001280
predicate	0.00031	0.000192
registerfile	0.00475	0.000432
select	0.00749	0.000553
squash	0.00051	0.000012

Table 6.6: Energy per inactive cycle of functional units

Determining the dynamic power per active cycle for functional units is somewhat more complex than for inactive cycles, due to the variable nature of active cycle energy consumption. To ensure that the calculated values are representative of the actual functional unit energy consumption, each unit is analysed over a range of applications. The MediaBench suite is used for this analysis, as it is considered to be highly representative of typical applications targeted by Cascade; full details of the process of accelerating MediaBench applications with Cascade is covered in chapter 5. For each functional unit, the average energy per activation is calculated, alongside the variance from the average over all applications. Monitoring the variance (in the form of standard deviation) allows any units that are not consistent in the energy per activation to be flagged for more detailed analysis.

Performing such calculations over a large number of applications is time consuming and error prone, therefore a script has been created to automate the process. It reads the analysis summary generated for each application by Cascade, and matches that with the appropriate hierarchical power report generated by Power Compiler. The figures for all functional units present in each test are parsed, and after subtracting energy attributable to inactive cycles, the average energy per active cycle is determined. This script is listed in Appendix D.3.

Table 6.7 lists the units analysed using this technique, along with their energy per active cycle for 130 nm and 90 nm technologies. The values listed in Table 6.6 and Table 6.7 are stored in an XML file accessible to Cascade, which uses them, along with details of the active and inactive cycle counts for each unit, to estimate the energy used by the functional units.

Functional unit	**Energy (nJ)**	
	130 nm technology	**90 nm technology**
access_st_1	0.0453	0.0191
access_st_1r	0.0635	0.0303
access_1x	0.0656	0.0313
access_1	0.0633	0.0302
access_1r	0.0799	0.0382
access_2	0.0868	0.0415
access_assoc_1	0.0675	0.0322
access_assoc_1r	0.0864	0.0413
access_stream_1	0.0347	0.0166
access_stream_1r	0.0462	0.0221
access_stream_1x	0.0399	0.0190
access_stream_st_1	0.0046	0.0022
access_stream_st_1r	0.0125	0.0060
access_remap_1	0.0236	0.0120
arithmetic	0.0755	0.0321
bitshift	0.0346	0.0592
branch	0.0040	0.0013
combine	0.0448	0.0657
coreregfile	0.0226	0.0108
immediate32	0.0891	0.0186
immediate8	0.0997	0.0310
logical	0.0778	0.0151
predicate	0.0243	0.0386
registerfile	0.0296	0.0141
select	0.0832	0.0362
squash	0.0077	0.0118

Table 6.7: Energy per active cycle of functional units

6.3 Output banks

The output banks that form a part of many of the functional units available to Cascade play a significant role in the power and energy consumption of those units. For most of the smaller functional units, the output banks dominate the unit's energy consumption, which is why these banks are being considered separately from the units themselves.

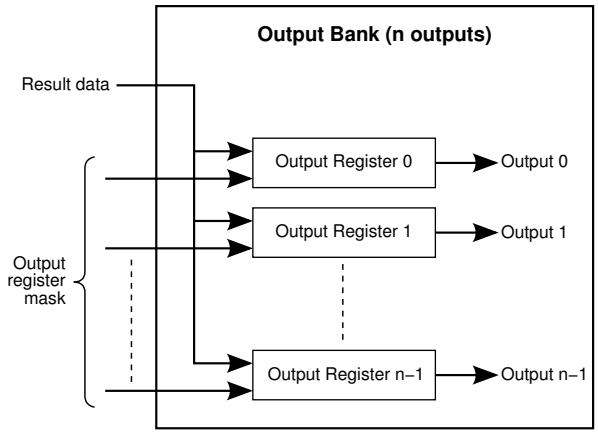


Figure 6.6: Output bank layout

A typical output bank is shown in Figure 6.6, consisting of a simple register array, usually 32 registers wide, used to store a single output value from its associated functional unit. The depth of each output bank can be configured depending on how many output values need to be stored simultaneously. A larger number of registers in the array can increase the utilisation efficiency of the associated functional unit, because a result can be calculated several cycles before it is required and the result stored in the output bank. The trade-off is increased area (which can be substantial when applied across all functional units) and increased power/energy consumption.

In addition to the registers, the output bank contains an output mask. This is effectively an enable signal that controls whether the value held in a particular line of registers is masked or propagated through the output to the other units to which it is connected. This allows results that are stored but not required until a future cycle to be masked from reaching the output.

As an array of registers, the energy consumption of output banks is likely to show a large variance depending on the data patterns being stored and the number of times the value stored changes, along with the behaviour of the output mask pattern. To determine the worst- and best-case power and energy values, the script in Appendix D.1, along with the testbench

in Appendix D.2, is used to generate switching activity information under a range of input stimulus conditions.

Unsurprisingly the lowest power and energy are consumed when the input values are all zero, as is the output mask. An excerpt of the stimulus file used to simulate this case is shown below in Figure 6.7.

```
# Stimulus file for gl_output_bank_tb.vhd
#
# First input to data_i in hexadecimal format:
# width dependent upon out_bank_register_width generic
# (default 32)
#
# Second input to out_reg_mask_i in hexadecimal format:
# width dependent upon out_bank_registers generic
# (default 16)

00000000 0000
00000000 0000
00000000 0000
00000000 0000
00000000 0000
00000000 0000
00000000 0000
00000000 0000
```

Figure 6.7: Output bank best case stimulus file

Finding the worst-case power and energy consumption requires some extra work, as it is not immediately obvious what input patterns will create the required conditions. Using the aforementioned script allows automated simulation of a large number of input stimulus files, the switching activity results of which can then be analysed from within Power Compiler to determine the worst-case average power figure. The analysis shows that this occurs when the input value is held high (all logic '1'), and all bits of the output mask are toggled between on and off during alternate cycles. An excerpt of the stimulus file that simulates these conditions is shown in Figure 6.8.

It is important to realise that the actual effect of the output mask will be highly variable dependent upon the connectivity of the outputs. A register array connected to several other units, particularly if those units are located a significant distance from the array, will have a much larger output capacitance than a register array connected to a single nearby unit.

```
# Stimulus file for gl_output_bank_tb.vhd
#
# First input to data_i in hexadecimal format:
# width dependent upon out_bank_register_width generic
# (default 32)
#
# Second input to out_reg_mask_i in hexadecimal format:
# width dependent upon out_bank_registers generic
# (default 16)

FFFFFFFF FFFF
FFFFFFFF 0000
FFFFFFFF FFFF
FFFFFFFF 0000
FFFFFFFF FFFF
FFFFFFFF 0000
FFFFFFFF FFFF
FFFFFFFF 0000
```

Figure 6.8: Output bank worst case stimulus file

For this reason, getting accurate estimates of the power and energy consumed by the output banks within a coprocessor requires a more detailed knowledge of the connectivity of the coprocessor, in addition to the switching activity generally required for coprocessor energy analysis.

Unfortunately, the connectivity cost (in terms of energy consumption) cannot be accurately determined early in the design process when power and energy analysis is being carried out on coprocessor candidates, due to a lack of accurate load capacitance information. Therefore a simplified model is required, that considers the effect of output banks on the energy consumption of functional units, while also allowing analysis to be performed quickly at an early stage of coprocessor candidate generation. This simplified model makes assumptions about the likely connectivity of output banks, which are characterised in an "average case" value, and subsequently used for the aforementioned analysis. Trial runs have suggested that in most cases the average case values provide an acceptable level of accuracy, but in in some cases where the post-layout connectivity costs are unusually high, then the average case values may prove to be somewhat optimistic. Without a much more complex and therefore slower model, this possibility cannot be avoided.

In most cases, the output banks will be configured to be 32 bits wide, with each unit potentially having a different depth dependent on connectivity and utilisation requirements as determined by Cascade during coprocessor synthesis. It is also possible to have output banks of 8 and 16 bits wide, again with a range of depths, although these are less common.

Analysis of the output banks is split into three groups based on the bank width: 8, 16 and 32 bits. Each group is then analysed using best case, average case, and worst case operating modes, as described previously. Within each mode, bank depths of 2, 4, 8 and 16 banks are analysed for both TSMC 130 nm and TSMC 90 nm process technologies. The results are listed in Tables 6.8, 6.9 and 6.10. Although the bank depths can be configured in sizes other than those analysed, it can be seen from the results that energy consumption is close to linearly proportional to bank depth, therefore other values can be easily interpolated from the results listed.

Operating mode	**Bank depth**	**Energy (nJ)**	
		130 nm technology	**90 nm technology**
Best case	2	0.4063	0.1914
	4	0.8127	0.3828
	8	1.6254	0.7656
	16	3.2508	1.5313
Average case	2	0.8225	0.4441
	4	1.3417	0.7133
	8	2.3963	1.2523
	16	4.4713	2.3306
Worst case	2	0.9345	0.5193
	4	1.8695	1.0390
	8	3.7401	2.0786
	16	7.4799	4.1575

Table 6.8: Output bank energy per cycle (8-bit width)

Further analysis of the energy consumption pattern of output banks reveals that they continue to consume significant amounts of energy during the cycles where the parent functional unit is considered to be inactive; that is, not performing any useful computation. During active cycles, the execution unit within the functional unit tends to be the dominant consumer of energy. Therefore it is the inactive cycles that are of particular interest with regard to output banks, as the number and configuration of banks will have a significant effect on the inactive cycle energy for functional unit being considered.

Operating mode	**Bank depth**	**Energy (nJ)**	
		130 nm technology	**90 nm technology**
Best case	2	0.8127	0.3828
	4	1.6254	0.7656
	8	3.2508	1.5313
	16	6.5016	3.0627
Average case	2	1.6456	0.8886
	4	2.6841	1.4269
	8	4.7932	2.5049
	16	8.9444	4.6614
Worst case	2	1.9017	0.7514
	4	3.8044	1.5018
	8	7.6108	3.0027
	16	15.2220	6.0046

Table 6.9: Output bank energy per cycle (16-bit width)

Operating mode	**Bank depth**	**Energy (nJ)**	
		130 nm technology	**90 nm technology**
Best case	2	1.6254	0.7656
	4	3.2508	1.5313
	8	6.5016	3.0627
	16	13.0030	6.1254
Average case	2	3.2554	1.7774
	4	5.3332	2.8539
	8	9.5485	5.0100
	16	17.8510	9.3230
Worst case	2	3.0097	1.8390
	4	6.7426	3.7583
	8	14.2100	7.5973
	16	29.1430	15.2750

Table 6.10: Output bank energy per cycle (32-bit width)

To facilitate the calculation of inactive cycle energy for functional units that may have a range of output bank configurations, each functional unit is split into several entries in Cascade's energy analysis algorithm, with a separate entry for each output bank present within the functional unit. For example, the first `coreregfile` unit, containing four output banks, would be represented as:

```
coreregfile_0/0
coreregfile_0/1
coreregfile_0/2
coreregfile_0/3
```

In doing this, the inactive cycle energy is automatically multiplied up depending on the number of output banks present within the execution unit, thus improving analysis accuracy while minimising the increase in computational complexity. Although the accuracy remains quite variable due to the large potential variance in capacitive loading on output banks (which can only be reliably determined at a much later stage of the design), this improvement provides an acceptable trade-off between that accuracy, and the much quicker early-stage analysis capability provided here. This analysis can be carried out on a large number of coprocessor candidates before selecting the preferred candidate to synthesise. The relatively small contribution of standard functional units, compared with multiplier units and memory blocks, means that the comparatively lower accuracy of these units does not present a problem to the overall accuracy of early stage analysis.

6.4 Summary

In this chapter, models were created for the various functional units used within Cascade, taking into account the energy used during both active and inactive cycles. These will be used within the unified power analysis model to be integrated into Cascade, allowing automated estimations of the power and energy consumption of coprocessors to be generated early in the design process.

Particular focus was placed on those units that consume a large proportion of the overall coprocessor energy—specifically multiplier units—ensuring that the most effort is expended in the areas that will show the greatest effect on improving overall accuracy, while maintaining a high speed analysis that can be used on a large number of coprocessor candidates during

early design space exploration. This involved implementing a more detailed model based on operation modes for units with higher energy consumption and variability.

In addition to the functional units themselves, models were created for the output banks found within most functional units. As these banks were found to account for a large proportion of the energy variance between instantiations of a particular functional unit, modelling the output banks independently enables a significant improvement in overall accuracy with minimal increase in computational complexity.

Although the work on functional units in this chapter is specific to the particular type of configurable processor implemented by Cascade, the partitioning and analysis techniques developed offer a base from which models could be developed for other types of modular configurable processors. Similarly, the method of allocating analysis resources dependent upon the importance of each module within the processor could bring similar benefits to the performance and accuracy trade-off for other configurable processor designs.

7. Characterising memories and register files

Cache memories and register files are often the most significant consumers of power and energy in coprocessors generated by Cascade, as seen in the results listed in chapter 3. Combined with the nature of generated memories, these components require special consideration during power and energy analysis to ensure maximum possible accuracy.

Due to the large size and density of on-chip caches, it is important to ensure a high level of efficiency in the design of such blocks to minimise area and energy requirements. Implementing large memories as inferred register arrays using standard HDL code causes the synthesis tool to produce very inefficient hardware, both in terms of area and energy usage. As a result, most memory blocks used by Cascade, with the exception of small register files, are created by a memory generator that produces optimised memories targeted to the physical process technology that the design will be synthesised on. Such memories, implemented as hard macro blocks, are much more efficient than inferred memories built during the compile stage of synthesis.

In this chapter, hard macro blocks and register files are analysed. Both are physically very similar, with the main difference being that macro blocks are much bigger than register files, and are used as the main storage within the coprocessor in the form of data and instruction caches. Register files are smaller local storage units designed for data that needs to be held live for several clock cycles. An analysis framework for each type of memory is developed in the following sections, allowing automated analysis of memories within Cascade coprocessors to the highest level of accuracy possible with the visibility offered by the memory blocks.

7.1 Hard macro memory blocks

In all cases the memory blocks used by Cascade are created by Artisan Memory Generator [70], with different versions for TSMC 180 nm, TSMC 130 nm and TSMC 90 nm process technologies. The Artisan-generated memory blocks used by Cascade to create the caches and register files used in coprocessors is listed in Table 7.1. Smaller register files are generated at synthesis time using DesignWare IP; those are considered in section 7.2.

One issue raised by the use of a memory generator is that the memories are hard macro blocks, therefore synthesis and power analysis tools cannot see the internal structure of the block to perform analysis. As a result, a simple look-up table based on the state of the memory block in each cycle, stored within the `.lib` or `.db` file representing the memory block, is the only power and energy information available for generated memory blocks.

Tag ram (single port RW): 16 depth, 30 width
Register file (dual port RO + WO) 32-bit width: 16, 32, 64, 128
Register file (single port RW): 16 depth, 48 width 64 depth, 32 width 512 depth, 8 width
Data cache (dual port 2xRW) 32-bit width: depths: 512, 1024, 2048, 4096, 8192
Data cache (single port RW) 32-bit width: depths: 512, 1024, 2048, 4096, 8192, 16382
Instruction cache (single port RW): widths: 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128 depths: 256, 512, 1024, 2048, 4096, 8192, 16384

Table 7.1: Artisan memory blocks used by Cascade

Artisan Memory Generator can create a plain text data file with details of all relevant values for each generated memory block—the same values that are available to Power Compiler within the `.db` files. An example of such a data file for one memory block is shown in

Figure 7.1. The data file includes current drawn at a specific frequency and voltage under various states of operation, such as read, write, deselected and standby—these are listed in the data file as `icc_r`, `icc_w`, `icc_desel` and `icc_standby` respectively.

Since these are the most detailed energy figures available for memory macro blocks, it is clearly desirable to use them within Cascade for calculations relating to such blocks. The approach taken is to parse the data file for all memory blocks that can be used by Cascade, extracting the desired data relating to energy use (specifically current, with the related frequency and voltage values), and placing it into a look-up table for later reference by Cascade. The parsing is done by the script listed in Appendix E.1, which outputs a comma separated values (CSV) file to be referenced as required during later analysis. A small section of the memory library output file is shown below:

```
#, Look-up table for memory energy values (dual-port memories)
#, type,words,bits,volt,freq,icc_rw_a,icc_rw_b,
#, icc_desel_a,icc_desel_b,icc_standby
#,
dp,128,32,1.20,200.000,2.151,1.974,0.650,0.650,0.025
dp,16,32,1.20,200.000,1.553,0.892,0.340,0.340,0.007
dp,32,32,1.20,200.000,1.638,1.047,0.384,0.384,0.009
dp,64,32,1.20,200.000,1.809,1.356,0.473,0.473,0.015
```

The memory library in CSV format effectively provides a look-up table used by other applications to fetch the required information related to each memory based on usage statistics. A shell script has been written to fetch data from the CSV file on a manual basis, although a more useful Java class has been written that can be integrated with other applications, allowing parsing of the data file directly. This Java class is listed in Appendix E.2.

Although initially it was intended that Cascade would reference the generated CSV files directly for the purpose of calculating memory energy usage, it was later decided to integrate the information stored in the CSV file into the `technology.xml` files used by Cascade for storing the per-cycle energy values for other functional units. A different `technology.xml` file is used for each process technology, making it a simple process to transfer the data from the CSV file for each process technology to the corresponding XML file.

Once the XML files have been annotated, calculating the energy used by a memory block is very similar to the process used for functional units. The current implementation considers only active and inactive cycles for the memories, as Cascade coprocessors do not activate

```
name    fast,1.32,-40.0 typical,1.20,25.0 slow,1.08,125.0
S    N    N    N
# sp_rw_s_instrmax1024 words=1024 bits=128 mux=8 drive=6
# pipeline=No frequency=200.000 ring width=6
geomx       1549.295    1549.295    1549.295
geomy         334.570     334.570     334.570
ring_size      13.200      13.200      13.200
icc            34.150      29.399      24.202
icc_r          31.211      26.726      21.852
icc_w          37.089      32.071      26.551
icc_peak      616.400     354.700     188.600
icc_desel       4.180       3.607       3.308
icc_standby     0.028       0.033       0.189
tcyc            1.012       1.509       2.361
ta              0.330       1.309       2.123
tas             0.354       0.526       0.795
tah             0.000       0.000       0.000
tcs             0.471       0.701       1.195
tch             0.000       0.000       0.000
tws             0.172       0.244       0.366
twh             0.000       0.000       0.000
tds             0.054       0.096       0.186
tdh             0.048       0.069       0.099
tckh            0.055       0.083       0.140
tckl            0.339       0.525       0.888
tckr            4.000       4.000       4.000
load_q          0.342       0.475       0.696
icap_a          0.022       0.021       0.020
icap_d          0.001       0.001       0.001
icap_clk        0.090       0.086       0.083
icap_cen        0.004       0.004       0.004
icap_wen        0.007       0.007       0.007
pwn_ck         10.000      10.000      10.000
vn_ck           0.450       0.447       0.430
vn_pwr          0.132       0.120       0.108
vn_gnd          0.132       0.120       0.108
```

Figure 7.1: Example Artisan memory data file

the deselected state while the coprocessor is running. Standby current is independent of, and in addition to, dynamic current consumed during active and inactive cycles, and is therefore also included in the XML file under the label of leakage energy. It will later be used as a component in calculations of coprocessor leakage energy—see chapter 9.

During the functional simulation of a coprocessor candidate, Cascade keeps track of the number of accesses made to each memory block present within the coprocessor, and multiplies that access count with the corresponding active cycle energy value stored in the `technology.xml` file. Inactive cycle energy is calculated using the same method, and both active and inactive energy values are added together to get the total energy used by that memory block over the simulation run.

This approach provides a quick means of estimating the energy usage of memory blocks within coprocessor candidates, allowing a large number of candidates to be considered early in the design space exploration. It also allows most of the accuracy available for energy calculations within black-box memory blocks to be retained, therefore keeping the results close to those that would result from a much longer power analysis flow using Power Compiler or a similar RTL or gate-level analysis tool.

Verification of the memory block energy utilisation calculation was carried out, using both the results determined using the look-up table described in this section, as well as a complete synthesis and gate-level analysis using Power Compiler. In all cases there are minor differences between the two sets of results (typically less than 5%), which does not present a concern due to the several orders of magnitude speed-up offered by using the look-up table method. It is not possible to examine the source of the discrepancy in the results, as Power Compiler's exact method of calculating power consumption for black box units is unknown.

7.2 Register files and tag RAM

Larger register files used by Cascade are created using the Artisan Memory Generator as described for cache memories (the register files generated this way are listed in Table 7.1), therefore the analysis approach taken is as described in the previous section. Smaller registers are created using Synopsys DesignWare IP, which are inferred from standard cells at the synthesis stage. Three different types of IP blocks, listed in Table 7.2, are used depending on

the requirements of each implementation. Further details on these IP blocks can be found in the DesignWare IP Family Reference [79].

In addition to register files, a small tag RAM is used within the coprocessor to store information on data being held in the cache, and to facilitate fast look-up of cached data lines—a feature common to many modern processors [80]. The tag RAM is 27 bits wide, and has one synchronous write port and two asynchronous read ports, therefore it is built using the `DW_ram_2r_w_s_dff` DesignWare IP block. It can be implemented in depths of 8, 16, 32, 64, 128 or 256 bits, depending on requirements with regards to the coprocessor data cache. There is one exception to this: the single-port static cache memory `fu_access_st_1` uses a single port tag RAM, built using the built using the `DW_ram_rw_s_dff` DesignWare IP block, with the same range of depths as the other tag RAM blocks.

IP block	Memory configuration
`DW_ram_r_w_s_dff`	Synchronous Write-Port, Asynchronous Read-Port
`DW_ram_2r_w_s_dff`	Synchronous Write Port, Asynchronous Dual Read Port
`DW_ram_rw_s_dff`	Synchronous Single Port Read/Write

Table 7.2: DesignWare IP memory blocks used by Cascade

Inferred memories have a higher level of visibility to analysis tools throughout the design process compared with hard macro blocks, allowing a more comprehensive analysis to be performed if detailed activity statistics are available. The trade-off with more detailed analysis is much higher computation time in both collection of activity statistics and subsequent calculation of power and energy consumption. In light of this, and taking into account the much lower comparative energy of inferred register files compared with larger hard macro blocks, it was decided that the best approach is to apply a similar state-based analysis as that used for hard macro memories.

Figure 7.2 shows an excerpt from the testbench used with DesignWare memory IP blocks (in this example, the `DW_ram_2r_w_s_dff` block) to determine power and energy consumption under similar conditions to those listed for the Artisan generated memories. The data sheet for Artisan memory blocks [70] states that energy values for read and write cycles are determined by switching all data pins, and half the address pins, on each cycle. Therefore the testbench for DesignWare memories replicates that behaviour. Elsewhere in the testbench, the `we_i` signal can be controlled to place the memory block in read mode or write mode, allowing the power and energy values to be determined for these modes.

```
stimulus: process (clk_i)
variable i : integer := 0;
begin

   if falling_edge(clk_i) then

      i := 0;
      --invert all data inputs on each cycle
      data_i <= not data_i;

      --invert 50% of address bits on each cycle
      --by looping through and inverting all even numbered bits
      while (i < address_bits) loop

         addr_i(i)   <= not addr_i(i);
         addr_b_i(i) <= not addr_b_i(i);

         i := i + 2;

      end loop;

   end if;

end process;
```

Figure 7.2: Excerpt from DesignWare SRAM IP testbench

The power consumption values determined for the `DW_ram_2r_w_s_dff` tag RAM unit at all the allowable depths, under each operating condition, are listed in Table 7.3. Note that the idle case produces very similar power consumption to read power, this is because the DesignWare memories do not have an explicit enable signal. There is an active-low chip select input, however this does not have any significant effect on the power consumption during inactive cycles, as the memory block still presents the contents of the current address at the outputs. Holding chip select high simply prevents the write enable signal from placing the memory into write mode, therefore it is redundant in this use as the write enable input is not shared between memories within Cascade coprocessors.

The tag RAM power values listed in Table 7.3 were determined with a clock frequency of 10 MHz. The corresponding energy per cycle values can easily be determined by multiplying the power consumption by the clock period (equivalent to dividing by the clock frequency). For example, the energy consumed by the 256-bit deep tag RAM during a read cycle is 0.1487 nJ.

Memory depth	**Operating mode**	**Power (μW)**	
		130 nm technology	**90 nm technology**
8 bits	idle	42.072	19.822
	read	46.156	21.746
	write	57.786	27.226
16 bits	idle	84.230	39.684
	read	91.456	43.089
	write	105.352	49.636
32 bits	idle	168.567	79.420
	read	182.808	86.129
	write	201.095	94.746
64 bits	idle	337.352	158.943
	read	367.645	173.215
	write	386.046	181.885
128 bits	idle	675.099	318.072
	read	741.966	349.576
	write	771.852	363.657
256 bits	idle	1351.300	636.663
	read	1487.000	700.598
	write	1663.500	783.756

Table 7.3: Tag RAM power consumption (100 MHz operation)

As with the other memory blocks and register files, the energy per cycle values for each operation mode are initially annotated into a CSV file, one for each of the two target process technologies—TSMC 130 nm and TSMC 90 nm. These files are listed in Appendices E.3 and E.4 respectively. The values from the CSV file are then transferred to an XML file, to be read by Cascade during coprocessor candidate energy analysis. The relevant values are then multiplied by the cycle counts for each operation mode to determine the approximate energy consumed by the tag RAM.

7.3 Summary

The work undertaken in this chapter provides an analysis framework for calculating the energy used by memory blocks and register files within Cascade generated coprocessors. Black-box macro memory blocks have limited visibility, meaning that the level of detail available for power and energy analysis is limited to that provided by the vendor-supplied data sheets. Therefore a memory energy model for these blocks was created based on the data provided in the data sheets. A shell script was created to extract and process the information for each memory block used by Cascade, placing the results into several CSV files—one for each target process technology.

For memory units created using DesignWare IP, full visibility is available for power and energy analysis, therefore a conventional analysis was carried out for these blocks using Power Compiler. For all DesignWare IP memories used by Cascade, each operating mode was analysed, and the results annotated into the same CSV file as that used for the macro memory blocks.

Shell scripts and a Java class have been written to allow automated analysis to take place, based on the characteristics and access patterns of a particular memory block or register file, combined with the previously created CSV file.

Finally, the information contained in the CSV files was transferred to an XML file to be referenced by Cascade, allowing memory energy analysis to be performed automatically as part of early stage coprocessor design space exploration.

Most configurable processors contain memory blocks and register files, composed of hard macro blocks and/or synthesised memories. Therefore the work carried out in this chapter is applicable to configurable processors in the generic sense, allowing rapid development of models representing the memory components of such processors.

8. Clock tree power and clock gating

Power and energy dissipated in the clock tree contributes a significant proportion to the overall power and energy consumed by modern microprocessors and embedded devices. In this chapter, a detailed examination of the energy consumed in the clock tree of several configurations of Cascade coprocessors is undertaken, and an analysis of clock tree gating, a technique to reduce energy consumption in both the clock network and logic blocks, is undertaken.

8.1 Clock tree power

The power consumed by the clock tree in a modern SoC device is often a substantial proportion of the total interconnect power consumption. This is a result of the high switching frequency (usually the highest frequency interconnect on the chip), and high capacitance due to fanout [81]. As the clock tree is such a specialised net, it is typically synthesised as part of the back-end flow using optimised clock tree synthesis algorithms. Although it is possible to obtain estimates of the clock tree power after RTL synthesis (using, for example, Power Compiler), the accuracy of such an analysis will be too poor to be useful, and as such it will typically be disabled by default. If desired, Power Compiler's clock tree power analysis result can be shown by adding the flag `-include_input_nets` to the `report_power` command.

8.1.1 SPICE and Nanosim analysis

There are several ways to obtain a more accurate analysis of the clock tree power. One is to extract the clock tree in SPICE format after it has been synthesised, which can later be analysed in a SPICE simulator such as Synopsys' HSPICE. This approach allows for a slow but highly accurate analysis of the clock tree timing and power performance.

An alternative that provides a similar level of accuracy, but an order of magnitude higher speed, is Synopsys' Nanosim. A standard Verilog netlist can be used with Nanosim, along with SPICE models. Nanosim greatly simplifies the process of performing transistor level timing and power analysis, driven through a graphical interface, interactive command line, or a Tcl-based script input.

The problem with both SPICE and Nanosim methods is that they require SPICE models for the process technology being used—in the case of this project, TSMC 130 nm and 90 nm models. Such libraries are not supplied by TSMC as part of their standard cell library package, therefore they are unavailable for this project. For this reason it is not possible to undertake a detailed clock tree power analysis using either of the aforementioned tools while targeting TSMC process technologies.

UMC libraries do come supplied with SPICE models, allowing clock tree power analysis to be undertaken using HSPICE or Nanosim. However, as the power figures determined using UMC libraries are not comparable to those that will be observed when using TSMC libraries, such an analysis would not be beneficial to creating a clock tree power model for commercial coprocessors generated by Cascade.

8.1.2 Design Compiler topographical mode analysis

Toward the end of the project period, Synopsys added a feature to Design Compiler known as *topographical mode*, beginning with version Y-2006.06. The version used in this section is Z-2007.03 [82]. Topographical mode integrates into Design Compiler some of the technology from Physical Compiler, enabling fast automated physical synthesis as part of logic synthesis. The information derived from physical synthesis is used to calculate estimates of the resistance and capacitance of wires within the design, superseding inaccurate wire-load

models. While not as accurate as the results that could be obtained by SPICE or Nanosim analysis, topographical mode within Design Compiler requires only the addition of a physical technology library to the existing flow, which is supplied by TSMC. The required inputs and outputs for topographical mode are shown in Figure 8.1.

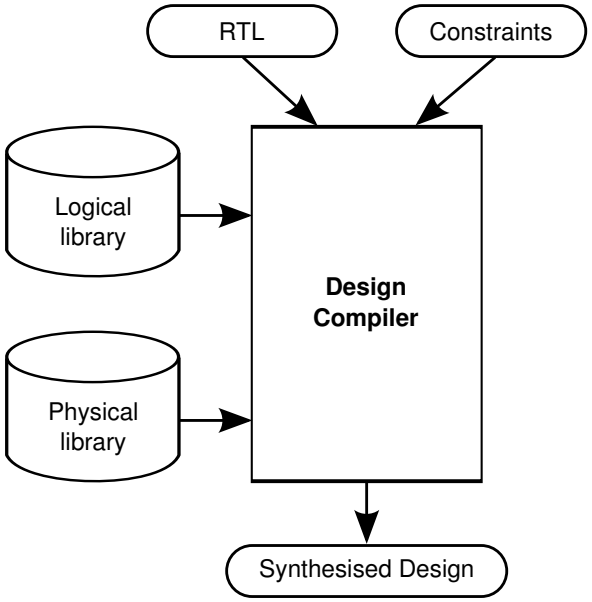


Figure 8.1: Design Compiler topographical mode inputs and outputs [82]

Some minor modifications are required to the existing Synopsys flow scripts listed in section 3.2. After testing whether the shell is in topographical mode, either the standard wire-load model is defined, or the physical technology library is declared using the commands below.

```
if {![shell_is_in_topographical_mode]} {
  set_wire_load_model -name "tsmc090_wl10" -library "typical"
} else {
  set use_pdb_lib_format true
  set physical_library "../libraries/tsmc90/tsmc090nvtlk_9lm_2thick.pdb"
}
```

Similarly, if the shell is in standard mode then the standard `compile` command is used. In topographical mode, power prediction must be explicitly enabled, and the `compile_ultra` command is used instead of `compile`, which is not supported in topographical mode. The commands used are shown below.

```
if {![shell_is_in_topographical_mode]} {
  compile
} else {
  set_power_prediction
  compile_ultra
}
```

Running a clock tree power analysis on the same design using both standard (wireload model) mode, and topographical mode, highlights the difference in results generated by each mode. The `pgp_decode` MediaBench test, taken from chapter 5, is used in this example. Using standard mode in Design Compiler with wireload model `tsmc090_wl10` (the least pessimistic model available in the library), Power Compiler reports a clock tree power of 10.1 mW at a clock frequency of 10 MHz—clearly a crude estimate based on fan-out. The fan-out value itself is artificially limited to 1000 by default, as the clock tree is classed as a "high fan-out net"; actual clock tree fan-out is reported as 12,320 loads.

By comparison, using topographical mode allows Design Compiler to perform a crude physical layout, and in doing so it can make better estimates of wire lengths and sizes. Thus all interconnect power estimates, including those for the clock tree, are better correlated to the results that are likely to be seen in the final physical design. For the aforementioned example, the clock tree power is estimated as 6.319 mW, with the overall total dynamic power of the design estimated as 8.7997 mW. Although the clock tree power comprises a large proportion of the overall power, this is based on a non-optimised clock tree. Normally a full clock tree synthesis will be undertaken in a dedicated physical synthesis tool, such as Synopsys JupiterXT or Cadence Encounter (as described in chapter 11), however for our comparative purposes the use of a non-optimised clock layout is sufficient.

One major drawback to using topographical mode for power analysis is run time. In the analyses undertaken in this section, topographical mode typically increases the run time of a synthesis and power analysis by a factor of 10–20×. For example, the `jpeg_decode` MediaBench test takes over 20 hours of CPU time on a 2.0 GHz Intel Xeon Linux system with 4 Gb RAM. A detailed examination revealed that this delay is not due to topographical mode itself, it is due to the use of the `compile_ultra` command, which creates a highly optimised synthesis mapping. Even in wire-load mode, `compile_ultra` increases the run-time for most tests by an order of magnitude or more, compared with the standard `compile` command. However, topographical mode requires the use of `compile_ultra`, therefore there appears to be no option to improve the run-time of synthesis and power analysis using topo-

graphical mode. As a result, topographical mode is used sparingly, primarily for the analysis of clock tree power in this section.

A comparison of the area and clock tree power estimates generated by Design Compiler using both wire load mode and topographical mode was undertaken using coprocessors generated from a selection of test cases from Cascade's test suite, running at 10 MHz. The results are listed in Table 8.1; clearly the estimate of clock tree power in wire load mode has been artificially clamped to 10.1 mW in all tests, due to the aforementioned default limit of 1000 nets for high fan-out nets such as the clock network. It is notable that topographical mode produces lower area estimates in all cases tested here, indicating that wire load mode is overestimating the area. The margin is relatively small, with the overestimates in the range of 3.5–8.7% for these cases.

Test	**Topographical mode**		**Wire load mode**	
	Area (μm)	**Clock power (mW)**	**Area (μm)**	**Clock power (mW)**
adpcm_encode	825360	6.480	868949	10.1
dct	990126	6.842	1045513	10.1
g721_decode	789697	5.539	844396	10.1
g721_encode	617896	5.166	665004	10.1
gsm_decode	646700	5.703	686200	10.1
jpeg_decode	1051382	6.609	1119008	10.1
pgp_decode	875195	6.319	931407	10.1
st_modified	1360079	6.869	1407133	10.1
test_nop	539615	5.691	586773	10.1
video_blur	1200872	6.613	1252118	10.1

Table 8.1: Area and clock tree power using topographical and wire load models

Of prime interest is establishing whether there is a useful deterministic relationship between area and clock tree power that will allow crude clock tree power estimations to be made at an early stage in Cascade's coprocessor flow. Area is much easier to estimate accurately without performing complex analysis, and as such area estimates have long been implemented as an early stage design space exploration feature of Cascade. To facilitate examination of the relationship, the results from the area and clock tree power analysis from Table 8.1 are plotted on the scatter graph shown in Figure 8.2.

As the relationship appears to have a linear element, linear regression analysis is applied to determine the coefficients of the linear equation, and the correlation coefficient, indicating

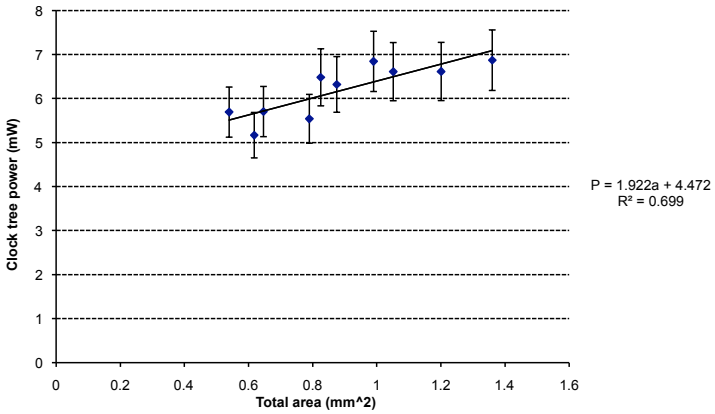


Figure 8.2: Clock tree power as a function of total area

how good a fit the approximation is. Linear regression analysis is applied using the following equations, where m is the slope, b is the intercept, a is area, and P is clock tree power:

$$m = \frac{n\sum(aP) - \sum a \sum P}{n\sum(a^2) - (\sum a)^2} \tag{8.1}$$

$$b = \frac{\sum P - m\sum a}{n} \tag{8.2}$$

Additionally, the correlation coefficient can be calculated as follows:

$$r = \frac{n\sum(aP) - \sum a \sum P}{\sqrt{[n\sum(a^2) - (\sum a)^2][n\sum(P^2) - (\sum P)^2]}} \tag{8.3}$$

These values can be calculated automatically using a spreadsheet such as Excel, and the results show that the slope has a value of 1.992 with the intercept 4.472. These figures give a squared confidence interval of 0.699—far short of the ideal value of 1 indicating significant variance between the trend line and the actual results.

Based on the linear regression approximation, the relationship between total area and clock tree power can thus be estimated as:

$$P_{clk} = 4.442 + 1.922 \times A_{total} \tag{8.4}$$

Where P_{clk} is the clock tree power, and A_{total} is the total coprocessor area in mm^2.

To test the accuracy of this relationship, estimates of the clock tree power are calculated using Equation 8.4 and the results compared against those generated by Power Compiler in topographical mode, as listed in Table 8.1. The results of this comparison, including the percentage error for each test, are listed in Table 8.2.

Test	**Area (μm)**	**Clock power (mW)**	**Clk power est. (mW)**	**% error**
adpcm_encode	825360	6.480	6.058	-6.51
dct	990126	6.842	6.375	-6.83
g721_decode	789697	5.539	5.990	8.14
g721_encode	617896	5.166	5.660	9.55
gsm_decode	646700	5.703	5.715	-0.21
jpeg_decode	1051382	6.609	6.493	-1.76
pgp_decode	875195	6.319	6.154	-2.61
st_modified	1360079	6.869	7.086	3.16
test_nop	539615	5.691	5.509	-3.20
video_blur	1200872	6.613	6.780	2.53

Table 8.2: Clock tree power estimate calculated from total area

The results from this analysis show that the worst-case accuracy is within $\pm$10% of the figure calculated by Power Compiler in topographical mode. This is accurate enough to be useful for early-stage analysis (such as during design space exploration), particularly for comparative analysis between coprocessor implementations, although it must be noted that the low confidence interval indicates that this relationship cannot be relied upon too heavily. The error has a standard deviation of 5.60%.

Further examination of the detailed area reports generated using the `-hier` switch of the `report_area` command (introduced in Design Compiler version Y-2006.06) indicates that the clock tree power results may be more closely correlated with only the logic area, excluding the area consumed by black-box memories. To investigate this possible trend, the black

box memory area was subtracted from the total area to get the logic area. Table 8.3 lists both the total area and logic area, along with the clock tree power, for each of the tests examined previously.

Test	**Area (μm)**	**Logic area (μm)**	**Clock power (mW)**
adpcm_encode	825360	364202	6.480
dct	990126	387138	6.842
g721_decode	789697	314638	5.539
g721_encode	617896	293377	5.166
gsm_decode	646700	311944	5.703
jpeg_decode	1051382	401758	6.609
pgp_decode	875195	371067	6.319
st_modified	1360079	394731	6.869
test_nop	539615	321876	5.691
video_blur	1200872	368467	6.613

Table 8.3: Total area and logic area compared with clock tree power

As before, the results are plotted on a scatter graph and linear regression analysis performed to determine the relationship between the two characteristics. The resultant graph is shown in Figure 8.3.

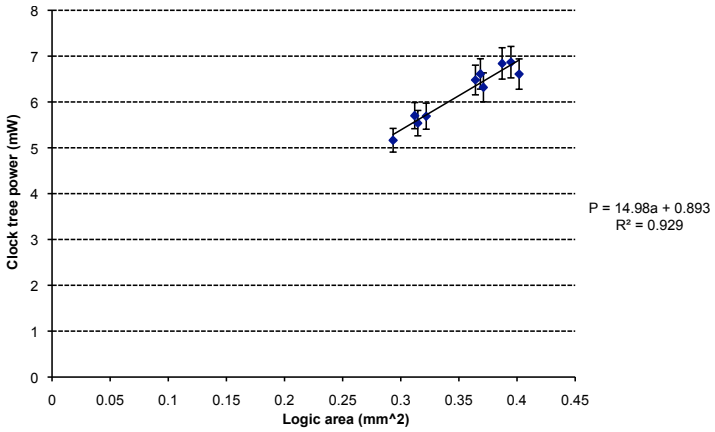


Figure 8.3: Clock tree power as a function of logic area

A more strongly-correlated linear relationship is apparent compared with that shown in Figure 8.2. These figures give a squared confidence interval of 0.929—much better than the value of 0.699 determined for the model using total area. The trend line slope of 14.98 $\mathrm{mW\,mm^{-2}}$ gives the multiplier factor for the linear relationship between total area and clock tree power. Combining the multiplier with the intercept value of 0.893 gives a relationship of:

$$P_{clk} = 0.893 + 14.98 \times A_{logic} \tag{8.5}$$

Where P_{clk} is the clock tree power, and A_{logic} is the total coprocessor area in mm^2.

Similar to the approach used with Equation 8.4, the accuracy of clock tree power estimates calculated using Equation 8.5 are compared against those generated by Power Compiler in topographical mode, shown in Table 8.1. Results and percentage errors for each test are listed in Table 8.4.

Test	**Logic area (μm)**	**Clk pow. (mW)**	**Clk pow. est. (mW)**	**% error**
adpcm_encode	364202	6.480	6.385	-1.47
dct	387138	6.842	6.728	-1.66
g721_decode	314638	5.539	5.642	1.86
g721_encode	293377	5.166	5.324	3.05
gsm_decode	311944	5.703	5.602	-1.77
jpeg_decode	401758	6.609	6.947	5.12
pgp_decode	371067	6.319	6.488	2.67
st_modified	394731	6.869	6.842	-0.39
test_nop	321876	5.691	5.751	1.05
video_blur	368467	6.613	6.449	-2.49

Table 8.4: Clock tree power estimate calculated from logic area

These results show a definite improvement in accuracy over those generated by analysing the total area—the percentage error range has fallen to just over ±5%, from ±10% previously. The error also has a lower standard deviation of 2.54%.

8.1.3 Integrating clock tree power analysis into Cascade

Implementing clock tree power analysis calculation into Cascade, using the previously determined relationship with logic area, is a trivial matter. Two values are added to the technology file for each target technology, one for the constant power and another for the area-dependent multiplier. To align with the rest of the calculations performed by Cascade, the power values are converted to energy per cycle values, which is done by multiplying the previously determined values by the clock period—in this case 100 ns. Therefore the constant component of the energy per cycle becomes:

$$\begin{aligned} & 0.893 \times 10^{-3} \times 100 \times 10^{-6} \\ = \quad & 0.893 \times 10^{-7} J/cycle \\ = \quad & 0.0893 nJ/cycle \end{aligned} \tag{8.6}$$

and the area-dependent multiplier becomes:

$$\begin{aligned} & 14.98 \times 10^{-3} \times 100 \times 10^{-6} \\ = \quad & 1.498 \times 10^{-6} J/cycle \\ = \quad & 1.498 nJ/mm^2/cycle \end{aligned} \tag{8.7}$$

These two values are annotated into Cascade's `technology.xml` file for the target technology, labelled as `clockEnergyConst` and `clockEnergyMult` respectively. Thus, to calculate the energy attributable to the clock tree, Cascade simply adds the first value with the second value multiplied by the area. The clock tree power is reported in the analysis summary report for comparative purposes, however it is not included in the overall energy total for the coprocessor due to the unoptimised nature of the estimates as explained in subsection 8.1.2. Thus, the user can decide whether to include the clock tree energy estimate, or instead carry out a full analysis, post-layout and clock tree synthesis, once the desired coprocessor candidate has been selected.

8.2 Clock gating

Employing any form of logic on the clock line, such as boolean gates or latches, for the purposes of realising a desired functionality has traditionally been considered poor design form due to issues that may appear during verification or timing closure. However, as a means of reducing both power and energy consumption with minimal, or zero, adverse effect on area and timing performance, carefully employed clock gating has emerged as a highly effective technique.

Clock gating has been utilised in mainstream microprocessor design for over a decade. Early techniques involved manually inserting coarse-grain gating, employed to shut down parts of the chip during idle periods, rather than attempting to reduce the clock tree power itself [83]. As feature sizes have reduced, interconnect power has become increasingly important in proportional terms, due to increased interconnect capacitance and higher switching frequencies. As mentioned in section 8.1, the clock tree has the highest switching frequency of all signals in a synchronous design, and it also tends to have a very high fan-out, resulting in large switching capacitances. This manifests itself in a high average power consumption in typical use, for example the 90 nm Hitachi SuperH processor consumes 15% of the total chip power in the clock tree [84].

As an example of clock gating in research literature, a method for reducing the power in finite state machines (FSMs) is presented in [85]. The authors offer an algorithm that detects cycles when the FSM is in a self-loop—that is, the state following a clock edge is the same as the previous state—and halts the clock input to the FSM during that cycle. The result is a functionally equivalent design that consumes less power because the switching capacitance of the clock tree is reduced, and none of the registers within the FSM switch during that cycle. The algorithm was initially restricted to Moore (fully synchronous output) FSMs, but later work extended it to include Mealy FSMs [86].

8.2.1 Clock gating methods applicable to Cascade

Cascade coprocessors do not make extensive use of FSMs in the logic, therefore a more general approach is required than that described in the work listed above. Working at the register level, which is the basic building block of synchronous circuits, allows clock gating to be applied to a wide range of logic blocks available to Cascade. Often registers are designed with a load-enable signal, which prevents any new data being clocked in when driven low. A typical implementation example of such a D-type register with enable signal is shown in Figure 8.4, with the load-enable signal toggling a multiplexer that maintains the current value via a feedback loop from the output.

Such a design is wasteful of energy during idle cycles (that is, while the load-enable signal is low), as the register is repeatedly clocking in the same value on each cycle. Clock gating provides an ideal solution to this energy wastage, utilising the load-enable signal as an activation mechanism for the clock gating circuitry. In Figure 8.5, the load-enable multiplexer

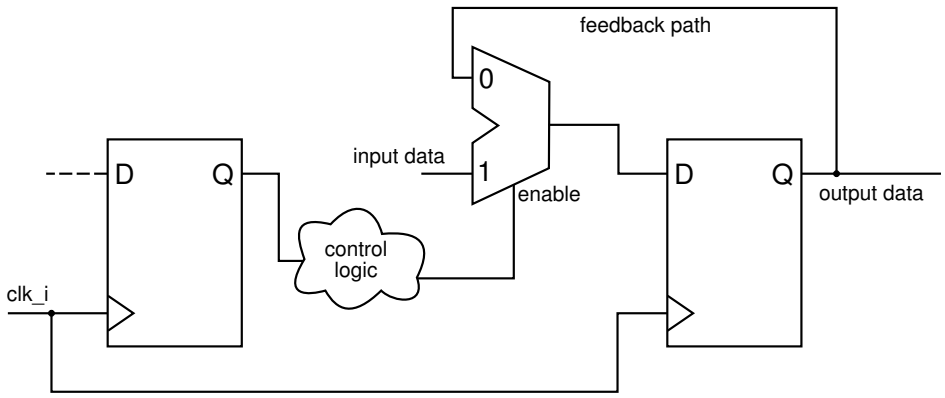


Figure 8.4: D-type register with load-enable signal

has been replaced by a D-type transparent latch combined with an AND gate, which control whether the clock signal is passed to the register depending on the load-enable signal. As the latch will only allow its input to change while the clock is high, the possibility of glitching due to unexpected changes in the load-enable signal is eliminated. It is common for recent ASIC technology libraries to contain optimised cells specifically for the purposes of clock gating; for example, the TSMC 90 nm technology library offers the cell *TLATNCA*, which effectively combines the transparent latch with the AND gate into a single macro implementation.

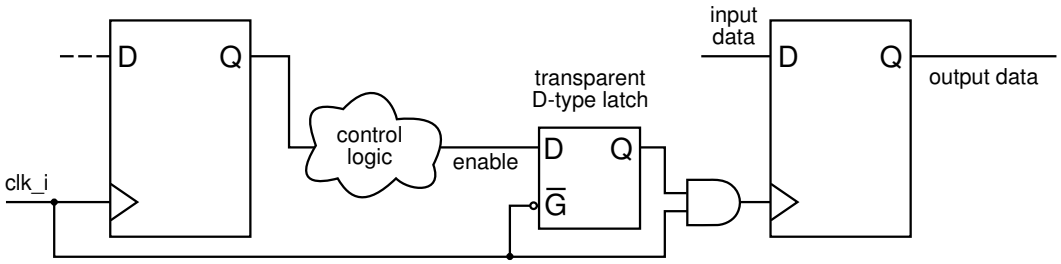
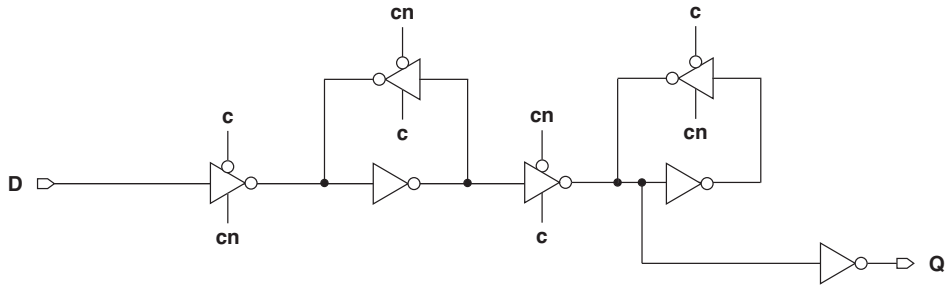


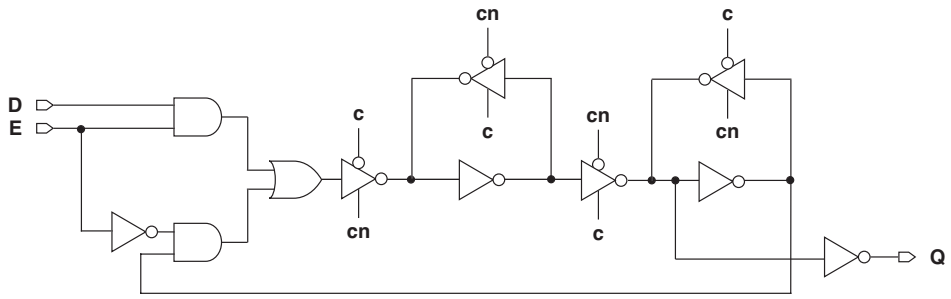
Figure 8.5: Latch based clock gated register

Similarly to the clock gating latch cell, many technology libraries provide an optimised D-type register with enable signal, allowing a more efficient implementation than that shown in Figure 8.4. In the TSMC 90 nm technology library, *EDFFHQ* provides an equivalent to the *DFFHQ*. Schematics of both types of register are shown in Figure 8.6. Such a cell will be automatically utilised during synthesis for any register that requires an enable signal, avoiding the need for a separate multiplexer.

It can clearly be seen that the addition of an enable signal to the base D-type register adds complexity to the cell, and this is reflected in the area, power and timing figures. However,



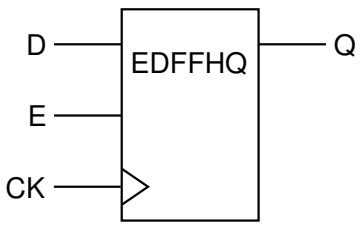
(a) DFFHQ D-type register without enable signal



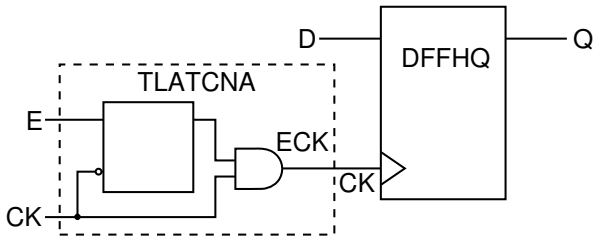
(b) EDFFHQ D-type register with enable signal

Figure 8.6: TSMC 90 nm register schematics [87]; c = clk_i, cn = $\overline{clk\_i}$

the additional area and power consumed by the clock gating latch cell must be considered when evaluating the differences between traditional load-enable signal registers, and gated clock replacements. Table 8.5 lists some of the attributes of each type of cell, along with those of the clock gating cell. All values are based on the X2 drive strength configuration of each cell.



(a) Enable signal register



(b) Clock gated register

Figure 8.7: Implementation of both register types

Based on these figures, it is possible to construct a formula to estimate the energy consumption for both the non-gated and gated clock implementations, and thus make decisions on which approach is preferable under each circumstance.

		DFFHQ	**EDFFHQ**	**TLATNCA**
Area[1] (pm^2)		16.9344	22.5792	9.1728
Delay[2] (ns)	(Rising)	0.0754	0.0727	0.0486
	(Falling)	0.0799	0.0773	0.0609
Power[3] μW/MHz	(D)	0.0072	0.0107	-
	(CK)	0.0155	0.0173	0.0057
	(E)	-	0.0112	0.0055
	(Q) / (ECK)	0.0051	0.0060	0.0059

Table 8.5: Area, delay and power figures for TSMC 90 nm cells [87]

$$E_{non-gated} = n \times 0.0155 \times F_{CK} \times t + \sum_{i=1}^{n} 0.0107 \times T_D(i) + 0.0112 \times T_E(i) + 0.006 \times T_Q(i) + 0.0173 \times f_{CK} \times t \quad (8.8)$$

$$E_{gated} = 0.0055 \times T_E + (E \times 0.0059 + n \times 0.0155) \times F_{CK} \times t + \sum_{i=1}^{n} 0.0072 \times T_D(i) + E \times 0.0051 \times T_Q(i) \quad (8.9)$$

[1] Area units pm^2 = squared picometres. 1 pm = 1×10^{-12} m

[2] CK→Q (or CK→ECK for TLATNCA) transition delay between a clock edge and the associated output change

[3] Internal power dissipated each time the associated pin changes

By simultaneously analysing Equations 8.8 and 8.9 it transpires that to reduce power due to *CK* switching, a register bank would need to have a width of 5 or more bits gated by a single latch to show an improvement over the non-gated design. On the other hand, switching *D* and *E* is always more efficient in the clock gated implementation, due to the use of DFFHQ registers rather than the more complex EDFFHQ registers. Since the clock signal is likely to be the most active signal in most cases, additional weighting should be given to the effect of this signal, hence making the "break even" point for gated clock banks a width of around 3 or 4 bits. If detailed switching statistics are available for individual banks, they can be used for a more accurate analysis of the ideal minimum bank width for clock gating.

8.2.2 Automated clock gating using Power Compiler

The aforementioned technique provides a fine grained approach to reducing the switching power in both registers and the clock tree itself. Although it is possible to implement such a scheme at the RTL level, in most cases it is preferable to allow dedicated tools to perform fine grained clock gating at the gate level due to greater knowledge of timing requirements and ASIC library specifications becoming available post-synthesis [88] [89]. There are several commercial tools available to automatically insert clock gating logic, usually as part of the synthesis flow. Examples of such tools include Synopsys' Power Compiler and Sequence Design's Power Theater.

Power Compiler's automated clock gating functionality can be used to reduce the dynamic power and energy consumption of coprocessors generated by Cascade. Through a combination of calculated predictions (in the previous subsection) and experimentation, it has been determined that for most cases the optimal minimum register bank width for clock gating is 3 bits. This value can be defined as a constraint in Power Compiler, preventing the tool from gating any register banks less than 3 bits wide. The setup command:

```
set_clock_gating_style -pos integrated -neg nor -minimum_bit_width 3
```

configures Power compiler to use an integrated cell (such as *TLATNCA* from the TSMC library) for gating clocks on the positive edge, a NOR cell with latch for negative-edge gated clocks, and a minimum bank width of 3 bits. The command to insert clock gating is simply:

```
insert_clock_gating -global
```

which is called after the design has been elaborated. This will instruct Power Compiler to analyse the design and automatically insert clock gating logic under the previously defined constraints.

A test case based on a coprocessor generated by Cascade for performing the SHA-1 hashing algorithm was run both with and without automatic clock gate insertion enabled. The results are listed in Table 8.6, showing the effect of clock gating on both dynamic (internal) and leakage power for the 20 most power hungry cells in the design.

Cell	**Cell internal power (mW)**		**Cell leakage power (pW)**	
	Clock gated	**Ungated**	**Clock gated**	**Ungated**
native_0	0.8742	1.0920	1.2283×10^{8}	1.2164×10^{8}
fu_Cache1	0.7712	0.9486	2.0290×10^{8}	2.0492×10^{8}
fu_registerfile_0	0.4243	0.4870	2.7305×10^{7}	2.8066×10^{7}
fu_arithmetic_Z	0.0984	0.2219	5.7328×10^{6}	4.3928×10^{6}
fu_bitshift_0	0.0833	0.2423	6.7646×10^{6}	4.5289×10^{6}
fu_mult64_b_0	0.0808	0.1090	8.8200×10^{6}	9.5805×10^{6}
fu_Cache0	0.0794	0.1479	4.8770×10^{7}	4.7835×10^{7}
fu_logical_0	0.0405	0.1091	4.0186×10^{6}	3.3486×10^{6}
fu_arithmetic_Y	0.0348	0.1475	6.0530×10^{6}	4.3019×10^{6}
fu_immediate8_0	0.0262	0.1605	5.5140×10^{6}	3.0243×10^{6}
fu_logical_2	0.0252	0.0701	3.2036×10^{6}	2.4553×10^{6}
fu_immediate8_1	0.0156	0.0937	3.1886×10^{6}	1.8280×10^{6}
fu_immediate32_0	0.0086	0.0640	2.4267×10^{6}	1.3865×10^{6}
fu_copy_0	0.0077	0.0941	3.6860×10^{6}	2.1189×10^{6}
fu_bitshift_1	0.0069	0.0289	1.6745×10^{6}	1.2193×10^{6}
fu_arithmetic_2	0.0063	0.0294	2.2592×10^{6}	1.9027×10^{6}
fu_logical_1	0.0033	0.0228	1.3725×10^{6}	1.0066×10^{6}
fu_predicate_0	0.0023	0.0132	5.4643×10^{5}	2.5141×10^{5}
fu_addrlink_0	0.0022	0.0141	8.5375×10^{5}	6.4835×10^{5}
fu_select_0	0.0019	0.0130	6.7834×10^{5}	4.7041×10^{5}
Totals (20 cells)	2.593 mW	4.109 mW	458.613 μW	444.936 μW

Table 8.6: Cell power comparison between gated and ungated clock designs

It can be seen from the table that there is a reduction in dynamic power for listed cells of almost 37%; this is accompanied by a modest increase in leakage power of 3%—partly due to the small increase in cell area from 1.4979 μm^2 to 1.5055 μm^2. At the TSMC 130 nm technology node used for this test, the small increase in leakage power is insignificant, particularly as it is offset by a much larger reduction in dynamic power. However, the increase

is worth noting, as leakage power is projected to contribute a much larger proportion of overall power as process technology size continues to shrink. Further analysis of leakage power issues are undertaken in chapter 9.

In addition to the fine grained approach to clock gating described above, a more coarse grained approach can be implemented at the system and RTL levels. Such an approach involves a more active technique; rather than simply changing the implementation of an existing signal (such as the enable signal, as employed by Power Compiler) a new signal is created to indicate when particular units should be disabled. Clock gating logic can then be applied using that signal to disable the clock to an entire block of logic, providing a very effective and efficient method of reducing dynamic power in units that spend significant periods of time in an idle state. Such a signal can be used in a similar way to an enable signal—rather than simply halting the operation of a unit, for example during stall cycles, the clock is halted instead resulting in a greater power and energy saving.

At this point in time, clock gating is disabled by default in the power analysis results generated by Cascade. As the coprocessors are generated at the RTL level, implementing full clock gating depends on such a capability being available in the logic synthesis tool being used by the end user, a scenario that cannot be guaranteed. There are also differences between the various tools available to perform clock gating, and the configuration of the tool will affect the results obtained. Therefore Cascade assumes that clock gating is not being used, and the user can later apply clock gating to the design as part of their synthesis flow, to further reduce the power and energy consumption of the design. The gated power and energy values are available to Cascade, should there be a particular desire to switch the energy estimates to those representing a clock-gated coprocessor.

8.3 Summary

In this chapter, a detailed examination of the power and energy consumed by the clock tree has been undertaken, and the benefits that clock gating can offer to Cascade coprocessors have been analysed.

The average power consumed by the clock tree was analysed for a range of coprocessors using Design Compiler's topographical mode. The results were then examined to determine

any pattern that could assist in estimating clock tree power with a much lower analysis cost. Initially a correlation with overall coprocessor area was discovered and an algorithm implemented to make use of it, but afterwards a tighter correlation was discovered with logic area, excluding black-box macro blocks. Thus the clock tree power estimation implemented in Cascade makes use of this correlation, allowing early stage estimates to be calculated for a large number of coprocessor candidates, and the energy value added to the overall coprocessor energy consumption estimate.

A low-level examination of clock gating techniques was undertaken, with a comparison of the standard cells used in both standard and clock-gated implementations. The area, delay and energy cost of each approach was considered, and it was determined that a register bank would have to be a minimum of 3 bits wide to show a net benefit from clock gating, in terms of both energy consumption and area reduction. In addition, a range of functional units were analysed to determine the difference in both active internal power and leakage power, for clock gated and non-gated designs. The results of these analyses have been made available to Cascade, should it be desired that the coprocessor energy estimations generated by Cascade assume that logic synthesis will be initiated with clock gating enabled.

The techniques developed and analysis undertaken in this chapter are very generic, and as a result are largely applicable to any configurable processor implementation with only minor adaptations likely to be required.

9. Leakage power issues

Before the deep sub-micron era, leakage power was of little concern to CMOS device designers. The ideal CMOS gate dissipates no power during steady states, therefore the only time power is dissipated is during state transitions, i.e. active periods. In reality, all CMOS gates dissipate power continuously, due to the physical realities of transistor design. Historically, the power dissipated during steady state periods has been so small in comparison to active power that it was insignificant, other than in exceptional cases where the device spent a very large proportion of its time in standby mode. Reductions in feature size through process technology improvements have resulted in a sharp increase in leakage currents due to two fundamental issues: increased transistor density resulting in thinner gate oxides and more transistors per unit area; and reduction in supply voltage leading to corresponding reductions in threshold voltage to maintain performance, meaning the biasing of transistors in the off state is reduced.

9.1 Sources of leakage power

Leakage current can be broken down to sub-threshold leakage current, and gate-oxide tunnelling current. The former is due to transistors not being completely off even when in the off state, and is characterised by Chandrakasan *et al*. [90] in Equation 9.1.

$$I_{sub} = K_1 W e^{-V_{th}/nV_\theta}(1 - e^{-V/V_\theta}) \tag{9.1}$$

Where V is the supply voltage, V_{th} is the threshold voltage, K_1 and n are experimentally derived constants, W is the gate width, and V_θ is the thermal voltage—around 25 mV at room temperature, increasing linearly with temperature.

Examining ways to reduce subthreshold leakage current referring to Equation 9.1, V and V_{th} are the only variables that can be indirectly controlled in an ASIC design process. Control logic can be added to enable the supply voltage, V, to be reduced to zero during idle cycles, resulting in subthreshold leakage current also dropping to zero. However, doing so causes a loss of state and may also carry a "wake-up" penalty, diminishing the benefit of such an approach for shorter idle periods. Similarly, most ASIC libraries provide high V_{th} cells, the use of such results in a negative exponential reduction in I_{sub}. The drawback of doing so is a reduction in switching speeds due to the smaller voltage differential between the supply voltage and threshold voltage, meaning that their use is restricted to non-critical paths [91]. Alternative approaches, such as using the "stack effect" with multiple transistors [92], offer similar benefits to scaling V_{th}, but also bring similar drawbacks.

The use of multiple threshold voltage transistors is typically applied at the post-synthesis stage, once path cycle times and available slack have been estimated. However, the effectiveness of multi-V_t implementations has a significant dependence upon the design techniques used at the RTL and higher levels. Designs with a significant number of complex paths between register blocks are likely to leave little slack, making the use of high V_t transistors unsuitable. For example, a single cycle multiplier would likely have a stringent timing budget, whereas substituting a pipelined equivalent will loosen those timing requirements at the cost of additional latency.

Similarly, implementing a parallel architecture will usually result in a lower frequency requirement to achieve the same throughput as an equivalent sequential implementation. This type of approach has been employed by Cascade from the outset, primarily to allow higher performance as required, but it also provides the benefit of lower power dissipation in the clock tree and the option to use slower, more energy efficient, transistors. Although this approach can be highly effective in reducing active power, the increase in transistor count presents a problem for leakage power reduction.

9.2 Calculating coprocessor leakage power

Previous sections have focused on automating active power analysis within Cascade. A similar functionality is required for leakage power, particularly for 90 nm technologies and beyond, to ensure that Cascade's overall power and energy estimates are accurate.

Leakage power typically shows a much smaller variance for any one particular hardware design, such as a single coprocessor generated by Cascade. In contrast, active power is highly dependent on both the hardware of the coprocessor, as well as activation patterns triggered by the software being run, as described in chapter 6.

In order to obtain the variance between different implementations of each functional unit, while running different software, a leakage power analysis was performed across all tests present in the MediaBench suite. The average of all tests was computed, along with the standard deviation to determine a statistical variance between implementations. Both values were computed for each test using the script listed in Appendix F. Table 9.1 shows the average leakage power recorded for each unit using both TSMC 130 nm and TSMC 90 nm technology libraries, along with the number of occurrences of each unit over the entire set of benchmarks. The occurrence count does not directly affect the result, but gives an indication of the "quality" of the results for each unit—an average calculated from a larger spread of results is more likely to be representative of the general case, dependent on whether the standard deviation is high or low.

To determine how closely the average reflects the likely value over a large number of designs, the standard deviation is calculated for each unit under both 130 nm and 90 nm technology libraries. This provides a statistical measure of how far each case deviates from the mean value averaged over all cases. Note that standard deviation is calculated for only the two most commonly used access units. The reason for this is primarily because the other access units are not instantiated in enough cases to get a meaningful figure for their standard deviation—they occur only four times each during the tests detailed above. Additionally, the access units are all fundamentally similar in design, therefore the variance is likely to be similar across all access units.

Functional unit	**Average leakage (nW)**		**Occurrences**
	TSMC 130 nm	**TSMC 90 nm**	
access_st_1	3.228×10^5	5.174×10^5	14
access_st_1r	4.519×10^5	7.244×10^5	14
access_1x	4.668×10^5	7.483×10^5	4
access_1	4.510×10^5	7.229×10^5	4
access_1r	5.689×10^5	9.120×10^5	4
access_2	6.180×10^5	9.120×10^5	4
access_assoc_1	4.804×10^5	7.701×10^5	4
access_assoc_1r	6.153×10^5	9.863×10^5	4
access_stream_1	2.473×10^5	3.965×10^5	4
access_stream_1r	3.291×10^5	5.276×10^5	4
access_stream_1x	2.838×10^5	4.550×10^5	4
access_stream_st_1	3.337×10^4	5.350×10^4	4
access_stream_st_1r	8.951×10^4	1.434×10^5	4
access_remap_1	1.685×10^5	2.702×10^5	4
arithmetic	2.761×10^3	9.804×10^3	55
bitshift	2.096×10^3	8.409×10^3	20
branch	2.557×10^2	1.840×10^3	14
combine	1.834×10^2	8.488×10^2	14
coreregfile	5.210×10^4	1.763×10^5	14
immediate32	2.213×10^3	6.656×10^3	14
immediate8	2.302×10^3	6.936×10^3	30
logical	1.801×10^3	7.445×10^3	17
multiplier64	2.464×10^4	7.570×10^4	14
predicate	3.206×10^2	1.620×10^3	14
registerfile	3.964×10^4	1.957×10^5	14
sat_arithmetic	1.432×10^3	5.138×10^3	14
select	1.839×10^3	7.017×10^3	21
squash	5.522×10^2	3.005×10^3	14
CBNative_Slave_Generic	1.819×10^5	1.013×10^6	9
AMBA_AHB_Slave_Generic	1.642×10^5	8.944×10^5	5

Table 9.1: Average functional unit leakage current in 130 nm & 90 nm technology

Standard deviation is calculated using Equation 9.2.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}} \tag{9.2}$$

where σ is the standard deviation, x_i is the difference for each test, $\bar{x}$ is the mean of the differences from all tests, and n is the number of tests.

Functional unit	**Standard deviation (nW)**			
	TSMC 130 nm		**TSMC 90 nm**	
access_st_1	9.994×10^{4}	(30.9%)	9.632×10^{4}	(18.6%)
access_st_1r	1.148×10^{5}	(23.6%)	1.225×10^{5}	(16.5%)
arithmetic	1.188×10^{3}	(43.0%)	3.831×10^{3}	(39.0%)
bitshift	4.182×10^{2}	(19.9%)	1.472×10^{3}	(17.5%)
branch	3.310×10^{1}	(12.9%)	1.960×10^{2}	(10.6%)
combine	1.050×10^{1}	(5.7%)	7.690×10^{1}	(9.0%)
coreregfile	2.666×10^{3}	(5.1%)	8.585×10^{3}	(4.8%)
immediate32	6.229×10^{2}	(28.1%)	1.761×10^{3}	(26.4%)
immediate8	9.396×10^{2}	(40.8%)	2.652×10^{3}	(38.2%)
logical	9.053×10^{2}	(50.2%)	3.267×10^{3}	(43.8%)
multiplier64	4.624×10^{3}	(18.7%)	1.470×10^{4}	(19.4%)
predicate	1.701×10^{2}	(53.0%)	7.494×10^{2}	(46.2%)
registerfile	1.015×10^{3}	(2.5%)	4.228×10^{3}	(2.1%)
sat_arithmetic	1.420×10^{1}	(0.9%)	2.353×10^{2}	(4.5%)
select	6.580×10^{2}	(35.7%)	2.431×10^{3}	(34.6%)
squash	4.360×10^{1}	(7.9%)	3.410×10^{2}	(11.3%)
CBNative_Slave_Generic	4.511×10^{4}	(24.8%)	2.857×10^{5}	(28.2%)
AMBA_AHB_Slave_Generic	3.735×10^{4}	(22.7%)	2.407×10^{5}	(26.9%)

Table 9.2: Standard deviation across all leakage tests in 130 nm & 90 nm technology

Table 9.2 shows a large variance in the standard deviation of leakage power for different units. The relatively small deviation in some units, such as registerfile and sat_arithmetic, implies that for these units the average leakage power will provide a reasonable estimate for the leakage power of an arbitrary test. Less consistent units, such as fu_logical, will require a more detailed examination that considers the changes in underlying structure between implementations. In most cases, this is as a result of differing numbers of banks—primarily

output banks—therefore a large improvement in accuracy is available by examining individual banks within each unit. The influence of output banks on leakage power will be taken into account later in the analysis.

The greater variance shown in the results using TSMC's 90 nm library can be attributed to characterisation of the library being far more detailed with regard to leakage power, compared with the 130 nm library. As an example, shown below is an excerpt from the 90 nm *liberty* (.lib) format library file showing the leakage power entry for the NAND2X1 cell:

```
cell_leakage_power : 16369.830000;
leakage_power() {
  when :"!A & !B";
  value : 1835.910000;  }
leakage_power() {
  when :"!A & B";
  value : 7199.181000;  }
leakage_power() {
  when :"A & !B";
  value : 6005.619000;  }
leakage_power() {
  when :"A & B";
  value : 16369.830000; }
```

By comparison, the equivalent entry in the 130 nm library file lists only:

```
cell_leakage_power : 1914.672600;
```

Thus, it is clear that while the 130 nm library lists only one fixed value for the leakage power of each cell, the 90 nm library takes into account the change in leakage power that is reflected by different input patterns. This change to a higher accuracy format reflects the increasing importance of leakage power in the overall power consumption as device sizes shrink, and must also be considered in Cascade's analysis. This presents an additional challenge, in that such a detailed leakage power analysis should take into account input state at any particular point of a simulation run, regardless of the clock signal. Thus all changes to any signal within a unit must be monitored. Such additional complexity must additionally be abstracted to a higher level in a similar manner to that implemented for active power calculation.

Feasibility analysis of such an approach indicates that the potential increase in accuracy does not justify the very large increase in complexity required to implement the more detailed

analysis. Variance between different instantiations of functional units has not increased significantly when moving from 130 nm to 90 nm implementations, despite the improved detail available for leakage power analysis at 90 nm. This can be explained by the variance caused by differences in the units themselves (such as output bank configuration) being a much more significant source of variance between units than that revealed by monitoring input patterns to each gate within a unit. Thus the improvement in accuracy that is directly attributable to a more detailed gate-level leakage power analysis is quite small.

Additionally, a significant performance penalty would be incurred in implementing such functionality at a higher level within Cascade, reducing the ability to perform quick analysis of a large number of potential coprocessor candidates at an early stage of design space exploration. As a result, the more detailed leakage power analysis available within the 90 nm libraries will be ignored at this point in time, in favour of a basic high-speed implementation similar to that provided for the 130 nm libraries.

To examine the amount of leakage power variance between lower level blocks that are largely unchanged between instantiations, a similar approach to that used to calculate the variance between functional units was employed. In this case, the output bank unit was selected as it is a commonly occurring but largely consistent unit. The leakage power of 1042 occurrences of the unit in all MediaBench tests was considered, and the mean and standard deviation calculated for both TSMC 130 nm and TSMC 90 nm libraries. The mini script used to perform this analysis is shown below.

```
#!/bin/sh
average=1021749
grep result_o_banks reports/*_power.txt -h | \
   awk --assign average=$average '
        BEGIN {numlines=0; print "0 "}
        {print $5 " ^ * " average " - 2 ^ +";
         numlines = numlines + 1}
        END {print numlines" 1 - / v p"}' - | \
   sed -e 's/e+/ 10 /' > average_leakage_calc.txt
dc -f average_leakage_calc.txt
```

Results are shown in Table 9.3. Unsurprisingly both the mean leakage power and the absolute variance between instantiations is higher with 90 nm libraries, however in both cases the standard deviation indicates that variance is low enough to be near insignificant for the purposes of making high level power estimations—for this particular unit the average value provides good accuracy in almost all cases.

	TSMC 130 nm	**TSMC 90 nm**
Mean	1.0217×10^6pW	3.4769×10^5pW
Standard deviation	0.0202×10^5pW (0.58%)	0.2924×10^6pW (2.86%)

Table 9.3: Output bank leakage statistics in 130 nm & 90 nm technology

In order to improve the accuracy of leakage power estimates, it is necessary to consider architectural differences between instantiations of functional units. An excerpt from the power analysis report for the ADPCM encode test, using TSMC 90 nm technology, is shown in Figure 9.1. The hierarchical breakdown of average power for `fu_select` shows that there are two instances of output banks in this particular instantiation of the unit. Within the same coprocessor, there are two other select units each containing five instances of output banks. As a result, the average leakage power for those units is over 9×10^6 pW, compared with 5.43×10^6 pW for the unit listed below—a difference of over 50%. Clearly the number of output banks contributes a significant proportion of the leakage power consumed by functional units, and as such it should be taken into account in leakage energy estimates.

```
                                  Switch   Int      Leak     Total
Hierarchy                         Power    Power    Power    Power     %
-------------------------------------------------------------------------
fu_select_1 (fu_select_1)       8.58e-04 1.16e-02 5.43e+06 1.79e-02  0.4
  result_o_banks_1 (bank_32_6)  3.78e-05 4.11e-03 9.30e+05 5.08e-03  0.1
  result_o_banks_0 (bank_32_5)  1.89e-05 3.98e-03 9.31e+05 4.93e-03  0.1
  ex_select_1 (ex_select_1)     1.43e-04 1.32e-04 4.50e+05 7.25e-04  0.0
  setup_banks (bank_r_17)       2.64e-04 2.52e-03 5.68e+05 3.35e-03  0.1
```

Figure 9.1: Excerpt of power analysis report for ADPCM encode test

The average leakage power values listed in Table 9.1 are re-determined with the exclusion of output banks, with the results listed in Table 9.4. A single global figure is then used for the leakage power of each output bank, since they are the same across all functional units. In the case of TSMC 130 nm process technology, each 32-bit output bank results in a leakage power of 348 nW. For TSMC 90 nm process technology, the value is 926 nW. The leakage power for each unit is calculated as a combination of the "base" value excluding output banks, plus the output bank value multiplied by the number of output banks present in that unit.

The values listed in Table 9.4 are for the base functional units only, excluding any memory blocks or register files. The leakage power for memories and register files was determined in chapter 7, under the guise of standby power.

9.3 Implementing leakage calculation in Cascade

Leakage power calculation from within Cascade is a relatively simple extension of the work carried out in section 9.2. The average leakage power values for each functional unit have already been determined, therefore all that is required to automate the calculation of energy dissipated due to leakage power over a coprocessor execution period is to multiply the power for all units present with the run time of the coprocessor.

Taking $P_{leak}(i)$ as the leakage power for functional unit i, $N_{banks}(i)$ as the number of output banks in functional unit i, P_{bank} as the leakage power per bank, T is the time period that the coprocessor is running, n as the total number of execution cycles carried out by the coprocessor, and f_{clk} is the clock frequency, the total leakage energy E_{leak} can be calculated:

$$\begin{aligned} E_{leak} &= \sum_{i=0}^{j} [P_{leak}(i) \times N_{banks}(i) \times P_{bank}] \times T \\ &= \sum_{i=0}^{j} [P_{leak}(i) \times N_{banks}(i) \times P_{bank}] \times \frac{n}{f_{clk}} \end{aligned} \tag{9.3}$$

Alternatively, the leakage energy per cycle can be calculated as:

$$E_{leak\_cyc} = \frac{\sum_{i=0}^{j} P_{leak}(i) \times N_{banks}(i) \times P_{bank}}{f_{clk}} \tag{9.4}$$

The latter approach allows the calculation of leakage energy to be performed in a similar manner to that for functional unit dynamic power (as detailed in chapter 6). That is, each cycle incurs an energy cost attributable to each functional unit, and that cost is added to the total energy cost, which is summarised at the end of the coprocessor simulation run. The key difference with the leakage energy calculation is that each unit consumes the same amount of leakage energy on every cycle; that is, there is no distinction between active and inactive cycle energy, as is the case for dynamic energy calculation.

The average leakage power values for functional units, listed in Table 9.4, are annotated into the same `technology.xml` files used for dynamic energy calculation—one file for each target process technology. For each coprocessor candidate being analysed during design space exploration, Cascade applies Equation 9.4 to the list of functional units present in the coprocessor, parsing the relevant values for each unit from the XML file. The execution cycle count is determined by Cascade as part of the existing coprocessor synthesis and analysis process, therefore no additional work is required to obtain this value.

In some cases, the target clock frequency may not be known during the early stages of the coprocessor synthesis flow. Therefore to solve the problem that total leakage energy cannot be determined without knowing the coprocessor operating frequency, Cascade can omit the frequency element of the calculation and produce results in terms of energy consumed per second of operation—effectively the leakage power consumed while the coprocessor is operational. This value allows easy comparison between coprocessor candidates, and can easily be converted back to a total energy consumption figure once the coprocessor clock frequency has been set.

9.4 Summary

In this chapter, the sources of leakage power were examined, and the reasons why it is becoming an increasingly important component of power and energy consumption, particularly at smaller dimensioned process technologies such as 90 nm and beyond, were considered.

An automated leakage energy analysis process was developed, similar to that developed for dynamic energy in chapter 6. Creating such a process for leakage energy proved to be a lot simpler than that for dynamic energy, mainly because the power consumption for each device instantiation is largely constant, with no dependence on the rate of gate switching within each unit.

The TSMC 90 nm technology libraries provide a higher level of detail in their leakage power characterisation than previous libraries, in that leakage power is dependent on the input pattern. However it was decided that the increase in accuracy offered by this facility is not sufficient enough to justify the huge increase in computational requirements that would be

placed upon Cascade in order to calculate the more accurate leakage energy—both the monitoring of input pattern changes, and the leakage calculation itself, carry a significant cost.

Functional unit leakage energy values were integrated into Cascade's XML files, which are used to calculate leakage energy for each coprocessor candidate. The result of this calculation is added to the previously determined dynamic energy to get an overall energy consumption value for each candidate. Coprocessor leakage energy results are reported in both verbose and summarised forms.

The leakage power analysis techniques developed in this chapter could easily be adapted to other types of configurable processor, as the approach used is quite generic with only a small degree of focus on Cascade-specific functionality.

Functional unit	**Leakage power (nW)**	
	TSMC 130 nm	**TSMC 90 nm**
access_st_1	1.397×10^3	3.200×10^3
access_st_1r	1.956×10^3	4.480×10^3
access_1x	2.021×10^3	4.628×10^3
access_1	1.952×10^3	4.471×10^3
access_1r	2.462×10^3	5.640×10^3
access_2	2.675×10^3	6.126×10^3
access_assoc_1	2.079×10^3	4.762×10^3
access_assoc_1r	2.663×10^3	6.099×10^3
access_stream_1	1.071×10^3	2.452×10^3
access_stream_1r	1.425×10^3	3.263×10^3
access_stream_1x	1.229×10^3	2.814×10^3
access_stream_st_1	1.445×10^2	3.309×10^2
access_stream_st_1r	3.874×10^2	8.873×10^2
access_remap_1	7.296×10^2	1.671×10^3
arithmetic	1.144×10^3	5.240×10^3
bitshift	5.600×10^2	4.520×10^3
branch	2.280×10^2	1.650×10^3
combine	1.870×10^2	7.440×10^2
coreregfile	2.132×10^3	1.456×10^4
immediate32	6.140×10^2	2.155×10^3
immediate8	2.480×10^2	1.300×10^3
logical	5.880×10^2	4.940×10^3
multiplier64	1.944×10^3	8.330×10^4
predicate	2.300×10^2	1.120×10^3
registerfile	8.200×10^2	5.500×10^3
sat_arithmetic	1.062×10^3	4.404×10^3
select	5.300×10^2	3.863×10^3
squash	5.060×10^2	2.700×10^3
CBNative_Slave_Generic	1.002×10^4	3.800×10^4
AMBA_AHB_Slave_Generic	2.150×10^4	6.450×10^4

Table 9.4: Functional unit leakage power excluding memory blocks and output banks

10. Power and energy optimisations

This chapter details the optimisations that have been examined and implemented within Cascade to reduce the power and energy consumption of generated coprocessors. In some cases the optimisations are purely beneficial; that is there is no significant disadvantage associated with their implementation, and as such they will typically be used universally in all coprocessors. Other cases may introduce trade-offs against area or timing criteria, requiring that the energy optimisation and its consequential effects be considered in the context of overall system requirements.

10.1 Multiplier optimisation

As examined in section 6.1, the multiplier unit is often the highest energy consuming logic block (excluding memory blocks) within a coprocessor. Therefore it is a prime candidate for optimisation, providing an opportunity for substantial savings that significantly influence the overall power budget of a coprocessor.

Analysis of usage patterns of multiplier units over a range of coprocessor applications reveals a large variance in the proportion of active cycles for the unit. In some multiplication-intensive applications, the multiplier is active during more than 50% of the cycles in which the coprocessor is active. At the other end of the scale, many coprocessors contain a multiplier unit that remains inactive throughout a typical application run. Such an arrangement is obviously inefficient, in terms of area as well as energy, and one option is removing the multiplier unit from the coprocessor thus reducing its energy consumption to zero.

Examining the coprocessor synthesis method employed by Cascade explains why seemingly redundant multiplier units are instantiated into application specific coprocessors. Each co-

processor starts with a template, that defines a minimum structure from which the entire coprocessor will be built after analysis of the application to which it will be targeted. A number of templates are defined within Cascade by default—these are listed in Table 10.1. The purpose of using such templates is to ensure a particular degree of reprogrammability within the hardware of a coprocessor; that is, even if a particular unit is not required for the targeted application, a template can enforce the instantiation of that unit ensuring it is available if required for a future reprogramming of the coprocessor.

Template name	**Description**
32_Bit_Multiplier	No ARM v5 multiply instruction support
64_Bit_Multiplier	Full ARM v5 instruction support
Minimal	No multiplier, saturating arithmetic or combine support
Minimal_Regfile	As "Minimal", plus minimal register file sizes
No_Multiplier	No multiplier support
Single_Cycle_Multiplier	Single-cycle 64-bit multiplier, full ARM v5 support

Table 10.1: Coprocessor synthesis templates provided by Cascade

Template functionality manifests itself most prominently in the case of multiplier units. Such units tend to be quite large and energy hungry, but the availability of them is paramount to obtaining acceptable performance in software that implements a non-negligible number of multiplications. In addition, the provision of a 64-bit output multiplication instruction is required to implement the ARM v5 instruction set, therefore any coprocessor generated by Cascade must take this into account if full ARM v5 instruction support is desired.

For those reasons, coprocessors generated by Cascade typically use the "64_Bit_Multiplier" template, ensuring ARM v5 compatibility and the ability to reprogram with code that performs multiplications without a substantial performance penalty. However, this approach means that often a coprocessor will be generated for an application that does not use the multiplier unit, resulting in that unit remaining idle for the entire application run. In such a case it is paramount that the multiplier idle energy consumption is minimised.

10.1.1 Reducing multiplier idle power

Typical power of a 64-bit multiplier unit under varying operating conditions was determined in section 6.1. An idle but enabled unit was found to consume around 22% more power than

an identical unit with the enable input held low. Therefore a substantial saving is available simply by ensuring that the enable input is automatically switched low during idle cycles; there is no significant start-up penalty involved with returning to enable mode using this method [78]. In actual use the savings available may be higher than those indicated in section 6.1, due to multiplier inputs toggling during idle cycles. Therefore during such cycles the multiplier is performing meaningless calculations that will be discarded; ensuring the enable line is held low during these cycles prevents such wasteful calculations. A sample test was undertaken using a coprocessor generated by Cascade with representative input stimuli; the results are shown in Table 10.2. In this case the average power of the multiplier unit has dropped by 36%, reflected in an overall coprocessor dynamic power reduction of 5%.

In almost all cases, the multiplier used is pipelined like that shown in Figure 10.1—the only exception for multiplier-enabled coprocessors being the "Single_Cycle_Multiplier" template used only in very low frequency designs due to its long critical path. In pipelined multipliers, the state of the pipeline and its associated latency must be considered by the mechanism controlling the enable signal. This is achieved by creating a register bank one bit smaller than the pipeline length, and using this register bank along with the current input to monitor the movement of valid data through the multiplier pipeline. Each time valid data is loaded, a '1' is fed into the register bank, or for invalid data (i.e. the multiplier is not being used for new inputs on the current cycle) a '0' is fed in. On each cycle the register bank is shifted, and the values at the end of the bank are allowed to fall off. Thus, any time there is a '1' present anywhere within the register bank or at the current input, valid data is present in the pipeline and the multiplier must remain enabled. On the other hand, if all values are '0' then the multiplier is idle and should be disabled to save energy.

	Without enable	**With enable**
Total dynamic power	4.840 mW	4.574 mW
Cell leakage power	881.08 μW	875.33 μW
Multiplier power	0.755 mW	0.482 mW

Table 10.2: Multiplier power savings

Further savings are possible by halting the clock signal. Again referring to section 6.1, even a disabled idle multiplier continues to consume approximately half of its full-load power. Stopping the clock of an idle multiplier can reduce active power to zero, although some multiplier implementations have input latches that continue to toggle whenever the inputs change regardless of whether the unit is disabled or the clock is halted.

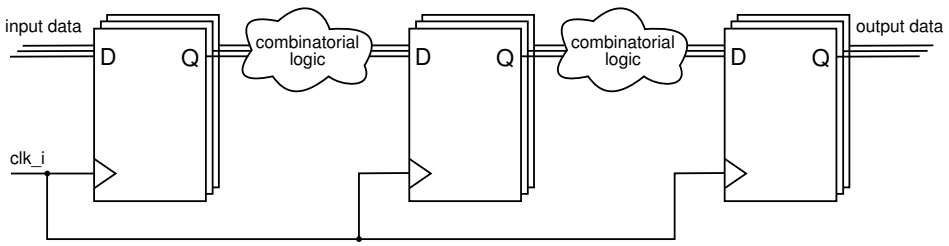


Figure 10.1: Pipelined multiplier stages

10.1.2 Preventing wasteful input latch toggling

In cases where the multiplier continues to consume significant energy during idle cycles due to input latch toggling, it is possible to mask the inputs to the multiplier on an enable signal with, for example, an array of AND gates like that shown in Figure 10.2, preventing the input latches from toggling while the multiplier unit is disabled. This approach obviously creates an increase in area, and also dissipates power during normal operation. Careful consideration of the energy savings available, combined with predictions of likely multiplier utilisation patterns for the target application, must be analysed before deploying such a method—thus it is not enabled by default within Cascade. This method completely eliminates active power within the multiplier unit during disabled cycles, leaving only leakage power as a concern. Even leakage power can be almost completely eliminated by controlling the power supply to the unit, however this introduces other issues that are detailed in chapter 9.

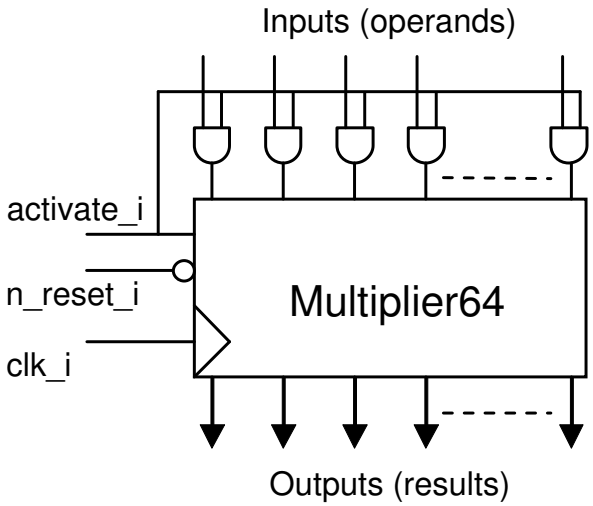


Figure 10.2: Multiplier input latch masking on enable signal

10.2 Instruction cache width reduction

The implementation of the instruction word encoding mechanism described in this section, was written in Java and integrated into Cascade by Richard Taylor at Critical Blue. All background research, tests and analysis of results as detailed in this section were carried out and documented by Paul Morgan. The code written by Richard is not listed in this thesis.

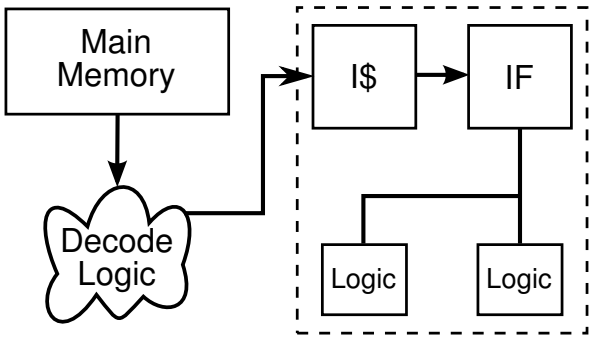
The principle single most power and energy hungry component in many coprocessor implementations is the instruction cache. Similarly, a significant proportion of the chip area is attributable to the instruction cache. Therefore there are substantial improvements available in reducing the both area requirement and energy consumption of instruction caches through more efficient utilisation, improving the efficiency of the cache without affecting performance in terms of both hit ratio and latency. Simply reducing the size of the cache to reduce energy at the cost of performance is ineffective—it is important to maintain the hit ratio of a cache from an energy optimisation viewpoint, as misses result in off-chip memory accesses, which are very expensive in terms of both lost cycles (stalls) and energy consumed.

10.2.1 Existing approaches

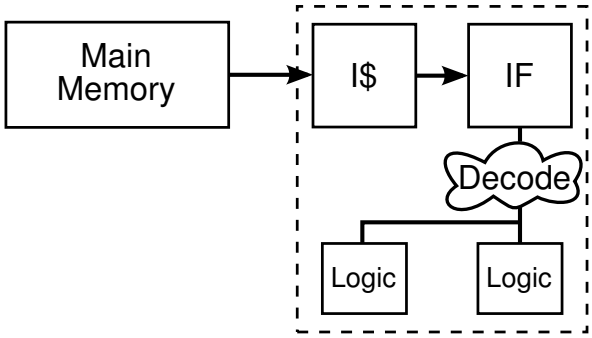
One approach to tackling instruction cache size and energy, often found in commercial embedded processors, is that of code compression, a technique that allows more efficient use of the instruction cache with the aim of maintaining a similar level of performance. There are two distinct categories of implementing instruction compression. The first involves compressing instructions in main memory only, decompressing as instructions are loaded into the cache, as illustrated in Figure 10.3(a). An example of this approach is IBM's CodePack [93]. This type of implementation has the advantage that the core does not need to be modified, and the decompression hardware is not on the critical path of the processor, minimising performance constraints. However, as it does not reduce the size or activity associated with the instruction cache, no on-chip energy savings can be made (although external bus activity is reduced). The second approach involves compression of both main memory and the instruction cache, shown in Figure 10.3(b). Doing so improves upon the previous approach in that on-core area and energy consumption are reduced, but performance may be affected depending upon the implementation used due to decompression taking place during the instruction decode stage which is on the critical path. Examples of this approach include Huffman cod-

ing [94], dictionary-based coding [95], and arithmetic coding using a Markov-model based instruction compression framework [96].

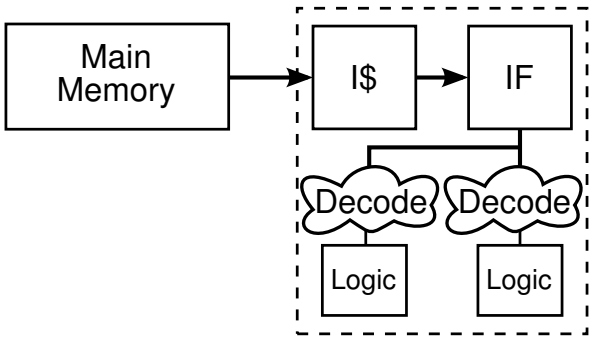
The problem with employing the aforementioned methods for coprocessors generated by Cascade is that it would be difficult to implement the required mechanism in a single cycle due to the characteristically long instruction words used by Cascade coprocessors. As such, the decompression hardware required would be complex, somewhat negating the gains made by reducing the instruction size in terms of both energy and area reduction.



(a) Global decompression between memory and I$



(b) Global decompression after I$



(c) Local decompression in front of FUs

Figure 10.3: Approaches to code decompression at run-time

10.2.2 Leveraging the application-specific instruction word

One significant advantage available to Cascade is the ability to customise the instruction word layout for each coprocessor, tailored to the application to which it is targeted; this forms a key stage in the process of architecting an application-specific VLIW processor. Thus a trade-off can be made: a wider word allows more parallel instruction issues to be made but at the cost of memory energy dissipation and area associated with the instruction cache. If the selected instruction format leans toward allowing maximal parallelism extraction, potential performance bottlenecks are avoided. The disadvantage of a wider word is that the average entropy of the instruction word tends to be poor, meaning that the processor instruction cache and instruction fetch mechanism are both area and energy inefficient.

Developing a technique similar to code compression that allows for the use of a significantly reduced instruction word width, with a correspondingly narrower instruction cache, while retaining the full performance of the underlying architecture, is a key part of improving the energy efficiency of coprocessors generated by Cascade. This can be achieved by using an encoding scheme targeted to the application for which the coprocessor is being architected; rather than attempting to arbitrarily compress long VLIW instructions.

Analysis of the nature of the application-specific code executed on the processor allows modifications to be made to the instruction set, making efficient use of commonly utilised instructions by means of shortened instructions substituted for full width instructions within the VLIW instruction. This technique has similarities to that applied by Schmitz *et al*. in the context of mode execution probabilities [97]. These shortened instructions may be easily decoded to the coprocessor's original internal microcode instruction format in front of the relevant functional unit, without impacting the overall critical path timing of the design. The instruction word layout is flexible, allowing each instruction issue slot to control any execution unit, and the ability is included to bypass the encoding mechanism so that infrequently executed instructions are not required to be encoded. Thus, provided that frequently occurring instructions are correctly identified and mapped to short instructions, the instruction word can be narrowed without limiting coprocessor throughput.

Rather than examining only the VLIW instructions and attempting to perform compression on complete words, this approach observes how instructions within each instruction word effect operations at a deeper level within the processor. During software analysis, instructions dispatched to individual functional units are examined for repetition that may enable efficient

short instruction substitutions to be made. A dictionary-like encoding scheme is then implemented, similar to that described in [98], but at a more fine-grained level within the target architecture. Thus, rather than having a single decode unit for entire VLIW words as illustrated in Figure 10.3(b), individual decode units for each functional unit are implemented as shown in Figure 10.3(c).

Coprocessors generated by Cascade have a heterogeneous VLIW architecture (i.e. the instruction word format is flexible in that each slot is capable of issuing an instruction to any functional unit). For each functional unit that is amenable to instruction encoding, effective redundancy can be created in the instruction word as that unit's average bandwidth requirement is reduced through the encoding scheme. As redundancy is created by applying the aforementioned method to multiple functional units, the width of the instruction word can be reduced, adjusting the instruction decode mechanism as appropriate, while still retaining the same level of throughput. In practice many applications permit the instruction word to be reduced to 50% or less of its original width, with no adverse impact on throughput.

To ensure that the size and energy consumption of the short instruction decode look-up table (LUT) logic remains reasonable, and that the number of bits required for encoded instructions is kept small, the algorithm does not attempt to encode all possible instructions for each functional unit. Rather, a profile-based analysis is performed that results in infrequently-used (or unused) instructions being identified and removed from the LUT. In order that these instructions can still be executed, several escape codes are implemented at the processor instruction level that allow short instructions to be bypassed and the full instruction be passed directly from other bits in the instruction word. Due to the reduced instruction word width, only a small number of full instruction can be issued simultaneously, potentially as low as one depending on how aggressively instruction width reduction has been pursued; consequently use of this facility can have a detrimental effect on performance if it results in an instruction fetch bottleneck. The selection of instruction word width is therefore one of the key decisions of this approach with regards to the trade-off between reducing area and energy, and maintaining performance.

To create a framework for the encoding algorithm, the instruction word is modified as follows: A count field indicates how many short (i.e. encoded) instructions are present in that particular instruction word, counting each slot from the most significant bits in the instruction word. In the case that a full instruction is required, the required number of bits will be made available from the least significant bits in the instruction word. Depending on the

width of the full instruction, any number of short instruction slots may overlap with the full instruction. Thus the corresponding short instruction bits are not used during that cycle if a full instruction should be required, which temporarily reduces the short instruction count value. Cascade's instruction scheduling algorithm is aware of these restrictions and assigns the layout of the instruction word appropriately.

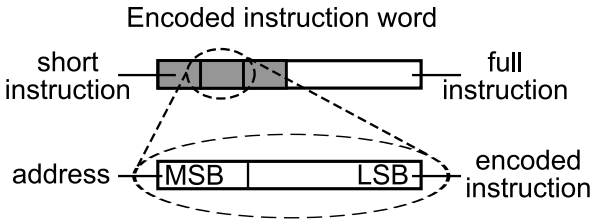
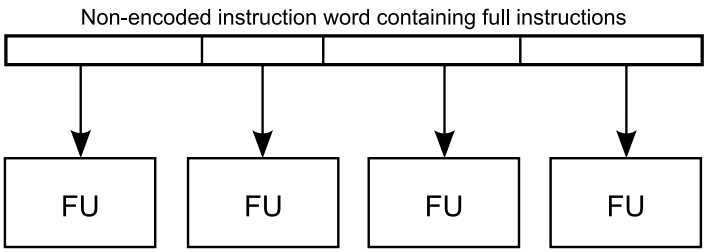


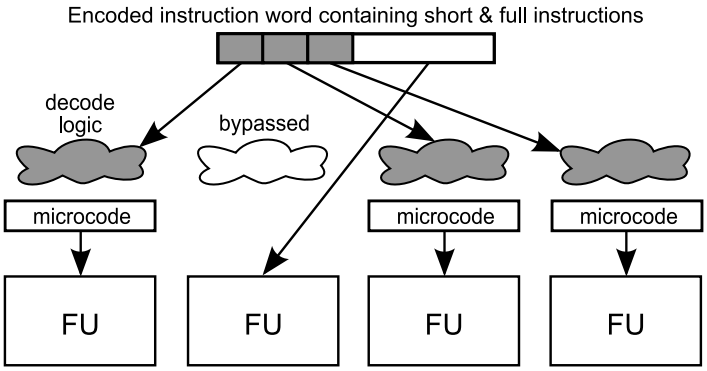
Figure 10.4: Encoded instruction word layout

Each short instruction within the instruction word is itself split into two sections: an "address" indicating for which FU the short instruction is intended, and the encoded instruction itself, as illustrated in Figure 10.4. Variable-width addresses are allocated in priority order of FU usage using a Huffman-type encoding so that heavily used FUs require fewer address bits and thus have more active instruction bits. This allows more instruction encodings to be allocated to higher priority units, and also enables the routing of each short instruction to the correct FU using simple equality comparison logic. The remainder of the short instruction is then translated to the target FU's microcode by look-up table decode logic within the FU, with decode mappings unique to each FU. There is at least one escape code instruction for each FU that indicates no entry is available for the desired setup pattern in the look-up table; in this case that FU will fetch the full instruction from the instruction word, bypassing the decode mechanism. In the case that the instruction word potentially contains multiple full instruction, a corresponding number of escape code instructions will indicate which full instruction should be fetched. Figure 10.5 (b) shows three FUs executing decoded short instructions and one FU bypassing the decode logic, executing a full instruction from the instruction word.

The key to ensuring that this approach achieves the desired goals of reduced area utilisation and energy consumption with no performance penalty, is effective selection of full instructions to be assigned to short instructions. The number of instructions can be varied for each individual FU, but the instruction value range is always aligned on a power of 2 boundary. The reason for this restriction is to ensure that when all short instructions from all FUs are combined within the instruction word, simple equality comparison hardware can be used



(a) Instruction layout without encoding



(b) Instruction layout with encoding

Figure 10.5: Comparison of instruction word formats

with the short instruction to select the correct individual FU, improving area and energy efficiency of the decode logic.

10.2.3 Instruction word encoding algorithm

The algorithm is implemented as follows: First an execution trace is generated by simulating the application program that has previously been compiled to run on the host processor, such as an ARM. This should be driven by a typical stimulus for the application and thus provide a representative profile to drive the short instruction allocations. The execution trace contains a list of activations for the code regions present within the target application. The functional units and instructions used within any particular code region can be determined from the microcode, therefore the list and frequency of instructions used for each functional unit can be derived from a combination of the activation trace and microcode.

Experimentation over a wide range of coprocessors and input stimulus has led to the conclusion that a short instruction width in the range of 8 to 11 bits wide is always optimal for

all coprocessors; any fewer is too restrictive on the number of short instructions giving little benefit, and any greater results in the decode hardware becoming too large and inefficient. Therefore a choice of between 256 and 2048 short instructions are available that can later be decoded to microcode. An overview of the algorithm flow is shown in Figure 10.6.

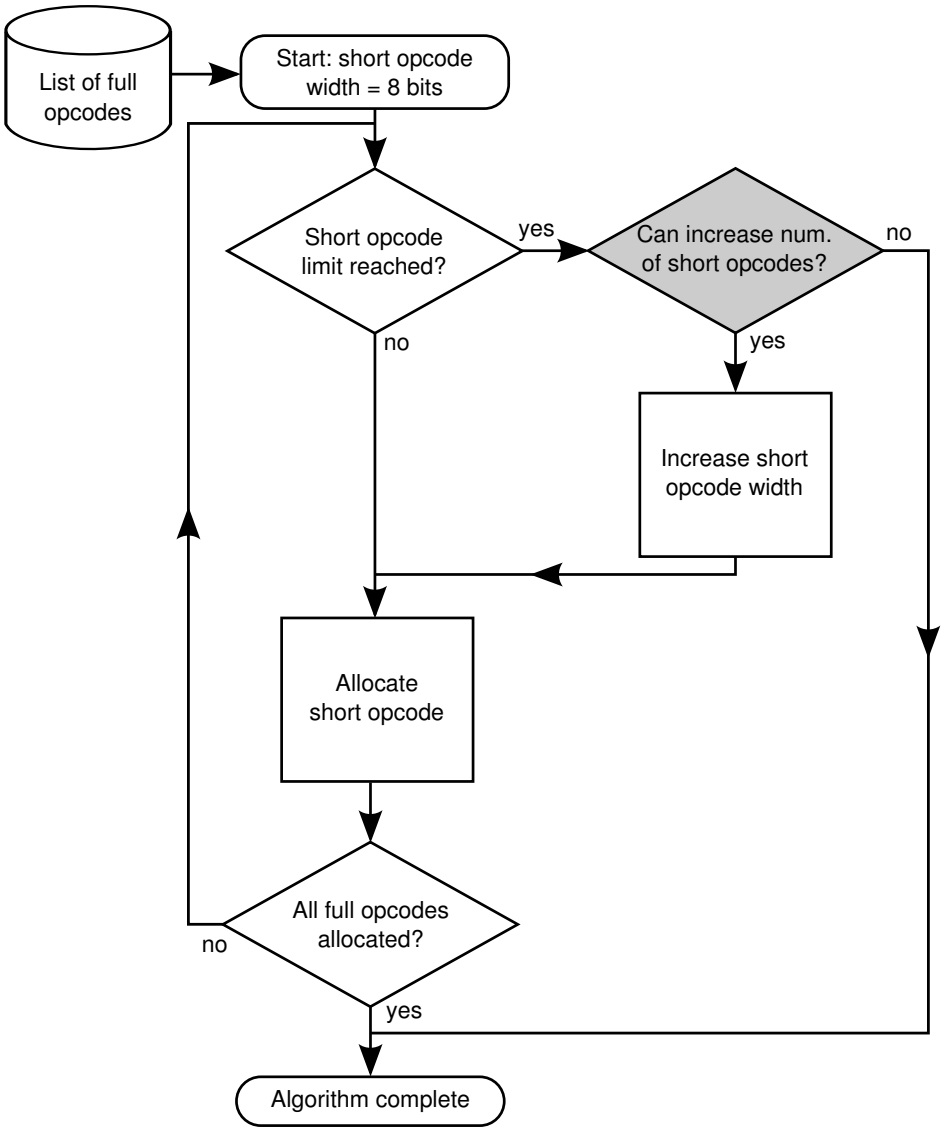


Figure 10.6: Opcode assignment algorithm flow (in instruction priority order)

The algorithm proceeds to iterate through a loop. Initially, short instructions are 8 bits wide, giving 256 slots available; this can be increased by the algorithm if deemed beneficial at a later stage. At least one short instruction is used as an escape pattern to allow a non-encoded full instruction to be executed, and the remaining short instructions are assigned to full instructions in priority order of the aforementioned list. Assignment of instructions progresses until either the list is exhausted, meaning all full instructions present in the list

have been assigned to a short instruction and this part of the algorithm is complete, or all available short instructions have been used.

In the latter case, the algorithm has to decide whether to increase the FU short instruction width by a single bit, doubling the number of short instructions available and thus allowing more full instructions to be assigned. If taken, this decision comes at a cost of increased instruction width and decode logic complexity, both of which negatively impact area and energy performance. This is a crucial decision in the algorithm, which is why it is highlighted in Figure 10.6. One of the key factors taken into consideration in this decision is how many full instructions will be efficiently encoded should the available number of short instructions be doubled by increasing the width by one bit. The aim is to expand the number of encoded instructions only when it will result in a significant reduction in the number of full instruction bypasses required, therefore resulting in a net energy reduction—that is, greater energy savings from the reduction in use of full instructions, compared with the additional energy consumed by the larger decode logic.

If the decision is taken to not increase the short instruction width, the algorithm is complete. Otherwise the algorithm proceeds to assign full instructions to the newly created short instructions in priority order as before, until either all full instructions have been assigned and the algorithm is complete, or all short instructions have again been used and another decision to extend the short instruction width is taken. If the short instruction reaches 11 bits wide (equivalent to 2048 instruction mappings including bypasses and NOPs) then it is not possible to increase the width any further, and the algorithm automatically completes with no further decision required.

When the instruction mapping is complete, hardware decode logic is created for each functional unit from the list of short instructions assigned to full instructions. This also incorporates the bypass mechanism for non-encoded full instructions. Application-specific processor RTL is then automatically generated, integrating the narrower instruction format and decode logic, and the executable code is recompiled by Cascade using the short instruction mapping logic. The result is a coprocessor with a narrower instruction path that is functionally identical to the original coprocessor from the software design perspective—Cascade hides all complexity involved in reducing the instruction width, in both the hardware and software domains.

Shown in Figure 10.7 is a truncated example of VHDL code created by Cascade incorporating the instruction decode mechanism implemented at the front of a functional unit. Each short instruction is decoded to the full microcode instruction that would otherwise have been stored in the instruction word had instruction encoding not been enabled. The full instruction is then transparently passed to the execution unit, so no other changes are required to the functional units to implement instruction encoding.

10.2.4 Experimental analysis

To examine the viability of this instruction encoding approach, analysis of the performance, area and energy consumption is undertaken for both the instruction cache and complete coprocessor created with encoded and non-encoded instruction formats. These experiments are based on benchmarks present in the MediaBench suite, the offloading of which onto Cascade coprocessors is detailed in chapter 5. To ensure a fair comparison, the area constraint, cache size restrictions, and effort levels are set at their defaults throughout all tests with the only change being the application of the encoding algorithm under evaluation during the second set of experiments.

Benchmarks are initially run through Cascade with no encoding of the instruction format. In this case, the word width is effectively unconstrained other than as part of overall coprocessor area constraints. This approach results in large variations in the word width as Cascade tries to optimise for performance within an area limit, meaning that the instruction word layout is strongly influenced by the peak level of parallelism extracted from the target function.

The instruction encoding algorithm is then enabled within Cascade, and new coprocessors are created for each of the benchmarks. Cycle-accurate simulations are run for both original and encoded instruction format coprocessors to get the number of cycles taken to complete the benchmark using the supplied MediaBench data sets. Instruction cache stalls are taken into account during this simulation, with estimated cache fill times based upon a typical external memory connected to an AMBA AHB bus [64].

Each coprocessor is synthesised to obtain area estimates using Design Compiler on a TSMC 130 nm process with Artisan cache memories. Gate-level simulations are run on Synopsys VCS to obtain switching activity information before performing gate-level power and energy

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity select_config is
   Port (
      -- 5 bit select giving 30 configurations, plus one
      -- select override and one NOP.
      select_i      : in std_logic_vector(4 downto 0);
      instruction_i : in std_logic_vector(31 downto 0);
      instruction_o : out std_logic_vector(31 downto 0));
end select_config;

architecture behavioral of select_config is

begin

   main_process: process (select_i, instruction_i)
   begin
      case select_i is
         -- Pass through input instruction (select override)
         when "00001" => instruction_o <= instruction_i;

         -- Encoded microcode setup patterns
         when "00010" => instruction_o <= X"638F32AE";
         when "00011" => instruction_o <= X"FB5674DA";
         when "00100" => instruction_o <= X"8C77B129";
         ...
         when "11101" => instruction_o <= X"2AF929E1";
         when "11110" => instruction_o <= X"556236A8";
         when "11111" => instruction_o <= X"98BE2138";

         -- Perform a NOP. Covers both intentional "00000"
         -- case and unlikely case of selection failure
         when others => instruction_o <= (others => '0');

      end case;
   end process main_process;
end architecture;
```

Figure 10.7: Instruction decode VHDL code excerpt

analysis using Power Compiler. A more detailed description of the power analysis flow used can be found in chapter 3.

With the instruction encoding algorithm enabled, cache area falls considerably in all except one test: JPEG encode. Further investigation reveals the reason that the instruction width doesn't fall significantly in this test—the original design has a narrow width of 128 bits, leaving little room for reduction of the instruction width while still allowing full instructions to be passed. However this appears to be an unusual exception. The largest saving in instruction cache size was achieved for MPEG2 decode, falling from a width of 320 bits to 104 bits, with a depth of 256 words in both cases. The average instruction width over all benchmarks without implementation of the encoding algorithm is 231 bits; with the encoding scheme enabled that drops significantly to an average of 94.5 bits.

On the other hand, the cache depth increases in some cases to compensate for the additional instructions required when the bypass mechanism is used for instructions that have not been encoded. The average instruction cache depth with the encoding algorithm disabled is 384 words; with encoding enabled the average depth rises to 496 words. Overall the reduction in width is much more sizeable than the increase in depth, resulting in the total cache memory size dropping from an average of 92.5K bits to 49.25K bits.

Overall coprocessor area falls to a lesser degree than cache area in all cases due to the additional decode logic placed in front of functional units reducing the benefit of the smaller cache to some degree—average gate count rises from 93.75K gates to 102.56K gates, an increase of just under 10%. Figure 10.8 shows the area of both the instruction cache and total coprocessor area, after application of the encoding algorithm, relative to the non-encoded case. For example, a value of 70% means there has been a 30% drop in area as a result of applying instruction encoding. In all cases overall synthesised area is lower after the application of instruction encoding, as the saving in cache area more than compensates for the additional decode logic.

Perhaps somewhat unexpectedly, performance also improves slightly in all cases even though this is not a primary goal of the algorithm—the key focus is on reducing area and energy, while endeavouring to have no detrimental effect on performance. Further examination reveals that reduced cycle counts are a welcome side-effect created by two factors: more efficient use of the instruction cache resulting in fewer instruction cache capacity stalls; and

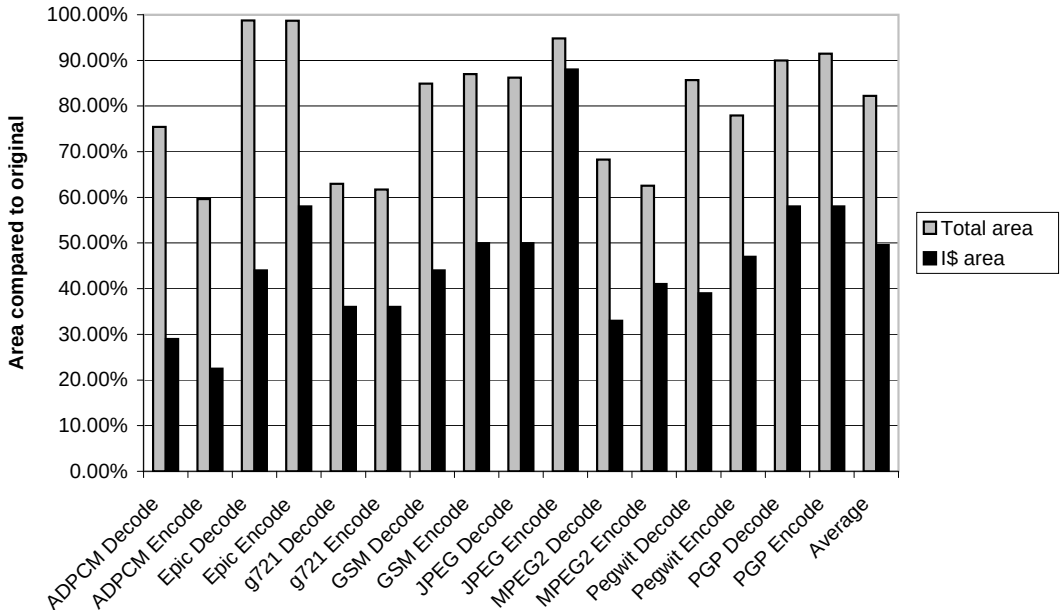


Figure 10.8: Coprocessor area using encoded instructions compared to base design

fewer cycles being required to fetch a narrower VLIW word from 32-bit main memory than are required to fetch a wider word, resulting in reduced stall cycles per cache miss.

As an example of this effect, the JPEG encode benchmark, which shows the largest instruction cache performance improvement, takes a total of 8,674,122 instructions to complete. In the original case 235,764 cycles are consumed due to instruction cache stalls (both cold start and capacity stalls). After implementing instruction encoding, the stall cycle count falls to 13,834—just 5.87% of the original count.

Figure 10.9 highlights the improvements in both instruction cache performance and overall cycles when instruction encoding is enabled. It should be noted that the small number of tests showing very large improvements, such as JPEG encode as discussed previously, are the result of small kernels that fit into the cache after applying the encoding algorithm. In such cases, almost all capacity stalls are eliminated, leaving only cold start stall cycles. However the number of instruction stalls is often a small proportion of the overall cycle count, therefore the large reduction in instruction stalls is not significantly reflected in overall cycle count reduction.

Observation of the Epic benchmarks reveals what appears to be an anomaly—the total cycle count falls further than the reduction in instruction cache stall cycles. A more thorough

examination of the results reveals that the reason for this unexpected performance improvement is due to the original instruction cache width being limited by the default area constraint within Cascade, reducing the peak parallelism that can be exploited by the coprocessor. The reduction in instruction width enabled by the encoding algorithm allows more simultaneous instructions to be issued from the cache while remaining within the area constraint.

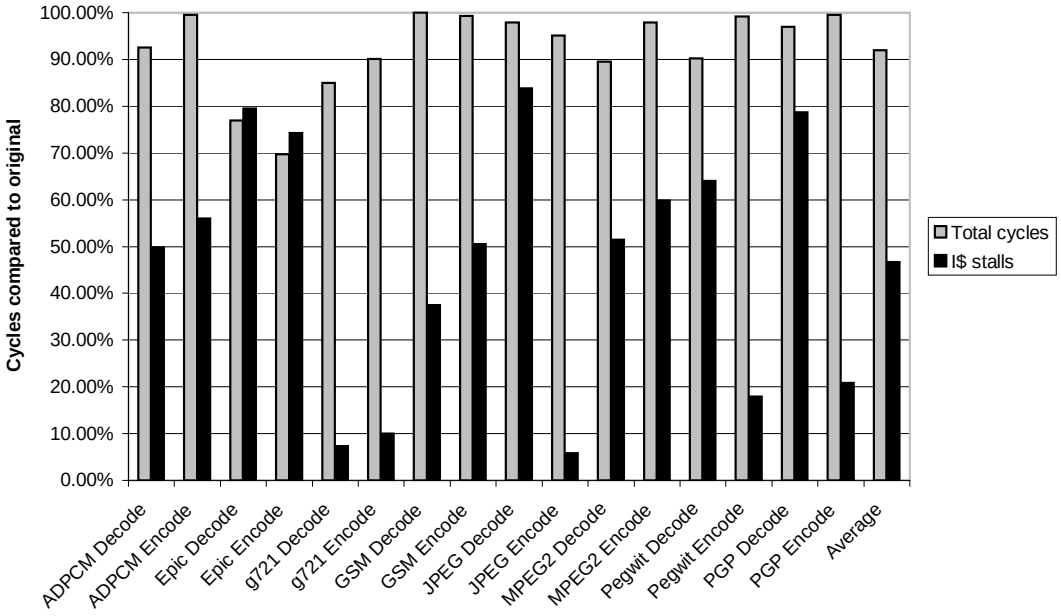


Figure 10.9: Coprocessor cycle counts using encoded instructions compared to base design

Overall energy reduction is observed in all examined tests, driven by the substantial savings in energy consumed by the instruction cache. There is a large variance between tests in the proportion of energy consumed by the instruction cache compared to the overall energy consumption for the entire processor. This is reflected in the results presented in Figure 10.10, highlighting the varying influence that a sizeable reduction in cache energy has on the overall processor energy performance. The average energy consumed over the tests is just over 80% of the original energy, a saving of almost 20%. Energy consumption of the instruction cache itself drops to 42.8% of the original value using instruction encoding.

It is important to note that the overall energy reduction is much lower than that saved in the instruction cache because the encoding mechanism introduces an additional energy consumption element with the look-up tables required to decode the encoded instructions. This is traded off against the energy saved in the instruction cache, and in all examined cases results in an overall energy saving, albeit a small saving of less than 5% in some less favourable tests. However, the algorithm does not guarantee an energy saving, meaning that in some cases the energy consumed by the decode logic may exceed the savings in the cache result-

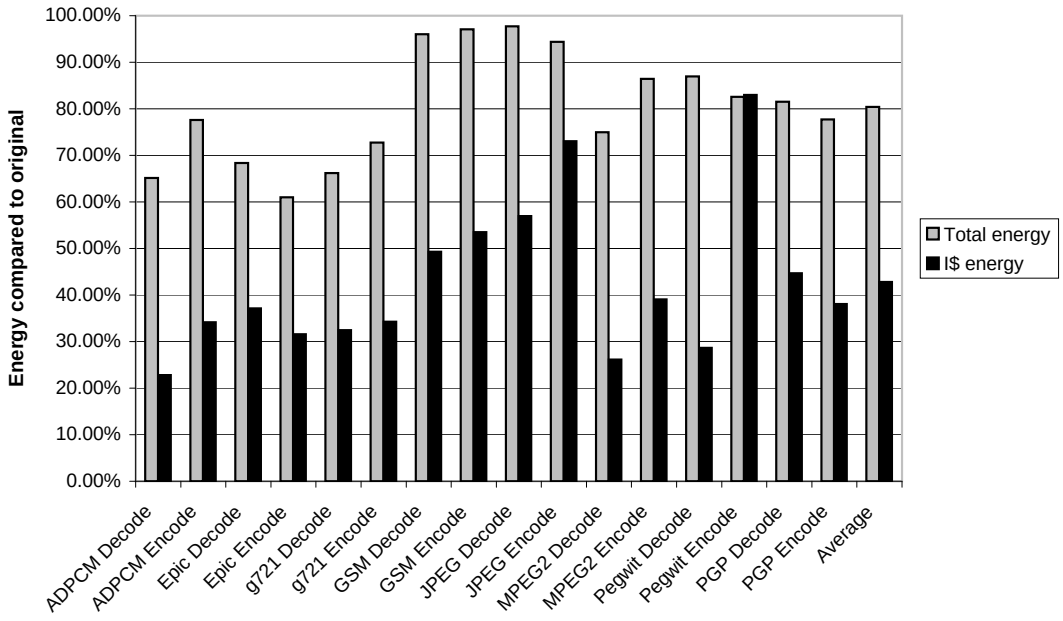


Figure 10.10: Coprocessor energy use using encoded instructions compared to base design

ing in a net energy consumption gain. Therefore it is currently necessary to carefully monitor results to ensure the instruction encoding scheme is providing favourable results.

A brief examination of leakage power was undertaken for the coprocessors both before and after enabling instruction cache compression. Leakage power was found to have fallen in all tests, primarily as a consequence of the smaller instruction cache. The average leakage power without applying instruction encoding is 878.16 μW, whereas it drops to 758.15 μW with the application of the encoding scheme—a reduction of 14%. Although the reduction is small compared to that observed for dynamic power, leakage power becomes far more significant at 90 nm and smaller process technologies, as explained in chapter 9.

Overall system energy consumption is likely to be further improved beyond that observed in the coprocessor itself, as this technique reduces the amount of system memory required to store instructions, and similarly a corresponding reduction in memory bus traffic will be observed. The total microcode size for all MediaBench tests is 274,056 bytes without instruction encoding, and 161,476 bytes with encoding. Thus, the microcode stored in main memory is reduced in size by 41% on average; a similar reduction in bus traffic due to instruction transfers is observed.

These factors are not taken into account in the results presented in this section, as they are dependent on the configuration of the system external to the coprocessor, and as such are

generally non-deterministic at the RTL level where Cascade operates. Applying instruction encoding will never result in an increase in bus traffic; therefore it can safely be assumed that system level energy external to the coprocessor will always fall regardless of the system configuration, meaning that this approach will offer additional energy savings on top of those presented in this section.

10.3 Idle and sleep modes

Wasteful dissipation of energy during non-active coprocessor cycles can result in significant increases in the overall energy budget for the coprocessor. This is largely dependent upon the proportion of time the coprocessor spends waiting on input from the host; in cases where this proportion is high there is a pressing requirement to reduce such energy wastage.

The coprocessor maintains an internal state to maintain, among other attributes, whether it is *sleeping*, *waking*, *running*, *stalled*, etc. During the sleeping state, the coprocessor checks all inputs coming over the bus for a wake-up signal; any other data on the bus is ignored. Thus only a small proportion of the coprocessor needs to be active to detect this signal, the presence of which can then activate the rest of the coprocessor.

A standard coprocessor implementation simply ignores any toggles on the input bus during the sleeping state, with the exception of the wake-up signal described previously. However, analysis of the continuous power consumed during this sleeping state, with all parts of the coprocessor essentially still active but stalled, has shown it to be a very inefficient approach. Table 10.3 shows a breakdown of the power consumption of a typical coprocessor implemented in TSMC 130 nm process technology and running at 10 MHz. The results are listed for a fully operating coprocessor, and the same coprocessor in idle (stalled) mode.

	Running power	**Idle power**
Cell internal power	3.918 mW	290.75 µW
Net switching power	569.39 µW	299.71 µW
Total dynamic power	4.487 mW	590.46 µW
Cell leakage power	703.51 µW	703.51 µW

Table 10.3: Idle power comparison

Although dynamic power has fallen to just over 13% in the idle state compared to dynamic power consumed during normal operation, this is still a significant amount of power to consume in return for no useful work. Table 10.4 shows what proportion of energy would be consumed by the coprocessor in both active and idle cycles, based on the proportion of execution time that is spent idle. It can be seen that as long as the coprocessor utilisation is kept high, energy wasted during idle cycles is not a hugely significant issue. However, at lower levels of utilisation, the time spent idling can consume more energy than that used for performing useful computation.

% execution time spent idle	**% energy attributable to**	
	active cycles	**idle cycles**
10	98.56	1.44
20	96.81	3.19
30	94.66	5.34
40	91.93	8.07
50	88.37	11.63
60	83.51	16.49
70	76.51	23.49
80	65.51	34.49
90	45.78	54.22

Table 10.4: Proportion of energy consumed in active and idle states

This poses a particular problem for coprocessors used in a blocking configuration—that is, the coprocessor runs only under the control of the host processor, causing the host to block while the coprocessor is executing. As a result, there may be extended periods when the host processor is executing code that has not been offloaded, during which time the coprocessor will be stalled. Although this is not a desirable scenario from a performance perspective, it may sometimes be necessary to implement such a system based on the requirements of the whole system of which the coprocessor is only one component, and as such the coprocessor energy consumption must be minimised during those stalled periods.

An ideal solution to the problem of wasted energy during stalled cycles is complete power gating of the coprocessor. This technique involves additional logic that controls the power supply to the coprocessor, effectively isolating the power supply and reducing power consumption (including leakage power) to zero during periods of inactivity [89]. Power gating, however, is an architectural solution that should be implemented as part of RTL to gate-level synthesis or at lower levels of abstraction, therefore it is not suitable for implementation into coprocessors at the RTL level.

Halting the master clock signal to the coprocessor is an alternative approach that can be implemented at the RTL level, while providing much of the benefit of power gating. The key drawbacks are that it has no effect on leakage power, reducing the effectiveness at smaller process technologies, and that care must be taken to avoid corruption of any data stored in the coprocessor. In addition, if the coprocessor input pins continue to toggle, some dynamic power will continue to be dissipated by the coprocessor.

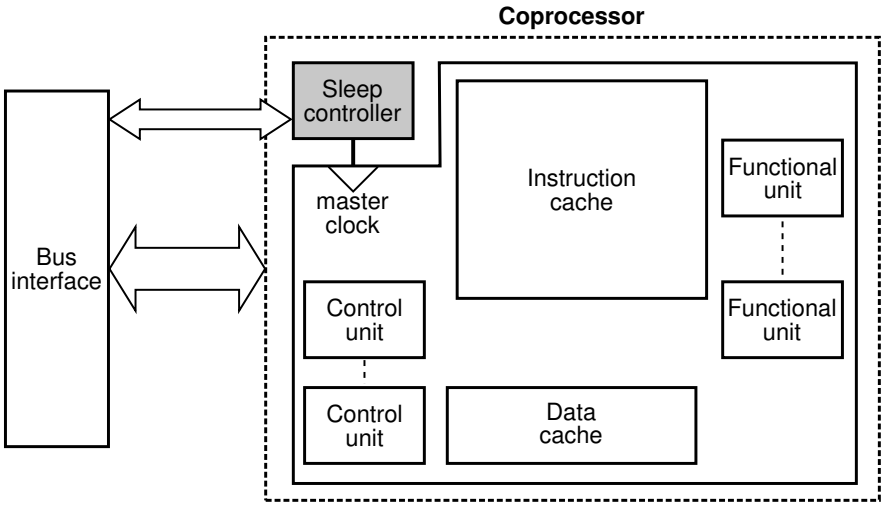


Figure 10.11: Coprocessor with sleep controller to reduce idle energy

Figure 10.11 shows one method of implementing a sleep controller that is used to reduce the dynamic power consumption to near zero during idle cycles. The sleep controller unit remains active at all times, and can be controlled using instructions within the coprocessor microcode to tell the controller to place the coprocessor into sleep mode or wake it up. It does this by halting the main clock signal into the coprocessor, and optionally also masking any input signals, although the latter comes at the cost of increased logic area and power consumption during active cycles, due to the additional masking logic required. The advantage of this approach is that entry to sleep mode is software controlled, therefore the number of instances where the coprocessor enters sleep mode for only a very short period (thus conveying no benefit) can be minimised via coprocessor usage pattern analysis at the instrumentation stage.

The coprocessor sleep mode detailed here comes at a low area cost, however the savings available from it are overshadowed by those available from implementing a full shut-down of the coprocessor at the architectural level. As leakage power continues to form an increasing proportion of the overall power and energy budget with newer process technologies, the limitations of an RTL-level clock halting sleep mode are clear. Therefore this feature is

not enabled by default as part of the coprocessors generated by Cascade, allowing more sophisticated techniques, that may clash with RTL level functionality, to be implemented at lower levels of the design process.

10.4 Summary

The aim of the work in this chapter is to improve the overall power and energy performance of coprocessors generated by Cascade, in some cases at the cost of a higher area requirement, in others with a simultaneous improvement in area. This has been achieved by focusing on components that typically contribute a high proportion of the coprocessor's energy consumption (multiplier and cache units), as well as making modifications to reduce energy wastage during idle cycles.

A relatively simple improvement to the control of the multiplier enable signal produced an energy reduction of 36% within the multiplier, which manifests as a 5% overall saving in the coprocessor. Further savings are possible by gating the input signals to the multiplier, although this technique must be applied with care as the additional gates consume energy during active cycles, therefore any improvement is dependent on the proportion of active and inactive cycles.

The most substantial savings are found in improvements to the instruction cache, due to the inherently inefficient nature of the VLIW instruction layout. The work in this chapter resulted in an average 57% reduction in instruction cache energy, and 47% reduction in area. These contribute an overall coprocessor energy saving of 20% and area saving of 18%. The results of this work were published at the 2007 Design Automation Conference; the paper is listed in Appendix L.

Idle power consumption was analysed, and it was discovered that coprocessors that spend a large proportion of time in the idle (stalled) state can, in some cases, consume more energy during those idle cycles than are consumed doing useful work. Several solutions were considered, and one in particular—using a sleep controller to halt the master clock signal to the coprocessor—was implemented. After the initial analysis was carried out, it was decided to avoid implementing an RTL level sleep controller, due to the inability to comprehensively tackle leakage power at this level of abstraction. With the increasing dominance of leakage

power during idle cycles when using smaller process technologies, an architectural solution is required. Therefore the best solution is to ensure RTL coprocessors are amenable to complete power gating during sleep cycles, which is best served by eliminating RTL-level sleep circuitry.

11. Physical layout and place & route

Analysis of power and energy performance at high levels of abstraction, such as RTL or gate level, provides results orders of magnitude faster than analysis at the physical level. However, accuracy suffers due to loss of detail, and assumptions made at higher abstraction levels. To obtain accurate power and energy consumption estimates it is important to perform analysis at lower levels for the purposes of comparison with higher level results, allowing an error margin to be established and improvements in analysis results to be back-annotated into the higher level analyses. In addition, more accurate area and timing figures can be derived from a post-layout design which can also be used to improve RTL level estimation accuracy.

This chapter details both the steps required to perform a physical layout (including floorplanning) and place & route on a gate-level coprocessor, along with power analysis of the physical-level coprocessor.

11.1 Physical layout using Synopsys Astro

Synopsys Astro platform tools, including Astro, Milkyway, JupiterXT and Physical Compiler are designed to work with the Synopsys front-end tools, such as Design Compiler and VCS, used in the earlier stages of this project. The key foundation of the process is detailed in the Synopsys' Recommended Astro Methodology application note [99]. JupiterXT contains script generation capabilities that enable much of the physical layout process to be automated, with manual intervention required only to modify values within script files and to perform graphical floorplanning [100]. After floorplanning, macro placement and power planning, the design is transferred to Physical Compiler for placement and optimisation. JupiterXT's Virtual Flat Flow is used throughout this section.

11.1.1 Library creation

The first step in the Synopsys back-end flow is creating a new library containing the modules from front-end design. This requires Library Exchange Format (LEF) files for the target technology library (in this case TSMC 130 nm), standard cell macros, and any hard macros used in the design. It is strongly advisable to create a map file indicating which layers in the technology library LEF file should be mapped to which layers in the Milkyway library. Synopsys provides a Perl script, `lef_layer_tf_number_mapper.pl` that examines the technology and LEF files, automating the creation of this layer mapping file.

It is then necessary to convert LEF files to Synopsys' physical library (PLIB) format. This is done using the following command (paying particular attention to the order in which LEF file arguments are passed):

```
lef2plib -lib <library_name>
         -output <plib_filename>
         -lef <technology_library>.lef
         -lef <cell_macros>.lef
         -lef <memory_macro_1>.{lef,vclef}
         -lef ...
         -lef <memory_macro_n>.{lef,vclef}
```

Before any of the aforementioned technology and library files can be used within JupiterXT, they must be combined into a Milkyway library that can later be referenced from JupiterXT, in accordance with the Milkyway Environment Data Preparation User Guide [101]. An overview of the flow for creating a Milkyway library is shown in Figure 11.1.

The particular steps taken in this case involve creating a new library from within the Astro tool, and specifying the technology file that forms the basis of this library. The LEF files can then be read into the library using the `read_lef` scheme command. This allows technology and cell LEF files to be specified, along with the previously generated layer mapping file. The `read_lib` command is then used to specify reference libraries for the target technology under various operating conditions—in this case fast, typical and slow .db files. Once this operation has been completed the Milkyway reference library is available to be read from within JupiterXT. The entire library creation process has been automated in a script listed in Appendix H.1.

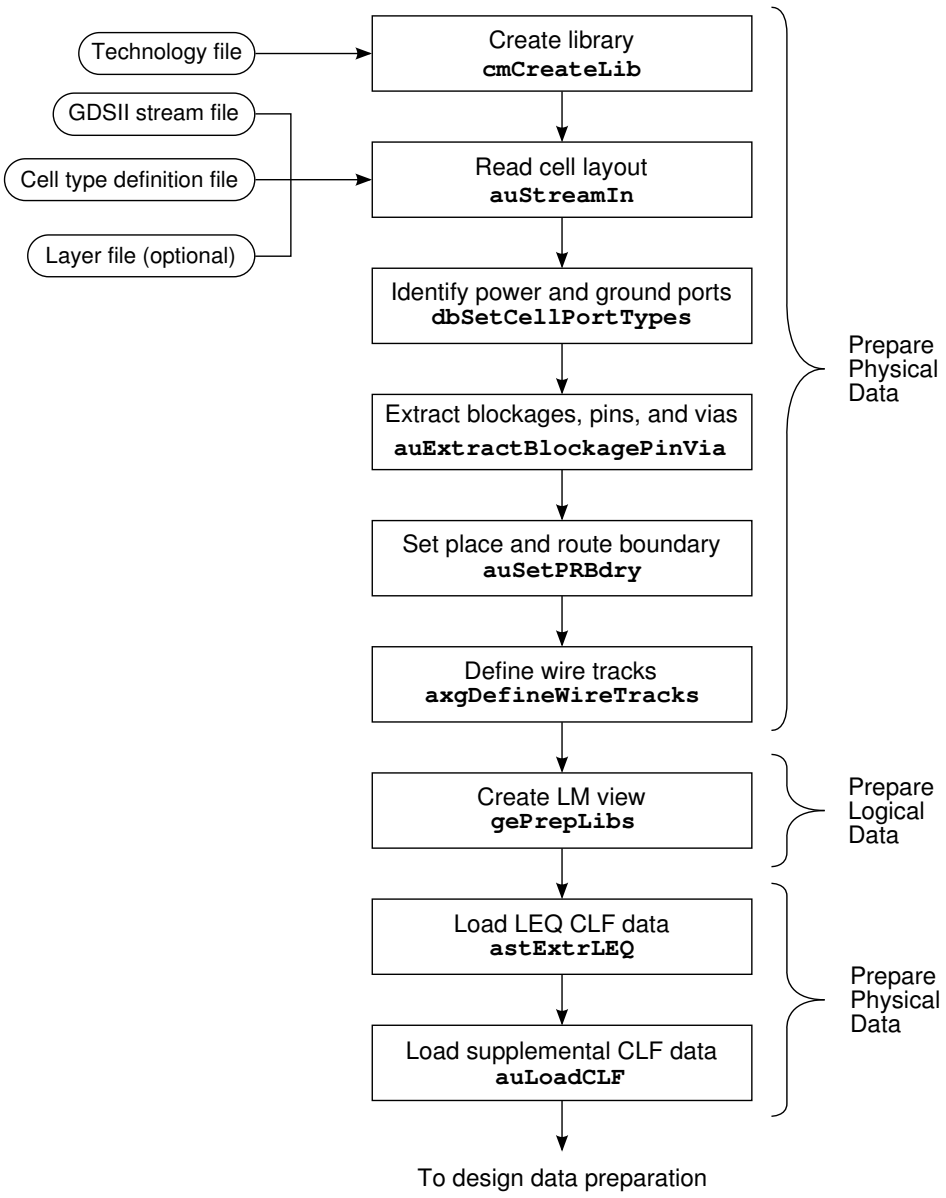


Figure 11.1: Synopsys Astro platform Milkyway library creation [102]

11.1.2 Floorplanning

Floorplanning can then be done in JupiterXT. This involves creating a new library from the synthesised Verilog netlist file using the `auVerilogToCell` command within JupiterXT. The Verilog file is specified along with the technology file and previously created Milkyway reference library. Upon completion, the new library and cell can be opened and the standard design constraints (`.sdc`) file loaded. Power and ground nets are connected to `VDD` and `VSS` respectively, after which the floorplan can be specified. Initial setup of the floorplan parameters, such as core utilisation, aspect ratio, and macro placement, completes successfully.

However beyond this stage the visible layout does not appear to be correct, and attempting to complete the flow further proves problematic, occasionally resulting in the tool spontaneously exiting. The script used to perform floorplanning up to this point can be found in Appendix H.2.

Although it was originally intended that the entire back-end layout and place & route would be performed using Synopsys tools, the aforementioned issues arose while attempting to perform layout using JupiterXT. It is suspected that the problem may originate from the Milkyway reference libraries containing TSMC memory macros, although this cannot be confirmed and there is no known workaround. After significant time was spent attempting to overcome these problems, it was decided to instead concentrate back-end layout and place & route efforts on Cadence's Encounter platform.

11.2 Physical layout using Cadence Encounter

Floorplanning, placement and routing are carried out using Cadence's First Encounter Ultra platform. RTL Compiler Ultra, a VHDL and Verilog synthesis tool, is included as part of the Encounter platform, however as this tool essentially overlaps Synopsys' Design Compiler (which is used throughout this project) it will be largely unused, with synthesised netlist input provided by Design Compiler. Many of the commands used with Design Compiler have similar equivalents in RTL Compiler [103]. An overview of the physical layout flow used in First Encounter is shown in Figure 11.2.

11.2.1 Initial configuration

This section uses a previously synthesised coprocessor design targeted at a benchmark from the MediaBench suite [74], specifically the PGP encryption algorithm in encode mode. Further details of the MediaBench suite, and the process of offloading functions onto coprocessors, can be found in chapter 5. The coprocessor was created with instruction width optimisation enabled within Cascade, as detailed in section 10.2. Before beginning the First Encounter flow, it is necessary to ensure that all required files have been prepared.

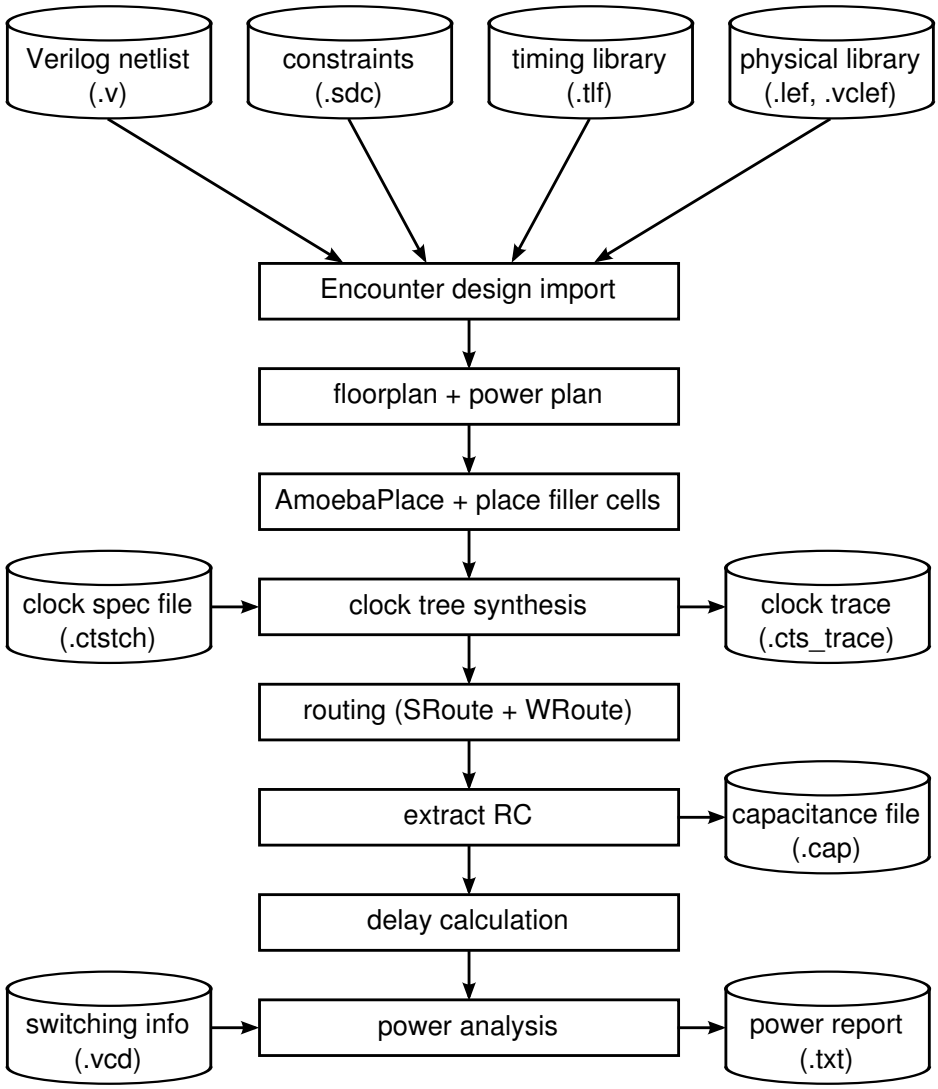


Figure 11.2: First Encounter physical layout flow

The process is detailed in the Encounter User Guide [104]; a summary of the required files for this project are listed below:

- Technology information files for cells and macros in Library Exchange Format (LEF)
- A synthesised design netlist in Verilog format
- Design constraints for the netlist in Standard Design Constraint (SDC) format
- Standard cell timing information in Timing Library Format (TLF)

LEF files required for a particular design depend on the technology used and the presence of any hard macros in the design. The physical process information is provided in a LEF file—in this example TSMC's 130 nm, 8-layer metal process information is provided in the file *tsmc13fsg_8lm_tech.lef*. Similarly, TSMC 130 nm standard cell macro information is

provided in *tsmc13nvt_macros.lef*. Additionally, any hard macros present in the design must also have a corresponding LEF file, since they cannot be built from the leaf cells present in the standard library macros LEF. Cascade coprocessors typically have several hard macro memory blocks created using the Artisan memory generator tool, which can create VCLEF files for each memory block. VCLEF is a subset of LEF, therefore with some minor modifications they can be used with First Encounter. In this example case, hard macro information is provided by the files *wr_wr_s_s_8192x32.vclef*, *sp_rw_s_instr256x104.vclef* and *sp_rw_s_16x48.lef*; respectively these represent the data cache, instruction cache and link memory instantiated in the coprocessor.

The netlist of the design is usually an unmodified Verilog output from a synthesis tool such as Design Compiler or RTL Compiler. One potential issue is that all instantiated cell types in the netlist must be unique. This can be achieved by running the `uniquify` command from within Design Compiler or RTL Compiler before writing out the netlist.

An SDC file provides constraints such as operating conditions, wireload models, clocks and input/output delays. Although not strictly necessary as an input file to First Encounter, results are improved if this file is included. It can be easily created from within Design Compiler or RTL Compiler using the command `write_sdc` after synthesis.

Timing information for standard cells is provided by the TLF file. This is usually provided by the technology vendor, and indeed TSMC provide the relevant TLF files for their 130 nm process. However, the provided files conform to TLF version 4.1, which is largely incompatible with First Encounter version 5.2 (requiring TLF files to be version 4.3 or newer). This problem can be easily rectified by converting Synopsys Liberty (.lib) format files into TLF using the *syn2tlf* tool provided by Cadence. This tool generates version 4.4 TLF files, which cause no problems with First Encounter.

Initial design import is controlled using a configuration file, which directs First Encounter to the necessary files listed above, as well as defining some settings in advance of floorplanning and layout taking place. For example, the power and ground nets are set to VDD and VSS respectively, representing the node names within the TSMC technology file, cells and macros. The configuration file used is shown in Figure 11.3.

```
##############################################
#                                            #
#  FirstEncounter Input configuration file.  #
#                                            #
##############################################

# Created by First Encounter v05.20-p002_1
# Modified to match correct macro files by paulm

global rda_Input

set cwd /crux/paulm/encounter_pgp_encode

set rda_Input(import_mode) {-treatUndefinedCellAsBbox 0 \
                            -keepEmptyModule 1 -useLefDef56 1 }
set rda_Input(ui_netlist) "pgp_encode_synth.v"
set rda_Input(ui_netlisttype) {Verilog}
set rda_Input(ui_settop) {1}
set rda_Input(ui_topcell) {test_copro}
set rda_Input(ui_timelib) "typical.tlf"
set rda_Input(ui_timingcon_file) "pgp_encode.sdc"
set rda_Input(ui_leffile) "tsmc13fsg_8lm_tech.lef tsmc13nvt_macros.lef \
                           wr_wr_s_s_8192x32.vclef sp_rw_s_instr256x104.vclef \
                           sp_rw_s_16x48.vclef"
set rda_Input(ui_core_cntl) {aspect}
set rda_Input(ui_aspect_ratio) {1.0}
set rda_Input(ui_core_util) {0.5}
set rda_Input(ui_isHorTrackHalfPitch) {0}
set rda_Input(ui_isVerTrackHalfPitch) {1}
set rda_Input(ui_ioOri) {R0}
set rda_Input(ui_isOrigCenter) {0}
set rda_Input(ui_delay_limit) {1000}
set rda_Input(ui_net_delay) {1000.0ps}
set rda_Input(ui_net_load) {0.5pf}
set rda_Input(ui_in_tran_delay) {0.0ps}
set rda_Input(ui_pwrnet) {VDD}
set rda_Input(ui_gndnet) {VSS}
```

Figure 11.3: First Encounter input configuration file

11.2.2 Floorplanning

Once the netlist, constraints and library files have been loaded, floorplanning can commence. Several parameters need to be set at this stage, which define the basic structure of the chip. These are summarised in Table 11.1.

Parameter	**Value**
Aspect ratio	1.0
Core utilisation	0.5
Core to IO boundary	15 µm
Row spacing	0 µm

Table 11.1: Floorplanning parameters for coprocessor in 130 nm technology

Each of the values in Table 11.1 has to be carefully selected to ensure that the design is both feasible and area efficient. For most designs a square aspect ratio of 1.0 provides the most efficient use of die area (unless there are considerations due to macro blocks or external blocks that are added to the design at a later stage). Core utilisation determines what proportion of the core will be allocated to cell placement, with the remainder left available for routing. This value plays a significant role in the ability to successfully route a design, and also the interconnect length distribution (ILD) of routed interconnects [105]. A degree of trial and error is required to find an optimal value for core utilisation, typically a more interconnect dominated design will require a lower value. Similarly, core to IO boundary provides routing area for connections to IO pads, and needs to be adjusted for the connectivity requirements of each design. Row spacing allows gaps to be inserted between each standard cell row, a feature that is not necessary for this design.

11.2.3 Power planning

Power planning is performed to supply power to standard cells and macros. The first step is to add power rings, which are placed around the perimeter of the core and supply power to the stripes, which carry power across the chip. A separate ring is used for power (VDD) and ground (VSS). For the TSMC 130 nm library, higher numbered metal layers are thicker making them more suited to carrying power rings, so METAL7 is used for the horizontal parts of the power ring, and METAL8 is used for vertical parts. Rings are centred in channel,

and widths are set to 2 µm with a spacing of 1 µm, based on values in the LEF technology file for the selected metal layers. Power stripes are added on METAL2, with width and spacing half that of power rings (1 µm and 0.5 µm respectively). The stripes are placed at a distance of 100 µm apart; this value is a trade-off between providing sufficient distribution of power lines while minimising routing blockage caused by the space consumed by the power lines, and again can be optimised through trial and error.

Normally at this stage automatic floorplanning can be performed by First Encounter, during which the tool will perform a detailed heuristic analysis in an attempt to determine the most efficient layout for macro cells. Unfortunately, the licence available (First Encounter Ultra) does not include automatic floorplanning functionality. Therefore a quick manual floorplan is performed instead, which retains a large degree of freedom for layout operations to be carried out at a later stage.

11.2.4 Macro placement and clock tree synthesis

With the chip layout and power supply in place, standard cell and macro placement can begin. First Encounter's timing-driven amoeba placement is initiated with high effort level. Once completed, standard cells and macro blocks that implement the netlist functionality are placed at appropriate locations within the core.

The clock tree supplying all cells needs to be synthesised as it is such a complex net requiring balancing using buffers. This is performed using automatic clock tree synthesis (CTS) functionality built into First Encounter. To direct CTS, a *coprocessor.ctstch* file is provided to the tool, the contents of which are listed below:

```
# Clock Synthesis File

AutoCTSRootPin    clk_i
NoGating          NO
MaxDelay          5ns
MinDelay          4ns
MaxSkew           200ps
MaxDepth          20
Buffer            CLKBUFX2 CLKBUFX4  \
                  CLKBUFX8 CLKBUFX12 \
                  CLKBUFX16
End
```

After the clock tree has been synthesised, it can be routed as part of a standard routing algorithm. Before routing, filler cells are added to complete any gaps in the physical layout. Doing so provides decoupling capacitance and completes power and ground connections to standard cells. Special routing is then performed on the power and ground nets, VDD and VSS. Once complete, global and final route can be performed on the entire design using WRoute. NanoRoute is a newer algorithm that would be preferable to WRoute, however the First Encounter Ultra licence does not cover the use of NanoRoute.

The complete design has now been placed and routed, with the final step being to connect power and ground nets to their respective pins. This is performed by issuing the following two commands:

```
globalNetConnect VDD -type pgpin -pin VDD -all -override
globalNetConnect VSS -type pgpin -pin VSS -all -override
```

A Tcl script has been written to automate the entire flow as described in this section; it can be found in Appendix H.3.

11.2.5 Post-layout analysis

Analysis can now be undertaken on the design, with a higher level of accuracy and detail than that provided by the gate-level flow described in chapter 3. In order to perform accurate power analysis, switching activity information is required. This is already available in SAIF format, used as part of the Synopsys Power Compiler gate-level analysis in section 3.4.

First Encounter does not recognise SAIF files, so it is necessary to recreate switching activity in Value Change Dump (VCD) format. VCD can be created from within Synopsys VCS, therefore the simulation method described in section 3.3 can be re-used with some minor changes to the Verilog testbench files to instruct VCS to dump VCD output instead of SAIF. At the start of the testbench, the command `$dumpvars;` is issued, and all SAIF commands are removed. VCS is called with the additional switch `-vcd coprocessor.vcd`, which instructs the simulator to dump all toggles into the VCD file.

Once the VCD file has been generated, the following command can be issued from within First Encounter to perform a power analysis based on the VDD net within the post-layout design:

```
updatePower -vcd coprocessor.vcd -vcdTop copro\_testbench/copro    \
            -noRailAnalysis -postCTS -report power\_report.txt VDD
```

This operation is usually quite time consuming, especially when a large VCD file is used. In this example case the VCD file is 37 GB for 30 ms of simulation time, highlighting that the VCD format is not particularly well suited to long simulation runs on complex designs. The VCD file can be compressed with `gzip` and First Encounter will automatically decompress it as required, but the trade-off is a further increase in run time. Once completed, First Encounter produces an output report as shown in Figure 11.4.

```
##############################################
# The Power Analysis Report for VDD net      #
##############################################
power supply: 1.2 volt
average power between 0.0000e+00 S and 3.0000e-02 S
Total id in vcd file: 506272
    In module copro_testbench/copro valid id: 294362
         redundant id: 40718
    In module copro_testbench/copro invalid id: 118260
        redundant id: 36269
Total activity in vcd file: 5.90059e+09
    In module copro_testbench/copro valid activity: 5.14162e+09
    In module copro_testbench/copro invalid activity: 7.57257e+08
average power(default): 1.9719e+00 mw
    average switching power(default): 9.5497e-01 mw
    average internal power(default): 7.8880e-01 mw
    average leakage power(default): 2.2809e-01 mw
    user specified power(default): 0.0000e+00 mw
average power by cell category:
    core:  1.8905e+00 mw
    block: 8.1184e-02 mw
    io:    0.0000e+00 mw
biggest toggled net: fu_multiplier64_0/enable
    no. of terminal: 785
    total cap: 3.5747e+03 ff
```

Figure 11.4: Post-layout power analysis report

Total area of Standard cells	6931843.817 μm^2
Total area of Macros	1469033.134 μm^2
Total area of Blockages	0.000 μm^2
Total area of Pad cells	0.000 μm^2
Total area of Core	8272207.388 μm^2
Total area of Chip	8446693.253 μm^2
Effective Utilization	1.0514e+00
Number of Cell Rows	779
% Pure Gate Density #1 (Substracting BLOCKAGES)	83.797%
% Pure Gate Density #2 (Substracting MACROS)	101.891%
% Pure Gate Density #3 (Substracting MACROS & BLOCKAGES)	101.891%
% Core Density (Counting Std Cells and MACROs)	101.555%
% Chip Density (Counting Std Cells and MACROs and IOs)	99.458%

Table 11.2: Floorplan and placement area report

The results of post-layout power analysis are particularly interesting when compared with those provided by gate-level analysis in section 3.4. For the same design analysed using the same input stimulus, operating conditions and technology library, the results from gate-level analysis using Synopsys Power compiler are shown in Figure 11.5. For comparison, the key results have been extracted from Figures 11.4 and 11.5, and are shown below. The upper entries represent post-layout power, and the lower entries gate-level power.

```
average power(default)          : 1.9719e+00 mw
average switching power(default): 9.5497e-01 mw
average internal power(default) : 7.8880e-01 mw
average leakage power(default)  : 2.2809e-01 mw

------------------------------------------------

Cell Internal Power             =     5.5776 mW
Net Switching Power             =   858.9359 uW
Total Dynamic Power             =     6.4365 mW
Cell Leakage Power              =   948.7441 uW
```

The results from post-layout average power analysis are clearly significantly lower than those produced by gate-level analysis. Although a higher accuracy is expected from analysis at a lower level of abstraction, the proportional difference is much larger than would normally be expected. Therefore it is necessary to examine both results in more detail to determine whether there are any discrepancies in each analysis method that may have skewed either one

or both sets of figures. For example, the post-layout analysis summary may have excluded certain types of cell or other power sources.

```
Design        Wire Load Model            Library
------------------------------------------------
test_copro             tsmc13_wl10       typical


Global Operating Voltage = 1.2
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  =   5.5776 mW   (87%)
  Net Switching Power  = 858.9359 uW   (13%)
                       ---------
Total Dynamic Power    =   6.4365 mW  (100%)

Cell Leakage Power     = 948.7441 uW
```

Figure 11.5: Gate level power analysis report

One known issue with the gate-level analysis figures is that of hard macro memory cells. These are effectively black boxes at gate level, so produce very coarse figures that are often higher than the actual consumption figures. Examining the detailed power report shows the average power values attributed to macro cells:

```
------------------------------------------------------------------------
                              Switch   Int      Leak     Total
Hierarchy                     Power    Power    Power    Power    %
------------------------------------------------------------------------
test_copro                       0.859    5.578 9.49e+08    7.385 100.0
------------------------------------------------------------------------
    ex_access_st_1r_0 (ex_access_st_1r_0_6_1_5_3)
                                 0.142    0.751 6.17e+08    1.510  20.5

    Inst_cu_direct_inst_cache (cu_direct_inst_cache_8_224_32_3_0_8)
                              7.74e-02    1.510 1.89e+08    1.777  24.1
```

If the total power figures for both macro blocks are subtracted from the overall coprocessor total power, the remaining figure is 4.098 mW. This is still significantly higher than the

2.2 mW total average power figure, including leakage power, obtained from post-layout analysis. The switching power figure is similar at both levels of abstraction (within 10%), but the internal power is where the large difference lies—the reasons for such are not clear at this point.

Unfortunately a detailed report of the power analysis carried out is not available from within Encounter, making it difficult to further examine the source of the large disparity between the two figures.

```
****************************************
Report : area
Design : test_copro
Version: W-2004.12-SP5
Date   : Wed Apr 18 12:07:51 2007
****************************************

Number of ports:                 81
Number of nets:                4393
Number of cells:                 77
Number of references:            30

Combinational area:       478952.812500
Noncombinational area:    1929511.125000
Net Interconnect area:    8566131.000000

Total cell area:          2408435.500000
Total area:               10974566.000000
```

It was originally intended to perform further work on obtaining results from post physical layout designs, allowing the information gained from these to be back-annotated to the higher level analysis, with a view to improving the accuracy of early stage analysis. Further examination of the back-end flow may also have pinpointed the reasons for the large power disparity found between post-layout and gate-level analysis.

Unfortunately, the licence for Cadence Encounter products expired while the work in this chapter was being undertaken. The licence was not subsequently renewed, meaning it was not possible to carry out any further work using these tools.

11.3 Summary

In this chapter, a full back-end physical layout and place & route flow was carried out on a coprocessor gate-level netlist. This was undertaken with the intention of allowing comparisons to be made between results obtained at higher levels of abstraction, and those obtained from a fully-annotated post layout coprocessor ready for tape-out.

Initial layout work was carried out using the Synopsys Astro platform tools—in particular, the JupiterXT physical layout tool. The required Milkyway libraries were created, and initial floorplanning carried out. However a subsequent problem that caused the tool to crash, suspected to be related to the use of TSMC macro cells in the Milkyway libraries, proved to be insurmountable. As a result the use of Astro platform tools was abandoned.

The Cadence SOC Encounter platform was subsequently used to complete the back-end flow. Initial configuration and floorplanning were completed successfully, and subsequent power planning, macro placement and clock tree synthesis were also successful.

Analysis of the post-layout design was then undertaken, with both area and power consumption figures obtained. These were then compared with the values obtained in previous chapters for the equivalent gate-level netlist. Unfortunately the licence for SOC Encounter tools expired before a more detailed analysis could be completed. However, the importance of post-layout analysis for obtaining accurate clock tree power figures was somewhat reduced with the introduction of topographical mode to Design Compiler, detailed in chapter 8.

12. Case study

Most of the work carried out in previous chapters has been undertaken with the goal of developing a fully automated power and energy consumption analysis model, for integration into Cascade. In this chapter, the building blocks that were formed from work in previous chapters are consolidated into a unified model within Cascade, resulting in a fully automated energy analysis capability at an early stage in the coprocessor design cycle. This capability allows a large number of coprocessor candidates to be compared, enabling the inclusion of energy consumption as a selection criterion alongside the existing criteria of hardware area and performance (cycle count).

To demonstrate this capability, several example cases are considered. The entire coprocessor generation flow is detailed, showing how the energy analysis functionality fits into the design flow. The selected coprocessor is then taken through a complete RTL synthesis, simulation and gate-level power analysis, as described in chapter 3, and the results are compared with estimates provided by Cascade early in the design flow. All tests from the MediaBench suite, listed in chapter 5, are run through the Cascade flow using both 90 nm and 130 nm process technologies, and the results compared with those from the traditional power analysis flow. The time taken for each approach is also noted, allowing accuracy to be considered taking into account the speed-up offered.

12.1 Cascade energy analysis flow overview

The introduction of energy analysis into Cascade does not change the design process significantly, an intentional design decision—the process should be largely transparent to the end user. Figure 12.1 shows the new flow, a development of the flow shown in Figure 2.2 and detailed in section 2.1. The main difference that is apparent to a user of Cascade is that the

candidate selection stage shows estimates for the energy used by each coprocessor candidate to complete execution of the software offloaded to the coprocessor. The energy estimates are in addition to previously present estimates of area and cycle count. Therefore the user can select an appropriate candidate to be synthesised, depending on their particular requirements around all three parameters.

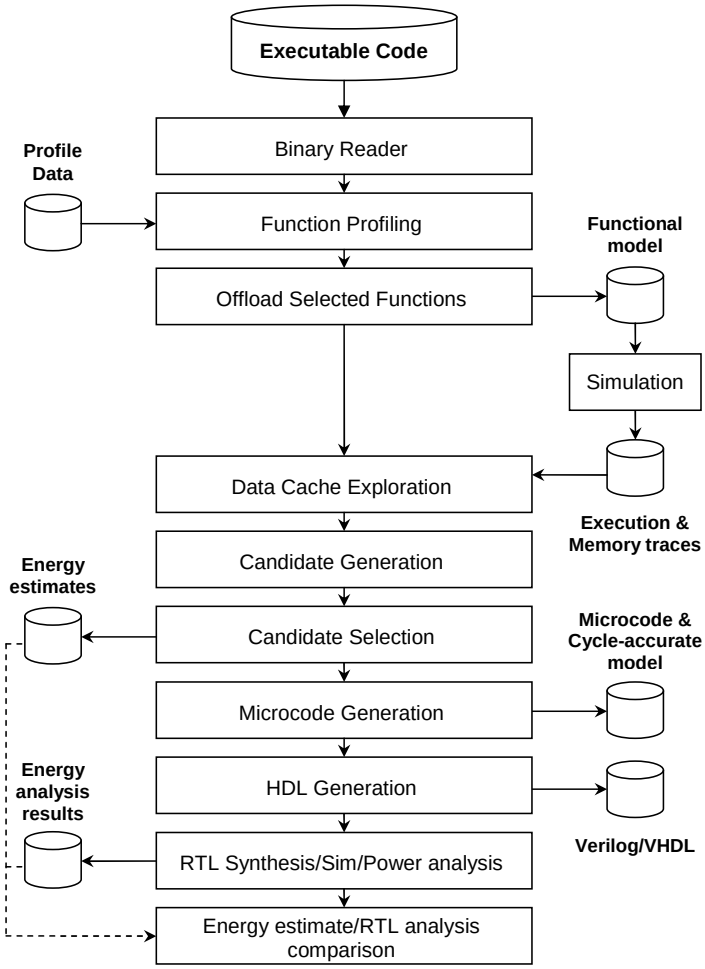


Figure 12.1: Cascade design flow incorporating energy analysis

There are three components that contribute to coprocessor energy consumption, as determined by Cascade. These are dynamic, leakage and clock tree energy. The methods used to determine each component have been covered in previous chapters—leakage energy in chapter 9, clock tree energy in chapter 8, and dynamic energy in a number of chapters, particularly chapter 6 for functional units and chapter 7 for memory blocks and register files. The clock tree energy model is self-contained and is intended only as a guide estimate due to the high level of variance possible in clock tree energy depending on the synthesis method used during layout. Therefore only the implementation of the combined dynamic and leak-

age energy model is considered here, with the dynamic energy model broken down to energy used during active and inactive cycles.

Much of the information used by Cascade to determine energy estimates for each coprocessor candidate is stored within an XML file specific to the current process technology being used. The file used with TSMC 90 nm technology coprocessors is listed in Appendix I.1, and that used with TSMC 130 nm is listed in Appendix I.2. Each top-level functional unit available to Cascade has three entries in the file: one for energy per active cycle, another for energy per inactive cycle, and finally an entry for leakage energy. The first two were determined in chapter 6, and they represent the amount of dynamic energy dissipated in a single cycle by one of the units (excluding memory blocks), depending on whether that unit is active or inactive during that cycle. The leakage energy entries were determined in chapter 9; they represent energy dissipated per second, regardless of whether the unit is active or inactive. An excerpt of the XML file listing two of the top level functional units is shown below:

```
<table name="execUnitActiveEnergy">
  <estimate key="access_st_1" value="0.01913"/>
  <estimate key="arithmetic" value="0.03218"/>
</table>
<table name="execUnitInactiveEnergy">
  <estimate key="access_st_1" value="0.000456"/>
  <estimate key="arithmetic" value="0.000120"/>
</table>
<table name="execUnitLeakageEnergy">
  <estimate key="access_st_1" value="448000"/>
  <estimate key="arithmetic" value="5240"/>
</table>
```

Another set of entries in the XML file represent the memory blocks and register files contained within a number of top level functional units, including both Artisan memories and DesignWare IP. The values contained in these entries were determined as part of the work undertaken in chapter 7. For each memory block, there are a range of entries representing the possible depths of that particular block available to Cascade. It was discovered that, within certain limits for each block, the energy used per access scales close to linearly with the memory width. Therefore the entries for memory accesses represent a single bit width at the specified depth; Cascade then multiplies up the value to the appropriate width for that particular memory. The majority of memory blocks used by Cascade are 32 bits wide, notable exceptions being the instruction cache and tag RAM, so the values for those memories are originally determined using a 32 bit width memory, then divided by 32 to get the single

bit width value. This is mainly for consistency within the XML file, although it also allows for future changes to the allowable memory width to be represented without modifying the XML file, rather than assuming that these memories will always be 32 bits wide. A small portion of the entries for a memory block are shown below:

```
<memory type="ram_rsws_rsws_bw">
  <table name="activeEnergy">
    <entry key="512" value="0.0005828"/>
    <entry key="1024" value="0.0006067"/>
  </table>
  <table name="inactiveEnergy">
    <entry key="512" value="0.000201"/>
    <entry key="1024" value="0.000205"/>
  </table>
  <table name="leakageEnergy">
    <entry key="512" value="8812"/>
    <entry key="1024" value="11312"/>
  </table>
</memory>
```

The final set of entries in the XML file relating to energy calculation are those for bus energy. Coprocessors created by Cascade can be connected to a number of bus interfaces, and each of these interfaces results in a different energy cost for both active and inactive cycles. An entry for each bus type for both active and inactive cycles represents the energy cost for each cycle. The energy entries for two bus types are shown below:

```
<table name="busTypeActiveEnergy">
  <estimate key="CBNative_Slave_Generic" value="0.01970"/>
  <estimate key="AMBA_AHB_Slave_Generic" value="0.02507"/>
</table>
<table name="busTypeStalledEnergy">
  <estimate key="CBNative_Slave_Generic" value="0.0197"/>
  <estimate key="AMBA_AHB_Slave_Generic" value="0.0250"/>
</table>
<table name="busTypeLeakageEnergy">
  <estimate key="CBNative_Slave_Generic" value="380000"/>
  <estimate key="AMBA_AHB_Slave_Generic" value="645000"/>
</table>
```

The values contained within the XML files are referenced by Cascade during the coprocessor candidate generation stage, as part of the overall coprocessor synthesis flow shown in Figure 12.1. Each category of information is mapped to a table by Java routines, where they are

later referenced in calculations along with the access pattern information of the coprocessor being analysed.

12.2 Cascade design flow

This section describes the basic flow used to create an application-specific coprocessor and offload some of the functionality from a host processor onto the coprocessor. Cascade offers a large range of configuration and optimisation options, which allow a user to tailor the coprocessor to specific requirements that may form part of a project. To avoid unnecessary complexity in the design flow, advanced options offered by Cascade are not modified unless they have particular relevance to the power and energy analysis functionality.

12.2.1 Initial configuration

The Cascade design flow starts with a binary, typically an executable for an ARM processor, from which a coprocessor and the corresponding microcode to run on the coprocessor are derived. Although starting the analysis from binary rather than source code complicates the analysis, Cascade uses this approach for several reasons; most importantly that it allows proprietary library functions to be offloaded, in cases where the source code may not be available. Working from the binary also allows Cascade to directly control the communication between the host and coprocessor, without concern about any compiler-induced side effects.

In addition to the binary itself, it is important to ensure that data to be processed by the binary is available and is representative of typical usage of the application to be accelerated. This is because Cascade analyses instruction usage patterns within the executable to create a coprocessor that is optimised to that application, attempting to maximise performance while minimising logic area and memory size.

Once a target binary has been loaded, a coprocessor can be created, onto which some of the functions from the binary will be offloaded. At this stage, several parameters of the coprocessor are defined—this includes the target technology (such as TSMC 130 nm or 90 nm), the system bus type (such as AMBA AHB or AXI, with or without DMA) and the address

location on the bus where the coprocessor is located. There are several system parameters that must be configured at this stage, such as wait cycles and burst lengths, but the choice of these values is dependent on the parameters of the system into which the coprocessor will be integrated, and as such is immaterial to this case study.

12.2.2 Architectural synthesis

The process of creating a hardware configuration for the coprocessor is known as architectural synthesis. Several parameters for the coprocessor must be configured prior to the commencement of architectural synthesis. The most important of these is selecting the desired base architecture from the list shown in Table 12.1. Each option provides a trade-off between low area/energy requirements and high reprogrammability, providing for high performance across a range of applications. For example, the `64_Bit_Multiplier` template provides a fully-featured coprocessor that can be subsequently reprogrammed with performance likely to remain high. However that performance and flexibility comes at a cost of increased area and energy requirements. On the opposite end of the scale, the `Minimal_Regfile` template will result in the smallest coprocessor that is capable of implementing the offloaded functions with reasonable performance. However, if the coprocessor is reprogrammed, performance may suffer considerably due to the limited execution resources available.

Required Units Template	Description
No_Multiplier	Supports ARM v5E ISA but will only add the multiplier type required by the offloaded functions
32_Bit_Multiplier	Supports ARM v5E ISA and at least a multiplier to support instructions requiring a 32-bit result
64_Bit_Multiplier	Supports ARM v5E ISA and a full 64-bit result Multiplier
Minimal	No saturating arithmetic (ARM v5E) or multiplier unless required
Minimal_Regfile	No saturating arithmetic (ARM v5E) or multiplier unless required, also forces minimised register file
Single_Cycle_64_Bit_Multiplier	As for 64_Bit_Multiplier, but the multiplier is not pipelined so only suitable for low frequency designs

Table 12.1: Coprocessor architectural synthesis required units templates

Building on top of the base template configuration, Cascade can generate a number of potential candidates for the coprocessor configuration. Alternatively, the user can choose single candidate generation mode, which requires the desired resource usage weight to be selected in the range of 0.0–1.0, where a low value optimises for maximum performance and a high value optimises for area. Single candidate generation mode was used extensively throughout this project, usually with a weight of 0.05, to allow the coprocessor synthesis process to be fully automated while maintaining consistency between synthesis runs, and at the same time minimising the run time required for each coprocessor synthesis flow to complete.

12.2.3 Function offloading

After the fundamental configuration options for the coprocessor candidates have been selected, it is necessary to indicate which functions within the code should be offloaded from the host processor to the coprocessor. Usually the desired functions will be those that consume a significant proportion of processor cycles, and as such will offer the greatest benefit from acceleration on a dedicated coprocessor. The use of profiling tools, such as GNU `gprof`, aids in the selection of functions to offload. Cascade can automatically call an external profiling tool, and offer a graphical representation of the results from where the desired functions can be selected, greatly simplifying the selection process.

There are two basic types of offloaded functions: *Entry* and *Local*. An Entry function is one that, when encountered on the host processor, will be offloaded to be executed on the coprocessor. Control returns to the host processor once the function completes, or if another function is called from within the Entry function. A Local function will not be offloaded to the coprocessor if it is encountered on the host processor; however if a Local function is encountered while execution is taking place on the coprocessor, the Local function will also be executed on the coprocessor. This allows small functions, such as mathematical operations, that are called from several points in the code, to execute on whatever device the calling function is executing on, reducing the overhead of frequently switching execution between host and coprocessor.

To aid in the offloading of a function tree, Cascade offers a *Group* offload option. When a function is selected for Group offload, that function is offloaded as an Entry function, and any other functions within its call graph are selected as Local functions. When using the Group offload feature, the entire parent function, including any sub-functions, will be executed on

the coprocessor before returning control to the host. This is the recommended method of function offloading in most scenarios, and the appropriate Local functions are automatically determined by Cascade using call graph analysis.

12.2.4 Functional simulation

When the desired functions to be offloaded to a coprocessor have been selected, it is necessary for Cascade to determine the runtime behaviour of those functions to allow the creation of a range of optimised candidate architectures for the coprocessor. This is achieved by first running an instrumented binary—a modified version of the original application, run on an instruction set simulator—which allows the generation of an instruction trace detailing the application's behaviour during execution of the offloaded functions.

A functional simulation of the coprocessor is then generated as a C model and compiled for the host processor, allowing memory access statistics and execution paths to be analysed. These statistics are used by Cascade to determine trade-offs between the performance improvement offered by the addition of a certain type of functional unit, and the area cost associated with that unit. Similarly, the information stored regarding memory access patterns allows Cascade to balance accesses across the available memory ports, with the aim of minimising conflicts and thus cache stalls. Data collected during this analysis is also used to provide performance estimates for each coprocessor candidate. Figure 12.2, taken from the Cascade User Guide, shows the flow used to generate the memory and execution trace files used in the aforementioned analysis.

12.2.5 Data cache configuration

Cascade can automatically configure the data caches for each candidate coprocessor, giving the user a choice of potential solutions with varying area requirements and performance estimates. Analysis of the suitability of each cache configuration is performed using the memory trace generated during the aforementioned functional simulation.

A large selection of cache configurations are available to Cascade. The four basic cache types are shown below, along with a description of their typical usage within a coproces-

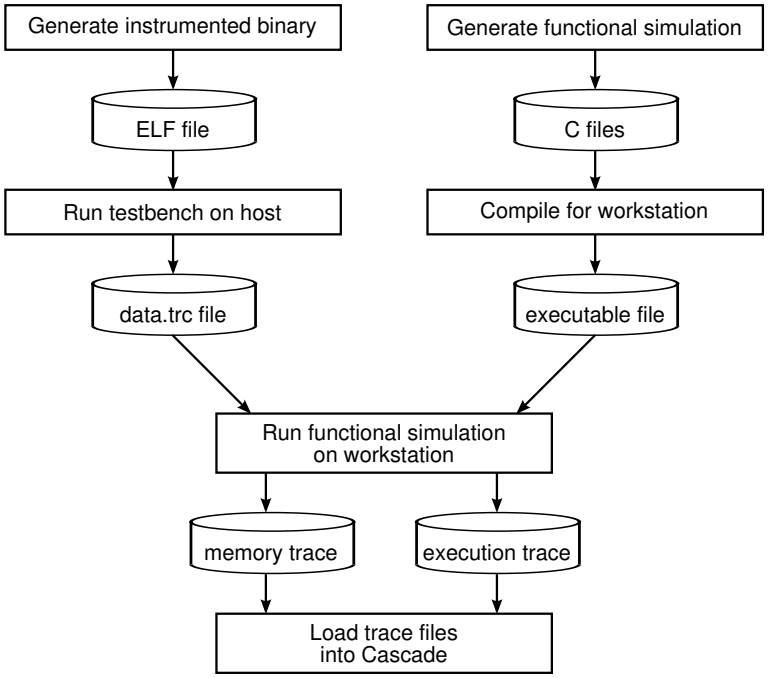


Figure 12.2: Functional simulation and instrumented binary flow

sor. Additionally, each cache type offers a number of port configuration options. A single coprocessor can have between one and four data caches, resulting in a sizeable number of combinations of possible cache configurations.

Window cache Optimised for large sequential access patterns. Typically used when the coprocessor is operating on a large stream of continuous data, common in many multimedia applications. The window cache stores a range of sequential memory addresses, and automatically fetches subsequent addresses as the location being read nears the end of the cached range.

Static cache The simplest type of cache available to Cascade. Static caches are very similar to scratch pad memories, intended to hold data from static areas of memory

Remapping cache A simpler cache than the window or associative cache. However this simplicity restricts the suitable applications of the remapping cache—all memory accesses in offloaded functions must access only static data areas or the stack.

Associative cache A fully functional cache, similar to that used in most high-end embedded processors and all desktop processors. The associative cache is extremely flexible and can be used for any type of access pattern, but it comes with a high area and energy cost.

The automated cache configuration feature analyses data access patterns of the offloaded functions, and presents the user with a graph showing each configuration as a point on the graph, with the axes representing cache size (area) against cycles (performance). With this information, the user can select the best trade-off from the available configurations, depending on the requirements of the target system.

Alternatively, manual selection of the cache can be enabled if desired. This is the approach taken throughout most of this project, for reasons of consistency, repeatability, and minimising run times. The following procedure, which is included in the `test.tcl` file, forces the coprocessor to be generated with a single dual-port static cache (comprised of one read-only port, one read-write port) with a resource usage weight of 0.35.

```
proc ConfigureCustomMemoryConfig {} {
        generate_memory_config "0.35" {access_st_1r}
}
```

As can be seen from the above procedure, manual cache selection sets only the cache unit(s)

used and the resource usage weight. The configuration of the cache is still automatically determined by Cascade analysing various cache sizes and bank configurations to determine the performance against size trade-off. For each configuration, memory operations for the offloaded functions are allocated to banks to determine performance in terms of the number of miss cycles that will be incurred. In the case of the above configuration, Cascade will evaluate 495 potential configurations, and automatically select the optimal one based on the resource usage setting. The approach of manually selecting a cache greatly reduces the number of configurations that have to be evaluated, and as a result enables a corresponding reduction in run time.

Cascade provides several analysis features to help evaluate and, if necessary, modify cache configurations. For example, for each cache configuration candidate, Cascade can list the performance breakdown in terms of hits, capacity misses, and compulsory misses. If there are any issues with the memory accesses made by the offloaded functions that either degrade performance, or prevent the use of certain cache types, those issues will be highlighted and a list provided, containing possible steps that can be taken to resolve them.

The cache analysis features are intended for use as part of an interactive coprocessor generation flow, rather than an automated (scripted) flow as used throughout this project. Therefore these features are rarely used in this project; the aforementioned manual cache selection and automated configuration, based on resource usage weight, are instead employed.

12.2.6 Candidate architecture generation

After the cache configuration has been determined, Cascade will proceed to generating a range of candidate architectures for the coprocessor. Each architecture will have a different set of execution units and interconnections between those units, built upon one of the base architectural templates selected from those listed in Table 12.1 during the architectural synthesis stage.

The candidate coprocessor architectures are optimised to execute the critical code regions present in the offloaded functions, identified during functional simulation. The candidate architectures cover a range of performance and area metrics, as reported by Cascade, and it is at this point that the addition of energy analysis functionality is particularly useful.

Coprocessor candidate generation is an automated process, and depending on the number of candidates being generated (which can be configured using the *effort level* parameter), this step may take some time. Once synthesis is complete, Cascade estimates the performance and area requirements of each candidate architecture, to guide the user in selecting the best option for the requirements of the overall system. The addition of energy consumption to the existing performance/area trade-off allows the user to make a much more informed decision as to the best candidate, particularly with regard to coprocessors that will be used in low power platforms.

Once the desired candidate is selected, the coprocessor design stage is complete. Cascade then creates the coprocessor hardware RTL in VHDL and/or Verilog, along with the appropriate testbench. The microcode to run on the coprocessor is also generated at this stage, along with a modified binary for the host processor that deals with function offloads and all communication between the two processors.

12.3 Energy analysis within Cascade

This section examines how the energy analysis functionality, the various components of which have been developed in previous chapters, is integrated into Cascade. Comparisons are made with the accuracy of the analysis functionality, against the results obtained when undertaking a complete power analysis using Synopsys tools as described in chapter 3.

12.3.1 Obtaining coprocessor energy results

The automated nature of the energy analysis functionality integrated into Cascade means that there are no changes required to the design flow in order to obtain energy estimates for each coprocessor candidate. The flow detailed in the previous section is followed verbatim, and Cascade will generate a report for each candidate in the following location:

`Projects/<project_name>/<coprocessor_name>/candidates/<candidate_name>`

Within each candidate directory, a file named `AnalysisSummary_<candidate_name>.txt` will be generated, containing the details of that particular candidate's characteristics. A

sample analysis summary file is listed in Appendix I.3. Of particular interest in this file are the following entries:

Total Logic Usage	Candidate area requirements (number of gates)
Total Cycles	Number of cycles to complete the test stimulus
Energy Usage	Dynamic energy consumed by the coprocessor
Clock Tree	Energy consumed by the clock tree
Total Leakage	Leakage energy consumed by the coprocessor

Where a number of candidates have been generated, the information provided in these entries can be used to compare each candidate's area, cycle count and energy performance. Alternatively, if Cascade is being used in interactive mode, this information can be presented in graphical format, making it easier to interpret, particularly for a large number of candidates. In the following analysis, typically 10–20 candidates are examined, and the optimal candidate is automatically selected by Cascade.

12.3.2 Analysis of results produced by Cascade

In order to validate the accuracy of energy estimates provided by the functionality integrated into Cascade, a selection of tests are taken through the complete coprocessor synthesis flow using TSMC 90 nm technology, which is the primary target technology for the majority of coprocessor platforms at the time of undertaking this case study. The energy estimate for each coprocessor is automatically extracted from the analysis summary file. Each coprocessor is then taken through a complete gate-level power analysis flow, consisting of RTL synthesis, netlist simulation, and gate-level power analysis, similar to that described in chapter 3. To allow the two values to be compared, the energy values provided by Cascade are converted to average power values, by dividing the total energy by the coprocessor run time. The latter figure can be determined by dividing the cycle count by the clock frequency.

Clock tree power is not included in these results, as it can change substantially depending on the clock tree synthesis and physical layout configuration options selected during the back-end flow. The estimates are provided as a useful comparison between coprocessor candidates on the basis of several assumptions about the back-end flow; therefore they are reported separately from the main coprocessor energy estimate, allowing the user to optionally include

clock tree power in the overall coprocessor energy estimates if desired. The method for estimating clock tree power, and an analysis of its accuracy, was detailed in section 8.1.

As with in previous chapters, the MediaBench benchmark suite is used as the basis for analysing the accuracy and performance of the new functionality integrated into Cascade, due to it being considered representative of typical applications targeted by Cascade. In addition to the tests provided as part of MediaBench, the analysis is extended to include several additional tests, with the aim of widening the test envelope and thus improving the confidence level in the general applicability of the functionality under analysis. The additional tests, along with a short description of the functionality of each, are listed below.

colour_interpolation	Function to colourise Bayer-encoded images
mp3_encode	Audio encoder using the free *Shine* libraries
speech_LPC55	LPC-10 version 55 speech encoding algorithm
motion_estimation	Video motion estimation application
fibonacci_sequence	Fibonacci sequence calculator
mersenne	Mersenne prime number determining algorithm
idea_encrypt	IDEA encryption algorithm

These applications were chosen to provide a range of test cases, comprising a range of sizes, complexity, and algorithmic composition, with the aim of maximising the scope of analysis within the potential application space targeted by Cascade. Ideally a much larger range of tests would be analysed, however the number is limited by the length of time taken to perform complete gate-level analysis for each test.

Results from each test using TSMC 90 nm process technology libraries are listed in Table 12.2, along with the difference between Cascade's results and those determined using the complete power analysis flow. It can be seen from these results that the energy analysis estimates provided by Cascade are within $\pm 10\%$ of those produced using the complete netlist flow. A notable observation is that Cascade tends to over-estimate the energy consumption in most cases, with a underestimation in several others, but no apparent pattern could be found when examining the detailed breakdown of the constituent components contributing to coprocessor energy. Further extensive analysis of the underlying causes is a potential route for future work that could offer an improvement in the accuracy level. Communication cost over the bus is not examined, as it is highly dependent on the configuration of the system exter-

nal to Cascade. Information on communication statistics is provided to allow calculation of estimated energy, if the external system parameters are known.

Test application	**Average power (mW)**		**Difference**
	Cascade	**Synopsys**	
adpcm_encode	5.852	5.347	9.44%
g721_decode	4.873	4.467	9.09%
g721_encode	4.495	4.126	8.93%
gsm_decode	3.845	3.559	8.04%
gsm_encode	4.030	3.725	8.18%
jpeg_decode	5.246	4.808	9.11%
jpeg_encode	4.407	4.059	8.57%
mpeg2_decode.fft	5.500	5.027	9.42%
mpeg2_decode.ref	5.422	4.951	9.50%
mpeg2_encode	5.456	4.988	9.39%
pgp_decode	5.401	4.945	9.22%
pgp_encode	6.775	6.168	9.84%
colour_interpolation	5.757	5.818	-1.05%
mp3_encode	4.451	5.051	-9.90%
speech_LPC55	5.283	4.834	9.28%
motion_estimation	4.393	4.581	-2.83%
fibonacci_sequence	3.779	4.136	-8.62%
mersenne	2.749	3.008	-8.62%
idea_encrypt	5.717	5.230	9.32%

Table 12.2: Average power consumption estimates (TSMC 90 nm technology)

For comparative purposes, the same tests are taken through the coprocessor synthesis flow using TSMC 130 nm process technology. The approach is identical to that used for the TSMC 90 nm tests, and the results are shown in Table 12.3. In this case, the accuracy is somewhat less than that obtained for the 90 nm coprocessor energy estimates, decreasing to around $\pm 18\%$ of the values obtained using the complete netlist flow. Examining the detailed energy figures provided by the analysis summary report explains the reason for the lower accuracy—a much higher proportion of the total energy consumed by coprocessors targeted at 130 nm process technology is due to dynamic power, for which energy estimates are inherently less accurate than those for leakage power due to software variances. In the tests undertaken using TSMC 130 nm process technology, 86.83% of the total energy consumed is attributable to dynamic power; for TSMC 90 nm coprocessors, that value falls to 54.56%. In light of this, it is expected that the energy estimates for 90 nm coprocessors will be more accurate as a result of the leakage component of high-level power estimation being more

accurate than the dynamic component. Although 130 nm process technology was the key target until around the mid point of the project, in the latter stages its significance has rapidly decreased, and it is unlikely that many new coprocessor designs will target this technology. Therefore it is not considered to be of concern that analysis of coprocessors synthesised using 130 nm technology allows for a lower accuracy than those targeted at a 90 nm process.

Test application	**Average power (mW)**		**Difference**
	Cascade	**Synopsys**	
adpcm_encode	6.380	5.375	18.70%
g721_decode	5.591	5.587	0.09%
g721_encode	4.978	4.565	9.05%
gsm_decode	3.435	4.179	-17.81%
gsm_encode	3.783	4.560	-17.05%
jpeg_decode	6.148	6.125	0.37%
jpeg_encode	4.820	5.901	-18.32%
mpeg2_decode.fft	6.994	6.044	15.72%
mpeg2_decode.ref	6.417	5.532	16.00%
mpeg2_encode	6.914	6.099	13.36%
pgp_decode	5.828	5.401	7.89%
pgp_encode	8.150	8.040	1.36%
colour_interpolation	7.340	6.700	9.55%
mp3_encode	8.375	7.153	17.07%
speech_LPC55	6.780	6.700	1.19%
motion_estimation	5.556	6.039	-8.00%
fibonacci_sequence	4.919	5.611	-12.33%
mersenne	3.249	3.674	-11.56%
idea_encrypt	7.098	6.061	16.79%

Table 12.3: Average power consumption estimates (TSMC 130 nm technology)

The reduction in analysis time using Cascade's functionality compared with a traditional power analysis flow is difficult to characterise, as a large proportion of the overall time taken when using the netlist power analysis flow is attributable to the simulation stage, where switching activity statistics are collected. Simulation time is closely correlated with the size of the input stimulus file in combination with the complexity of the coprocessor hardware, making it highly variable depending on the quantity of data processed by the functions off-loaded to the coprocessor.

On the contrary, Cascade's energy analysis functionality is largely unaffected by the size of the data set. This is because activity statistics are gathered during functional simulation and instrumentation which, while dependent on the data set size, are performed as part of

Cascade's coprocessor synthesis flow regardless of whether energy analysis is taking place. Therefore the additional run-time attributable to the energy analysis functionality is limited to dynamic and leakage power calculations, based on data gathered during previous stages of Cascade's flow, which already carry out most of the work required by the energy analysis functionality with little additional penalty.

As a result, the automated analysis flow typically offers a speed-up in the range of two to three orders of magnitude (100–1000×), compared to a traditional power analysis tool chain, using a suite of synthesis, simulation and power analysis tools. Such a large performance increase enables the comparison of a much larger selection of potential candidates for each coprocessor. In addition to the improved computational performance, the integrated energy functionality is fully automated, and requires no external tool licences.

12.4 Summary

In this chapter, a walk through of the Cascade coprocessor synthesis flow was undertaken, with the creation of a detailed account of each of the steps in the flow. The integration of energy analysis functionality into Cascade was demonstrated, and the method of obtaining energy estimates for a number of coprocessor candidates shown.

Finally, the results provided by Cascade were compared against those produced using a complete gate-level power analysis flow with conventional tools. Accuracy was found to be within ±10% for coprocessors synthesised using TSMC 90 nm process technology, with an analysis speed up of two to three orders of magnitude. Thus the benefits of integrated analysis, including speed and automation, allowing analysis of a much larger range of potential candidates at an early stage of the flow, were highlighted, proving that high-level estimation of configurable processors is feasible, with accuracy within the bounds of what can be considered to be useful.

13. Conclusion

In this chapter, the work that has been carried out throughout the previous chapters is summarised, and consideration is given to the outcomes of that work and how it relates to the initial goals of the project. Further development potential, to advance what has currently been achieved, is then considered.

13.1 Project summary

This project has examined power and energy considerations, in a number of contexts, in relation to the automated coprocessor synthesis tool Cascade, which targets system on chip platforms. Fast energy consumption analysis capability has been implemented into Cascade at an early stage of the coprocessor synthesis flow, and several optimisations have been identified and implemented into the coprocessor architectures.

Prior to the commencement of this project, there existed no power or energy awareness capability within Cascade, with performance criteria for coprocessor candidates being limited to area and cycle count estimates. With the increasing importance of power and energy awareness in many consumer electronics products, which comprise much of the typical target market for Cascade, the implementation of such functionality is clearly a highly desirable attribute to add to Cascade's capabilities.

Estimating area and cycle count for a particular coprocessor candidate is a relatively simple task. Area is a static value unaffected by the software running on a coprocessor, that can be estimated as soon as the coprocessor architecture is finalised. Cycle count is a little more complex, but it can be determined using a cycle-accurate simulation, which forms part of the coprocessor verification suite.

By comparison, power and energy analysis is a far more complex problem, composed of many constituent components that must be analysed individually using different techniques. Leakage power is largely determined by the choice of hardware configuration, whereas dynamic power depends on both the hardware configuration of the coprocessor, and how the software being executed on the coprocessor exercises that hardware.

The work undertaken to implement automated energy analysis functionality within Cascade can be summarised by the following:

- Development of a gate-level coprocessor power evaluation tool flow using Synopsys tools, and comparison between Cascade coprocessors and open-source coprocessor cores using that tool flow. Chapters 3 and 4.
- Porting of the MediaBench suite of benchmarks initially to the ARM9 processor family, with subsequent offloading of key functions from each test to a Cascade coprocessor. This benchmark suite was identified as being strongly representative of the typical type of application that Cascade targets, making it an ideal basis for later analysis work on functionality added to Cascade. Chapter 5.
- Creation of energy models for the functional units that make up the fundamental building blocks of coprocessors, with specific attention paid to the multiplier unit and output banks, due to their high significance to the overall energy consumption picture. Chapter 6.
- Characterisation of memory blocks and register files used by Cascade, allowing energy models that represent the consumption of these units under relevant operating conditions. Chapter 7.
- Examination of the power consumed by the clock tree within a Cascade coprocessor, with automated analysis capability added to Cascade. Consideration of the benefits offered by clock gating at both the RTL and netlist levels. Chapter 8.
- Implementation of a leakage power model for coprocessors. Chapter 9.

With the integration of all the aforementioned functionality into Cascade, a fully automated, seamless energy analysis and optimisation capability is now available within Cascade. This capability allows the user to examine the energy characteristics of each coprocessor candidate, alongside the existing area and cycle count characteristics, and select the candidate that best matches the needs of the system into which the coprocessor will be integrated.

Without such functionality being available within Cascade, a user with a requirement to estimate the energy consumption of a coprocessor would have to undertake a complete analysis flow, similar to that described in chapter 3. If the user does not already have appropriate licences for the tools used in the analysis, acquisition of those licences would represent a substantial cost. In addition, the time consumed in such an analysis, even sacrificing accuracy by performing the analysis at the RTL rather than the gate level, makes it infeasible for more than a very small number of coprocessor candidates.

The analysis functionality integrated into Cascade during this project offers a speed-up of several orders of magnitude over a gate-level simulation and power analysis, while typically maintaining accuracy within 10% at the TSMC 90 nm process technology node. This analysis is completely automated, requiring no additional user input or changes to the flow, and the high performance of the analysis allows it to be performed on a large number of coprocessor candidates with minimal effect on the overall runtime of Cascade. This allows the user to make an informed decision on selection of the most appropriate coprocessor candidate, particularly with regard to coprocessors that will be implemented into energy sensitive SoC platforms.

In addition to the automated analysis capabilities added to Cascade, several energy consumption optimisations were identified during the project. The components that consumed the largest proportion of energy within the coprocessor architecture were examined, and several possible solutions considered with a view to reducing the consumption of those units. Two optimisations—multiplier idle cycles and instruction cache width reduction—were found to offer substantial reductions in energy consumption, particularly the latter, with no performance penalty. Both of these optimisations have since been implemented and are currently present in coprocessors synthesised by Cascade. Idle and sleep modes for coprocessors were also investigated, but not implemented by default at this time due to issues identified at the time of the analysis. Full details of these optimisations are in chapter 10.

Although much of the project has focused on energy analysis and optimisation as applicable to Cascade, and specifically with reference to coprocessors synthesised by Cascade, at a higher level the techniques developed are generally applicable to system-on-chip processor development. The modular nature of the approach taken throughout the project ensures that any modifications necessary to adapt the analysis model to different processor types can be carried out without disruption to other parts of the model.

The following points highlight the achievements of this project that are universally applicable to configurable SoC processor design:

- Application of low-level analysis to determine the energy consumption profile of the processor, enabling informed partitioning of the high-level model with focus directed towards components with the most significant contribution to the overall energy budget
- Identification and isolation of different fundamental types of component that make up the processor, some of which may require specialised analysis techniques—for example, hard macro memory blocks, which are likely to be present in most processor designs as cache memory
- Recognition of the significance of clock tree power in deep sub-micron synchronous designs, and development of models to estimate the energy consumed by the clock tree at a high level. This work is generally applicable to most SoC processors.
- Acknowledgement of the importance of leakage power to the overall SoC energy picture, particularly at the 90nm process technology node and beyond. The high level leakage energy model developed is applicable to most SoC devices, as it is much less variable (and therefore less specialised) than the dynamic energy model.

In summary, the work undertaken in this project furthers existing knowledge in the field of high-level power and energy modelling for configurable SoC processors. Prior to the work carried out here, high-level analysis would typically have considered only the hardware element of the configurable processor, neglecting the influence of software changes that are required when the hardware is changed. By taking into account the entire hardware and software combination, this research has enabled high-level energy estimations with accuracy within the bounds of what is considered useful for making decisions early in the design space exploration phase, allowing simple and effective optimisation of processor energy consumption within the target platform.

13.2 Future work

The work detailed in this thesis has met the key goals identified in the initial stages of the project, and has solved some significant problems. As a result, very desirable functionality, in both the technical and commercial realms, has been added to Cascade. A logical extension of this work would be to improve the accuracy of energy estimates. Some of the analysis techniques were deliberately simplified to reduce computational complexity; it would be desirable to investigate methods of analysis that allow recovery of that accuracy without significantly lengthening run times.

Current analysis functionality is targeted at platforms with one or more host processors, but only a single Cascade coprocessor. Although the functionality is extensible to multiple coprocessors within a single platform, it does this by simply repeating the analysis over each coprocessor. Multi-core capability is becoming increasingly important in the embedded market, and as a result a more sophisticated analysis capability for multiple coprocessors platforms is a highly desirable attribute. Features such as the ability to analyse the energy usage of various software partitioning schemes between coprocessors, or the capability to suggest the optimal number of coprocessor cores for a particular application, from an energy perspective, would be highly desirable in a multi-core environment.

As an electronic design automation tool, Cascade is continually evolving as the underlying technology evolves. Thus, the functionality developed in this thesis and integrated into Cascade must also concurrently evolve with Cascade. The next process technology node that will be targeted by Cascade is 65 nm. It may become apparent when coprocessors are synthesised using 65 nm technology that the leakage power analysis functionality within Cascade is not of a sufficient accuracy due to the increased proportion of the overall energy budget that is attributable to leakage at smaller process technology nodes. In such a scenario, one solution would be to adapt the leakage power analysis model to account for input state changes, and possibly reducing the complexity of the dynamic power model to offset the increased run-times incurred by the more detailed leakage power model. This is one example; there are a multitude of reasons why the energy analysis functionality may become inaccurate due to changes elsewhere in Cascade, underlying the necessity that the functionality continues to develop with Cascade to maintain the accuracy that has been achieved at this point targeting a 90 nm process technology platform.

Appendices

A. Top-level Tool Flow Script

The functionality of the scripts listed in section A.1 and A.2 effectively overlap, however they are maintained as independent scripts for practical reasons: Synopsys licence locations prevent running the entire analysis on a single machine, due to there being no single machine with access to both VCS and Power Compiler. Therefore the script in Section A.1 can be called with the `-no_power` flag, thus skipping the Power Compiler stage and resulting in the generation of a tarball archive ready to be transferred to a suitable machine. After the transfer, the script in section A.2 will automatically extract the relevant files from the archive before performing power and energy analysis.

A.1 Integrated synthesis, simulation and power script

```
#!/bin/sh
#
# Power/energy analysis script that operates on one test case in three stages:
#
# 1. RTL synthesis in Design Compiler, outputs gate-level netlist
# 2. Netlist simulation in VCS, outputs backward-SAIF file
# 3. Power/energy analysis in Power Compiler
#
# This is the Verilog version of the script (both RTL and netlist)
#
# Written by Paul Morgan, 2005-2006
#
##############################################################################
# Command-line parameters:                                                   #
#  -s, --no_synth   : Do not perform synthesis, use existing synthesised design  #
#  -S, --no_sim     : Do not simulate                                         #
#  -p, --no_power   : Do not perform power analysis                           #
#  -a, --no_archive : Do not consolidate netlist and SAIF files into tar.gz archive #
##############################################################################

usage()
{
   echo "Usage: $0 <test1_name> [test2_name] ... [testN_name] [options]"
```

```
   echo
   echo "Options:"
   echo "-s, --no_synth   : Do not perform synthesis, use existing synthesised design"
   echo "-S, --no_sim     : Do not simulate"
   echo "-p, --no_power   : Do not perform power analysis"
   echo "-a, --no_archive : Do not consolidate netlist and SAIF files into tar archive"
   echo
   echo "Single-letter short options can be combined into a single option flag"
   echo
   echo "Examples:"
   echo "$0 test_mp3 --no_power"
   echo "$0 test_sha1 test_md5 -Spa"
   exit 1
}

# Set location of root directory containing test directories
# - usually the same place as where this script is located
root_dir="`pwd`"

# Reset optional flags used to disable parts of the script
unset $NO_SYNTH $NO_SIM $NO_POWER $NO_ARCHIVE

# Output a blank line before any messages to improve readability
echo

# Parse command line input and set appropriate flags
if [ $# -lt 1 ]; then
   usage
fi

# Use GNU getopt to parse the input string, and store the output status. This is
# done to allow any errors indicated by getopt to be temporarily ignored so that
# we can parse the input string and highlight the offending option flag. Getopt's
# output status is then checked in case any errors aren't caught by the parsing
# done within the script
input_string=(`getopt -q -osSpa -lno_synth,no_sim,no_power,no_archive -- $@`)
getopt_status=$?

for (( i = 0; i < ${#input_string[*]}; i++ )); do
   if [ ! ${input_string[$i]} == "--" ]; then
     case ${input_string[$i]} in
       -s | --no_synth)   NO_SYNTH="TRUE";;
       -S | --no_sim)     NO_SIM="TRUE";;
       -p | --no_power)   NO_POWER="TRUE";;
       -a | --no_archive) NO_ARCHIVE="TRUE";;
       *)                 echo "Invalid command line option \"${input_string[$i]}\"."
                          echo
                          usage;;
     esac
   else
     TEST_LIST=${input_string[*]:($i+1)}
     break
   fi
done

if [ $getopt_status -ne 0 ]; then
   echo "Invalid command line input: $@"
```

```
   usage
fi


for current_test_name in $TEST_LIST ; do

   # Get the current test path, to do this we have to strip the single-quotes
   # from the test name added by getopt
   current_test="`echo $current_test_name| cut -d \' -f 2`"

   # Check whether the current test directory exists. If not, the continue command
   # causes the for loop to skip it and the next item is processed. In the case of
   # the final item in the list the loop exits
   if [ ! -d $current_test ]; then
      echo "Warning: Test directory $current_test doesn't exist, skipping test."
      # Unsetting current_test isn't really necessary, just a safety net
      unset current_test
      continue
   fi

   # Set location of secondary directories in relation to current test directory
   test_dir="$root_dir/$current_test"
   report_dir="$root_dir/reports/$current_test"

   echo "Current test is $current_test; test directory $test_dir"

   # Set the appropriate call to dc_shell depending on host due to lack of XG licence
   # required for the older version of DC on bootes. Newer versions no longer need a
   # licence for XG mode.
   if [ `hostname -s` = "bootes" ]; then
      dc_shell_exec="dc_shell-t"
   else
      dc_shell_exec="dc_shell-xg-t"
   fi

   # Set directory to store synthesised netlist and SAIF files necessary for
   # power analysis if exporting to another machine
   if [ ! $NO_ARCHIVE ]; then
      consolidate_dir="$root_dir/netlist+saif"
      echo "Using $consolidate_dir to store netlist and SAIF output files."
      echo

      # Ensure consolidate directory exists
      if [ ! -d "$consolidate_dir" ]; then
         mkdir -p "$consolidate_dir"
      fi
   fi

   # Automatically disable power analysis for now as we don't have a
   # Power Compiler licence at this site
   echo "Automatically disabling power analysis due to lack of required licence."
   NO_POWER="TRUE"
   NO_SIM="TRUE"

   # Ensure reports directory is available
   if [ ! -d $report_dir ]; then
     mkdir -p $report_dir
```

```
fi

echo "`date`: Starting test $current_test"
echo

###############################################################################
#                                                                             #
# Synthesis section                                                           #
#                                                                             #
###############################################################################

# Command line option to skip synthesis for previously synthesised designs
if [ $NO_SYNTH ]; then
  echo "Skipping synthesis."

else

  cd "$test_dir/Verilog_Impl"

  # By default the .synopsys_dc.setup file is in Example_Scripts
  # and we need it in the current directory. First need to change
  # any references currently pointing to the slow library to instead
  # point to typical, to ensure realistic power figures
  sed s/slow/typical/g < Example_Scripts/.synopsys_dc.setup > .synopsys_dc.setup

  # Start DC-shell with synthesis script
  echo "Synthesis starting:" `date`
  echo "Synthesis started:" `date` > $report_dir/synth.txt
  $dc_shell_exec -wait 10 \
                 -x "set current_test $current_test; set report_dir $report_dir" \
                 -f $root_dir/synth.tcl >> $report_dir/synth.txt

  # Check for errors in the synthesis process
  if [ ! $? = 0 ]; then
    echo "Synthesis failed, see $report_dir/synth.txt; exiting..."; exit;
  fi

  echo "Synthesis completing:" `date`
  echo "Synthesis completed:" `date` >> $report_dir/synth.txt
  echo
fi

###############################################################################
#                                                                             #
# Simulation section                                                          #
#                                                                             #
###############################################################################

# Command line option to skip simulation
if [ $NO_SIM ]; then
  echo "Skipping simulation."

else

   # Find the relevant testbench files and insert SAIF commands
   # using the Verilog API
```

```
# Delete any pre-existing temp file
rm -f /tmp/insert_saif_data.txt

# Create a temp file with commands to be inserted into top-level testbench
cat > /tmp/insert_saif_data_top.txt << "      EOF"
 // ************* ADDED SECTION FOR SAIF MONITORING **************
 initial
 begin
     $set_gate_level_monitoring("on");
     $set_toggle_region(copro_testbench.copro);
     $toggle_start();
     $display("Starting toggle.");
     //full simulation is too long, therefore do the test over
     //30 ms for now to allow comparisons in switching activity
     //#30000000;
     //$toggle_stop();
     //$display("Simulation reached limit of 30 ms");
     //$display("Stopping toggle, generating SAIF file.");
     //$toggle_report("backward.saif",1e-9,"copro");
     //finish simulation after SAIF file has been written
     //$finish;
 end
 // ******************** END OF ADDED SECTION ********************

EOF

# Create a temp file with commands to be inserted into 2nd-level testbenches
cat > /tmp/insert_saif_data_generic.txt << "      EOF"
            // ************* ADDED SECTION FOR SAIF MONITORING **************
            $toggle_stop();
            $display("Stopping toggle, generating SAIF file.");
            $toggle_report("backward.saif",1e-9,"copro_testbench");
            //finish simulation after SAIF file has been written
            $finish;
            // ******************** END OF ADDED SECTION ********************
EOF

# Declare an integer variable used to calculate the correct line number
# in which to insert the SAIF commands for each file
declare -i linenum

if [ -f "$test_dir/Testbench/Verilog_Testbench/copro_testbench.v" ]; then
   tb_file="$test_dir/Testbench/Verilog_Testbench/copro_testbench.v"
   # Check to ensure the SAIF commands aren't already inserted in this file, then
   # create a backup before inserting the SAIF commands from saif_code.v
   grep -q set_gate_level_monitoring $tb_file
   if [ $? -ne 0 ]; then
      # For this testbench the SAIF info needs to be placed 2 lines after the line
      # containing wait(0);
      linenum=`grep 'wait(0);' $tb_file -n|cut - -d : -f 1`+2

      mv $tb_file $tb_file\.bak
      cat $tb_file\.bak | sed -e "$linenum r /tmp/insert_saif_data_top.txt" \
         > $tb_file
      echo "Updated $tb_file"
   fi
fi
```

```
# Also insert SAIF write files into generic and AHB testbenches, in case the
# simulation completes before hitting the time-out in the above file. If this
# happens, we hit a $stop command in the generic testbench, causing the simulation
# to wait indefinitely for user input.

testbench_dir="$test_dir/Testbench/Verilog_Testbench"

if [ -f "$testbench_dir/cbnative_slave_generic_testbench.v" ]
then
   tb_file="$testbench_dir/cbnative_slave_generic_testbench.v"

   # For this testbench the SAIF info needs to be placed 2 lines after the line
   # containing $display("waiting"
   linenum=`grep '$display("waiting"' $tb_file -n|cut - -d : -f 1`+1
fi

if [ -f "$testbench_dir/amba_ahb_slave_generic_testbench.v" ]
then
   tb_file="$testbench_dir/amba_ahb_slave_generic_testbench.v"

   # For this testbench the SAIF info needs to be placed 2 lines after the line
   # containing $display("waiting"
   linenum=`grep '$display("waiting"' $tb_file -n|cut - -d : -f 1`+1
fi

if [ -f "$testbench_dir/cbnative_dma_streaming_testbench.v" ]
then
   tb_file="$testbench_dir/cbnative_dma_streaming_testbench.v"

   # For this testbench the SAIF info needs to be placed 2 lines after the line
   # containing $display("instruction"
   linenum=`grep '$display("instruction"' $tb_file -n|cut - -d : -f 1`+1
fi

if [ -f "$testbench_dir/amba_ahb_dma_streaming_testbench.v" ]
then
   tb_file="$testbench_dir/amba_ahb_dma_streaming_testbench.v"

   # For this testbench the SAIF info needs to be placed 2 lines after the line
   # containing $display("instruction"
   linenum=`grep '$display("instruction"' $tb_file -n|cut - -d : -f 1`+1
fi

if [ -f "$testbench_dir/amba_ahb_master_generic_testbench.v" ]
then
   tb_file="$testbench_dir/amba_ahb_master_generic_testbench.v"

   # For this testbench the SAIF info needs to be placed 2 lines after the line
   # containing $display("instruction"
   linenum=`grep '$display("instruction"' $tb_file -n|cut - -d : -f 1`+1
fi

# Check to ensure the SAIF commands aren't already inserted in this file, then
# create a backup of the file before inserting the SAIF commands from saif_code.v
grep -q toggle_report $tb_file
if [ $? -ne 0 ]; then
```

```
    mv $tb_file $tb_file\.bak
    cat $tb_file\.bak | sed -e "$linenum r /tmp/insert_saif_data_generic.txt" \
       > $tb_file
    echo "Updated $tb_file"
 fi

 # Delete temporary files
 rm -f /tmp/insert_saif_data_top.txt /tmp/insert_saif_data_generic.txt


cd "$test_dir/Verilog_Impl"

# Start simulation process
echo "Simulation starting:" `date`
echo "Simulation started:" `date` > $report_dir/sim.txt

# Delete data files from previous simulation
rm -rf simv simv.daidir csrc

 # Create a link to the stimulus input and results files
 if ! [ -f SimInput.txt ]; then
    # Have to create links for all SimInput*.txt files as streaming copros
    # may have more than 1
    for txtfile in $test_dir/Testbench/Sim*.txt; do
       ln -s -f $txtfile
    done
 fi

 if ! [ -f SimExpectedResults.txt ]; then
    ln -s $test_dir/Testbench/SimExpectedResults.txt
 fi

 echo tb_file is $tb_file >>$report_dir/sim.txt

 # Build and run simulation
 vcsi -R +v2k +cli+1 ../Testbench/Verilog_Testbench/copro_testbench.v  \
    -v $tb_file -v synth/$current_test\.v                               \
    -v ~/synopsys/libraries/tsmc13/tsmc13_no_timing_check.v             \
    -y "/opt/Artisan/CompiledMemories/TSMC_130/MemoryModels/*.v"        \
         >> $report_dir/sim.txt

# Check for errors in the simulation process, and exit before running scsim
if [ $? -ne 0 ]; then
  echo "Simulation failed, exiting..."; exit 1
fi

 # Check that the backward SAIF file has been generated, and if so rename it to
 # reflect the current design name. This is necessary to allow multiple SAIF files
 # from different tests to be stored in the same directory, and cannot be done
 # within the simulation as the Verilog PLI doesn't have access to the
 # $current_test variable
 if [ -f backward.saif ]; then
    mv backward.saif $current_test\.saif
 fi
```

```
    # Check simululation output matches expected output
    # First check whether the simulation was stopped early by the sim time limit
    # In that case results won't match as they are incomplete so don't bother checking

    grep -q "Simulation reached limit" $report_dir/sim.txt
    if [ $? -ne 0 ]; then

       diff -iw SimExpectedResults.txt SimResults.txt > /dev/null
       if [ $? -ne 0 ]; then
          echo "Simulation results differ, exiting..."; exit 1
       else
          echo "Simulation results compared successfully to expected output."
       fi
    else
       echo "Simulation reached time limit, skipping diff check."
    fi

  echo "Simulation complete:" `date`
  echo "Simulation completed:" `date` >> $report_dir/sim.txt
  echo

fi

################################################################################
#                                                                              #
# Power analysis section                                                       #
#                                                                              #
################################################################################

if [ $NO_POWER ]; then
   echo "Skipping power analysis..."
   if [ -f $current_test\.saif ]; then
     echo "Compressing SAIF file to save disk space."
     gzip $current_test\.saif
     echo "Done."
   fi
else

  echo "Power analysis starting:" `date`
  echo "Power analysis started:" `date` > $report_dir/power.txt

  #Check if SAIF file is compressed, and uncompress it
  if ! [ -f $current_test\.saif ] && [ -f $current_test\.saif.gz ]; then
    gunzip $current_test\.saif.gz
  fi

  $dc_shell_exec -x "set current_test $current_test" -f power.tcl \
     >> $report_dir/power.txt

  # Recompress SAIF file once complete
  if [ -f $current_test\.saif ]; then
    echo "Compressing SAIF file to save disk space."
    gzip $current_test\.saif
    echo "Done."
  fi

  # Check for errors in the power analysis process
```

```
      if [ $? = 1 ]; then
        echo "Power analysis failed, exiting..."; exit;
      fi

      echo "Power analysis finishing:" `date`
      echo "Power analysis completed:" `date` >> $report_dir/power.txt
      echo
    fi

    echo "`date`: Finished test $current_test"
    echo

    # Check whether the NO_ARCHIVE flag has been set, and if not place
    # output netlist and SAIF files into the consolodation dir
    if [ ! $NO_ARCHIVE ]; then
       cp "$test_dir/Verilog_Impl/$current_test.saif.gz" \
       "$test_dir/Verilog_Impl/synth/$current_test.v" "$consolidate_dir"
       cd "$consolidate_dir"
       gunzip "$current_test.saif.gz"
       tar -zcf "$current_test.tar.gz" "$current_test.saif" "$current_test.v" \
          --remove-files
       cd -
       echo "Copied Verilog netlist and SAIF files for $current_test to \
          $consolidate_dir/$current_test.tar.gz"
       echo
    fi

    unset current_test
    cd $root_dir

done

echo "Completed all accessible tests."
exit 0
```

A.2 Independent power and energy analysis script

```
#!/bin/sh
#
# Script to run power analysis for all tests that have been previously
# synthesised and simulated. It is required that a SAIF file has been
# generated for each test, and either the SAIF file can be gzipped alone
# or combined with the netlist into a gzipped tarball. Any files decompressed
# as part of this script will be recompressed once the script completes.
#
# Written by Paul Morgan, 2005–2006

# Initialise variables used to keep track of input file format
tarball=0
tarsaif=0

if [ $# -lt 1 ]; then
   echo "Usage: $0 <test1_name> [test2_name] ... [testn_name]"
   exit 1
fi

for current_test in $*; do
   echo ----------------------------------------------------------------
   echo
   echo "Starting $current_test"

   # Check if overall gzipped tar package exists, and if so decompress it
   # Note that some versions of tar installed on UNIX systems cannot
   # decompress gzip archives, so this is done in two steps
   if [ -f $current_test\.tar.gz ]; then
      gunzip -c $current_test\.tar.gz | tar -xf -
      tarball=1
   fi

   # Check if saif file is compressed and if so decompress it
   if [ -f $current_test\.saif.gz ]; then
      gunzip $current_test\.saif.gz
      tarsaif=1
   fi

   # Get the simulation run time from the SAIF file and add this to the top of
   # the power report to allow easy calculation of energy
   simtime_ns=`grep DURATION $current_test.saif | cut -b 11- | cut -d . -f 1`
   simtime_seconds=`echo -e "9\nk\n\n$simtime_ns\n1000000000\n/\np " | dc`
   echo "Run duration: $simtime_ns\ns" > reports/$current_test\_power.txt
   echo "            : $simtime_seconds\s" >> reports/$current_test\_power.txt

   # Convert simulation time into seconds using the desk calculator (dc) tool
   simtime_seconds=`echo -e "9\nk\n\n$simtime_ns\n1000000000\n/\np " | dc`

   dc_shell-t -f power.tcl -x "set design_name $current_test" \
      > reports/$current_test.txt

   # Check that the level 1 hierarchical power report has been produced
   if [ ! -f reports/$current_test\_powerh1.txt ]; then
      echo "No hierarchical power report found, skipping energy calculation"
```

```
else
   gawk --assign simtime=$simtime_ns '{

      # Check for the existence of the Hierarchy label. This indicates
      # the start of the section containing the values we want to
      # monitor. Set the "hier" flag.
      if ($1 ~ /Hierarchy/)
      {
         hier=1; startline = FNR
      }

      # The line starting with 1 indicates the end of the Hierarchy
      # section, so clear the flag
      if ($1 == "1")
      {
         hier=0
      }

      # Start processing values within the Hierarchy section, but only
      # after the first two lines as these contain only header information
      if (hier==1 && FNR >= (startline+2))
      {
            # If the second field contains no brackets, then there is no
         # instantiation name field meaning that internal and switching
         # power are in fields 2 and 3. This generally only occurs with
         # the top-level design as it is not instantiated by any higher-level.
         # In all other cases the desired values are found in fields 3 and 4.
         if (!($2 ~ /\(*\)/))
         {
            printf "%-30s %25.5f nJ\n", $1, (($2+$3)*simtime)/1000
         }
         else
         {
            printf "%-30s %25.5f nJ\n", $1, (($3+$4)*simtime)/1000
         }
      }
   }' reports/$current_test\_powerh1.txt > reports/$current_test\_energy.txt
fi

# If current test files were tar/gzipped check archive exists then delete
# saif and netlist files
if [ $tarball -eq 1 ]; then
   if [ -f $current_test\.tar.gz ]; then
      rm $current_test.saif $current_test.v
   fi
fi

#If only SAIF file was gzipped then re-zip it
if [ $tarsaif -eq 1 ]; then
   gzip $current_test\.saif
fi

# Reset local variables as they will be re-used in the next loop iteration
unset tarball
unset tarsaif

done
```

B. Open-source cores support files

The files listed below are created for use with the various open-source processors evaluated in chapter 4. The `device.vhd` and `config.h` files, listed in section B.1 and B.2 respectively, are automatically generated by the configuration tool provided with the LEON processor, which is invoked with the command `make xconfig`; these reflect the configuration options selected within that tool. The shell script listed in section B.4 automatically downloads, builds and installs the GNU tool chain and uClinux for the OpenRISC 1200 processor.

B.1 LEON processor device.vhd

```
------------------------------------------------------------------------------
--  This file is a part of the LEON VHDL model
--  Copyright (C) 1999  European Space Agency (ESA)
--
--  This library is free software; you can redistribute it and/or
--  modify it under the terms of the GNU Lesser General Public
--  License as published by the Free Software Foundation; either
--  version 2 of the License, or (at your option) any later version.
--
--  See the file COPYING.LGPL for the full details of the license.


------------------------------------------------------------------------------
-- Entity:      device
-- File:        device.vhd
-- Author:      Jiri Gaisler - Gaisler Research
-- Description: package to select current device configuration
------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use work.target.all;

package device is
```

```
------------------------------------------------------------------------------
-- Automatically generated by tkonfig/mkdevice
------------------------------------------------------------------------------

  constant apbslvcfg_tkconfig : apb_slv_config_vector(0 to APB_SLV_MAX-1) := (
  --   first        last      index  enable     function           PADDR[9:0]
  ( "0000000000", "0000001000",  0,   true), -- memory controller, 0x00 - 0x08
  ( "0000001100", "0000010000",  1,   false), -- AHB status reg.,   0x0C - 0x10
  ( "0000010100", "0000011000",  2,   true), -- cache controller,  0x14 - 0x18
  ( "0000011100", "0000100000",  3,   false), -- write protection,  0x1C - 0x20
  ( "0000100100", "0000100100",  4,   true), -- config register,   0x24 - 0x24
  ( "0001000000", "0001101100",  5,   true), -- timers,            0x40 - 0x6C
  ( "0001110000", "0001111100",  6,   true), -- uart1,             0x70 - 0x7C
  ( "0010000000", "0010001100",  7,   true), -- uart2,             0x80 - 0x8C
  ( "0010010000", "0010011100",  8,   true), -- interrupt ctrl     0x90 - 0x9C
  ( "0010100000", "0010101100",  9,   true), -- I/O port           0xA0 - 0xAC
  ( "0010110000", "0010111100", 10,   false),-- 2nd interrupt ctrl 0xB0 - 0xBC
  ( "0011000000", "0011001100", 11,   false), -- DSU uart          0xC0 - 0xCC
  ( "0100000000", "0111111100", 12,   false), -- PCI configuration  0x100- 0x1FC
  ( "1000000000", "1011111100", 13,   false), -- PCI arbiter        0x200- 0x2FC
     others => apb_slv_config_void);

  constant apb_tkconfig  : apb_config_type := (table => apbslvcfg_tkconfig);

  constant ahbslvcfg_tkconfig : ahb_slv_config_vector(0 to AHB_SLV_MAX-1) := (
  -- first    last  index  split  enable  function          HADDR[31:28]
    ("0000", "0111",  0,   false, true), -- memory controller,   0x0- 0x7
    ("1000", "1000",  1,   false, true), -- APB bridge, 128 MB   0x8- 0x8
    ("1001", "1001",  2,   false, false), -- DSU         128 MB   0x9- 0x9
    ("1010", "1111",  3,   false, false), -- PCI initiator        0xA- 0xF
    ("0110", "0110",  4,   false, false), -- AHB RAM module       0x4- 0x4
      others => ahb_slv_config_void);

  constant ahb_tkconfig : ahb_config_type := ( masters => 1, defmst => 0,
    split => false, slvtable => ahbslvcfg_tkconfig, testmod => false);

  constant syn_tkconfig : syn_config_type := (
    targettech => tsmc25, infer_pads => false,
    infer_ram => false, infer_regf => false, infer_rom => true,
    infer_mult => false, rftype => 1);

  constant iu_tkconfig : iu_config_type := (
    nwindows => 8, multiplier => none, mulpipe => false, divider => none,
    mac => false, fpuen => 0, cpen => false, fastjump => true, icchold => true,
    lddelay => 1, fastdecode => true, watchpoints => 0, impl => 0,
    version => 0, rflowpow => false);

  constant fpu_tkconfig : fpu_config_type :=
    (core => meiko, interface => none, fregs => 0, version => 0);

  constant cache_tkconfig : cache_config_type := (
    isets => 1, isetsize => 4, ilinesize => 4, ireplace => rnd, ilock => 0,
    dsets => 1, dsetsize => 4, dlinesize => 4, dreplace => rnd, dlock => 0,
    dsnoop => none, drfast => false, dwfast => false, cachetable => cachetbl_std);

  constant mctrl_tkconfig : mctrl_config_type := (
    bus8en => true, bus16en => false, wendfb => false, ramsel5 => false,
```

```
    sdramen => false, sdinvclk => false);

  constant peri_tkconfig : peri_config_type := (
    cfgreg => true, ahbstat => false, wprot => false, wdog => false,
    irq2cfg => irq2none, ahbram => false, ahbrambits => 11);

  constant debug_tkconfig : debug_config_type := ( enable => true, uart => false,
    iureg => false, fpureg => false, nohalt => false, pclow => 2,
    dsuenable => false, dsutrace => false, dsumixed => false,
    dsudpram => false, tracelines => 64);

  constant boot_tkconfig : boot_config_type := (boot => memory, ramrws => 0,
    ramwws => 0, sysclk => 25000000, baud => 19200, extbaud => false,
    pabits => 11);

  constant pci_tkconfig : pci_config_type := (
    pcicore => none, ahbmasters => 0, ahbslaves => 0,
    arbiter => false, fixpri => false, prilevels => 4, pcimasters => 4,
    vendorid => 16#0000#, deviceid => 16#0000#, subsysid => 16#0000#,
    revisionid => 16#00#, classcode =>16#000000#, pmepads => false,
    p66pad => false, pcirstall => false);

  constant tkconfig : config_type := (
    synthesis => syn_tkconfig, iu => iu_tkconfig, fpu => fpu_tkconfig,
    cp => cp_none, cache => cache_tkconfig, ahb => ahb_tkconfig,
    apb => apb_tkconfig, mctrl => mctrl_tkconfig, boot => boot_tkconfig,
    debug => debug_tkconfig, pci => pci_tkconfig, peri => peri_tkconfig);

------------------------------------------------------------------------------
-- end of automatic configuration
------------------------------------------------------------------------------


------------------------------------------------------------------------------
-- This is the current device configuration
------------------------------------------------------------------------------
  constant conf : config_type := tkconfig;

end;
```

B.2 LEON processor config.h

```
/*
 * Automatically generated C config: don't edit
 */
#define AUTOCONF_INCLUDED
#define CONFIG_PERI_LCONF 1
/*
 * Synthesis
 */
#define CONFIG_CFG_NAME "tkconfig"
#undef  CONFIG_SYN_GENERIC
#undef  CONFIG_SYN_ATC35
#undef  CONFIG_SYN_ATC25
#undef  CONFIG_SYN_ATC18
#undef  CONFIG_SYN_FS90
#undef  CONFIG_SYN_UMC018
#define CONFIG_SYN_TSMC025 1
#undef  CONFIG_SYN_PROASIC
#undef  CONFIG_SYN_AXCEL
#undef  CONFIG_SYN_VIRTEX
#undef  CONFIG_SYN_VIRTEX2
#undef  CONFIG_SYN_INFER_RAM
#undef  CONFIG_SYN_INFER_REGF
#undef  CONFIG_SYN_INFER_PADS
#undef  CONFIG_SYN_INFER_MULT
#undef  CONFIG_SYN_TRACE_DPRAM
/*
 * Processor and caches
 */
/*
 * Interger unit
 */
#define CONFIG_IU_NWINDOWS (8)
#undef  CONFIG_IU_V8MULDIV
#define CONFIG_IU_LDELAY (1)
#define CONFIG_IU_FASTJUMP 1
#define CONFIG_IU_ICCHOLD 1
#define CONFIG_IU_FASTDECODE 1
#undef  CONFIG_IU_RFPOW
#define CONFIG_IU_WATCHPOINTS (0)
#define CONFIG_IU_IMPL 0x0
#define CONFIG_IU_VER 0x0
/*
 * Floating-point unit
 */
#undef  CONFIG_FPU_ENABLE
/*
 * Co-processor
 */
#undef  CONFIG_CP_ENABLE
/*
 * Cache system
 */
/*
 * Instruction cache
```

```
 */
#define CONFIG_ICACHE_ASSO1 1
#undef  CONFIG_ICACHE_ASSO2
#undef  CONFIG_ICACHE_ASSO3
#undef  CONFIG_ICACHE_ASSO4
#undef  CONFIG_ICACHE_SZ1
#undef  CONFIG_ICACHE_SZ2
#define CONFIG_ICACHE_SZ4 1
#undef  CONFIG_ICACHE_SZ8
#undef  CONFIG_ICACHE_SZ16
#undef  CONFIG_ICACHE_SZ32
#undef  CONFIG_ICACHE_SZ64
#define CONFIG_ICACHE_LZ16 1
#undef  CONFIG_ICACHE_LZ32
/*
 * Data cache
 */
#define CONFIG_DCACHE_ASSO1 1
#undef  CONFIG_DCACHE_ASSO2
#undef  CONFIG_DCACHE_ASSO3
#undef  CONFIG_DCACHE_ASSO4
#undef  CONFIG_DCACHE_SZ1
#undef  CONFIG_DCACHE_SZ2
#define CONFIG_DCACHE_SZ4 1
#undef  CONFIG_DCACHE_SZ8
#undef  CONFIG_DCACHE_SZ16
#undef  CONFIG_DCACHE_SZ32
#undef  CONFIG_DCACHE_SZ64
#define CONFIG_DCACHE_LZ16 1
#undef  CONFIG_DCACHE_LZ32
#undef  CONFIG_DCACHE_SNOOP
/*
 * Memory controller
 */
#define CONFIG_MCTRL_8BIT 1
#undef  CONFIG_MCTRL_16BIT
#undef  CONFIG_MCTRL_WFB
#undef  CONFIG_MCTRL_5CS
#undef  CONFIG_MCTRL_SDRAM
/*
 * AMBA AHB configuration
 */
#define CONFIG_AHB_DEFMST (0)
#undef  CONFIG_AHB_SPLIT
/*
 * Optional modules
 */
#undef  CONFIG_PERI_AHBSTAT
#undef  CONFIG_PERI_WPROT
#define CONFIG_PERI_LCONF 1
#undef  CONFIG_PERI_IRQ2
#undef  CONFIG_PERI_WDOG
#undef  CONFIG_AHBRAM_ENABLE
/*
 * Debug support unit
 */
#undef  CONFIG_DSU_ENABLE
```

```
/*
 * PCI interface
 */
#undef  CONFIG_PCI_ENABLE
/*
 * Fault-tolerance configuration
 */
#undef  CONFIG_FT_ENABLE
/*
 * Boot options
 */
#define CONFIG_BOOT_EXTPROM 1
#undef  CONFIG_BOOT_INTPROM
#undef  CONFIG_BOOT_MIXPROM
/*
 * VHDL Debugging
 */
#undef  CONFIG_DEBUG_UART
#undef  CONFIG_DEBUG_IURF
#undef  CONFIG_DEBUG_NOHALT
#undef  CONFIG_DEBUG_PC32
```

B.3 LEON processor synthesis script

```
#################################################################
# Script to compile leon with synopsys DC                       #
# Jiri Gaisler, Gaisler Research, 2001                          #
# Converted from dcsh to tcl to allow XG mode to be used - paulm #
#################################################################

# libraries are referenced from .synopsys_dc.setup in the usual way - paulm

set frequency 10
set clock_skew 0.10
set input_setup 2.0
set output_delay 4.0

set hdlin_ff_always_sync_set_reset true
set hdlin_translate_off_skip_text true
set sourcedir "../../leon"

if {![file isdirectory WORK]} {file mkdir WORK}
define_design_lib WORK -path WORK
foreach hdlfile [glob -nocomplain -directory $sourcedir -- *.vhd]
  {analyze -format vhdl -library WORK $hdlfile}

elaborate leon
current_design leon
uniquify

ungroup [find cell "*pad*"] -flatten

current_instance mcore0
group [find cell [list "wp*" "asm*" "apb*" "uart*" "timer*" "irq*" \
  "iopo*" "ahb*" "mctrl*" "lc*" "reset*" "dcom*"]]                 \
  -design_name amod -cell_name amod0

current_instance amod0
ungroup -all -flatten
current_instance ../proc0/iu0

ungroup -all -flatten
current_instance ../rf0
ungroup -all -flatten
current_instance ../c0
ungroup -all -flatten
current_instance ../cmem0
ungroup -all -flatten
current_instance ../../..

set peri [expr 1000.0 / $frequency]
set input_delay [expr $peri - $input_setup]
set tdelay [expr $output_delay + 2]
create_clock -name "clk" -period $peri -waveform \
  [list 0.0 [expr $peri / 2.0]] [list "clk"]
set_wire_load_mode segmented

set_clock_uncertainty -hold $clock_skew "clk"
```

```
set_clock_uncertainty -setup $clock_skew "clk"

set_input_delay $input_delay -clock clk [list {pio[15]} {pio[14]}    \
  {pio[13]} {pio[12]} {pio[11]} {pio[10]} {pio[9]} {pio[8]} {pio[7]} \
  {pio[6]} {pio[5]} {pio[4]} {pio[3]} {pio[2]} {pio[1]} {pio[0]}     \
  {data[31]} {data[30]} {data[29]} {data[28]} {data[27]} {data[26]}  \
  {data[25]} {data[24]} {data[23]} {data[22]} {data[21]} {data[20]}  \
  {data[19]} {data[18]} {data[17]} {data[16]} {data[15]} {data[14]}  \
  {data[13]} {data[12]} {data[11]} {data[10]} {data[9]} {data[8]}    \
  {data[7]} {data[6]} {data[5]} {data[4]} {data[3]} {data[2]}        \
  {data[1]} {data[0]} "brdyn" "bexcn"]

set_max_delay $tdelay -to [list "errorn" "wdogn" {pio[15]} {pio[14]} \
  {pio[13]} {pio[12]} {pio[11]} {pio[10]} {pio[9]} {pio[8]} {pio[7]} \
  {pio[6]} {pio[5]} {pio[4]} {pio[3]} {pio[2]} {pio[1]} {pio[0]}     \
  {data[31]} {data[30]} {data[29]} {data[28]} {data[27]} {data[26]}  \
  {data[25]} {data[24]} {data[23]} {data[22]} {data[21]} {data[20]}  \
  {data[19]} {data[18]} {data[17]} {data[16]} {data[15]} {data[14]}  \
  {data[13]} {data[12]} {data[11]} {data[10]} {data[9]} {data[8]}    \
  {data[7]} {data[6]} {data[5]} {data[4]} {data[3]} {data[2]}        \
  {data[1]} {data[0]}]

set_max_delay $output_delay -to [list "writen" {romsn[1]} {romsn[0]} \
  "read" "oen" "iosn" {rwen[3]} {rwen[2]} {rwen[1]} {rwen[0]}        \
  {ramsn[3]} {ramsn[2]} {ramsn[1]} {ramsn[0]} {ramoen[3]}            \
  {ramoen[2]} {ramoen[1]} {ramoen[0]} {sdcsn[1]} {sdcsn[0]}          \
  "sdwen" "sdrasn" "sdcasn" {sddqm[3]} {sddqm[2]} {sddqm[1]}         \
  {sddqm[0]} {address[27]} {address[26]} {address[25]} {address[24]} \
  {address[23]} {address[22]} {address[21]} {address[20]}            \
  {address[19]} {address[18]} {address[17]} {address[16]}            \
  {address[15]} {address[14]} {address[13]} {address[12]}            \
  {address[11]} {address[10]} {address[9]} {address[8]} {address[7]} \
  {address[6]} {address[5]} {address[4]} {address[3]} {address[2]}   \
  {address[1]} {address[0]}]

set_max_area 0
set_max_transition 2.0 leon
set_flatten false -design [list "leon.db:leon"]
set_structure true -design [list "leon.db:leon"] -boolean false -timing true

compile -map_effort medium -boundary_optimization

write -f ddc -hier leon -output leon.ddc

report_timing

current_design mcore
report_area
current_design leon

#Write out both VHDL and Verilog netlists
change_names -rule vhdl -hierarchy
write -format vhdl -hierarchy -output leon_synth.vhd
change_names -rule verilog -hierarchy
write -format verilog -hierarchy -output leon_synth.v

quit
```

B.4 OpenRISC 1200 toolchain build script

```
#!/bin/sh

##############################################################################
#                                                                            #
# Script to download and build GNU/uClinux toolchain for OpenRISC 1200 processor     #
# Written by Paul Morgan, 2004-2007                                          #
# Created with guidance from http://www.meansoffreedom.net/opencores.html    #
#                                                                            #
##############################################################################

TARGET_DIR=/crux/paulm/or1k
TOOLS_DIR="$TARGET_DIR/tools"
OR32_DIR="$TOOLS_DIR/or32-uclinux"
LOG_DIR="$TARGET_DIR/log"

# Pre-existing configuration files can be specified here to avoid having to enter
# settings using the configuration menus
LINUX_CONFIG=/crux/paulm/or1k_archive/linux_config
UCLIBC_CONFIG=/crux/paulm/or1k_archive/uclibc_config

# If the archive files have already been downloaded and stored, the directory can
# be specified here and the existing files will be used, rather than re-downloaded
ARCHIVE_DIR=/crux/paulm/or1k_archive

# List of files that are required for the build
ARCHIVE_FILES="\
  binutils-2.16.1.tar.bz2 binutils_2.16.1_unified.diff_rgd_fixed.bz2     \
  gcc-3.4.4-or32-unified.diff.bz2 gcc-3.4.4.tar.bz2 linux-2.6.19.tar.bz2 \
  linux_2.6.19_or32_unified_simtested.bz2 uClibc-0.9.28.3.tar.bz2        \
  uClibc-0.9.28-or32-libc-support.bz2 uClibc-0.9.28-or32-unified.bz2     "


# If target directory exists from a previous build, remove it
if [ -d "$TARGET_DIR" ]; then
  rm -rf "$TARGET_DIR"
fi

mkdir "$TARGET_DIR"
mkdir "$LOG_DIR"

# Check whether the archive directory exists, and if so attempt to copy the
# list of required files from it. If successful the FILES_OK flag is set,
# otherwise the required files will be downloaded
if [ -d "$ARCHIVE_DIR" ]; then
  cd "$ARCHIVE_DIR"
  cp $ARCHIVE_FILES "$TARGET_DIR"
  if [ $? -eq 0 ]; then
    echo "Successfully copied archive files to target directory."
    FILES_OK=1
  fi
fi

cd "$TARGET_DIR"
```

```
# If the required files were not successfully copied from the archive directory,
# they are downloaded here. The success of each download is tested individually
# and if any fails the script cannot proceed so exits with an error.
if [ $FILES_OK -ne 1 ]; then
  echo "Complete file set not available from archive, downloading instead."

  # GNU binutils and gcc compiler, and or32 patches
  GNU_FILES="\
    http://ftp.gnu.org/gnu/binutils/binutils-2.16.1.tar.bz2                  \
    http://ftp.gnu.org/gnu/gcc/gcc-3.4.4/gcc-3.4.4.tar.bz2                   \
    http://www.meansoffreedom.net/binutils_2.16.1_unified.diff_rgd_fixed.bz2 \
    http://www.meansoffreedom.net/gcc-3.4.4-or32-unified.diff.bz2            "

  # Linux kernel 2.6.19 and or32 patch
  LINUX_FILES="\
    ftp://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.19.tar.bz2          \
    http://www.meansoffreedom.net/linux_2.6.19_or32_unified_simtested.bz2    "

  # uClibc and patches
  UCLIBC_FILES="\
    http://www.uclibc.org/downloads/uClibc-0.9.28.3.tar.bz2                  \
    http://www.meansoffreedom.net/uClibc-0.9.28-or32-unified.bz2             \
    http://www.meansoffreedom.net/uClibc-0.9.28-or32-libc-support.bz2        "

  touch "$LOG_DIR/downloads.txt"

  for current_file in $GNU_FILES $LINUX_FILES $UCLIBC_FILES; do
    wget $current_file >> "$LOG_DIR/downloads.txt" 2>&1
    if [ $? -ne 0 ]; then
      echo "Error downloading file $current_file."
      echo "Build failed."
      exit 1
    fi
  done

  # If the archive directory exists, copy the successfully downloaded files to it
  if [ -d "$ARCHIVE_DIR" ]; then
    cp $ARCHIVE_FILES "$ARCHIVE_DIR"
  fi

fi

# Unpack and patch GNU binutils
echo "Starting binutils unpack and patch." |tee "$LOG_DIR/binutils.txt"
tar -jxf binutils-2.16.1.tar.bz2
cd binutils-2.16.1
bunzip2 -c ../binutils_2.16.1_unified.diff_rgd_fixed.bz2 |patch -p1 --quiet
cd ..

# Build the GNU binutils and add the binary directory to the system path
echo "Starting binutils build." |tee -a "$LOG_DIR/binutils.txt"
mkdir b-b
mkdir tools
cd b-b
../binutils-2.16.1/configure --target=or32-uclinux --prefix="$OR32_DIR" \
  >> "$LOG_DIR/binutils.txt" 2>&1
make all install >> "$LOG_DIR/binutils.txt" 2>&1
```

```
export PATH="$OR32_DIR/bin:$PATH"
cd ..

# Unpack and patch the Linux kernel
echo "Starting linux unpack and patch." |tee "$LOG_DIR/linux.txt"
tar -jxf linux-2.6.19.tar.bz2
cd linux-2.6.19
bunzip2 -c ../linux_2.6.19_or32_unified_simtested.bz2 |patch -p1 --quiet


# Check whether a pre-existing .config file is specified and copy it to
# the appropriate location. Otherwise use the menu-based configuration
# utility. In the latter case or32 must be selected as the platform
if [ -f "$LINUX_CONFIG" ]; then
  cp "$LINUX_CONFIG" ./.config
  make oldconfig ARCH=or32 >> "$LOG_DIR/linux.txt" 2>&1
else
  make menuconfig ARCH=or32 2>> "$LOG_DIR/linux.txt"
fi
cd ..

# Copy the relevant files from Linux to uClinux for use with OpenRISC
mkdir -p tools/or32-uclinux/include/asm
mkdir tools/or32-uclinux/include/linux
cp -f -dR linux-2.6.19/include/linux/* tools/or32-uclinux/include/linux/
cp -f -dR linux-2.6.19/include/asm-or32/* tools/or32-uclinux/include/asm/
cd tools/or32-uclinux/
ln -s include sys-include
cd ../..

# Unpack and patch gcc
echo "Starting gcc unpack and patch." |tee "$LOG_DIR/gcc.txt"
tar -jxf gcc-3.4.4.tar.bz2
cd gcc-3.4.4
bunzip2 -c ../gcc-3.4.4-or32-unified.diff.bz2 |patch -p1 --quiet
cd ..

# Configure, build and install the gcc or32-uclinux cross-compiler
echo "Starting gcc build." |tee -a "$LOG_DIR/gcc.txt"
mkdir b-gcc
cd b-gcc
../gcc-3.4.4/configure --target=or32-uclinux --prefix="$OR32_DIR"          \
  --with-local-prefix="$OR32_DIR/or32-uclinux" --with-gnu-as --with-gnu-ld \
  --verbose --enable-languages=c >> "$LOG_DIR/gcc.txt" 2>&1
make all install >> "$LOG_DIR/gcc.txt" 2>&1
cd ..

# Cross-compile Linux using the or32-uclinux build tools
echo "Cross-compiling linux." |tee -a "$LOG_DIR/linux.txt"
cd linux-2.6.19
make vmlinux ARCH=or32 CROSS_COMPILE="$OR32_DIR/bin/or32-uclinux-" \
  >> "$LOG_DIR/linux.txt" 2>&1
cd ..

# Unpack and patch uClibc
echo "Starting uclibc unpack and patch." |tee "$LOG_DIR/uclibc.txt"
tar -jxf uClibc-0.9.28.3.tar.bz2
```

```
cd uClibc-0.9.28.3
bunzip2 -c ../uClibc-0.9.28-or32-unified.bz2 |patch -p1 --quiet
cd libc
bunzip2 -c ../../uClibc-0.9.28-or32-libc-support.bz2 |patch -p1 --quiet

cd ..
ln -s extra/Configs/Config.or32 Config

# Want to cross-compile uClibc for or32-uclinux so set the target C compiler
export CC=or32-uclinux-gcc

# Check whether a pre-existing .config file is specified and copy it to
# the appropriate location. Otherwise use the menu-based configuration
# utility. In the latter case or32 must be selected as the Target Arch,
# under Target Features the path to the Linux kernel must be set, under
# Library Installation Options set the development directory to the location
# of the or32-uclinux tool chain. Position Independent Code must be disabled
if [ -f "$UCLIBC_CONFIG" ]; then
  cp "$UCLIBC_CONFIG" ./.config
  make oldconfig >> "$LOG_DIR/uclibc.txt" 2>&1
else
  make menuconfig 2>> "$LOG_DIR/uclibc.txt"

fi

# Build and install the uClibc components, then reset the C compiler to
# the host native version
echo "Starting uclibc build." |tee -a "$LOG_DIR/uclibc.txt"
make clean >> "$LOG_DIR/uclibc.txt" 2>&1
make all install >> "$LOG_DIR/uclibc.txt" 2>&1
unset CC
cd ..

# Re-build the gcc cross-compiler, this time integrating the cross-compiled
# uClibc components
echo "Starting gcc with uclibc build." |tee -a "$LOG_DIR/gcc.txt"
cd b-gcc
../gcc-3.4.4/configure --target=or32-uclinux --prefix="$OR32_DIR"          \
  --with-local-prefix="$OR32_DIR/or32-uclinux" --with-gnu-as --with-gnu-ld \
  --verbose --enable-languages=c >> "$LOG_DIR/gcc.txt" 2>&1
make all install >> "$LOG_DIR/gcc.txt" 2>&1
cd ../tools/or32-uclinux/or32-uclinux
ln -s ../include sys-include
cd lib
cp ../../lib/*.* .

cd $TARGET_DIR

echo "Completed."
```

C. MediaBench build/test scripts

The files listed below are used by Cascade as part of the automated coprocessor test build process for the MediaBench suite, as detailed in chapter 5. The `build.tcl` file listed in section C.1 is used to build the pgp_encode test. Build files for other tests have a similar structure, with some minor specifics targeted to each individual test. Similarly, `test.tcl` listed in section C.2 runs the pgp_encode test after it has been built. Finally, the `default.xml` file listed in section C.3 defines the configuration used for all MediaBench tests.

C.1 Sample MediaBench build.tcl

```
##############################################################################
# Author:  paulm                                                             #
# Date  :  15/02/2006                                                        #
#                                                                            #
# Build file used by Cascade to build pgp_encode test                        #
# Based on build.tcl located in SystemTest/Tests/MediaBench/Test_g721_decode #
##############################################################################

# Ensure the object file directory is empty before starting
file delete -force obj
file mkdir obj

# Delete PGP temp files otherwise test will fail once number
# of temp files reaches 100 (2-digit filename limit)
foreach tempfile [glob -nocomplain data/pgptest.pl.\$*] {
  file delete $tempfile
}

puts stdout "Compiling Source Code..."

set cb_build true
set build_dir "[pwd]"
set source_dir $build_dir/src

# Build the CB Libraries
#
```

```
if {$cb_build} {
  puts stdout "Compiling CB Libraries..."
  foreach cfile $src_driver_list {
    set command "exec $gCC $compileFlags -o \
       obj/[file rootname [file tail $cfile]].o $cfile"
    puts $command
    eval $command
  }
  puts stdout "Assembling CB Libraries..."
  foreach asmfile $asm_driver_list {
    set asmCommand "exec $gASM $asmFlags -o \
       obj/[file rootname [file tail $asmfile]].o $asmfile"
    puts $asmCommand
    eval $asmCommand
  }
}

set compileFlags [concat $compileFlags  -Irsaref/source \
   -Irsaref/test -DUSEMPILIB -O -DPORTABLE -DMPORTABLE -DIDEA32]
set csrc_list [glob -nocomplain $source_dir/*.c]
foreach cfile $csrc_list {
  set command "exec $gCC $compileFlags -I$source_dir -o \
     obj/[file rootname [file tail $cfile]].o $cfile"
  puts $command
  eval $command
}

cd obj
set build_cmd "exec $gLINK $linkFlags [glob -nocomplain -- *.o] \
   ../rsaref/test/rsaref.a"
puts $build_cmd
eval $build_cmd
cd $build_dir

puts stdout "Done!"

# Ensure randseed.bin is set to read-only status, otherwise PGP changes it
# on each run resulting in a diff failure due to session key changes. We
# are effectively compromising the PGP security for testability.

global tcl_platform
if {$tcl_platform(host_platform) == "windows"} {
  exec attrib +r randseed.bin
}
else {
  exec chmod -w randseed.bin
}
```

C.2 Sample MediaBench test.tcl

```
##############################################################################
# Author:  paulm                                                             #
# Date  :  15/02/2006                                                        #
#                                                                            #
# Test file used by Cascade to offload function(s) from pgp_encode test      #
# Based on test.tcl located in SystemTest/Tests/MediaBench/Test_g721_decode  #
##############################################################################

global gConfig_usecommon

set gConfig_usecommon 0

# Procedure to select the functions to be offloaded to a Cascade
# coprocessor. Mapping an ENTRY function group will offload both
# the listed function and any children of that function, ensuring
# that control will only return to the host once the parent function
# has exited. Note that in cases where all child functions cannot
# be statically determined (e.g. due to function pointers), it is
# necessary to explicitly offload child functions as LOCAL.
proc Map {} {
        copro_map_function_group ENTRY ideaCfbEncrypt
}

# Relax configuration options to allow Cascade to determine the best solution
proc ConfigureArchSynth {} {
        setRelaxedAll
}

proc ConfigureCodeGen {} {
        setRelaxedAll
}

# Set up the memory configuration explicitly
proc ConfigureCustomMemoryConfig {} {
        generate_memory_config "0.35" {access_st_1r}
}
```

C.3 MediaBench configuration file—default.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<testconfig xmlns="http://www.criticalblue.com/CascadeNS"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.criticalblue.com/CascadeNS
   TestConfig.xsd">

        <configuration>
                <opt mode="2" />
                <connection mode="local" />
                <instrumentation mode="true" />
                <midas mode="single" weight="0.05" />
                <keepSeparateProjects mode="false" />
                <enableARMLinkedStage mode="true" />
                <exclusions ignore="false" />

<!-- For CodeSourcery version 3.4.4-8 -->
                <toolchain env="gcc"
                        iss="arm-none-eabi-run -m 33554432"
                        compile="arm-none-eabi-gcc"
                        assemble="arm-none-eabi-as"
                        link="arm-none-eabi-gcc"
                        compileFlags="-mcpu=arm9e -g"
                        assembleFlags="-mcpu=arm9e -g"
                        linkFlags="-Wl,-M"
                        version="4.1"/>
        </configuration>

        <tests>
                <test mode="case" name="MediaBench/Test_pgp_decode" />
        </tests>

        <scripts>
                <testscript file="InitTestPre" />
                <testscript file="Profile" />
                <testscript file="Instrumentation" />
                <testscript file="VMSim" />
                <testscript file="RegionSim" />
                <testscript file="InitMemoryConfig" />
                <testscript file="CandidateGeneration" />
                <testscript file="InitCand" />
                <testscript file="Microcode" />
                <testscript file="BaseCDFG" />
                <testscript file="IdealCDFG" />
                <testscript file="TargetCDFG" />
                <testscript file="AllocatedCDFG" />
                <testscript file="ScheduledCDFG" />
                <testscript file="ArchSim" />
                <testscript file="Hardware" />
                <testscript file="Snapshot" />
                <testscript file="QoR" />
                <testscript file="DebugFiles" />
                <testscript file="Linked" />
        </scripts>
</testconfig>
```

D. Functional unit analysis files

The files listed below are used for analysing functional units, as detailed in chapter 6. The simulation script listed in section D.1 is adapted from that listed in section A.1. The key changes allow for a single hardware configuration to be driven by multiple stimulus files in individual simulations, allowing the switching activity of the output banks to be characterised under varying operating conditions. The testbench in section D.2 is used within the simulation, parsing the stimulus file and supplying the inputs to the unit under test as required.

D.1 Output bank simulation script

```
#!/bin/sh
#
# Output bank test script, used to build and analyse the power consumption of
# an output bank under various conditions of input stimulus
#
# Written by Paul Morgan, 2005-2006
#
###############################################################################
# Command-line parameters:                                                    #
#  -n, --no_build   : Do not build a new simulation, use existing simulation  #
###############################################################################

usage()
{
   echo "Usage: $0 [options]"
   echo
   echo "Options:"
   echo "-n, --no_build   : Do not build a new simulation, use existing simulation"
   echo
   echo "Examples:"
   echo "$0 --no_build"
   echo "$0 -n"
   exit 1
}
```

```
# Reset optional flag used to disable part of the script
unset $NO_SIM_BUILD

# Output a blank line before any messages to improve readability
echo

# Parse command line input and set appropriate flags
if [ $# -gt 1 ]; then
   usage
fi

# Create a variable with name of current design
design_name="gl_output_bank"

# Ensure reports directory is available
if [ ! -d reports ]; then
        mkdir reports
fi

# Use GNU getopt to parse the input string, and store the output status. This is
# done to allow any errors indicated by getopt to be temporarily ignored so that
# we can parse the input string and highlight the offending option flag. Getopt's
# output status is then checked in case any errors aren't caught by the parsing
# done within the script
input_string=(`getopt -q -on -lno_build -- $@`)
getopt_status=$?

for (( i = 0; i < ${#input_string[*]}; i++ )); do
   if [ ! ${input_string[$i]} == "--" ]; then
     case ${input_string[$i]} in
       -n | --no_build)   NO_SIM_BUILD="TRUE";;
       *)                 echo "Invalid command line option \"${input_string[$i]}\"."
                          echo
                          usage;;
     esac
   fi
done

if [ $getopt_status -ne 0 ]; then
   echo "Invalid command line input: $@"
   usage
fi

#####################################################################
#                                                                   #
# Simulation build section                                          #
#                                                                   #
#####################################################################

cd sim

# Command line option to skip simulation build for previously built designs
if [ $NO_SIM_BUILD ]; then
        echo "Skipping simulation build."

else
```

```
        # Start simulation process
        echo "Simulation build starting:" `date`
        echo "Simulation build started:" `date` > ../reports/sim_main.rpt

        # Ensure the work directory exists and is empty
        if [ -d work ]; then
                rm -rf work/*
        else
                mkdir work
        fi

        # Ensure SAIF output directory exists and is empty
        if [ -d saif ]; then
                rm -rf saif/*
        else
                mkdir saif
        fi

        # Analyse TSMC gate-level library
        vhdlan -nc -event \
                ~/synopsys/libraries/tsmc13/tsmc13.vhd

        # Analyse unit under test
        vhdlan -nc -event \
        ../$design_name\_synth.vhd

        # Analyse testbench
        vhdlan -nc -event \
                ../$design_name\_tb.vhd

        # Build simulation
        scsi $design_name\_tb >> ../reports/sim_main.rpt

        # Check for errors in the build process, and exit before running scsim
        if [ $? = 1 ]; then
        echo "Simulation build failed, exiting..."; exit;
        fi

        echo "Simulation build complete:" `date`
        echo "Simulation build completed:" `date` >> ../reports/sim_main.rpt

fi

#######################################################################
#                                                                     #
# Simulation execution section                                        #
#                                                                     #
#######################################################################


# Remove old saif file if it exists to allow later check that new file
# has been generated successfully
rm -f saif/backward*.saif

# Perform loop to process all stimulus files present within stimulus directory
for FILE in stimulus/stimulus*.txt; do
```

```
  echo "Simulating for input stimulus $FILE:" `date` >> ../reports/sim_main.rpt
  echo "Simulating for input stimulus $FILE:" `date`

        # Create a symbolic link to current stimulus file for testbench
        ln -sf $FILE stimulus.txt

        # Pass unique part of stimulus name to scsim for naming backward SAIF file
        FILE_EXT=`echo $FILE|sed -e "s/stimulus\/stimulus//" -e "s/.txt//"`

        # Using a blank variable for FILE_EXT trips up scsim so temporarily change it
        if [[ $FILE_EXT = "" ]]; then
                FILE_EXT="_no_name"
        fi

        # Botch to allow passing of design_name and stimulus_file
  # environmental variables into scsim
        printf "set design_name $design_name\nset file_ext
    $FILE_EXT\nsource sim.include_main\n" > sim.include_top

        scsim -i sim.include_top > ../reports/sim$FILE_EXT\.rpt

        # Check that similation has completed
        grep "\"Simulation complete.\"" ../reports/sim$FILE_EXT\.rpt >/dev/null
        if [ $? = 1 ]; then
                echo "Simulation not completed processing $FILE, exiting..."; exit
        fi

        # Check that simulation did not encounter errors
        grep error -i ../reports/sim$FILE_EXT\.rpt >/dev/null
        if [ $? = 0 ]; then
                echo "Simulation encountered errors processing $FILE,
       see reports/sim.rpt; exiting..."; exit
        fi

        # Check that the backward SAIF file has been successfully generated
        if [ ! -f saif/backward$FILE_EXT\.saif ]; then
        echo "SAIF file not generated processing $FILE, exiting..."; exit;
        fi

done

# If temporary "_no_name" was used return to the correct matching name
if [ -f saif/backward_no_name.saif ]; then
        mv saif/backward_no_name.saif saif/backward.saif
fi

if [ -f ../reports/sim_no_name.rpt ]; then
        mv ../reports/sim_no_name.rpt ../reports/sim.rpt
fi

echo "Simulation finishing:" `date`
echo "Simulation completed:" `date` >> ../reports/sim_main.rpt

cd ..
```

```
#######################################################################
#                                                                     #
# Power analysis section                                              #
#                                                                     #
#######################################################################

echo "Power analysis starting:" `date`
echo "Power analysis started:" `date` > reports/power.rpt
dc_shell-xg-t -x "set design_name $design_name" -f power.tcl >> reports/power.rpt

# Check for errors in the power analysis process
if [ $? = 1 ]; then
   echo "Power analysis failed, exiting..."; exit;
fi

echo "Power analysis finishing:" `date`
echo "Power analysis completed:" `date` >> reports/power.rpt

unset design_name
```

D.2 Output bank testbench

```
-- ******************************************************************************
-- Project:     CascadeLibrary
-- File:        gl_output_bank_tb.vhd
-- Original:    Created on Oct 17, 2005 by paulm
-- ******************************************************************************

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_textio.all;
use std.textio.all;

entity gl_output_bank_tb is
  generic (
  out_bank_registers      : integer := 16;
  out_bank_register_width : integer := 32
  );

end gl_output_bank_tb;

architecture behavior of gl_output_bank_tb is

  component gl_output_bank

  generic(
    out_bank_registers      : integer;
    out_bank_register_width : integer
  );

  port(
    clk_i          : in std_logic;
    n_reset_i      : in std_logic;
    n_wait_flag_i  : in std_logic;
    data_i         : in std_logic_vector(out_bank_register_width-1 downto 0);
    out_reg_mask_i : in std_logic_vector(out_bank_registers-1 downto 0);
    data_o         : out std_logic_vector
                       ((out_bank_register_width*out_bank_registers)-1 downto 0)
  );
  end component;

  signal clk_i          : std_logic := '0';
  signal n_reset_i      : std_logic;
  signal n_wait_flag_i  : std_logic;
  signal data_i         : std_logic_vector(out_bank_register_width-1 downto 0);
  signal out_reg_mask_i : std_logic_vector(out_bank_registers-1 downto 0);
  signal data_o         : std_logic_vector
                     ((out_bank_register_width*out_bank_registers)-1 downto 0);

BEGIN


  uut: gl_output_bank
  generic map(
    out_bank_registers      => out_bank_registers,
```

```
  out_bank_register_width => out_bank_register_width
)

port map(
  clk_i             => clk_i,
  n_reset_i         => n_reset_i,
  n_wait_flag_i     => n_wait_flag_i,
  data_i            => data_i,
  out_reg_mask_i    => out_reg_mask_i,
  data_o            => data_o
);

clk_i <= not clk_i after 5 ns;
n_reset_i <= '1', '0' after 20 ns, '1' after 40 ns;


testbench : process

file     stimulus_file       : text is in "stimulus.txt";
variable stimulus_line       : line;
variable data_i_stim
          : std_logic_vector(out_bank_register_width-1 downto 0);
variable out_reg_mask_i_stim
          : std_logic_vector(out_bank_registers-1 downto 0);
variable read_check          : boolean;

begin


--initial configuration
n_wait_flag_i     <= '0';
data_i            <= (others => '0');
out_reg_mask_i    <= (others => '0');
wait for 105 ns;

--activate unit and cycle through input stimulus
n_wait_flag_i     <= '1';
wait for 10 ns;

while not endfile(stimulus_file) loop

  readline(stimulus_file, stimulus_line);
  if (stimulus_line(1) /= '#') then             -- ignore comment lines

    hread(stimulus_line, data_i_stim, read_check);
    assert read_check
      report "File read error reading data_i." severity error;

    hread(stimulus_line, out_reg_mask_i_stim, read_check);
    assert read_check
      report "File read error reading out_reg_mask_i." severity error;

    data_i         <= data_i_stim;
    out_reg_mask_i <= out_reg_mask_i_stim;

    wait for 10 ns;
  end if;
```

```
  end loop;

  data_i            <= (others => '0');
  out_reg_mask_i    <= (others => '0');

  wait for 20 ns;
  n_wait_flag_i     <= '0';

  wait for 5 ns;

  assert FALSE report "Simulation complete." severity failure;
  end process;

end behavior;
```

D.3 Functional unit active cycle energy script

```
#!/bin/sh
#
# Script to analyse dynamic power for a list of functional units. Examines all
# relevant hierarchical power reports for instances of the unit in question,
# and parses the result for that unit to generate an average dynamic power over
# all power reports. This is done by writing a script file for the Desk
# Calculator (dc) tool using Reverse Polish Notation, which is then executed by
# dc to generate the result.
#
# Written by Paul Morgan, 2007.
#
###############################################################################
# Command-line parameters:                                                    #
#  -v, --verbose   : Increase output verbosity, list intermediate values      #
#  -q, --quiet     : Display only final average values for each unit           #
###############################################################################

usage()
{
   echo "Usage: $0 [-v|--verbose] [-q|--quiet]"
   echo
   echo "Options:"
   echo "-v, --verbose   : Increase output verbosity, list intermediate values"
   echo "-q, --quiet     : Display only final average values for each unit"
   echo
   echo "Options verbose and quiet are mutually exclusive"
   echo
   echo "Examples:"
   echo "$0 --verbose"
   echo "$0"
   exit 1
}

# Reset optional flags used to control verbosity
unset $VERBOSE $QUIET

# Parse command line input and set appropriate flags
if [ $# -gt 1 ]; then
   usage
fi

# Use GNU getopt to parse the input string, and store the output status. This
# is done to allow any errors indicated by getopt to be temporarily ignored so
# that we can parse the input string and highlight the offending option flag.
# Getopt's output status is then checked in case any errors aren't caught by
# the parsing done within the script
input_string=(`getopt -q -ovq -lverbose,quiet -- $@`)
getopt_status=$?

for (( i = 0; i < ${#input_string[*]}; i++ )); do
  if [ ! ${input_string[$i]} == "--" ]; then
    case ${input_string[$i]} in
      -v | --verbose)    VERBOSE="TRUE";;
      -q | --quiet)      QUIET="TRUE";;
```

```
      *)                echo "Invalid option \"${input_string[$i]}\"."
                        echo
                        usage;;
    esac
  fi
done

# Check if both verbose and quiet options have been set simultaneously
if  [ $VERBOSE ] && [ $QUIET ]; then
  echo "Verbose and quiet options cannot be set simultaneously"
  usage
fi

if [ $getopt_status -ne 0 ]; then
  echo "Invalid command line input: $@"
  usage
fi

# Debug mode flag - set to 1 to increase verbosity of output
debug=0

clock_freq=10000000 # 10 MHz

ACTIVE_ENERGY_LOG=active_energy_list.txt

tests_list="adpcm_encode epic_decode g721_decode g721_encode gsm_decode \
            gsm_encode jpeg_decode jpeg_encode mpeg2_decode.fft         \
            mpeg2_decode.ref mpeg2_encode pgp_decode pgp_encode"

cat > $ACTIVE_ENERGY_LOG << EOF
# File recording the active energy per cycle calculated for each
# functional unit within each test. This is used to calculate the
# mean active energy per cycle across all units. These values are
# recorded in the format (comma seperated):
# <functional unit> <active energy per cycle> <test name>
EOF

# Function to determine the energy per active cycle for all the functional
# units present in each test being analysed. All these values are output
# to the log file $ACTIVE_ENERGY_LOG that can later be used to determine
# the average energy per active cycle for each functional unit type
get_active_energy_cycle()
{
  for current_test in $tests_list
  do

    # Remove any old copy of the log file for this test in case noclobber is set
    if [ -f dc_calc_$test.txt ]; then
      rm dc_calc_$test.txt
    fi

    # Define the files used for this test
    analysis_file=$current_test\_analysis_summary.txt
    power_file=$current_test\_power_hier1.txt

    # Check for presence of both files
    if [ ! -f $analysis_file ] || [ ! -f $power_file ]; then
```

```
  echo "File(s) missing for test ${current_test}; skipping"
  continue
fi

# Convert both files to UNIX format, as carriage returns cause problems
dos2unix -q $analysis_file
dos2unix -q $power_file

# Get the total cycle count of the current test
cycle_count=`grep -m 1 "Total Cycles" $analysis_file | cut -d " " -f 3`

# Get a list of the functional units used in this test's coprocessor
func_units_list=`grep fu_ $power_file | cut -d " " -f 3 | cut -b 4-`

# Cycle through each functional unit, extracting the required information
# from the text files, in order to calculate the active cycle energy

for func_unit in $func_units_list
do

  # Determine the switching power for this functional unit
  switch_power=`grep fu_$func_unit $power_file | awk '{printf $3}'`
  # Check whether current value has an exponent, if not add one
  echo $switch_power | grep -q e
  if [ $? -eq 1 ]; then
    switch_power=$switch_power\e+00
  fi

  # Need to change the value into a format usable by dc
  switch_power=`echo $switch_power|sed -e 's/e+/ /' -e 's/e-/ _/'`
  switch_power=`dc -e "10 k $switch_power 10 r ^ * p"`

  if [ $VERBOSE ]; then
    echo "Switching power for unit $func_unit is $switch_power"
  fi

  # determine the internal power for this functional unit
  int_power=`grep fu_$func_unit $power_file | awk '{printf $4}'`
  # Check whether current value has an exponent, if not add one
  echo $int_power | grep -q e
  if [ $? -eq 1 ]; then
    int_power=$int_power\e+00
  fi

  # Need to change the value into a format usable by dc
  int_power=`echo $int_power|sed -e 's/e+/ /' -e 's/e-/ _/'`
  int_power=`dc -e "10 k $int_power 10 r ^ * p"`

  if [ $VERBOSE ]; then
    echo "Internal power for unit $func_unit is $int_power"
  fi

  # Determine the number of active cycles for this functional unit
  active_cycles=$(echo $(grep $func_unit $analysis_file | grep /active | \
   grep -v ram_ | cut -d " " -f 2 | grep -o "[0-9]*")                  \
   + + + + + + + + + + p | dc 2> /dev/null)
  if [ $VERBOSE ]; then
```

```
      echo "Active cycles for $func_unit is $active_cycles"
    fi

    # Determine the inactive energy for this functional unit
    inactive_energy=$(echo 10 k $(grep $func_unit $analysis_file |        \
      grep /inactive | grep -v ram_ | cut -d " " -f 2 |                   \
      grep -o "[0-9]*[.]*[0-9]*") + + + + + + + + + + p | dc 2> /dev/null)
    if [ $VERBOSE ]; then
      echo inactive energy for $func_unit: `grep $func_unit $analysis_file \
        | grep /inactive | grep -v ram_ | cut -d " " -f 2 |                \
        grep -o "[0-9]*[.]*[0-9]*"`
      echo "Inactive energy for $func_unit is $inactive_energy"
      echo
    fi

    # Check whether this unit has no active cycles, as the energy per
    # active cycle cannot be calculated in this case
    if [ -z "$active_cycles" ]; then
      active_energy_cycle="N/A. No active cycles for this unit."
    else
      active_energy_cycle=`dc -e "10 k $switch_power $int_power + 1000000 * \
        $cycle_count * $clock_freq / $inactive_energy - $active_cycles / p"`
      echo "${func_unit},${active_energy_cycle},${current_test}" \
        >> $ACTIVE_ENERGY_LOG
    fi
    if [ $VERBOSE ]; then
      echo Total energy for $func_unit is `dc -e "10 k $switch_power \
        $int_power + 1000000 * $cycle_count * $clock_freq / p"`
    fi
    if [ ! $QUIET ]; then
      echo "Active energy per cycle for $func_unit is $active_energy_cycle"
      echo
    fi

  done

  # Output a dot to the screen after each test in quiet mode, to indicate
  # the script is still running in case of long runs
  if [ $QUIET ]; then
    printf "."
  fi

  done
}

# Function used to determine the average energy per active cycle for functional
# units. Reads in a list of functional units and their energy per active cycle
# from the log file $ACTIVE_ENERGY_LOG, and calculates the average for all
# functional unit types present in the log.
get_units_energy()
{
  if [ ! -f "$ACTIVE_ENERGY_LOG" ]; then
    echo "No log file found: $ACTIVE_ENERGY_LOG"
    exit
  fi

  units_list=`cut -s -d , -f 1 $ACTIVE_ENERGY_LOG | sed 's/[a-z0-9_]$//g' \
```

```
    | sort -u | sed 's/[a-z0-9_]$//g'`

  if [ -z "$units_list" ]; then
    echo "No units found in log file: $ACTIVE_ENERGY_LOG"
    exit
  fi

  if [ $QUIET ]; then
    echo
    echo "Average energy per active cycle for units:"
  fi

  for current_unit in $units_list
  do
    occurences=`grep -c $current_unit active_energy_list.txt`
    if [ ! $QUIET ]; then
      echo "Analysing unit $current_unit, $occurences occurences."
    fi

    dc_list="`grep $current_unit active_energy_list.txt | cut -d , -f \
       2` ++++++++++++++++"

    average_energy=`dc -e "10 k $dc_list ++++++++++++++++ $occurences / p" \
       2>/dev/null`

    if [ ! $QUIET ]; then
      echo "Average energy per active cycle for $current_unit is: \
         $average_energy"
      echo
    else
      echo "$current_unit: $average_energy"
    fi

  done
}

get_active_energy_cycle
get_units_energy
```

E. Memory energy analysis code

Analysis of the energy consumed by memory blocks, as detailed in chapter 7, is mainly carried out in two stages. First a memory library is built in CSV format using the shell script listed in section E.1. This is done by analysing the data files provided by the memory vendor, in the absence of a more detailed model. In the second stage, a Java class, listed in section E.2 takes as input statistics on memory accesses, based on which it parses the aforementioned memory library and calculates the total energy consumed by the memory blocks. The Java class uses the opencsv library developed by Glen Smith to parse text files in comma separated value (CSV) format. This library is open-source and distributed under the Apache License v2.0 which allows commercial use; it can be freely downloaded from `http://opencsv.sourceforge.net`.

The generated CSV memory libraries for 130 nm and 90 nm TSMC process technologies are listed in section E.3 and section E.4 respectively.

E.1 Memory library creation script

```
#!/bin/sh
#
# Script to filter memory data files and produce a look-up table
# text file containing the power figures for all memories available
# in the library.
#
# First section applies only to single-port memories
# and "wr_wr*" dual-port memories in this script
#
# Second section applies to "dp_*" dual-port memories due to different data format
#
# This section creates a text file in the format:
# No. Words, No. Bits, Voltage, Frequency, Read Current, Write Current,
# Deselected Current, Standby Current
#
```

```
# The first two values are integer, all others are floating-point
#
# Written by Paul Morgan, 2005-2006

# Set the location of the memory library (generated memories)
mem_lib_location="/opt/Artisan/CompiledMemories/TSMC_130"

output_file="memory_library.csv"

echo "#, Look-up table for memory energy values (except \"dp_*\" memories)" \
      > $output_file
printf "#, type,words,bits,volt,freq,icc_read,icc_write,icc_desel,icc_standby\n#,\n"
      >> $output_file

for currentfile in `find $mem_lib_location \
   -name "sp*.dat" -o -name "rw_s*.dat"  -o -name "wr_wr*.dat"`
do

   case $currentfile in
      $mem_lib_location\/sp_rw* )
         printf "sp_rw," >> $output_file;
         ;;
      $mem_lib_location\/rw* )
         printf "rw," >> $output_file;
         ;;
      $mem_lib_location\/wr_wr* )
         printf "wr_wr," >> $output_file;
         ;;
   esac

   if [ -r $currentfile ]; then

      awk '{
      if ($5 ~ /typical,/) {
         volt=$6
         column=3
         }
      else if ($8 ~ /typical,/) {
         volt=$9
         column=4
         }
      else if ($3 ~ /words*/) {
         split ($3, wordsout, "=");
         split ($4, bitsout, "=");
         split (volt, voltout, ",");
         split ($8, freqout, "=");
         printf wordsout[2]",";
         printf bitsout[2]",";
         printf voltout[1]",";
         printf freqout[2]",";
         }
      else {
         # Total 4 columns, column 3 contains "typical" values
         if (/icc_r|icc_w|icc_desel/ && column == 3) {printf $3",";}
         # Final entry on line therefore dont place comma at the end
         else if (/icc_standby/ && column == 3) {printf $3;}
         # Total 5 columns, column 4 contains "typical" values
```

```
         else if (/icc_r|icc_w|icc_desel/ && column == 4) {printf $4",";}
         # Final entry on line therefore dont place comma at the end
         else if (/icc_standby/ && column == 4) {printf $4;}
         }
      }
      END {printf "\n"}
      ' $currentfile >> $output_file

   else

      echo "#, Cannot read input file $currentfile" >> $output_file

   fi

done

# This section applies only to "dp_*" dual-port memories in this script
#
# Creates a text file in the format:
# No. Words, No. Bits, Voltage, Frequency, Port A R/W Current, Port B R/W Current,
# Port A Deselected Current, Port B Deselected Current, Standby Current
#
# The first two values are integer, all others are floating-point

echo "#," >> $output_file
echo "#," >> $output_file
echo "#, Look-up table for memory energy values (dual-port memories)" \
      >> $output_file
printf "#, type,words,bits,volt,freq,icc_rw_a,icc_rw_b,"
printf "icc_desel_a,icc_desel_b,icc_standby\n#,\n" >> $output_file

for currentfile in `find $mem_lib_location -name "dp*.dat"`; do

   case $currentfile in
      $mem_lib_location\/dp* )
         printf "dp," >> $output_file;
         ;;
   esac

   if [ -r $currentfile ]; then

      awk '{
      if ($5 ~ /typical,/) {
         volt=$6
         column=3
         }
      else if ($8 ~ /typical,/) {
         volt=$9
         column=4
         }
      else if ($3 ~ /words*/) {
         split ($3, wordsout, "=");
         split ($4, bitsout, "=");
         split (volt, voltout, ",");
         split ($8, freqout, "=");
         printf wordsout[2]",";
         printf bitsout[2]",";
```

```
            printf voltout[1]",";
            printf freqout[2]",";
            }
         else {
            # Total 4 columns, column 3 contains "typical" values
            if (/icc_a|icc_b|icc_desel_a|icc_desel_b/ && column == 3) {printf $3",";}
            # Final entry on line therefore dont place comma at the end
            else if (/icc_standby/ && column == 3) {printf $3;}
            # Total 5 columns, column 4 contains "typical" values
            else if (/icc_a|icc_b|icc_desel_a|icc_desel_b/ && column == 4) {printf $4",";}
            # Final entry on line therefore dont place comma at the end
            else if (/icc_standby/ && column == 4) {printf $4;}
            }
         }
         END {printf "\n"}
         ' $currentfile >> $output_file

   else

      echo "#, Cannot read input file $currentfile" >> $output_file

   fi

done

# Add registerfile and tag ram data to end of file

if [ -f rf_tag.csv ]; then
   cat rf_tag.csv >> $output_file
fi

echo "Memory library build complete. Output file is $output_file"
```

E.2 Memory analysis Java source code

```
/* Tool to report the energy consumption of a memory block, given details
 * of that block's type and size, along with the number of cycles the block
 * spends in each state. Data regarding each block is read from a CSV file,
 * typically memory_library.csv.
 *
 * Required arguments:
 * For types sp, rw, sp_rw or wr_wr:
 * <type> <words> <bits> <read cycles> <write cycles>
 *   <deselected cycles> <standby cycles>
 *
 * For type dp:
 * <type> <words> <bits> <Port A RW cycles> <Port B RW cycles>
 *   <Port A deselected cycles> <Port B deselected cycles> <standby cycles>
 *
 * For types rf or tag:
 * <type> <words> <bits> <read cycles> <write cycles> <idle cycles>
 *
 * Written by Paul Morgan, 2005-2006
 */

import java.io.*;
import java.util.*;
import java.lang.Integer;

import au.com.bytecode.opencsv.CSVReader;

public class AnalyseMemoryMain {

   // Input file in CSV format containing the data values for the memory library
   // This is usually generated from vendor data sheets using a shell script
   private static final String INPUT_FILE="memory_library.csv";

   private enum MemType {sp, dp, rf_tag, undef};

   private static void usage() {
      System.out.println("Error: Invalid arguments passed to memory analysis method");
      System.out.println();
      System.out.println("Required arguments:");
      System.out.println("For types sp, rw, sp_rw or wr_wr:");
      System.out.println("String[] {<type>, <words>, <bits>, <read cycles>,
                         <write cycles>, <deselected cycles>, <standby cycles>}");
      System.out.println();
      System.out.println("For type dp:");
      System.out.println("String[] {<type>, <words>, <bits>, <Port A RW cycles>,
                         <Port B RW cycles>, <Port A deselected cycles>,
                         <Port B deselected cycles>, <standby cycles>}");
      System.out.println();
      System.out.println("For types rf or tag:");
      System.out.println("String[] {<type>, <words>, <bits>, <read cycles>,
                         <write cycles>, <idle cycles>}");
      System.out.println();
   }
```

```
private static void duplicateMatchError() {
   System.out.println("Error: More than one match exists in the lookup table");
   System.out.println("Conflict error.");
   System.out.println();
}

private static void noMatchError() {
   System.out.println("Error: No match for the desired unit exists
                      in the lookup table");
   System.out.println("Fatal error.");
   System.out.println();
}


public static double[] main (String[] args) throws IOException {

   // Array for storing values to be returned from this method
   double[] returnValues = {0,0};

   // Enumerated value for storing the memory type being analysed
   MemType memType = MemType.undef;

   /* Parse input arguments and ensure the correct number of inputs
    *
    * First ensure the array has at least two elements before checking
    * args[1] otherwise we'll get an out of bounds error. Then check for
    * the exact correct number of arguments once the memory type has been
    * determined, as input arguments are dependent upon this. Any error
    * calls the usage() method which displays the required input and exits
    */

   if (args.length < 2) {
      usage();
      //Return negative value error condition to calling method
      returnValues[0] = -1;
      return (returnValues);
   }

   if ((args[0].compareTo("sp") == 0) || (args[0].compareTo("sp_rw") == 0) ||
         (args[0].compareTo("rw") == 0)) {
      memType = MemType.sp;
      if (args.length != 7) {
         usage();
         //Return negative value error condition to calling method
         returnValues[0] = -1;
         return (returnValues);
      }
   }

   else if (args[0].compareTo("dp") == 0) {
      memType = MemType.dp;
      if (args.length != 8) {
         usage();
         //Return negative value error condition to calling method
         returnValues[0] = -1;
         return (returnValues);
      }
```

```
}

else if ((args[0].compareTo("rf") == 0) || (args[0].compareTo("tag") == 0)) {
   memType = MemType.rf_tag;
   if (args.length != 6) {
      usage();
      //Return negative value error condition to calling method
      returnValues[0] = -1;
      return (returnValues);
   }
}

else {
   usage();
   //Return negative value error condition to calling method
   returnValues[0] = -1;
   return (returnValues);

}

System.out.println("Memory unit energy analysis.");
System.out.println("Analyses overall energy consumption of an
   individual cache memory, tag ram");
System.out.println("or register file based upon read, write, deselected
   and sleep cycles.");
System.out.println();

/* External CSV parsing function called from au.com.bytecode.opencsv.CSVReader
 * This results in a List (memEntries) with each entry being a String array
 * representing a line of the CSV file. Each entry in the String array represents
 * an individual entry in the CSV file.
 */
CSVReader reader = new CSVReader(new FileReader(INPUT_FILE));
List memEntries = reader.readAll();

// Flag to check whether we matched the desired unit to one in the list
boolean matchedList = false;

/* Iterate through each line in the CSV file until we find a match
 * for the memory type specified as input argument. Assuming a match is
 * found, analysis is performed on that data and the method successfully exits
 */
iteratorLoop: for (Iterator i = memEntries.iterator(); i.hasNext(); ) {
   String[] s = (String[]) i.next();

   MemType currentLineMemType = MemType.undef;

   //Store the type of the current line being analysed
   if ((s[0].compareTo("sp") == 0)|| (s[0].compareTo("sp_rw") == 0)
         || (s[0].compareTo("rw") == 0))
      currentLineMemType = MemType.sp;
   else if (s[0].compareTo("dp") == 0)
      currentLineMemType = MemType.dp;
   else if ((s[0].compareTo("rf") == 0) || (s[0].compareTo("tag") == 0))
      currentLineMemType = MemType.rf_tag;

   // Check if the current line in the buffer matches what we're looking for
```

```
if ((currentLineMemType == memType) && (s[1].compareTo(args[1]) == 0)
      && (s[2].compareTo(args[2]) == 0)) {

   /* If the matched flag is already set, we have a duplicate match error
    * Not that currently this is redundant as the program exits after
    * performing calculations on the first match, but it may be
    * changed in future therefore this is left in as a safety net
    */
   if (matchedList)
      duplicateMatchError();
   else
      matchedList = true;


   // Perform analysis as required for the matched memory type
   if (memType == MemType.sp) {

      // Parse the command line inputs
      int readCycles    = Integer.parseInt(args[3]);
      int writeCycles   = Integer.parseInt(args[4]);
      int deselCycles   = Integer.parseInt(args[5]);
      int standbyCycles = Integer.parseInt(args[6]);

      // Parse the data file values from the current line
      double voltage    = Double.parseDouble(s[3]);
      double frequency  = Double.parseDouble(s[4]);
      double iccRead    = Double.parseDouble(s[5]);
      double iccWrite   = Double.parseDouble(s[6]);
      double iccDesel   = Double.parseDouble(s[7]);
      double iccStandby = Double.parseDouble(s[8]);

      int cycleCount  = readCycles + writeCycles + deselCycles + standbyCycles;

      double totalEnergy = ((readCycles * iccRead + writeCycles * iccWrite +
                           deselCycles * iccDesel + standbyCycles * iccStandby)
                            * voltage / frequency);

      // Finished successfully
      returnValues[0] = (double)cycleCount;
      returnValues[1] =  totalEnergy;

      return (returnValues);
   }

   // Perform analysis as required for the matched memory type
   else if (memType == MemType.dp) {

      // Parse the command line inputs
      int readWriteACycles = Integer.parseInt(args[3]);
      int readWriteBCycles = Integer.parseInt(args[4]);
      int deselACycles     = Integer.parseInt(args[5]);
      int deselBCycles     = Integer.parseInt(args[6]);
      int standbyCycles    = Integer.parseInt(args[7]);

      // Parse the data file values from the current line
      double voltage    = Double.parseDouble(s[3]);
      double frequency  = Double.parseDouble(s[4]);
```

```
            double iccRW_A    = Double.parseDouble(s[5]);
            double iccRW_B    = Double.parseDouble(s[6]);
            double iccDeselA  = Double.parseDouble(s[7]);
            double iccDeselB  = Double.parseDouble(s[8]);
            double iccStandby = Double.parseDouble(s[9]);

            // Calculate cycle count and energy from above values - obvious enough
            int cycleCount  = readWriteACycles + readWriteBCycles + deselACycles
                            + deselBCycles + standbyCycles;
            double totalEnergy = ((readWriteACycles * iccRW_A + readWriteBCycles
                                 * iccRW_B + deselACycles * iccDeselA + deselBCycles
                                 * iccDeselB + standbyCycles * iccStandby) * voltage
                                 / frequency);

            // Finished successfully
            returnValues[0] = (double)cycleCount;
            returnValues[1] =  totalEnergy;

            return (returnValues);
         }


         // Perform analysis as required for the matched memory type
         else if (memType == MemType.rf_tag) {

            // Parse the command line inputs
            int readCycles  = Integer.parseInt(args[3]);
            int writeCycles = Integer.parseInt(args[4]);
            int idleCycles  = Integer.parseInt(args[5]);

            // Parse the data file values from the current line
            // Don't need the voltage value as we already have energy values
            // for this memory type
            double energyRead  = Double.parseDouble(s[4]);
            double energyWrite = Double.parseDouble(s[5]);
            double energyIdle  = Double.parseDouble(s[6]);

            int cycleCount  = readCycles + writeCycles + idleCycles;

            double totalEnergy = (readCycles * energyRead + writeCycles * energyWrite
                                 + idleCycles * energyIdle);

            // Finished successfully
            returnValues[0] = (double)cycleCount;
            returnValues[1] =  totalEnergy;

            return (returnValues);
         }
      }
   }

   /* If we get this far then the requested unit type did not match one
    * in the list. However to be sure, check the matchedList flag
    */
   if (!matchedList)
      noMatchError();
```

```
        /* At this point we have to unconditionally return something to keep
         * the compiler happy, in every case reaching this point should be caused
         * by the no match condition above. Therefore we return a -2 which signifies
         * this error condition.
         */
        returnValues[0] = -2;
        return (returnValues);
    }
}
```

E.3 Memory library CSV file (130 nm)

```
#, Look-up table for memory energy values (except "dp_*" memories)
#, type,words,bits,volt,freq,icc_read,icc_write,icc_desel,icc_standby
#,
rw,1024,32,1.20,200.000,8.111,9.404,1.563,0.010
rw,16384,32,1.20,200.000,26.038,27.838,3.211,0.043
rw,2048,32,1.20,200.000,8.994,10.942,1.665,0.014
rw,4096,32,1.20,200.000,10.348,13.728,1.707,0.023
rw,512,32,1.20,200.000,7.800,8.703,1.497,0.008
rw,8192,32,1.20,200.000,13.872,15.710,2.205,0.038
sp_rw,128,32,1.20,200.000,2.572,2.704,0.259,0.002
sp_rw,16,48,1.20,200.000,2.246,2.534,0.295,0.004
sp_rw,512,8,1.20,200.000,3.270,3.457,0.990,0.004
sp_rw,64,32,1.20,200.000,2.477,2.565,0.239,0.002
sp_rw,1024,104,1.20,200.000,22.072,26.404,3.096,0.027
sp_rw,1024,112,1.20,200.000,23.624,28.293,3.267,0.029
sp_rw,1024,120,1.20,200.000,25.175,30.182,3.437,0.031
sp_rw,1024,16,1.20,200.000,5.008,5.626,1.223,0.007
sp_rw,1024,24,1.20,200.000,6.559,7.515,1.393,0.009
sp_rw,1024,32,1.20,200.000,8.111,9.404,1.563,0.010
sp_rw,1024,40,1.20,200.000,9.662,11.293,1.734,0.012
sp_rw,1024,48,1.20,200.000,11.213,13.182,1.904,0.014
sp_rw,1024,56,1.20,200.000,12.765,15.071,2.074,0.016
sp_rw,1024,64,1.20,200.000,14.316,16.960,2.245,0.018
sp_rw,1024,72,1.20,200.000,15.867,18.849,2.415,0.020
sp_rw,1024,8,1.20,200.000,3.457,3.738,1.052,0.005
sp_rw,1024,80,1.20,200.000,17.418,20.738,2.585,0.022
sp_rw,1024,88,1.20,200.000,18.970,22.627,2.756,0.023
sp_rw,1024,96,1.20,200.000,20.521,24.515,2.926,0.025
sp_rw,16384,16,1.20,200.000,14.322,15.194,2.126,0.024
sp_rw,16384,24,1.20,200.000,20.180,21.516,2.669,0.034
sp_rw,16384,32,1.20,200.000,26.038,27.838,3.211,0.043
sp_rw,16384,8,1.20,200.000,8.464,8.872,1.584,0.014
sp_rw,2048,104,1.20,200.000,24.431,30.673,3.214,0.039
sp_rw,2048,112,1.20,200.000,26.146,32.865,3.387,0.042
sp_rw,2048,120,1.20,200.000,27.861,35.057,3.559,0.045
sp_rw,2048,16,1.20,200.000,5.566,6.558,1.321,0.009
sp_rw,2048,24,1.20,200.000,7.279,8.750,1.493,0.012
sp_rw,2048,32,1.20,200.000,8.994,10.942,1.665,0.014
sp_rw,2048,40,1.20,200.000,10.709,13.134,1.837,0.017
sp_rw,2048,48,1.20,200.000,12.424,15.327,2.009,0.020
sp_rw,2048,56,1.20,200.000,14.139,17.519,2.181,0.023
sp_rw,2048,64,1.20,200.000,15.855,19.711,2.354,0.025
sp_rw,2048,72,1.20,200.000,17.570,21.904,2.526,0.028
sp_rw,2048,8,1.20,200.000,3.856,4.365,1.149,0.006
sp_rw,2048,80,1.20,200.000,19.285,24.096,2.698,0.031
sp_rw,2048,88,1.20,200.000,21.000,26.288,2.870,0.034
sp_rw,2048,96,1.20,200.000,22.715,28.480,3.042,0.036
sp_rw,256,104,1.20,200.000,21.052,23.460,2.979,0.018
sp_rw,256,112,1.20,200.000,22.541,25.138,3.147,0.019
sp_rw,256,120,1.20,200.000,24.031,26.817,3.316,0.020
sp_rw,256,16,1.20,200.000,4.666,4.996,1.127,0.005
sp_rw,256,24,1.20,200.000,6.156,6.674,1.295,0.006
sp_rw,256,32,1.20,200.000,7.645,8.353,1.463,0.007
sp_rw,256,40,1.20,200.000,9.135,10.031,1.632,0.008
```

```
sp_rw,256,48,1.20,200.000,10.624,11.710,1.800,0.009
sp_rw,256,56,1.20,200.000,12.114,13.388,1.969,0.011
sp_rw,256,64,1.20,200.000,13.604,15.067,2.137,0.012
sp_rw,256,72,1.20,200.000,15.093,16.746,2.305,0.013
sp_rw,256,8,1.20,200.000,3.177,3.317,0.958,0.003
sp_rw,256,80,1.20,200.000,16.583,18.424,2.474,0.014
sp_rw,256,88,1.20,200.000,18.072,20.103,2.642,0.015
sp_rw,256,96,1.20,200.000,19.562,21.781,2.811,0.017
sp_rw,4096,104,1.20,200.000,27.993,38.975,3.285,0.065
sp_rw,4096,112,1.20,200.000,29.954,41.780,3.460,0.069
sp_rw,4096,120,1.20,200.000,31.915,44.586,3.636,0.074
sp_rw,4096,16,1.20,200.000,6.438,8.117,1.357,0.013
sp_rw,4096,24,1.20,200.000,8.387,10.923,1.532,0.018
sp_rw,4096,32,1.20,200.000,10.348,13.728,1.707,0.023
sp_rw,4096,40,1.20,200.000,12.308,16.533,1.883,0.027
sp_rw,4096,48,1.20,200.000,14.269,19.338,2.058,0.032
sp_rw,4096,56,1.20,200.000,16.230,22.144,2.233,0.037
sp_rw,4096,64,1.20,200.000,18.190,24.949,2.409,0.041
sp_rw,4096,72,1.20,200.000,20.151,27.754,2.584,0.046
sp_rw,4096,8,1.20,200.000,4.495,5.312,1.182,0.009
sp_rw,4096,80,1.20,200.000,22.111,30.559,2.759,0.051
sp_rw,4096,88,1.20,200.000,24.072,33.365,2.935,0.055
sp_rw,4096,96,1.20,200.000,26.033,36.170,3.110,0.060
sp_rw,512,104,1.20,200.000,21.392,24.441,3.018,0.021
sp_rw,512,112,1.20,200.000,22.902,26.190,3.187,0.022
sp_rw,512,120,1.20,200.000,24.412,27.938,3.356,0.024
sp_rw,512,16,1.20,200.000,4.780,5.206,1.159,0.005
sp_rw,512,24,1.20,200.000,6.290,6.955,1.328,0.007
sp_rw,512,32,1.20,200.000,7.800,8.703,1.497,0.008
sp_rw,512,40,1.20,200.000,9.311,10.452,1.666,0.010
sp_rw,512,48,1.20,200.000,10.821,12.201,1.835,0.011
sp_rw,512,56,1.20,200.000,12.331,13.949,2.004,0.012
sp_rw,512,64,1.20,200.000,13.841,15.698,2.173,0.014
sp_rw,512,72,1.20,200.000,15.351,17.447,2.342,0.015
sp_rw,512,8,1.20,200.000,3.270,3.457,0.990,0.004
sp_rw,512,80,1.20,200.000,16.861,19.195,2.511,0.017
wr_wr,512,32,1.20,200.000,13.026,16.325,2.252,0.062
sp_rw,512,88,1.20,200.000,18.372,20.944,2.680,0.018
sp_rw,512,96,1.20,200.000,19.882,22.693,2.849,0.020
sp_rw,8192,16,1.20,200.000,8.241,9.248,1.605,0.021
sp_rw,8192,24,1.20,200.000,11.057,12.479,1.905,0.030
sp_rw,8192,32,1.20,200.000,13.872,15.710,2.205,0.038
sp_rw,8192,40,1.20,200.000,16.687,18.941,2.505,0.046
sp_rw,8192,48,1.20,200.000,19.503,22.172,2.805,0.055
sp_rw,8192,56,1.20,200.000,22.318,25.403,3.105,0.063
sp_rw,8192,64,1.20,200.000,25.134,28.634,3.405,0.072
sp_rw,8192,8,1.20,200.000,5.426,6.017,1.305,0.013
sp_rw,1024,128,1.20,200.000,26.726,32.071,3.607,0.033
sp_rw,2048,128,1.20,200.000,29.576,37.250,3.731,0.047
sp_rw,256,128,1.20,200.000,25.520,28.495,3.484,0.021
sp_rw,4096,128,1.20,200.000,33.875,47.391,3.811,0.078
sp_rw,512,128,1.20,200.000,25.922,29.687,3.525,0.025
wr_wr,1024,32,1.20,200.000,13.271,16.657,2.293,0.066
wr_wr,2048,32,1.20,200.000,13.809,17.323,2.374,0.073
wr_wr,4096,32,1.20,200.000,14.886,18.653,2.536,0.088
wr_wr,8192,32,1.20,200.000,17.038,21.314,2.860,0.116
#,
```

```
#,
#, Look-up table for memory energy values (dual-port memories)
#, type,words,bits,volt,freq,icc_rw_a,icc_rw_b,icc_desel_a,icc_desel_b,icc_standby
#,
dp,128,32,1.20,200.000,2.151,1.974,0.650,0.650,0.025
dp,16,32,1.20,200.000,1.553,0.892,0.340,0.340,0.007
dp,32,32,1.20,200.000,1.638,1.047,0.384,0.384,0.009
dp,64,32,1.20,200.000,1.809,1.356,0.473,0.473,0.015
#,
#,
#, Look-up table for simulation-derived memory energy values (nJ) - (reg file)
#, type,words,bits,volt,enj_read,enj_write,enj_idle
#,
rf,32,32,1.20,0.022104,0.023785,0.019789
rf,64,32,1.20,0.043537,0.045017,0.039709
#,
#,
#,Look-up table for simulation-derived memory energy values (nJ) - (tag ram)
#,type,words,bits,volt,enj_read,enj_write,enj_idle
#,
tag,8,27,1.20,0.004616,0.005779,0.004207
tag,16,27,1.20,0.009146,0.010535,0.008423
tag,32,27,1.20,0.018281,0.020110,0.016857
tag,64,27,1.20,0.036765,0.038605,0.033735
tag,128,27,1.20,0.074197,0.077185,0.067510
tag,256,27,1.20,0.148700,0.166350,0.135130
```

E.4 Memory library CSV file (90 nm)

```
#, Look-up table for memory energy values (except "dp_*" memories)
#, type,words,bits,volt,freq,icc_read,icc_write,icc_desel,icc_standby
#,
rw,1024,32,200.000,1.0,4.561,4.747,9.53E-1,1.16E-1
rw,16384,32,200.000,1.0,15.231,15.315,2.730,7.19E-1
rw,2048,32,200.000,1.0,4.899,5.069,1.004,1.65E-1
rw,4096,32,200.000,1.0,5.533,5.673,1.064,2.63E-1
rw,512,32,200.000,1.0,4.372,4.564,9.07E-1,9.12E-2
rw,8192,32,200.000,1.0,8.837,8.960,1.634,4.15E-1
sp_rw,128,32,200.000,1.0,1.459,1.742,3.09E-1,1.75E-2
sp_rw,16,48,200.000,1.0,1.439,1.742,2.20E-1,8.69E-3
sp_rw,512,8,200.000,1.0,1.628,1.674,4.59E-1,3.89E-2
sp_rw,1024,104,200.000,1.0,13.024,13.628,2.310,3.00E-1
sp_rw,1024,112,200.000,1.0,13.964,14.615,2.461,3.21E-1
sp_rw,1024,120,200.000,1.0,14.905,15.601,2.611,3.41E-1
sp_rw,1024,16,200.000,1.0,2.681,2.772,6.52E-1,7.46E-2
sp_rw,1024,24,200.000,1.0,3.621,3.760,8.03E-1,9.52E-2
sp_rw,1024,32,200.000,1.0,4.561,4.747,9.53E-1,1.16E-1
sp_rw,1024,40,200.000,1.0,5.501,5.734,1.104,1.36E-1
sp_rw,1024,48,200.000,1.0,6.442,6.721,1.255,1.57E-1
sp_rw,1024,56,200.000,1.0,7.382,7.708,1.406,1.77E-1
sp_rw,1024,64,200.000,1.0,8.322,8.694,1.556,1.98E-1
sp_rw,1024,72,200.000,1.0,9.262,9.681,1.707,2.18E-1
sp_rw,1024,8,200.000,1.0,1.742,1.785,5.01E-1,5.41E-2
sp_rw,1024,80,200.000,1.0,10.203,10.668,1.858,2.39E-1
sp_rw,1024,88,200.000,1.0,11.143,11.655,2.009,2.59E-1
sp_rw,1024,96,200.000,1.0,12.083,12.642,2.159,2.80E-1
sp_rw,16384,16,200.000,1.0,8.308,8.346,1.674,4.13E-1
sp_rw,16384,24,200.000,1.0,11.771,11.830,2.202,5.66E-1
sp_rw,16384,32,200.000,1.0,15.231,15.315,2.730,7.19E-1
sp_rw,16384,8,200.000,1.0,4.845,4.861,1.146,2.60E-1
sp_rw,2048,104,200.000,1.0,13.786,14.353,2.359,4.05E-1
sp_rw,2048,112,200.000,1.0,14.774,15.385,2.510,4.32E-1
sp_rw,2048,120,200.000,1.0,15.761,16.416,2.660,4.58E-1
sp_rw,2048,16,200.000,1.0,2.924,3.007,7.03E-1,1.11E-1
sp_rw,2048,24,200.000,1.0,3.911,4.038,8.54E-1,1.38E-1
sp_rw,2048,32,200.000,1.0,4.899,5.069,1.004,1.65E-1
sp_rw,2048,40,200.000,1.0,5.887,6.101,1.155,1.91E-1
sp_rw,2048,48,200.000,1.0,6.874,7.132,1.305,2.18E-1
sp_rw,2048,56,200.000,1.0,7.862,8.163,1.456,2.45E-1
sp_rw,2048,64,200.000,1.0,8.849,9.195,1.606,2.72E-1
sp_rw,2048,72,200.000,1.0,9.837,10.226,1.757,2.98E-1
sp_rw,2048,8,200.000,1.0,1.936,1.975,5.52E-1,8.46E-2
sp_rw,2048,80,200.000,1.0,10.824,11.258,1.907,3.25E-1
sp_rw,2048,88,200.000,1.0,11.812,12.290,2.058,3.52E-1
sp_rw,2048,96,200.000,1.0,12.799,13.321,2.209,3.78E-1
sp_rw,256,104,200.000,1.0,12.368,13.001,2.189,2.22E-1
sp_rw,256,112,200.000,1.0,13.270,13.950,2.336,2.38E-1
sp_rw,256,120,200.000,1.0,14.172,14.899,2.484,2.54E-1
sp_rw,256,16,200.000,1.0,2.454,2.551,5.68E-1,4.71E-2
sp_rw,256,24,200.000,1.0,3.355,3.502,7.15E-1,6.30E-2
sp_rw,256,32,200.000,1.0,4.256,4.452,8.63E-1,7.89E-2
sp_rw,256,40,200.000,1.0,5.156,5.402,1.010,9.48E-2
sp_rw,256,48,200.000,1.0,6.057,6.352,1.157,1.11E-1
```

```
sp_rw,256,56,200.000,1.0,6.958,7.302,1.305,1.27E-1
sp_rw,256,64,200.000,1.0,7.860,8.252,1.452,1.42E-1
sp_rw,256,72,200.000,1.0,8.761,9.202,1.600,1.58E-1
sp_rw,256,8,200.000,1.0,1.554,1.601,4.21E-1,3.12E-2
sp_rw,256,80,200.000,1.0,9.663,10.152,1.747,1.74E-1
sp_rw,256,88,200.000,1.0,10.564,11.102,1.894,1.90E-1
sp_rw,256,96,200.000,1.0,11.466,12.051,2.042,2.06E-1
sp_rw,4096,104,200.000,1.0,15.242,15.733,2.388,6.15E-1
sp_rw,4096,112,200.000,1.0,16.320,16.852,2.536,6.54E-1
sp_rw,4096,120,200.000,1.0,17.397,17.971,2.683,6.93E-1
sp_rw,4096,16,200.000,1.0,3.373,3.440,7.70E-1,1.85E-1
sp_rw,4096,24,200.000,1.0,4.453,4.557,9.17E-1,2.24E-1
sp_rw,4096,32,200.000,1.0,5.533,5.673,1.064,2.63E-1
sp_rw,4096,40,200.000,1.0,6.613,6.790,1.211,3.02E-1
sp_rw,4096,48,200.000,1.0,7.692,7.907,1.358,3.41E-1
sp_rw,4096,56,200.000,1.0,8.772,9.025,1.506,3.80E-1
sp_rw,4096,64,200.000,1.0,9.851,10.142,1.653,4.19E-1
sp_rw,4096,72,200.000,1.0,10.930,11.260,1.800,4.58E-1
sp_rw,4096,8,200.000,1.0,2.292,2.324,6.23E-1,1.46E-1
sp_rw,4096,80,200.000,1.0,12.008,12.378,1.947,4.97E-1
sp_rw,4096,88,200.000,1.0,13.086,13.496,2.094,5.36E-1
sp_rw,4096,96,200.000,1.0,14.164,14.614,2.241,5.75E-1
sp_rw,512,104,200.000,1.0,12.609,13.232,2.252,2.48E-1
sp_rw,512,112,200.000,1.0,13.524,14.195,2.401,2.65E-1
sp_rw,512,120,200.000,1.0,14.440,15.157,2.550,2.83E-1
sp_rw,512,16,200.000,1.0,2.542,2.637,6.08E-1,5.63E-2
sp_rw,512,24,200.000,1.0,3.457,3.601,7.58E-1,7.37E-2
sp_rw,512,32,200.000,1.0,4.372,4.564,9.07E-1,9.12E-2
sp_rw,512,40,200.000,1.0,5.286,5.528,1.056,1.09E-1
sp_rw,512,48,200.000,1.0,6.201,6.491,1.206,1.26E-1
sp_rw,512,56,200.000,1.0,7.116,7.454,1.355,1.43E-1
sp_rw,512,64,200.000,1.0,8.032,8.418,1.505,1.61E-1
sp_rw,512,72,200.000,1.0,8.947,9.381,1.654,1.78E-1
sp_rw,512,8,200.000,1.0,1.628,1.674,4.59E-1,3.89E-2
sp_rw,512,80,200.000,1.0,9.862,10.344,1.803,1.96E-1
sp_rw,512,88,200.000,1.0,10.778,11.306,1.953,2.13E-1
sp_rw,512,96,200.000,1.0,11.693,12.269,2.102,2.31E-1
sp_rw,8192,16,200.000,1.0,5.059,5.118,1.086,2.61E-1
sp_rw,8192,24,200.000,1.0,6.948,7.039,1.360,3.38E-1
sp_rw,8192,32,200.000,1.0,8.837,8.960,1.634,4.15E-1
sp_rw,8192,40,200.000,1.0,10.725,10.882,1.908,4.92E-1
sp_rw,8192,48,200.000,1.0,12.613,12.803,2.182,5.69E-1
sp_rw,8192,56,200.000,1.0,14.500,14.725,2.456,6.46E-1
sp_rw,8192,64,200.000,1.0,16.387,16.646,2.730,7.23E-1
sp_rw,8192,8,200.000,1.0,3.170,3.197,8.11E-1,1.84E-1
sp_rw,1024,128,200.000,1.0,15.846,16.588,2.762,3.62E-1
sp_rw,2048,128,200.000,1.0,16.748,17.448,2.811,4.85E-1
sp_rw,256,128,200.000,1.0,15.074,15.848,2.631,2.69E-1
sp_rw,4096,128,200.000,1.0,18.474,19.090,2.830,7.32E-1
sp_rw,512,128,200.000,1.0,15.356,16.120,2.700,3.00E-1
wr_wr,1024,32,200.000,1.0,3.726,4.040,1.865,3.17E-1
wr_wr,2048,32,200.000,1.0,3.903,4.219,1.928,3.86E-1
wr_wr,4096,32,200.000,1.0,4.123,4.443,1.920,5.26E-1
wr_wr,512,32,200.000,1.0,3.574,3.887,1.770,2.82E-1
wr_wr,8192,32,200.000,1.0,4.422,4.750,1.762,8.05E-1
#,
#,
```

```
#, Look-up table for memory energy values (dual-port memories)
#, type,words,bits,volt,freq,icc_rw_a,icc_rw_b,icc_desel_a,icc_desel_b,icc_standby
#,
dp,16,32,200.000,1.0,9.23E-1,5.19E-1,5.33E-2,6.20E-2,1.54E-2
dp,128,32,200.000,1.0,1.140,9.83E-1,1.61E-1,1.70E-1,6.92E-2
dp,32,32,200.000,1.0,9.58E-1,5.89E-1,7.20E-2,8.07E-2,2.30E-2
dp,64,32,200.000,1.0,1.019,7.22E-1,1.03E-1,1.11E-1,3.84E-2
#,
#,
#, Look-up table for simulation-derived memory energy values (nJ) - (reg file)
#, type,words,bits,volt,enj_read,enj_write,enj_idle
#,
rf,32,32,1.20,0.010414,0.011206,0.009323
rf,64,32,1.20,0.020512,0.021209,0.018708
#,
#,
#,Look-up table for simulation-derived memory energy values (nJ) - (tag ram)
#,type,words,bits,volt,enj_read,enj_write,enj_idle
#,
tag,8,27,1.20,0.002174,0.002722,0.001982
tag,16,27,1.20,0.004309,0.004963,0.003968
tag,32,27,1.20,0.008613,0.009474,0.007942
tag,64,27,1.20,0.017321,0.018188,0.015894
tag,128,27,1.20,0.034957,0.036365,0.031807
tag,256,27,1.20,0.070059,0.078375,0.063666
```

F. Leakage power analysis script

```
#!/bin/sh
#
# Script to analyse leakage power for a list of functional units. Examines all
# relevant hierarchical power reports for instances of the unit in question,
# and parses the result for that unit to generate an average leakage power over
# all power reports. This is done by writing a script file for the Desk Calculator
# (dc) tool using Reverse Polish Notation, which is then executed by dc to
# generate the result.
#
# Written by Paul Morgan, 2005-2006

# Debug mode flag - set to 1 to retain intermediate dc command text files
debug=0

for test in fu_squash fu_select fu_sat_arithmetic fu_registerfile fu_predicate   \
            fu_mult64 fu_logical fu_immediate8 fu_immediate32 fu_copy fu_combine \
            fu_branch fu_bitshift fu_arithmetic fu_addrlink fu_Cache0 fu_Cache1
do
  # Remove any old copy of the log file for this test in case noclobber is set
  if [ -f dc_calc_$test.txt ]; then
    rm dc_calc_$test.txt
  fi

  # Add a zero to the stack to prevent a "stack empty" warning when trying to add
  # results from the first log file to non-existant previous results
  printf "10 k 0 " |tee individual_$test.txt > dc_calc_$test.txt

  declare -i count=0

  # Cycle through each report, filtering out the desired information and adding
  # it to the command file for dc to calculate the average leakage power
  for report in `grep $test *_hier1.txt | awk '{printf $6" "}'`; do
    printf $report | sed -e 's/e+/ /' | tee -a individual_$test.txt \
      >> dc_calc_$test.txt
    echo " 10 r ^ * +" >> dc_calc_$test.txt
    echo " 10 r ^ * MEAN - 2 ^ +" >> individual_$test.txt
    count=count+1
  done

  echo "$count / p" >> dc_calc_$test.txt

  # Perform the calculation using dc with the previously written script file
  # and output the result to the console
```

```
  mean=`dc -f dc_calc_$test.txt`
  echo "Average leakage for test $test: $mean ($count occurrences)"

  sed -e s/MEAN/$mean/ < individual_$test.txt > individual2_$test.txt
  echo " $count 1 - / v p" >> individual2_$test.txt

  stddev=`dc -f individual2_$test.txt`
  echo "Std deviation for $test: $stddev (`dc -e "3 k $stddev $mean / 100 * p"` %)"

  # Remove intermediate files unless in debug mode
  if [ $debug -ne 1 ]; then
    rm -f dc_calc_$test.txt individual_$test.txt individual2_$test.txt
  fi

done
```

G. Technology library power comparison script

```
#!/bin/sh
#
# Script to calculate both the mean change in average power, and the standard
# deviation of that change, for a selection of tests undertaken using both
# TSMC 130 nm and TSMC 90 nm technology libraries
#
# Written by Paul Morgan, 2005–2006

# Declare an integer to store the total number of test cases being analysed
declare -i count=0

# Create the initial setup commands that will be fed to the dc calculator.
# 10 k sets a precision of 10 decimal places, 0 simply provides a null value
# on the stack to prevent a stack empty error with initial calculations
printf "10 k 0 " > dc_calc_input.txt

# For each test, examine the relevant power report for 130 nm and 90 nm cases.
# Find the relevant line and cut the desired average power value, storing the
# value into a text file to be analysed by dc. Add the relevant dc commands
# to calculate the ratio between 130 nm and 90 nm for each test, and sum them.
for testcase in adpcm_decode adpcm_encode epic_decode epic_encode g721_decode    \
                g721_encode gsm_decode gsm_encode jpeg_decode jpeg_encode        \
                mpeg2_decode mpeg2_encode pegwit_decode pegwit_encode pgp_decode \
                pgp_encode
do
   grep "Total Dynamic Power" reports/$testcase\_comp_power.txt | cut -b 28-33 \
      >> dc_calc_input.txt
   printf " " >> dc_calc_input.txt
   grep "Total Dynamic Power" ../mediabench/reports/$testcase\_comp_power.txt \
      | cut -b 28-33 >> dc_calc_input.txt
   printf " / p + " >> dc_calc_input.txt
   count=$count+1
done

echo "Change in average power for each test going from 130 nm to 90 nm:"
dc -f dc_calc_input.txt | tee differences.txt

# Divide the total sum of differences by the test count to get the mean.
# To ensure that only the average value is stored as a variable, the dc
# commands are filtered through sed to remove any previous print commands
printf "$count / " >> dc_calc_input.txt
average=$(dc -e "$(cat dc_calc_input.txt|sed -e s'/p//g') p")
```

```
echo
echo "Average change    : $average"

# Calculate the standard deviation by subtracting the mean from each value,
# squaring the result, then adding all squared results. The total is then
# divided by (count - 1), then square rooted to get the final result.
awk --assign average=$average --assign count=$count '
    BEGIN {print "10 k 0 "}
    {print $1" "average" - 2 ^ + "}
    END {print count" 1 - / v p"}
    ' differences.txt > stddev.txt

echo "Standard deviation: `dc -f stddev.txt`"
```

H. Physical layout and place & route scripts

These scripts are used as part of the post-synthesis back-end flow. The script in section H.1 is intended to be run from within Synopsys Astro in scheme mode, while that in section H.2 is a Synopsys JupiterXT scheme mode script. Finally, section H.3 lists a Cadence First Encounter Tcl script.

H.1 Milkyway library creation script

```
;# Scheme
menuReload "astro_data_prep"

cmCreateLib

setFormField "Create Library" "Library Name" "ref_lib"
setFormField "Create Library" "Technology File Name" "tsmc13fsg_8lm.tf"
setFormField "Create Library" "Set Case Sensitive" "1"
formOK       "Create Library"

read_lef

setFormField "Read LEF" "Library Name" "ref_lib"
setFormField "Read LEF" "Tech LEF Files" "tsmc13fsg_8lm_tech.lef"
setFormField "Read LEF" "Layer Mapping" "8lm_tech_lef_tf.map"
setFormField "Read LEF" "Cell Options" "Overwrite Existing Cell"
setFormField "Read LEF" "Overwrite Existing Technology" "1"
setFormField "Read LEF" "Cell LEF Files" "tsmc13nvt_macros.lef      \
                                          sp_rw_s_instrmax256.vclef \
                                          sp_rw_s_instr256x96.vclef \
                                          wr_wr_s_s_4096x32.vclef"
formOK "Read LEF"

read_lib
readLibForm "logical"
gePrepLibs

setFormField "Library Preparation" "Library Name" "ref_lib"
formButton   "Library Preparation" "importLMDB"
```

```
formButton   "Library Preparation" "selectDB"
setFormField "Library Preparation" "Typical DB To Import" "typical.db"
setFormField "Library Preparation" "Max DB To Import" "slow.db"
setFormField "Library Preparation" "Min DB To Import" "fast.db"
formApply    "Library Preparation"
formButton   "Library Preparation" "setLMDB"
setFormField "Library Preparation" "Is Ref Library" "1"
setFormField "Library Preparation" "Ref Library Name" "ref_lib"
setFormField "Library Preparation" "Design Cell Name" ""
setFormField "Library Preparation" "Set DB To Max" "slow.db"
setFormField "Library Preparation" "Set DB To Min" "fast.db"
setFormField "Library Preparation" "Set DB To Typical" "typical.db"
formOK       "Library Preparation"

readLibForm  "hide"

exit
```

H.2 JupiterXT floorplanning script

```
;# Scheme
auVerilogToCell
setFormField "Verilog To Cell" "Library Name" "main_lib"
setFormField "Verilog To Cell" "Verilog File Name" "pgp_encode.v"
setFormField "Verilog To Cell" "Output Cell Name" "test_copro"
setFormField "Verilog To Cell" "Top Module Name" "test_copro"
setFormField "Verilog To Cell" "Tech File Name" "tsmc13fsg_8lm.tf"
formButton   "Verilog To Cell" "refLibOptions"
setFormField "Verilog To Cell" "Reference Library" "ref_lib"
setFormField "Verilog To Cell" "Handle Dirty Netlist" "1"
setFormField "Verilog To Cell" "Verilog IEEE 2001" "1"
formOK "Verilog To Cell"

geOpenLib
setFormField "Open Library" "Library Name" "main_lib"
formOK "Open Library"
geOpenCell
setFormField "Open Cell" "Cell Name" "test_copro"
formOK "Open Cell"
ataLoadSDC
formOK "Load SDC File"
setFormField "Load SDC File" "SDC File Name" "../pgp_encode/pgp_encode.sdc"
formOK "Load SDC File"

aprPGConnect
setFormField "Connect/Disconnect PG" "Net Type" "Power"
setFormField "Connect/Disconnect PG" "Net Name" "VDD"
formOK "Connect/Disconnect PG"
aprPGConnect
setFormField "Connect/Disconnect PG" "Net Type" "Ground"
setFormField "Connect/Disconnect PG" "Net Name" "VSS"
formOK "Connect/Disconnect PG"
geCloseWindow
formOK "Close Window"

axgPlanner
setFormField "Floor Planning" "Core Utilization" "0.5000"
formOK "Floor Planning"

exit
```

H.3 First Encounter physical layout script

```
#####################################################
#                                                   #
#  Encounter Command File                           #
#  Created on Fri Mar 30 14:34:21                   #
#                                                   #
#####################################################

# Load configuration file, this provides details on the Verilog netlist,
# library files, technology files and design constraints
  loadConfig /crux/paulm/encounter_pgp_encode/Default.conf 0
  commitConfig

# Initiate the design floorplan, specifying the site from the technology
# library, core size by aspect ratio (1) and core utilisation (0.5), and
# the core to IO boundaries (15 um in all dimensions).
  floorPlan -site TSM13SITE -r 1 0.5 15 15 15 15

# Add power rings to the die. These are configured with reference to the
# relevant technology file containing details on the metal layers available
# within the physical layout. Nets VDD and VSS correspond to the power and
# ground pins in TSMC cells.
  addRing -spacing_bottom 1 -width_left 2 -width_bottom 2 -width_top 2        \
    -spacing_top 1 -layer_bottom METAL7 -center 1                             \
    -stacked_via_top_layer METAL8 -width_right 2 -around core                 \
    -jog_distance 0.23 -offset_bottom 0.23 -layer_top METAL7 -threshold 0.23  \
    -offset_left 0.23 -spacing_right 1 -spacing_left 1 -offset_right 0.23     \
    -offset_top 0.23 -layer_right METAL8 -nets {VSS VDD}                      \
    -stacked_via_bottom_layer METAL1 -layer_left METAL8

# Add power stripes with similar criteria to those for power rings. Spacing
# is chosen to minimise routing blockage while still providing adequate
# power supply to meet the demands of the core.
  addStripe -block_ring_top_layer_limit METAL4                                \
    -max_same_layer_jog_length 0.88 -padcore_ring_bottom_layer_limit METAL1   \
    -set_to_set_distance 100 -stacked_via_top_layer METAL8                    \
    -padcore_ring_top_layer_limit METAL4 -spacing 0.5                         \
    -merge_stripes_value 0.23 -layer METAL2                                   \
    -block_ring_bottom_layer_limit METAL1 -width 1 -nets {VSS VDD}            \
    -stacked_via_bottom_layer METAL1

# Configure options for automatic placement of standard cells, particularly
# timing-driven placement with high effort. Then call the automatic
# placement and refine placement functions
  setPlaceMode -timingdriven -reorderScan -congHighEffort -noCongOpt          \
    -noModulePlan

  placeDesign -prePlaceOpt

  refinePlace

# Load the previously defined clock tree specification file, which contains
# directives to perform automatic clock tree synthesis
  specifyClockTree -clkfile test_copro.ctstch
```

```
  createSaveDir test_copro_cts

# Perform automatic clock tree synthesis, and once complete save detail
# and reports relevant to the clock tree
  ckSynthesis -rguide test_copro_cts/test_copro_cts.guide                    \
    -report test_copro_cts/test_copro_cts.ctsrpt

  saveClockNets -output test_copro_cts/test_copro_cts.ctsntf

  saveNetlist test_copro_cts/test_copro_cts.v

  savePlace test_copro_cts/test_copro_cts.place

# Call the special routing algorithm for power and ground nets VDD and VSS
  sroute -deleteExistingRoutes -noBlockPins -noPadRings -noPadPins           \
    -jogControl { preferWithChanges differentLayer } -nets { VDD VSS }

# Perform wroute and global routing on all remaining nets
  wroute -timingDriven

  globalRoute

# Connect power and ground pins to the power stripes
  globalnetconnect VDD -type pgpin -pin VDD -all -override
  globalnetconnect VSS -type pgpin -pin VSS -all -override

# Perform a simulation-based power analysis based on the toggle activity
# obtained from VCD file output during simulation
  updatePower -vcd /crux/paulm/tests/pgp_encode/Verilog_Impl/full_sim.vcd    \
    -vcdTop copro_testbench/copro -noRailAnalysis -report power.report VDD

# Set options and extract RC data from the post-routed design
  setExtractRCMode -detail -rcdb test_copro.rcdb -relative_c_t 0.03          \
    -total_c_t 5.0 -reduce 5 -noise

  setXCapThresholds -totalCThreshold 5.0 -relativeCThreshold 0.03

  extractRC -outfile test_copro.cap

  rcOut -spef test_copro.spef

# Obtain an area report of the placed and routed design
  reportGateCount -level 5 -limit 100 -outfile test_copro.gateCount

  saveDesign /crux/paulm/tests/pgp_encode/Verilog_Impl/test_copro.enc
  exit
```

I. Case study supporting files

The files listed below are used with the Cascade energy analysis case study undertaken in chapter 12. Listed in section I.1 is an excerpt of the `technology.xml` file used by Cascade for the analysis of TSMC 90 nm coprocessors. Specifically, the file listed details the energy values stored within the file that have been determined throughout this project. The complete file actually referenced by Cascade contains a lot of additional information used for other calculations, such as area estimates, that has been removed from the file listed here to keep it concise.

Similarly, section I.2 is an excerpt of the `technology.xml` file used by Cascade for the analysis of TSMC 130 nm coprocessors. Finally, section I.3 lists an example analysis summary file that is created by Cascade containing, among other details, the area, performance and energy statistics for a coprocessor candidate.

I.1 TSMC 90 nm technology.xml energy entries

```
<?xml version="1.0" encoding="UTF-8"?>
<technology xmlns="http://www.criticalblue.com/CascadeNS">
  <name>ASIC_90nm</name>
  <guideFrequency>300</guideFrequency>
  <estimates resourceUsageUnit="K gates" energyUnit="nJ" minResourceUsage="100"
     uncondControllerResourceUsage="0.1" condControllerResourceUsage="0.25"
     equalityComparatorResourceUsage="0.0025" encodedBitResourceUsage="0.00035"
     regResetOverhead="1.25" connectionSwitchEnergy="0.0"
     clockEnergyConst="0.165" clockEnergyMult="1.27"
     validEnergyEstimates="true">
    <units>
      <table name="execUnitActiveEnergy">
        <estimate key="access_st_1" value="0.01913"/>
        <estimate key="access_st_1r" value="0.03036"/>
        <estimate key="access_1x" value="0.03136"/>
        <estimate key="access_1" value="0.03029"/>
```

```
    <estimate key="access_1r" value="0.03821"/>
    <estimate key="access_2" value="0.04151"/>
    <estimate key="access_assoc_1" value="0.03227"/>
    <estimate key="access_assoc_1r" value="0.04133"/>
    <estimate key="access_stream_1" value="0.01661"/>
    <estimate key="access_stream_1r" value="0.02211"/>
    <estimate key="access_stream_1x" value="0.01907"/>
    <estimate key="access_stream_st_1" value="0.00224"/>
    <estimate key="access_stream_st_1r" value="0.00601"/>
    <estimate key="access_remap_1" value="0.01202"/>
    <estimate key="access_remap_1r" value="0.01202"/>
    <estimate key="arithmetic" value="0.03218"/>
    <estimate key="bitshift" value="0.05924"/>
    <estimate key="branch" value="0.00138"/>
    <estimate key="combine" value="0.06575"/>
    <estimate key="immediate32" value="0.01866"/>
    <estimate key="immediate8" value="0.03103"/>
    <estimate key="coreregfile" value="0.01082"/>
    <estimate key="logical" value="0.01511"/>
    <estimate key="multiplier64" value="0.50978"/>
    <estimate key="predicate" value="0.03864"/>
    <estimate key="registerfile" value="0.01417"/>
    <estimate key="select" value="0.03623"/>
    <estimate key="squash" value="0.01182"/>
  </table>
  <table name="execUnitInactiveEnergy">
    <estimate key="access_st_1" value="0.0003257"/>
    <estimate key="access_st_1r" value="0.0004561"/>
    <estimate key="access_1x" value="0.0004711"/>
    <estimate key="access_1" value="0.0004551"/>
    <estimate key="access_1r" value="0.0005741"/>
    <estimate key="access_2" value="0.0006236"/>
    <estimate key="access_assoc_1" value="0.0004848"/>
    <estimate key="access_assoc_1r" value="0.0006209"/>
    <estimate key="access_stream_1" value="0.0002496"/>
    <estimate key="access_stream_1r" value="0.0003321"/>
    <estimate key="access_stream_1x" value="0.0002864"/>
    <estimate key="access_stream_st_1" value="0.0000336"/>
    <estimate key="access_stream_st_1r" value="0.0000903"/>
    <estimate key="access_remap_1" value="0.0001701"/>
    <estimate key="access_remap_1r" value="0.0001701"/>
    <estimate key="arithmetic" value="0.0001201487"/>
    <estimate key="bitshift" value="0.0001680642"/>
    <estimate key="branch" value="0.0001140"/>
    <estimate key="combine" value="0.0001140"/>
    <estimate key="immediate32" value="0.0007540"/>
    <estimate key="immediate8" value="0.000168"/>
    <estimate key="coreregfile" value="0.00505005"/>
    <estimate key="logical" value="0.001280"/>
    <estimate key="multiplier64" value="0.001100"/>
    <estimate key="predicate" value="0.0001920"/>
    <estimate key="registerfile" value="0.000432"/>
    <estimate key="select" value="0.0005530"/>
    <estimate key="squash" value="0.0000120"/>
    <estimate key="sat_arithmetic" value="0.0004331"/>
  </table>
  <table name="execUnitLeakageEnergy">
```

```
      <estimate key="access_st_1" value="320000"/>
      <estimate key="access_st_1r" value="448000"/>
      <estimate key="access_1x" value="462800"/>
      <estimate key="access_1" value="447100"/>
      <estimate key="access_1r" value="564000"/>
      <estimate key="access_2" value="612600"/>
      <estimate key="access_assoc_1" value="476200"/>
      <estimate key="access_assoc_1r" value="609900"/>
      <estimate key="access_stream_1" value="245200"/>
      <estimate key="access_stream_1r" value="326300"/>
      <estimate key="access_stream_1x" value="281400"/>
      <estimate key="access_stream_st_1" value="33090"/>
      <estimate key="access_stream_st_1r" value="88730"/>
      <estimate key="access_remap_1" value="167100"/>
      <estimate key="access_remap_1r" value="167100"/>
      <estimate key="arithmetic" value="5240"/>
      <estimate key="bitshift" value="4520"/>
      <estimate key="branch" value="1650"/>
      <estimate key="combine" value="744"/>
      <estimate key="immediate32" value="2155"/>
      <estimate key="immediate8" value="1300"/>
      <estimate key="coreregfile" value="158560"/>
      <estimate key="logical" value="4940"/>
      <estimate key="multiplier64" value="83300"/>
      <estimate key="predicate" value="1120"/>
      <estimate key="registerfile" value="187500"/>
      <estimate key="select" value="3863"/>
      <estimate key="squash" value="2700"/>
      <estimate key="sat_arithmetic" value="4404"/>
    </table>
  </units>
  <buses>
    <table name="busTypeResourceUsage">
      <estimate key="CBNative_Slave_Generic" value="15.67"/>
      <estimate key="AMBA_AHB_Slave_Generic" value="15.83"/>
      <estimate key="AMBA_AHB_Master_Generic" value="17.81"/>
      <estimate key="CBNative_DMA_Streaming" value="8.74"/>
      <estimate key="AMBA_AHB_DMA_Streaming" value="9.02"/>
      <estimate key="AMBA_AHB_Master_Streaming" value="11.16"/>
    </table>
    <table name="busTypeActiveEnergy">
      <estimate key="CBNative_Slave_Generic" value="0.01970"/>
      <estimate key="AMBA_AHB_Slave_Generic" value="0.02507"/>
      <estimate key="AMBA_AHB_Master_Generic" value="0.02614"/>
      <estimate key="AMBA_AHB_DMA_Streaming" value="0.04843"/>
      <estimate key="AMBA_AXI_DMA_Streaming" value="0.04843"/>
    </table>
    <table name="busTypeStalledEnergy">
      <estimate key="CBNative_Slave_Generic" value="0.01970"/>
      <estimate key="AMBA_AHB_Slave_Generic" value="0.02507"/>
      <estimate key="AMBA_AHB_Master_Generic" value="0.02614"/>
      <estimate key="AMBA_AHB_DMA_Streaming" value="0.04843"/>
      <estimate key="AMBA_AXI_DMA_Streaming" value="0.04843"/>
    </table>
    <table name="busTypeLeakageEnergy">
      <estimate key="CBNative_Slave_Generic" value="916000"/>
      <estimate key="AMBA_AHB_Slave_Generic" value="645000"/>
```

```
      <estimate key="AMBA_AHB_Master_Generic" value="750789"/>
      <estimate key="AMBA_AHB_DMA_Streaming" value="380242"/>
      <estimate key="AMBA_AXI_DMA_Streaming" value="380242"/>
    </table>
  </buses>
  <memories>
    <memory type="ram_raws">
      <table name="resourceUsage">
        <entry key="32" value="0.276"/>
        <entry key="64" value="0.553"/>
        <entry key="128" value="1.169"/>
        <entry key="256" value="2.619"/>
      </table>
      <table name="activeEnergy">
        <entry key="32" value="0.000491"/>
        <entry key="256" value="0.002686"/>
      </table>
      <table name="inactiveEnergy">
        <entry key="32" value="0.0000835"/>
        <entry key="256" value="0.0004569"/>
      </table>
      <table name="leakageEnergy">
        <entry key="32" value="1415"/>
        <entry key="256" value="14133"/>
      </table>
    </memory>
    <memory type="ram_rsws_bw">
      <table name="resourceUsage">
        <entry key="32" value="0.088"/>
        <entry key="64" value="0.113"/>
        <entry key="128" value="0.164"/>
        <entry key="256" value="0.246"/>
        <entry key="512" value="0.666"/>
        <entry key="1024" value="1.035"/>
        <entry key="2048" value="1.776"/>
        <entry key="4096" value="3.286"/>
        <entry key="8192" value="5.833"/>
        <entry key="16384" value="10.92"/>
        <entry key="32768" value="21.00"/>
      </table>
      <table name="activeEnergy">
        <entry key="512" value="0.0006981"/>
        <entry key="1024" value="0.0007271"/>
        <entry key="2048" value="0.0007787"/>
        <entry key="4096" value="0.0008754"/>
        <entry key="8192" value="0.0013903"/>
        <entry key="16384" value="0.0023864"/>
      </table>
      <table name="inactiveEnergy">
        <entry key="512" value="0.0005741"/>
        <entry key="1024" value="0.0005980"/>
        <entry key="2048" value="0.0006617"/>
        <entry key="4096" value="0.0007579"/>
        <entry key="8192" value="0.0011247"/>
        <entry key="16384" value="0.0023304"/>
      </table>
      <table name="leakageEnergy">
```

```
      <entry key="512" value="2850"/>
      <entry key="1024" value="3625"/>
      <entry key="2048" value="5156"/>
      <entry key="4096" value="8218"/>
      <entry key="8192" value="12968"/>
      <entry key="16384" value="22468"/>
    </table>
  </memory>
  <memory type="ram_rsws_en">
    <table name="resourceUsage">
      <entry key="256" value="0.433"/>
      <entry key="512" value="0.598"/>
      <entry key="1024" value="0.926"/>
      <entry key="2048" value="1.585"/>
      <entry key="4096" value="2.916"/>
      <entry key="8192" value="5.463"/>
      <entry key="16384" value="10.688"/>
      <entry key="32768" value="20.172"/>
    </table>
    <table name="activeEnergy">
      <entry key="1024" value="0.0006334"/>
      <entry key="2048" value="0.0006678"/>
      <entry key="4096" value="0.0007336"/>
      <entry key="8192" value="0.0012903"/>
      <entry key="16384" value="0.0023864"/>
    </table>
    <table name="inactiveEnergy">
      <entry key="1024" value="0.0001078"/>
      <entry key="2048" value="0.0001098"/>
      <entry key="4096" value="0.0001105"/>
      <entry key="8192" value="0.0002132"/>
      <entry key="16384" value="0.0004265"/>
    </table>
    <table name="leakageEnergy">
      <entry key="1024" value="3625"/>
      <entry key="2048" value="5156"/>
      <entry key="4096" value="8218"/>
      <entry key="8192" value="12968"/>
      <entry key="16384" value="22468"/>
    </table>
  </memory>
  <memory type="ram_ra_ws">
    <table name="resourceUsage">
      <entry key="32" value="0.283"/>
      <entry key="64" value="0.578"/>
      <entry key="128" value="1.210"/>
      <entry key="256" value="2.671"/>
    </table>
    <table name="activeEnergy">
      <entry key="16" value="0.0005041"/>
      <entry key="32" value="0.0005041"/>
      <entry key="128" value="0.0006479"/>
      <entry key="256" value="0.0034380"/>
    </table>
      <table name="inactiveEnergy">
      <entry key="16" value="0.0000871"/>
      <entry key="32" value="0.0000871"/>
```

```
      <entry key="128" value="0.0001119"/>
      <entry key="256" value="0.0005940"/>
    </table>
    <table name="leakageEnergy">
      <entry key="32" value="1027"/>
      <entry key="64" value="2149"/>
      <entry key="128" value="4408"/>
      <entry key="256" value="11107"/>
    </table>
  </memory>
  <memory type="ram_raws_ra">
    <table name="resourceUsage">
      <entry key="32" value="0.332"/>
      <entry key="64" value="0.680"/>
      <entry key="128" value="1.414"/>
      <entry key="256" value="3.082"/>
    </table>
    <table name="activeEnergy">
      <entry key="32" value="0.001244"/>
      <entry key="256" value="0.004120"/>
    </table>
    <table name="inactiveEnergy">
      <entry key="32" value="0.0002149"/>
      <entry key="256" value="0.0007119"/>
    </table>
    <table name="leakageEnergy">
      <entry key="32" value="1295"/>
      <entry key="64" value="3772"/>
      <entry key="128" value="6475"/>
      <entry key="256" value="12424"/>
    </table>
  </memory>
  <memory type="ram_rsws_rsws_bw">
    <table name="resourceUsage">
      <entry key="128" value="0.473"/>
      <entry key="256" value="0.686"/>
      <entry key="512" value="1.583"/>
      <entry key="1024" value="2.236"/>
      <entry key="2048" value="3.530"/>
      <entry key="4096" value="6.125"/>
      <entry key="8192" value="11.377"/>
      <entry key="16384" value="21.536"/>
      <entry key="32768" value="41.072"/>
    </table>
    <table name="activeEnergy">
      <entry key="512" value="0.0005828"/>
      <entry key="1024" value="0.0006067"/>
      <entry key="2048" value="0.0006345"/>
      <entry key="4096" value="0.0006692"/>
      <entry key="8192" value="0.0007165"/>
    </table>
    <table name="inactiveEnergy">
      <entry key="512" value="0.0002016"/>
      <entry key="1024" value="0.0002054"/>
      <entry key="2048" value="0.0002138"/>
      <entry key="4096" value="0.0002304"/>
      <entry key="8192" value="0.0002638"/>
```

```
        </table>
        <table name="leakageEnergy">
          <entry key="512" value="8812"/>
          <entry key="1024" value="11312"/>
          <entry key="2048" value="12062"/>
          <entry key="4096" value="16437"/>
          <entry key="8192" value="25156"/>
        </table>
      </memory>
    </memories>
  </estimates>
</technology>
```

I.2 TSMC 130 nm technology.xml energy entries

```
<?xml version="1.0" encoding="UTF-8"?>
<technology xmlns="http://www.criticalblue.com/CascadeNS">
  <name>ASIC_130nm</name>
  <guideFrequency>300</guideFrequency>
  <estimates resourceUsageUnit="K gates" energyUnit="nJ" minResourceUsage="100.0"
     uncondControllerResourceUsage="0.1" condControllerResourceUsage="0.25"
     equalityComparatorResourceUsage="0.0025" encodedBitResourceUsage="0.00035"
     regResetOverhead="1.25" connectionSwitchEnergy="0.0"
     clockEnergyConst="0.0" clockEnergyMult="0.0"
     validEnergyEstimates="true">
    <units>
      <table name="execUnitActiveEnergy">
        <estimate key="access_st_1" value="0.04537"/>
        <estimate key="access_st_1r" value="0.06352"/>
        <estimate key="access_1x" value="0.06562"/>
        <estimate key="access_1" value="0.06339"/>
        <estimate key="access_1r" value="0.07996"/>
        <estimate key="access_2" value="0.08686"/>
        <estimate key="access_assoc_1" value="0.06752"/>
        <estimate key="access_assoc_1r" value="0.08648"/>
        <estimate key="access_stream_1" value="0.03477"/>
        <estimate key="access_stream_1r" value="0.04626"/>
        <estimate key="access_stream_1x" value="0.03990"/>
        <estimate key="access_stream_st_1" value="0.00469"/>
        <estimate key="access_stream_st_1r" value="0.01258"/>
        <estimate key="access_remap_1" value="0.02369"/>
        <estimate key="access_remap_1r" value="0.02369"/>
        <estimate key="arithmetic" value="0.05057"/>
        <estimate key="bitshift" value="0.03461"/>
        <estimate key="branch" value="0.00400"/>
        <estimate key="combine" value="0.13544"/>
        <estimate key="immediate32" value="0.03455"/>
        <estimate key="immediate8" value="0.03988"/>
        <estimate key="coreregfile" value="0.02264"/>
        <estimate key="logical" value="0.02093"/>
        <estimate key="multiplier64" value="1.24528"/>
        <estimate key="predicate" value="0.01430"/>
        <estimate key="registerfile" value="0.02965"/>
        <estimate key="select" value="0.04329"/>
        <estimate key="squash" value="0.00778"/>
      </table>
      <table name="execUnitInactiveEnergy">
        <estimate key="access_st_1" value="0.00342"/>
        <estimate key="access_st_1r" value="0.00480"/>
        <estimate key="access_1x" value="0.00495"/>
        <estimate key="access_1" value="0.00479"/>
        <estimate key="access_1r" value="0.00604"/>
        <estimate key="access_2" value="0.00656"/>
        <estimate key="access_assoc_1" value="0.00510"/>
        <estimate key="access_assoc_1r" value="0.00653"/>
        <estimate key="access_stream_1" value="0.00262"/>
        <estimate key="access_stream_1r" value="0.00349"/>
        <estimate key="access_stream_1x" value="0.00301"/>
        <estimate key="access_stream_st_1" value="0.00035"/>
```

```
      <estimate key="access_stream_st_1r" value="0.00095"/>
      <estimate key="access_remap_1" value="0.00179"/>
      <estimate key="access_remap_1r" value="0.00179"/>
      <estimate key="arithmetic" value="0.00024"/>
      <estimate key="bitshift" value="0.00334"/>
      <estimate key="branch" value="0.00023"/>
      <estimate key="combine" value="0.00027"/>
      <estimate key="immediate32" value="0.00196"/>
      <estimate key="immediate8" value="0.00222"/>
      <estimate key="coreregfile" value="0.00675/>
      <estimate key="logical" value="0.00239"/>
      <estimate key="multiplier64" value="0.00260"/>
      <estimate key="predicate" value="0.00031"/>
      <estimate key="registerfile" value="0.00275"/>
      <estimate key="select" value="0.00256/>
      <estimate key="squash" value="0.00051/>
      <estimate key="sat_arithmetic" value="0.00118"/>
    </table>
    <table name="execUnitLeakageEnergy">
      <estimate key="access_st_1" value="139700"/>
      <estimate key="access_st_1r" value="195600"/>
      <estimate key="access_1x" value="202100"/>
      <estimate key="access_1" value="195200"/>
      <estimate key="access_1r" value="246200"/>
      <estimate key="access_2" value="267500"/>
      <estimate key="access_assoc_1" value="207900"/>
      <estimate key="access_assoc_1r" value="266300"/>
      <estimate key="access_stream_1" value="107100"/>
      <estimate key="access_stream_1r" value="142500"/>
      <estimate key="access_stream_1x" value="122900"/>
      <estimate key="access_stream_st_1" value="14450"/>
      <estimate key="access_stream_st_1r" value="38740"/>
      <estimate key="access_remap_1" value="72960"/>
      <estimate key="access_remap_1r" value="72960"/>
      <estimate key="arithmetic" value="1144"/>
      <estimate key="bitshift" value="560"/>
      <estimate key="branch" value="228"/>
      <estimate key="combine" value="187"/>
      <estimate key="immediate32" value="614"/>
      <estimate key="immediate8" value="248"/>
      <estimate key="coreregfile" value="45332"/>
      <estimate key="logical" value="588"/>
      <estimate key="multiplier64" value="1944"/>
      <estimate key="predicate" value="230"/>
      <estimate key="registerfile" value="36320"/>
      <estimate key="select" value="530"/>
      <estimate key="squash" value="506"/>
      <estimate key="sat_arithmetic" value="1062"/>
    </table>
  </units>
  <buses>
    <table name="busTypeActiveEnergy">
      <estimate key="CBNative_Slave_Generic" value="0.04174"/>
      <estimate key="AMBA_AHB_Slave_Generic" value="0.08494"/>
      <estimate key="AMBA_AHB_Master_Generic" value="0.08940"/>
      <estimate key="AMBA_AHB_DMA_Streaming" value="0.08133"/>
      <estimate key="AMBA_AXI_DMA_Streaming" value="0.08133"/>
```

```
    </table>
    <table name="busTypeStalledEnergy">
      <estimate key="CBNative_Slave_Generic" value="0.02070"/>
      <estimate key="AMBA_AHB_Slave_Generic" value="0.08494"/>
      <estimate key="AMBA_AHB_Master_Generic" value="0.08494"/>
      <estimate key="AMBA_AHB_DMA_Streaming" value="0.08133"/>
      <estimate key="AMBA_AXI_DMA_Streaming" value="0.08133"/>
    </table>
    <table name="busTypeLeakageEnergy">
      <estimate key="CBNative_Slave_Generic" value="10020"/>
      <estimate key="AMBA_AHB_Slave_Generic" value="21500"/>
      <estimate key="AMBA_AHB_Master_Generic" value="22769"/>
      <estimate key="AMBA_AHB_DMA_Streaming" value="11531"/>
      <estimate key="AMBA_AXI_DMA_Streaming" value="11531"/>
    </table>
  </buses>
  <memories>
    <memory type="ram_raws">
      <table name="resourceUsage">
        <entry key="32" value="0.276"/>
        <entry key="64" value="0.553"/>
        <entry key="128" value="1.169"/>
        <entry key="256" value="2.619"/>
      </table>
      <table name="activeEnergy">
        <entry key="32" value="0.00102"/>
        <entry key="256" value="0.00562"/>
      </table>
      <table name="inactiveEnergy">
        <entry key="32" value="0.000177"/>
        <entry key="256" value="0.000972"/>
      </table>
      <table name="leakageEnergy">
        <entry key="32" value="309"/>
        <entry key="256" value="2645"/>
      </table>
    </memory>
    <memory type="ram_rsws_bw">
      <table name="resourceUsage">
        <entry key="32" value="0.088"/>
        <entry key="64" value="0.113"/>
        <entry key="128" value="0.164"/>
        <entry key="256" value="0.246"/>
        <entry key="512" value="0.666"/>
        <entry key="1024" value="1.035"/>
        <entry key="2048" value="1.776"/>
        <entry key="4096" value="3.286"/>
        <entry key="8192" value="5.833"/>
        <entry key="16384" value="10.92"/>
        <entry key="32768" value="21.00"/>
      </table>
      <table name="activeEnergy">
        <entry key="512" value="0.00146"/>
        <entry key="1024" value="0.00152"/>
        <entry key="2048" value="0.00168"/>
        <entry key="4096" value="0.00194"/>
        <entry key="8192" value="0.00260"/>
```

```
      <entry key="16384" value="0.00488"/>
    </table>
    <table name="inactiveEnergy">
      <entry key="512" value="0.00120"/>
      <entry key="1024" value="0.00125"/>
      <entry key="2048" value="0.00138"/>
      <entry key="4096" value="0.00158"/>
      <entry key="8192" value="0.00235"/>
      <entry key="16384" value="0.00488"/>
    </table>
    <table name="leakageEnergy">
      <entry key="512" value="300"/>
      <entry key="1024" value="375"/>
      <entry key="2048" value="525"/>
      <entry key="4096" value="862"/>
      <entry key="8192" value="1425"/>
      <entry key="16384" value="1612"/>
    </table>
  </memory>
  <memory type="ram_rsws_en">
    <table name="resourceUsage">
      <entry key="256" value="0.433"/>
      <entry key="512" value="0.598"/>
      <entry key="1024" value="0.926"/>
      <entry key="2048" value="1.585"/>
      <entry key="4096" value="2.916"/>
      <entry key="8192" value="5.463"/>
      <entry key="16384" value="10.688"/>
      <entry key="32768" value="20.172"/>
    </table>
    <table name="activeEnergy">
      <entry key="1024" value="0.00125"/>
      <entry key="2048" value="0.00138"/>
      <entry key="4096" value="0.00158"/>
      <entry key="8192" value="0.00235"/>
      <entry key="16384" value="0.00488"/>
    </table>
    <table name="inactiveEnergy">
      <entry key="1024" value="0.000293"/>
      <entry key="2048" value="0.000312"/>
      <entry key="4096" value="0.000320"/>
      <entry key="8192" value="0.000413"/>
      <entry key="16384" value="0.000602"/>
    </table>
    <table name="leakageEnergy">
      <entry key="1024" value="450.00"/>
      <entry key="2048" value="630.00"/>
      <entry key="4096" value="1035.00"/>
      <entry key="8192" value="1710.00"/>
      <entry key="16384" value="1935.00"/>
    </table>
  </memory>
  <memory type="ram_ra_ws">
    <table name="resourceUsage">
      <entry key="32" value="0.283"/>
      <entry key="64" value="0.578"/>
      <entry key="128" value="1.210"/>
```

```
      <entry key="256" value="2.671"/>
    </table>
    <table name="activeEnergy">
      <entry key="16" value="0.00105"/>
      <entry key="32" value="0.00105"/>
      <entry key="128" value="0.00135"/>
      <entry key="256" value="0.00719"/>
    </table>
    <table name="inactiveEnergy">
      <entry key="16" value="0.000182"/>
      <entry key="32" value="0.000182"/>
      <entry key="128" value="0.000234"/>
      <entry key="256" value="0.001244"/>
    </table>
    <table name="leakageEnergy">
      <entry key="32" value="267"/>
      <entry key="64" value="527"/>
      <entry key="128" value="1128"/>
      <entry key="256" value="2810"/>
    </table>
  </memory>
  <memory type="ram_raws_ra">
    <table name="resourceUsage">
      <entry key="32" value="0.332"/>
      <entry key="64" value="0.680"/>
      <entry key="128" value="1.414"/>
      <entry key="256" value="3.082"/>
    </table>
    <table name="activeEnergy">
      <entry key="32" value="0.00260"/>
      <entry key="256" value="0.00862"/>
    </table>
    <table name="inactiveEnergy">
      <entry key="32" value="0.00045"/>
      <entry key="256" value="0.00149"/>
    </table>
    <table name="leakageEnergy">
      <entry key="32" value="466"/>
      <entry key="64" value="667"/>
      <entry key="128" value="2948"/>
      <entry key="256" value="3030"/>
    </table>
  </memory>
  <memory type="ram_rsws_rsws_bw">
    <table name="activeEnergy">
      <entry key="512" value="0.00244"/>
      <entry key="1024" value="0.00248"/>
      <entry key="2048" value="0.00258"/>
      <entry key="4096" value="0.00279"/>
      <entry key="8192" value="0.00319"/>
    </table>
    <table name="inactiveEnergy">
      <entry key="512" value="0.000422"/>
      <entry key="1024" value="0.000430"/>
      <entry key="2048" value="0.000447"/>
      <entry key="4096" value="0.000482"/>
      <entry key="8192" value="0.000552"/>
```

```
        </table>
        <table name="leakageEnergy">
          <entry key="512" value="2325"/>
          <entry key="1024" value="2475"/>
          <entry key="2048" value="2737"/>
          <entry key="4096" value="3300"/>
          <entry key="8192" value="4350"/>
        </table>
      </memory>
    </memories>
  </estimates>
</technology>
```

I.3 Sample analysis summary

```
SUMMARY FOR CANDIDATE 21u_230c_9s_18b_176w_128d

Effort Level: 0%
Template: 64_Bit_Multiplier
Reprogrammability: 0.5
Chaining Aggressiveness: 0.5
Technology: ASIC_130nm
Bus Type: CBNative_Slave_Generic
Instruction Burst Length: 32 bytes
Stream Data Burst Length: 256 bytes
Static Data Burst Length: 256 bytes
Host Response Wait: 0 cycles
Initial Wait: 0 cycles
Inter-Burst Wait: 0 cycles
Inter-Word Wait: 0 cycles

Candidate Name: 21u_230c_9s_18b_176w_128d
Single-Port Memory Usage: 2.8K bytes
Dual-Port Memory Usage: 5.1K bytes
Total Logic Usage: 81K gates
Estimated Streaming Logic Usage: 70K gates

Code Mapping:
Total Cycles: 4952890
Dataflow Cycles: 4150654 (83%)
Chained Ideal Cycles: 3561756 (71%)
Unchained Ideal Cycles: 4743098 (95%)
Base Cycles: 3561756 (71%)
Post Alloc Cycles: 4612717 (93%)
Active Cycles: 4323587 (87%)
D$ Stall Cycles: 496268 (10%)
I$ Stall Cycles: 1242 (0%)
Offload Stall Cycles: 131793 (2%)
Energy Usage: 3278984.863nJ
Rendered Microcode Size: 1324 bytes
   Workload: Test_adpcm_encode_test_copro.trc
      Entry: f1
      Actual Activations: 591
      D$ Stall Cycles: 496268 (10% of total)
      D$ Compulsory Stall Cycles: 140916 (2% of total)
      D$ Capacity Stall Cycles: 355352 (7% of total)
      I$ Stall Cycles: 1242 (0% of total)
      I$ Compulsory Stall Cycles: 1242 (0% of total)
      I$ Capacity Stall Cycles: 0 (0% of total)
      Offload Stall Cycles: 131793 (2% of total)
      Dataflow Active Cycles: 4150654 (96% of active)
      Chained Ideal Active Cycles: 3561756 (82% of active)
      Unchained Ideal Active Cycles: 4743098 (109% of active)
      Base Cycles: 3561756 (82% of active)
      Post Alloc Cycles: 4612717 (106% of active)
      Active Cycles: 4323587 (87% of total)
      Total Cycles: 4952890
      Energy Usage: 3278984.863nJ
```

```
Architecture Attributes:
Instruction Memory: 128 x 176 (2.8K bytes)
Data Cache Slot 0: {access_st_1r 4096 bytes {1 1 1 1}}
Data Cache Accessed Lines = 43%
Data Cache Accessed Words = 16%
Data Cache Occupancy = 43%
Units: 21
Connections: 230
Sockets: 83
Output Registers: 107

Required IP Summary:
ram_ra_ws
ram_raws_ra
ram_rsws_en
ram_rsws_rsws_bw
multiplier

Logic Breakdown By Type:
Control Unit: 15.67K gates (19%)
Unit Instances: 23.655K gates (29%)
Input Selectors: 7.996K gates (10%)
Output Registers: 21.8K gates (27%)
Instruction Decoder: 12.523K gates (15%)

Logic Breakdown By Unit:
Control Unit: 15.67K gates (19%)
arithmetic_0: 2.959K gates (4%)
bitshift_0: 2.578K gates (3%)
branch_0: 0.737K gates (1%)
combine_0: 0.358K gates (0%)
coreregfile_0: 9.441K gates (12%)
immediate32_0: 2.423K gates (3%)
immediate8_0: 2.502K gates (3%)
logical_0: 2.645K gates (3%)
multiplier64_0: 10.332K gates (13%)
predicate_0: 0.45K gates (1%)
registerfile_0: 5.573K gates (7%)
sat_arithmetic_0: 1.176K gates (1%)
select_0: 3.046K gates (4%)
squash_0: 1.075K gates (1%)
access_st_1r_0: 7.741K gates (9%)
arithmetic_1: 2.773K gates (3%)
select_1: 1.99K gates (2%)
immediate8_1: 1.433K gates (2%)
arithmetic_2: 2.05K gates (3%)
select_2: 2.663K gates (3%)
arithmetic_3: 2.03K gates (2%)


Component Totals:
    Opcode Multiplexer = 2.702
    Selection Comparator = 3.352
    Hardwired Decoder = 0.245
    Escape Multiplexer = 2.245
    Setup Register = 3.978
```

```
Instruction Decoder Total = 12.523

System Parameters:
min_region_coverage: 0.95
target_unrolled_region_size: 500.0
route_cost: 0.025
access_promotion_groups: 0.0
max_mux_inputs: 16.0
max_output_registers: 16.0
region_partitioner_alpha: 0.0
chained_conn_coverage: 0.95
function_inlining_aggressiveness: 0.2
holding_input_threshold: 0.025
critical_conn_coverage: 0.95
additional_remapping_banks: 2.0
additional_remapping_bits: 2.0
min_conn_add_weight: 0.75

Total Regions: 5
Hot Region Coverage: 99% (100% Hot Microcode)

Total Energy Usage Breakdown By Component:
CBNative_Slave_Generic/active: 134984.5nJ (4%)
CBNative_Slave_Generic/stalled: 19647.15nJ (0%)
access_st_1r_0/0/active: 38385.35nJ (1%)
access_st_1r_0/0/inactive: 20873.44nJ (0%)
access_st_1r_0/0/ram_rsws_rsws_bw_1024x32/active: 48115.13nJ (1%)
access_st_1r_0/0/ram_rsws_rsws_bw_1024x32/inactive: 59903.62nJ (1%)
access_st_1r_0/1/inactive: 23773.92nJ (0%)
access_st_1r_0/1/ram_rsws_rsws_bw_1024x32/inactive: 68227.54nJ (2%)
arithmetic_0/0/active: 97774.96nJ (2%)
arithmetic_0/0/inactive: 743.72nJ (0%)
arithmetic_1/0/active: 29845.47nJ (0%)
arithmetic_1/0/inactive: 1074.49nJ (0%)
arithmetic_2/0/active: 29845.47nJ (0%)
arithmetic_2/0/inactive: 1074.49nJ (0%)
arithmetic_3/0/active: 14922.73nJ (0%)
arithmetic_3/0/inactive: 1147.15nJ (0%)
bitshift_0/0/active: 20428.06nJ (0%)
bitshift_0/0/inactive: 14571.98nJ (0%)
branch_0/0/active: 1782.89nJ (0%)
branch_0/0/inactive: 1036.87nJ (0%)
combine_0/0/inactive: 1342.23nJ (0%)
coreregfile_0/0/active: 6867.8nJ (0%)
coreregfile_0/0/inactive: 31384.64nJ (0%)
coreregfile_0/0/ram_ra_ws_128x32/active: 13158.49nJ (0%)
coreregfile_0/0/ram_ra_ws_128x32/inactive: 34895.79nJ (1%)
coreregfile_0/1/active: 6680.45nJ (0%)
coreregfile_0/1/inactive: 31440.49nJ (0%)
coreregfile_0/1/ram_ra_ws_128x32/active: 12799.54nJ (0%)
coreregfile_0/1/ram_ra_ws_128x32/inactive: 34957.89nJ (1%)
coreregfile_0/2/active: 20469.58nJ (0%)
coreregfile_0/2/inactive: 27329.79nJ (0%)
coreregfile_0/2/ram_ra_ws_128x32/active: 39219.08nJ (1%)
coreregfile_0/2/ram_ra_ws_128x32/inactive: 30387.31nJ (0%)
coreregfile_0/3/active: 10020.68nJ (0%)
coreregfile_0/3/inactive: 30444.73nJ (0%)
```

```
coreregfile_0/3/ram_ra_ws_128x32/active: 19199.31nJ (0%)
coreregfile_0/3/ram_ra_ws_128x32/inactive: 33850.73nJ (1%)
coreregfile_0/4/active: 10248.17nJ (0%)
coreregfile_0/4/inactive: 30376.91nJ (0%)
coreregfile_0/4/ram_ra_ws_128x32/active: 19635.18nJ (0%)
coreregfile_0/4/ram_ra_ws_128x32/inactive: 33775.32nJ (1%)
coreregfile_0/5/active: 120.44nJ (0%)
coreregfile_0/5/inactive: 33396.1nJ (1%)
coreregfile_0/5/ram_ra_ws_128x32/active: 230.75nJ (0%)
coreregfile_0/5/ram_ra_ws_128x32/inactive: 37132.29nJ (1%)
coreregfile_0/6/active: 23568.93nJ (0%)
coreregfile_0/6/inactive: 26405.84nJ (0%)
coreregfile_0/6/ram_ra_ws_128x32/active: 45157.35nJ (1%)
coreregfile_0/6/ram_ra_ws_128x32/inactive: 29359.99nJ (0%)
coreregfile_0/7/active: 6720.6nJ (0%)
coreregfile_0/7/inactive: 31428.52nJ (0%)
coreregfile_0/7/ram_ra_ws_128x32/active: 12876.46nJ (0%)
coreregfile_0/7/ram_ra_ws_128x32/inactive: 34944.58nJ (1%)
immediate32_0/0/active: 35761.25nJ (1%)
immediate32_0/0/inactive: 7694.73nJ (0%)
immediate8_0/0/active: 59067.13nJ (1%)
immediate8_0/0/inactive: 7735.13nJ (0%)
immediate8_1/0/active: 17649.43nJ (0%)
immediate8_1/0/inactive: 10049.02nJ (0%)
instruction_memory/ram_rsws_en_128x176/active: 653068.95nJ (19%)
instruction_memory/ram_rsws_en_128x176/inactive: 82004.96nJ (2%)
logical_0/0/active: 21616.1nJ (0%)
logical_0/0/inactive: 9369.5nJ (0%)
multiplier64_0/0/inactive: 12877.61nJ (0%)
predicate_0/0/active: 2110.12nJ (0%)
predicate_0/0/inactive: 1492.07nJ (0%)
registerfile_0/0/active: 75078.47nJ (2%)
registerfile_0/0/inactive: 6664.51nJ (0%)
registerfile_0/0/ram_raws_ra_128x32/active: 419904.19nJ (12%)
registerfile_0/0/ram_raws_ra_128x32/inactive: 69451.68nJ (2%)
registerfile_0/1/active: 17496.47nJ (0%)
registerfile_0/1/inactive: 12010.82nJ (0%)
registerfile_0/1/ram_raws_ra_128x32/active: 97855.5nJ (2%)
registerfile_0/1/ram_raws_ra_128x32/inactive: 125166.1nJ (3%)
sat_arithmetic_0/0/inactive: 5845.41nJ (0%)
select_0/0/active: 57502.09nJ (1%)
select_0/0/inactive: 9279.17nJ (0%)
select_1/0/active: 19158.84nJ (0%)
select_1/0/inactive: 11546.63nJ (0%)
select_2/0/active: 25545.11nJ (0%)
select_2/0/inactive: 11168.97nJ (0%)
squash_0/0/active: 4605.61nJ (0%)
squash_0/0/inactive: 2224.44nJ (0%)

Total Leakage Energy Usage Breakdown By Component:
CBNative_Slave_Generic/stalled: 916000nJ/s (41%)
access_st_1r_0/0/output_regs/leakage: 2778nJ/s (0%)
access_st_1r_0/1/output_regs/leakage: 926nJ/s (0%)
access_st_1r_0/leakage: 724480nJ/s (32%)
access_st_1r_0/ram_rsws_rsws_bw_1024x32/leakage: 0nJ/s (0%)
arithmetic_0/0/output_regs/leakage: 3241nJ/s (0%)
arithmetic_0/leakage: 5240nJ/s (0%)
```

```
arithmetic_1/0/output_regs/leakage: 3935.5nJ/s (0%)
arithmetic_1/leakage: 5240nJ/s (0%)
arithmetic_2/0/output_regs/leakage: 2893.75nJ/s (0%)
arithmetic_2/leakage: 5240nJ/s (0%)
arithmetic_3/0/output_regs/leakage: 2315nJ/s (0%)
arithmetic_3/leakage: 5240nJ/s (0%)
bitshift_0/0/output_regs/leakage: 4745.75nJ/s (0%)
bitshift_0/leakage: 4520nJ/s (0%)
branch_0/0/output_regs/leakage: 0nJ/s (0%)
branch_0/leakage: 1650nJ/s (0%)
combine_0/0/output_regs/leakage: 115.75nJ/s (0%)
combine_0/leakage: 744nJ/s (0%)
coreregfile_0/0/output_regs/leakage: 3704nJ/s (0%)
coreregfile_0/1/output_regs/leakage: 3704nJ/s (0%)
coreregfile_0/2/output_regs/leakage: 3704nJ/s (0%)
coreregfile_0/3/output_regs/leakage: 3704nJ/s (0%)
coreregfile_0/4/output_regs/leakage: 0nJ/s (0%)
coreregfile_0/5/output_regs/leakage: 0nJ/s (0%)
coreregfile_0/6/output_regs/leakage: 0nJ/s (0%)
coreregfile_0/7/output_regs/leakage: 0nJ/s (0%)
coreregfile_0/leakage: 158560nJ/s (7%)
coreregfile_0/ram_ra_ws_128x32/leakage: 0nJ/s (0%)
immediate32_0/0/output_regs/leakage: 6482nJ/s (0%)
immediate32_0/leakage: 2155nJ/s (0%)
immediate8_0/0/output_regs/leakage: 7408nJ/s (0%)
immediate8_0/leakage: 1300nJ/s (0%)
immediate8_1/0/output_regs/leakage: 3704nJ/s (0%)
immediate8_1/leakage: 1300nJ/s (0%)
instruction_memory/ram_rsws_en_128x176/leakage: 0.02nJ/s (0%)
logical_0/0/output_regs/leakage: 3819.75nJ/s (0%)
logical_0/leakage: 4940nJ/s (0%)
multiplier64_0/0/output_regs/leakage: 1967.75nJ/s (0%)
multiplier64_0/leakage: 83300nJ/s (3%)
predicate_0/0/output_regs/leakage: 57.88nJ/s (0%)
predicate_0/leakage: 1120nJ/s (0%)
registerfile_0/0/output_regs/leakage: 6482nJ/s (0%)
registerfile_0/1/output_regs/leakage: 4630nJ/s (0%)
registerfile_0/leakage: 187500nJ/s (8%)
registerfile_0/ram_raws_ra_32x32/leakage: 0nJ/s (0%)
sat_arithmetic_0/0/output_regs/leakage: 926nJ/s (0%)
sat_arithmetic_0/leakage: 4404nJ/s (0%)
select_0/0/output_regs/leakage: 5556nJ/s (0%)
select_0/leakage: 3863nJ/s (0%)
select_1/0/output_regs/leakage: 2778nJ/s (0%)
select_1/leakage: 3863nJ/s (0%)
select_2/0/output_regs/leakage: 5556nJ/s (0%)
select_2/leakage: 3863nJ/s (0%)
squash_0/0/output_regs/leakage: 0nJ/s (0%)
squash_0/leakage: 2700nJ/s (0%)
Total Leakage: 2212356.14nJ/s

Function Hot Spot Summary:
Function f1 100% (Chained Ideal 100%)

--------------------------------------------------------------------------------
Hotspot Region 0 now 61% (2655360/4323587 cycles), was estimated 66%
Hot Region, Relaxed Aliasing
```

```
Cycles: Unaliased=21, Dataflow=21, Chained Ideal=13, Unchained Ideal=22,
  Base=13, Post Allocation=19, Active=18
Size=292 bytes, Frequency=147520
Weights: Global=0.67, Entry adpcm_coder weight=0.67

Port Activation Map:
access_st_1r_0|32|/0  :  X           X
arithmetic_0|32|/0    :  X X XXX X XXX
arithmetic_1|32|/0    :     X X X X
arithmetic_2|32|/0    :    X    XX
arithmetic_3|32|/0    :           X
bitshift_0|32|/0      :   X XX
branch_0|32|/0        :          X     X
coreregfile_0|32|/0   :   X
coreregfile_0|32|/1   :       X
coreregfile_0|32|/2   :     X XXX
coreregfile_0|32|/3   :  X   X
coreregfile_0|32|/4   :     XXX
coreregfile_0|32|/6   :  XX   XX
coreregfile_0|32|/7   :        X
immediate32_0|32|/0   :   XXX   X
immediate8_0|32|/0    : XXXXXX
immediate8_1|32|/0    : XX
logical_0|32|/0       :        X X X
registerfile_0|32|/0  : XXX  X      X   XX
registerfile_0|32|/1  : X
select_0|32|/0        :     X X XXXXX XX
select_1|32|/0        :       X   X
select_2|32|/0        :      X X X
squash_0|32|/0        :               X

Instructions Per Clock (IPC)=1.83
Operations Per Clock (OPC)=4.33

Covered 61% of execution time

--------------------------------------------------------------------------------
Hotspot Region 1 now 37% (1622720/4323587 cycles), was estimated 32%
Hot Region, Relaxed Aliasing, Falls Through
Cycles: Unaliased=7, Dataflow=7, Chained Ideal=11, Unchained Ideal=10, Base=11,
  Post Allocation=12, Active=11
Size=204 bytes, Frequency=147520
Weights: Global=0.33, Entry adpcm_coder weight=0.33

Port Activation Map:
access_st_1r_0|32|/0  :     X  X
arithmetic_0|32|/0    : X X  XX
arithmetic_2|32|/0    :    X
arithmetic_3|32|/0    :   X
bitshift_0|32|/0      :    X
branch_0|32|/0        :         X
coreregfile_0|32|/0   :    X
coreregfile_0|32|/1   :  X
coreregfile_0|32|/2   :      XX
coreregfile_0|32|/3   :   X
coreregfile_0|32|/6   :  XX  X
coreregfile_0|32|/7   :    X
```

```
immediate32_0|32|/0   : XX  X
immediate8_0|32|/0    : XXXX
immediate8_1|32|/0    : X
logical_0|32|/0       :    XXXX
predicate_0|32|/0     :  X
registerfile_0|32|/0  : XXXXX XXXXX
registerfile_0|32|/1  : XXX
select_1|32|/0        :   X
select_2|32|/0        :    X
squash_0|32|/0        :      XXX

Instructions Per Clock (IPC)=1.45
Operations Per Clock (OPC)=4.55

Covered 98% of execution time
```

J. Embedded Systems Conference 2005 Paper

Using Coprocessor Synthesis to Accelerate Embedded Software

Richard Taylor, Paul Morgan[1]
CriticalBlue, 226 Airport Parkway, Suite 470, San Jose, CA 95110
408 467 5091
{richard.taylor, paul.morgan}@criticalblue.com

ABSTRACT

Design of complex embedded systems is becoming increasingly expensive, while product life cycles are shortening. Commercial viability requires that silicon platforms get to market quicker and stay in the market longer, making reprogrammability a necessity. We present Cascade, a tool for developing coprocessors that accelerate existing embedded software applications, with no requirement for detailed microprocessor knowledge. Cascade allows functionality to be extended quickly with minimal user intervention, retaining a high degree of reprogrammability of the software implementation thus lengthening the lifespan of the platform and delivering results promptly.

1 INTRODUCTION

The complexity of SoC design means that design cost and time to market (TTM) are significant factors in determining the success or failure of a product. Short life cycles mean that reprogrammability is a key requirement allowing derivative products to be produced in a short time and with low design cost.

Software solutions provide the highest degree of programmability with lower design and verification costs, a fact reflected in an increasing proportion of functionality being implemented in software rather than hardware in complex SoC systems. One major drawback of implementing a solution purely in software on a general-purpose microprocessor is that speed, silicon area and power/energy performance is usually poor compared to a dedicated hardware approach. For this reason it is often desirable to offload some of the functionality to hardware, trading off the flexibility of software for the performance of hardware.

The problem is that offloading even a small fraction of the overall functionality, generally the few functions that dominate the performance of the application, is expensive in terms of both design cost and design time. The fixed nature of the resulting hardware means that future derivative designs face either restrictions due to the limitations of the hardware functions, or an expensive and time-consuming redesign of the hardware to implement the new functionality.

[1] Paul Morgan is also based at the Institute for System Level Integration, Livingston, UK.

CriticalBlue's approach is to combine the advantages of both software and hardware with an automated coprocessor synthesis solution providing the performance, power consumption and area advantages of dedicated hardware along with the flexibility, low-cost and fast time to market of software.

This paper presents CriticalBlue's coprocessor synthesis methodology and our tool, Cascade, which has been developed to automate this process. In addition, we will present and discuss how Cascade generates coprocessors to accelerate real-world applications, confirm the benefits and present the conclusions of our approach.

2 COPROCESSOR SYNTHESIS

Many embedded applications have a number of key functions that encapsulate only a small proportion of the executed code but dominate execution time. Often these functions will contain significant instruction level parallelism that the general-purpose microprocessor cannot take advantage of due to not having the necessary execution resources. Superscalar and Very Long Instruction Word (VLIW) general purpose processors attempt to address this problem by providing a degree of parallelism extraction at either compile time or dynamically at run-time. However, this typically results in a larger hardware overhead as the processor must be designed for the exploitation of parallelism across a wide range of applications.

The approach taken by Cascade is to exploit the inherent parallelism in key functions that would most benefit from being offloaded. This is achieved by utilizing a key technology advantage of analyzing executable code compiled for the target platform, along with an instruction trace captured by the tool, the details of which are used to determine an optimized combination of execution, connectivity and control resources for the coprocessor. Cascade is able to balance temporal and spatial computation in the architecture depending on the inherent code parallelism of offloaded functions and user constraints. By having the freedom to optimize both the coprocessor architecture and the microcoded instructions that run upon it, a more optimized balance can be struck between the efficiency of custom hardware and the flexibility of a reprogrammable processor. Key to the Cascade methodology is that it neither attempts to replace nor replicate the functionality of the main processor nor the design flow, thus minimizing system design disruption and allowing the coprocessor to avoid much of the infrastructure overhead of general purpose processors. Instead the coprocessor is optimized for a particular task, but retains much of the flexibility implied by a software implementation.

2.1 Cascade design flow

Cascade's technology is illustrated in Figure 1. In the first step the compiled application software is analyzed using standard profiling tools. This process aids designers in identifying software functions that would benefit from acceleration.

Once the user has identified the software functions to be offloaded, Cascade analyzes the instruction code and automatically maps the chosen functions onto a dedicated coprocessor that has been architected to extract the maximum parallelism. Analysis is performed to extract both the control and data dependencies between instructions. At the end of this second step, information is provided to the user about the estimated performance of the co-processor. This includes estimations of communication overhead with the main processor.

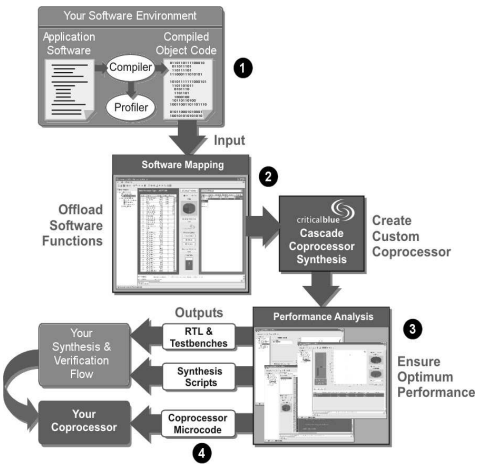


Figure 1. Cascade design flow

In the third step, Cascade produces an instruction and bit accurate C model of the coprocessor designed in step 2. By using the model in the context of a system level design environment, the user is able to understand the implications of offloading selected software functions within the context of the overall design. Users are able to perform rapid "what if" analysis with very quick turnaround.

Once satisfied with the coprocessor's performance, an RTL form of the coprocessor can be generated for simulation and synthesis using standard EDA tools. In this fourth and final step, the coprocessor microcode is generated. Microcode can be generated independently of the coprocessor hardware, allowing new microcode to be targeted at an existing coprocessor design. The original executable is modified automatically so that calls to the offloaded functions are automatically vectored to a communications library. This causes automatic handoff to the coprocessor, passing parameters and results automatically between the processor systems. Hardware developed through coprocessor synthesis is architected to communicate directly with the bus interface of the main processor such as AMBA.

2.2 Comparison with existing design methodologies

There are several approaches to accelerating embedded software on existing platforms with the aim to reducing design time compared to the traditional custom hardware development methodology.

Behavioral Synthesis facilitates rapid development of dedicated hardware by using a higher level of abstraction than RTL, allowing generation of RTL from C, C++ or SystemC descriptions of the desired hardware. Savings in area and power versus manually created RTL can be achieved in some cases. The resulting hardware is fixed, offering maximum performance at the cost of being non-programmable. In embedded designs it has become increasingly necessary to retain the programmability of software in

as much of the design as possible to broaden the potential application domains and extend the lifespan of the end-product. Behavioral synthesis is most applicable where the highest possible performance is required or where reprogrammability is not important.

Custom Instruction Set Processors provide a methodology for optimizing the entire processor around the application being targeted. This is potentially a very effective and hardware-efficient solution that retains complete software flexibility, but requires detailed knowledge of the desired specification of the processor and the use of a custom tool chain to utilize the processor. In cases where the application specific processor is being used alongside an existing general purpose processor, the designer must manually deal with issues of communication and coherency between the processors. Designing a new processor and the tools to support it is very time consuming, and requires detailed knowledge of both the underlying processor architecture and the application software, which will be running on it. EDA tools are available to greatly speed this process where this knowledge is present, resulting in significantly reduced design times compared to manual creation of a new processor.

The key advantage of the Cascade approach compared to existing methodologies is that the entire process of offloading existing software functionality is automated with minimal intervention required from the designer. The reprogrammability of software is maintained while complex issues such as coprocessor design optimization, code generation, communication between processors, and cache coherency, are dealt with transparently to the designer. Embedded software code originally targeted at the main embedded processor can be utilized directly without any additional effort to provide a seamless acceleration to the software application.

3 ACCELERATING EMBEDDED APPLICATIONS

To prove the performance of Cascade's automated coprocessor generation methodology, we examine the acceleration of three real-world embedded applications.

3.1 Bayer image processing

Digital imaging is becoming an increasingly common feature of many consumer electronics products, with embedded hardware required to process data captured by the image sensor. The majority of current sensors produce output in a Bayer mask pattern or another similar pattern, requiring interpolation to produce the color image[2]. This task is computationally intensive, particularly for video or streaming applications, making it an ideal candidate for offloading to a coprocessor to accelerate the task.

Analysis of the code by Cascade identifies two key functions of the algorithm: interpolation and defect correction. Both these functions are then flagged that they should

[2] Further details can be found at http://www.matrix-vision.com/support/articles/pdf/art_bayermosaic_e.pdf

be offloaded to a coprocessor by Cascade. The code is compiled for an ARM9 processor (compatible with the ARM v4T ISA), and the unmodified binary is then fed into Cascade to allow automated generation of an optimized coprocessor along with the accompanying microcode.

Performance of both the ARM9 processor and the coprocessor generated by Cascade is shown in the table below. This considers both the defect correction loop and the four loops of the patch interpolate function. Only defect correction and smooth green are performed for every pixel in the image, the other loops are performed every fourth pixel which is reflected in the weighting for each loop.

Loop	Weighting	ARM Cycles Per Iteration	CriticalBlue Cycles Per Iteration	CriticalBlue Performance
Defect Correction	1.00	113	32 per 2 iterations	16.00
Green Base	0.25	26	42 per 7 iterations	1.50
Smooth Red	0.25	53	74 per 7 iterations	2.64
Smooth Blue	0.25	51	80 per 7 iterations	2.86
Smooth Green	1.00	79	38 per 2 iterations	19.00
Weighted Total		**225**		**42.00**

The average number of clock cycles per pixel for the ARM9 is 225 whereas the Cascade generated coprocessor consumes 42 cycles, representing a speedup of **5.36**. It is estimated that the design time to take the code through the tool and generate a working and verified coprocessor is less than two days. When minor optimizations were made to the source code the coprocessor completed the same task in 15.2 clock cycles, an acceleration factor of **14.8** with design time of 4 days.

Targeting the coprocessor to a 0.18µm TSMC ASIC process, the area requirement of the coprocessor is 1.35mm^2 and the worst-case performance was 223.1MHz with typical speed of 260.1MHz.

3.2 BCH cyclic coding

The BCH algorithm is a forward error correction-coding scheme that allows a number of bit errors in a message to be corrected without requiring retransmission[3]. The code contains a number of elements that present potential difficulties to extracting parallelism including nested loops, complex control conditions, arbitrary pointer dereferencing, and variable strides through arrays.

[3] The source code used in this example can be found at http://www.eccpage.com/bch3.c

Two primary functions encapsulate the algorithm, namely encode_bch and decode_bch. Although many embedded systems are likely to employ only one of these functions depending on the application, we offload both to a coprocessor.

Cascade's analysis of the code highlights several points within each function that are consuming the majority of processing time. One example within the decoder is a loop

```
for (j=0; j < length; j++)
     if (recd[j] != 0) s[i] ^= alpha_to[(i*j)%n];
```

that is executed many times, and the serially-dependent operations within the loop cause the ARM9 pipeline to take tens of cycles per iteration.

Cascade exploits these loop characteristics to maximize speedup. The aforementioned loop is completed in just two cycles per iteration on the coprocessor due to the optimized arrangement of the functional units. The diagram to the right shows some of the functional units utilized within the coprocessor, with the darkened units simultaneously active within one particular cycle highlighting parallel execution of instructions.

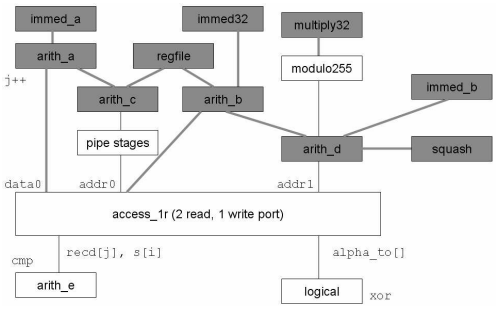


Figure 2. Coprocessor functional units

The results of offloading these functions are shown in the table below. It can be seen that with no modification to the source code Cascade has accelerated the application by a factor of **4.87**. This is using a BCH bit length of 256; slightly higher speedup is possible with longer bit lengths, and slightly lower speedup with shorter bit lengths.

Function	ARM Cycles	CriticalBlue Cycles	Acceleration
Encode	550233	125926	4.37
Decode	1430294	280432	5.10
Total	**1980527**	**406388**	**4.87**

3.3 MP3 encoding

Compression of audio is a processor-intensive task that often needs to be performed in real time for embedded applications, making it an ideal candidate for acceleration. MP3 (MPEG audio layer 3) is currently the most widely used audio compression algorithm and has therefore been selected for our analysis. Shine is a relatively simple fixed-point open source implementation of MP3 that performs well on

processors without a floating-point unit such as ARM[4]. Minor modifications are made to the code, replacing assembly code functions with the equivalent C code to make it platform independent. No optimizations are performed on the code.

A one second audio clip is compressed to MP3 using the algorithm. Profiling information revealed that the "L3_window_filter_subband" function is called many times during the encode process and is responsible for almost half of all processor cycles. Therefore this function is selected for offloading to a custom coprocessor.

The function contains a number of small loops that are executed many times during each call to the function. Cascade exploits available parallelism by unrolling the loops up to 30 times and executing several iterations of each loop concurrently.

Offloading this key MP3 encoding function results in a speedup of **5.13** as shown in the table below. This represents a design time of less than two days. The design is extended to include a hardware fixed-point multiplier block coded in VHDL and added as a user unit to be utilized by Cascade as a building-block for the coprocessor. Adding a user unit is an optional stage to further improve performance, in this case additional design time is minimal due to the simple design (10 lines of VHDL) and area cost is similar to the original coprocessor due to more efficient use of multiplication units. Resultant speedup with the user unit block is **6.90** compared to ARM9.

Function	ARM Cycles	CriticalBlue Cycles	Acceleration
L3_window_filter_subband	270146448	52637292	5.13
L3_window_filter_subband + HW	270146448	39143670	6.90

4 CONCLUSION

In this paper, CriticalBlue has presented its methodology for accelerating embedded software by means of Cascade, a coprocessor synthesis tool. Cascade's advantage is the simplicity with which this methodology can be integrated into existing design flows to enable designers to automatically extract parallelism directly from embedded software code at the compiled object level. Through the presented real-world application examples, it has been shown that Cascade can achieve significant speedups in a fraction of the time required for other embedded software acceleration techniques. Using Cascade also retains a high degree of the reprogrammability of the original software implementation and therefore represents a major breakthrough in realizing programmable hardware accelerators directly from embedded software.

[4] Shine fixed-point source code available at http://www.mp3-tech.org/programmer/sources/shinefixed.zip

K. CODES-ISSS 2005 Paper

Automated Data Cache Placement for Embedded VLIW ASIPs

Paul Morgan[1], Richard Taylor, Japheth Hossell, George Bruce, Barry O'Rourke
CriticalBlue Ltd
17 Waterloo Place, Edinburgh, UK
+44 131 524 0080
{paulm, richardt, jexh, georgeb, barryo} @criticalblue.com

ABSTRACT
Memory bandwidth issues present a formidable bottleneck to accelerating embedded applications, particularly data bandwidth for multiple-issue VLIW processors. Providing an efficient ASIP data cache solution requires that the cache design be tailored to the target application. Multiple caches or caches with multiple ports allow simultaneous parallel access to data, alleviating the bandwidth problem if data is placed effectively. We present a solution that greatly simplifies the creation of targeted caches and automates the process of explicitly allocating individual memory access to caches and banks. The effectiveness of our solution is demonstrated with experimental results.

Categories and Subject Descriptors
B.3.2 [**Memory Structures**]: Design Styles – *Cache memories;* C.1.1 [**Processor Architectures**]: Single Data Stream Architectures – *VLIW Architectures*

General Terms
Algorithms, Design, Performance, Theory

Keywords
Cache, cache optimization, embedded applications, ASIP.

1. INTRODUCTION
Embedded systems often employ application-specific instruction processors (ASIPs) that have been tailored to the domain in which they will be employed. In the interests of maximizing performance and minimizing energy consumption it is desirable to exploit instruction level parallelism inherent in the code. Employing a VLIW processor provides an ideal mechanism for extracting this parallelism. However, a significant number of instructions in many applications are loads or stores, in our experiments typically around 30% of all instructions, therefore data memory bandwidth issues are often a significant bottleneck to successfully exploiting instruction-level parallelism. Thus it is necessary to instantiate and effectively utilize data cache units that allow multiple concurrent accesses to maximize data bandwidth.

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

Access patterns for the instruction cache tend to be much more structured and predictable than those for the data cache leaving more scope for performance improvement in successful data cache configuration and data allocation. The key to achieving an optimal solution is maximally exploiting both temporal and spatial locality in memory accesses, which are application dependent. Factors such as cache size, bank configuration and number of ports present a highly configurable architecture. Multiple ports allow simultaneous access to a single cache, different banks hold different data sets within the cache, and multiple caches can have different properties each suited to different data access patterns within the application. Effectively utilizing cache architectures, both in terms of selecting the hardware configuration and optimizing data allocation to exploit maximum benefit from the chosen configuration, is a challenging and time consuming task.

We present an automated solution by way of a software tool for guiding the creation of a suitable hardware configuration and allocating data to optimally utilize the selected configuration. This is achieved by automatically generating and analyzing the memory trace of an application, taking advantage of the memory access information available at design and compile time to produce a more efficient allocation than would be possible by performing dynamic allocation using run-time logic. We provide a library of cache blocks to allow a wide range of architectures to be created tailored to the target application. Our tool guides the user towards an ideal hardware solution by performing allocation and analysis on a selection of candidate architectures, producing comparative results for each candidate architecture.

This document is presented as follows. First we examine a selection of related work in Section 2. In Section 3 we list the hardware blocks created to build our caches, and detail the software allocation algorithm used to optimize data allocation to the cache. In Section 4 we undertake experiments to show the cache performance benefits of our solution. Finally we present our conclusions and suggest future work that could be undertaken to further our research in Section 5.

2. RELATED WORKS
A great deal of research on the topics of cache configuration and mapping has been undertaken in the past with many of the methods being proposed targeted at application-specific architectures. Givargis [3] recognized that better cache performance can be obtained by considering the target application during the design phase of an ASIP. Similarly, Panda et al. [8]

[1] Paul Morgan is also based at the Institute for System Level Integration, Livingston, UK.

demonstrated a method of optimizing memory hierarchy, including data cache, for application-specific designs.

Single cache optimizations such as varying line size, set associativity or replacement algorithm have been covered for several goals, such as energy [14][15] or hit rate [7]. For application-specific architectures it is often beneficial to have one or more additional caches with a different configuration to the first, depending on the nature of the application being executed. A well-researched technique is that of the scratchpad memory [4][8], a small area of storage in which elements can be placed without disrupting the main cache. Gordon-Ross et al. [4] extend the analysis to a two-level cache hierarchy, proposing a simultaneous exploration technique for both cache levels that trades off power requirements and performance.

Sudarsanam and Malik [12] addressed the issue of memory bank assignment to optimize for simultaneous access in ASIPs with a tool called SPAM. This work tackles a similar problem to what we face but is targeted at single cache ASIPs with two identical banks whereas our tool targets highly configurable architectures that can have multiple caches of different types each with different sized banks.

Grun et al. have produced excellent work on memory architecture exploration in [5] culminating in a tool called APEX. This work considers the entire memory of an embedded system, rather than focusing on the data cache and does not provide for the parallel data access requirements of multiple-issue VLIW processors.

What we propose is to provide a library of customizable cache blocks that can be tailored at design time to a suitable configuration for the target application. Allocation of code to caches and banks is automated by a tool we have designed using a software algorithm that attempts to find an optimized solution for the selected hardware caching architecture taking into consideration the parallel access requirements of a multiple-issue VLIW processor. This approach allows multiple possible candidates for the hardware configuration to be quickly examined for suitability, overcoming the problem of attempting to simultaneously optimize both hardware configuration and software mapping, a problem which could not be solved in reasonable time with our level of configurability.

To the best of our knowledge no previous work has explored an automated software mapping for highly configurable hardware cache architectures as proposed here.

3. CACHE ALLOCATION

To facilitate the creation of an application-specific data cache, we provide a library of highly configurable cache blocks to allow our cache to be optimized for a wide range of applications. There are currently four cache styles in our library; three window caches and one direct-mapped static cache. A cache unit may contain a number of independent banks, each of which may hold a different data set. Using multiple banks allows different data areas to be held, addressed independently and accessed simultaneously depending on available ports.

Window caches hold a contiguous region of memory in each bank, and automatically attempt to keep the correct addresses in the cache by pre-fetching data from main memory in the background when accesses are ascending or descending and are nearing the edge of the cached region. The three window caches are distinguished by their port configuration, one with a single read/write port, the second with an additional read only port and the third with two read/write ports. Additional ports increase the complexity of the cache so the trade-off between area and performance must be considered. Window caches require no tag overhead due to the cached memory region being contiguous, significantly reducing the area footprint, and the pre-fetch mechanism greatly improves performance on favorable access patterns.

Static caches provide a more conventional direct-mapped cache, with the addition of software placement of data into banks. Such caches are simpler than window caches, with no pre-fetch mechanism, and anything between 8 and 64 lines each of 64 words to provide a more suitable cache for accesses of a sparsely spread pattern, with sizes of 2k, 4k, 8k or 16k bytes. All static caches are single-ported, utilizing around 40-45% less area than a dual-ported window cache depending on configuration options.

Each of the four cache units has further parameterized configuration options to ensure maximum flexibility to adapt to any application. The size of each window cache is configurable in powers of two from 512 to 64k words, with 1, 2, 3 or 4 banks. In addition to design-time choices, the number of banks and bank size ratios can be dynamically configured by the host at runtime.

All our cache blocks are directly mapped taking advantage of the lower latency, smaller area requirements and reduced power consumption offered compared with set- or fully-associative caches, as tag comparisons are not required with direct-mapped caches. We rely upon an effective software allocation and the pre-fetching abilities of our window caches to minimize cache misses that would otherwise be inherent in a direct-mapped cache.

The aforementioned cache blocks provide an enormous range of configuration options. There are 32 possible valid combinations of each window cache, and 4 static cache options, meaning a dual-cache design offers 1296 configurations. It would be an intractable task to attempt to fully automate the selection of an optimal cache configuration that meets all the required criteria for any custom application. Therefore the user selects a number of candidates for the cache configuration from the provided hardware blocks based upon area and performance requirements, and the type of application being accelerated.

The target application is run with a representative data set, and a memory access trace is automatically generated. The trace is then analyzed to determine ranges of memory that show independence in either the spatial or temporal ranges. Instructions are partitioned into groups whose access patterns interfere both spatially and temporally. Each group is allocated to a cache bank according to the algorithm described below. Hardware cache coherency logic ensures that the memory hierarchy will always be valid for any access pattern regardless of the memory configuration, relieving the allocation algorithm of this concern and providing resilience to any future changes in the executed code. Additional logic ensures that any location in the memory hierarchy can be accessed from any port, although an interference stall penalty is incurred if the data is cached in a location other than the identified bank allowing time for the hardware to transparently fetch the data from the correct bank. The software analysis optimizes the allocation of memory regions to the

available cache configurations providing post-allocation performance statistics on each candidate to guide the selection process towards an optimal solution.

3.1 Allocation Algorithm

The aim of the allocation algorithm is to assign grouped memory access instructions to appropriately sized banks to minimize cache misses, and minimize interference between groups by assigning concurrently active ranges into different banks where possible. In cases where both a window cache and static cache are available, the algorithm attempts to select the most appropriate cache type for each group. A flow diagram overview is illustrated in figure 1 opposite. The tool examines the original program and lists all load/store instructions, then records from the memory access trace the range of addresses accessed by each instruction.

An interference graph is built with nodes representing load and store instructions, with instructions accessing overlapping address ranges with respect to cache line boundaries being merged into a single node. An interference edge is added between nodes that access data in the same activation range (a temporal run-time value calculated by the algorithm dependent upon the varying access density at the trace point of current analysis), identifying those nodes as being simultaneously "live", which is analogous to a register allocation interference graph [2].

Critical analysis is then performed to identify memory accesses that may require to be issued in parallel by the VLIW processor. These accesses are identified by performing a scheduling step on a fully optimized version of the most executed portions of code, forming critical access groups (CAGs) from accesses issued on the same cycle. This information is added to the interference graph such that if instructions in a CAG are located on the same node then that node's criticality is set to a value representing the number of simultaneous accesses that must be issued from the node. That node can then be allocated if possible to a bank with the required parallel access capability. If instructions in the CAG span multiple nodes then the criticality value is applied to the edge linking those nodes, indicating that those nodes should be allocated to banks with sufficient ports to satisfy the criticality constraints of both nodes simultaneously.

Nodes are sorted into priority order depending upon their memory access frequency to be assigned to available cache banks, starting with the most important node. Each bank's attributes, such as its type (window or static), the bank size, and the number of ports the bank is accessible through, are known to the allocation algorithm and are used to influence the selection of a bank for each node.

Choosing whether a node should use a window cache is a crucial step in the algorithm, as significant performance benefits are possible for access patterns amenable to window caching but performance can be degraded for unsuitable access patterns. Analyzing the entire memory trace and recording the frequency of all sequential accesses would be extremely slow and memory hungry, therefore selection of the cache type for each node is based upon the access proportion of that node. This is calculated as the number of accesses represented by that node divided by the address range accessed by the node. Nodes with access proportion above a threshold based upon available window and static banks are likely to perform sequential accesses so are earmarked for window caching. Nodes below the threshold will be allocated to static banks. We have found this approach to work well for most applications although a more robust and comprehensive algorithm for cache type selection is under development.

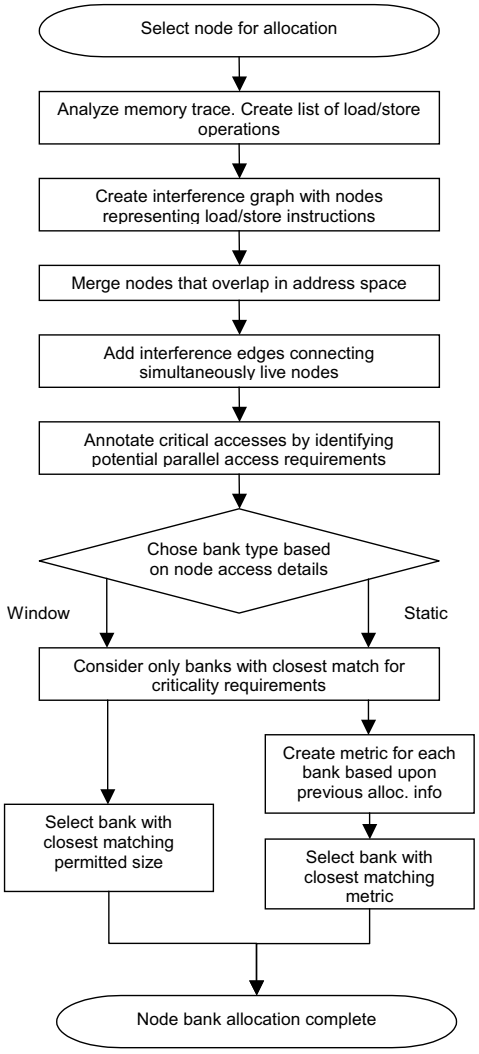


Figure 1. Allocation algorithm flow chart

Once the type of cache has been selected, the next step is choosing the bank that will be assigned to each node. The

criticality value determined previously, including analysis of neighboring nodes connected by critical edges, indicates the optimal number of ports for a node. The banks with the closest number of ports are selected for further consideration and all others are disregarded for that node. The remaining steps are dependent upon whether the node under consideration is targeted towards a window or a static cache. For window caches a preferred size is calculated based upon the total available size of window cache banks multiplied by the proportion of the overall access count generated by that node. The bank with size most closely matching the preferred size is selected.

Allocating static cache banks to nodes is more complex, requiring the generation of a metric for each bank to aid selection. All non-critical edges connected to nodes designated for static cache allocation are removed. This allows banks to be assigned to nodes that were connected by non-critical edges and is permitted because the metric generated for each bank contains information about where that bank has been previously allocated. Whenever a static bank is allocated a record of the lines used by the allocation is stored along with the tag(s) used for each line. Thus when the metric for a subsequent allocation is generated it consists of: the proportion of lines that negatively interfere with a previous allocation (two address lines map to the same cache lines with different tags); the proportion of lines that positively interfere with a previous allocation (similar to negative interference but with the same tags); and the proportion of the address range accessed by the node that does not completely fit into the cache. The bank with the best metric is selected for allocation. If two banks have identical metrics then the smaller bank is selected. If they are the same size a deterministic choice is made.

As nodes are assigned to banks in priority order less important nodes may be assigned to banks that do not necessarily fit their access pattern. The assumption is that a memory configuration can be found that allocates the most important accesses to suitable banks and any remaining accesses will have less influence on the overall performance of the memory. If there are no suitable banks available for a particular node, that node is assigned to the *default* bank which is designated the first time it is required. The default bank is chosen as the static cache bank with the least allocated accesses; if no static cache is available, the least accessed window cache bank is chosen instead. Once selected the default bank is then fixed for the rest of the allocation.

4. EXPERIMENTAL RESULTS

To evaluate our architecture and coupled allocation algorithm, we verify the performance of the system running real-world applications using instruction-set simulators (ISS). For our architecture we use a custom simulator that is part of our tool, and results are shown from an ARM920T using the ARMulator simulator. The ARM was chosen as it has 16Kb data cache arranged into a 64-way set associative configuration and mapped using a content addressable memory (CAM) [11] giving it a high level of adaptability for different applications. The ARM results are provided simply as a reference rather than a direct comparison, as our architectures are targeted towards specific applications in each case whereas the ARM is general-purpose. In addition, as we are targeting a multiple-issue VLIW processor that completes each experiment in fewer cycles, our target system places much higher demands on the data cache than the ARM processor.

By configuring ARMulator to produce verbose statistics during simulation, we can monitor cache activity such as hits, misses and fetches, for both instruction and data caches. We use our tool with a selection of potential cache configuration candidates which the tool cycles through performing allocation and producing results relating to the cycle count and cache hits and misses. Our tool is compatible with the ARM instruction set and can therefore utilize the same compiled code as that used on ARMulator.

To keep the design exploration simple and within the bounds of a realistic cache area for the selected applications, we limit the choices to one window cache or one static cache, or one of each, with a maximum size of 16Kb in total. Window caches are considered with varying bank numbers of 1, 2, 3 or 4, and have one read/write port and one read port. Static caches have a single read/write port. Even with this relatively small selection of that possible from the available hardware blocks, there are still a significant number of combinations to explore. The use of our tool greatly speeds this process, helping guide the user towards an optimal solution. Run-times for these examples are in the range of 2-10 minutes on a 2.8 GHz Pentium 4 PC with 1 Gb RAM.

To ensure that the results reflect the true cost of the miss penalties for each architecture, we have included an estimated number of stall cycles which indicates the number of bus cycles that the AMBA AHB bus consumes fetching or writing back cache lines. These estimates are based on factors such as initial transfer latency, burst transfer rate, cache line size, and bus contention. Our caches have been designed such that they do not increase the latency of accesses, maintaining overall system performance. Details of the AMBA AHB specification can be found in [1].

We ran several applications considered to be relevant to real-world embedded systems, which are also amenable to speedup on a VLIW ASIP and are therefore applicable to our target system. These are applications that we have previously targeted to some of our ASIP designs as part of other projects but in future we plan to extend our tests to relevant applications from the MediaBench suite. To ensure that results reflect only the monitored function, the caches are flushed and cleaned before entering the function so that the cache will have a cold start. This is achieved by inserting dedicated cache control assembly instructions immediately prior to entering the function. All caches were configured to use a write-back policy meaning that only a miss on a read or write requires a cache line to be synchronized with main memory causing a stall. One exception is an interference miss where the desired data is cached, but a read/write attempt is made on a bank or port other than where the data resides. The logic will automatically reference the correct location, but a shorter stall may be necessary in this case and this is taken into account in our "stall cycles" figures in the results.

We present the results of our experiments for each application. As expected some of our potential candidates did not produce competitive results, so due to space restrictions and the large number of candidates these were pruned and will not be considered further. Results for the three best candidates in terms of area, performance and energy, are shown for each application.

The selected candidates were synthesized for a TSMC 0.18µm process using Artisan memories to obtain the area requirements of each architecture including logic area. We then performed worst-case dynamic power analysis with high switching activity rates at

200MHz using Synopsys Power Compiler for logic cells and CACTI [10] for SRAM cells. Real-world power is likely to be lower as these figures are intended only for rough comparison between our architectures. The power figure for multi-ported architectures is shown per-port, as this provides a more realistic representation of the energy contribution over the entire application. This is because the instantaneous power of a dual-port cache performing two simultaneous accesses will be higher than that of a single-port cache, but the single-port cache will require two accesses on separate cycles to achieve the same result. More accurate integrated energy modeling within the tool based upon cache activity is a planned future development.

Worst-case dynamic power figures for a cache equivalent to that in the ARM were estimated using a combination of CACTI and the information in [16] regarding CAM-tag lookup caches. Area information for the ARM cache is not publicly available so we estimate the cache area based upon its configuration and available data on the arm architecture. The result appears to be high, but it agrees with the value calculated by extrapolating the difference in areas quoted by ARM for the ARM9 with different cache sizes. For comparison, the area of a 16K 4-way set associative cache with one bank and a 32 byte line size is 2.28mm^2.

For reference the logic overhead of our first cache architecture (Custom1 in section 4.1 below) is under 8%, relatively low even allowing for our pre-fetch logic due to lack of tag lookup overhead. Actual logic overhead will vary depending upon cache configuration and memory technology used.

4.1 Color Interpolation

The first application is a color interpolation function with approximately 300 lines of C. It performs integer colorization of Bayer-encoded images commonly produced by digital image sensors. It inputs an 8-bit intensity encoded bitmap image and outputs the full 24-bit image using interpolation. A detailed overview is available at [6]. This function relies heavily upon array manipulations therefore placing significant demands on the memory subsystem that must be satisfied to achieve good performance. The input image is CIF resolution (352x288) with file size 100Kb. The architectures selected for the color interpolation application are as follows:

- Custom1 – 4k Static cache; 8k Window cache, 1 bank
- Custom2 – 8k Static cache; 8k Window cache, 1 bank
- Custom3 – 16k Window cache, 1 8k bank + 2 4k banks

Results for this application are shown in Table 1.

Table 1. Results for color interpolation (access count 2822608)

Cache	Misses	Hit Rate	Stall Cycles	Area	Power
ARM	319830	88.67%	4797450	3.69mm^2	206mW
Custom1	29527	98.95%	236216	1.46mm^2	104mW
Custom2	14898	99.47%	119184	1.74mm^2	107mW
Custom3	123	**99.99%**	**3321**	1.95mm^2	129mW

Using our tool, we find the optimal configuration for this application is a single 16K window cache with one 8k bank and two 4k banks, resulting in a hit rate greater than 99.99%. This is largely due to the effectiveness of the pre-fetching mechanism fitting well with the bank configuration and access pattern. The allocation algorithm ensures that interferences between simultaneous accesses to different memory locations are minimized by allocating those locations to separate banks.

4.2 Run-Length Encoding

The second application is a run-length encoding function, a basic lossless compression algorithm that is simple to implement (approx. 200 lines of C) and has low computational and storage requirements. For this experiment, we compress an arbitrary data stream stored in a text file with a size of 50Kb. The architectures selected for the RLE application are as follows:

- Custom1 – 2k Static; 4k Window cache, 1 2k + 2 1k banks
- Custom2 – 4k Static cache; 4k Window cache, 2 2k banks
- Custom3 – 4k Static cache; 8k Window cache, 2 4k banks

Results for this application are shown in Table 2.

Table 2. Results for RLE function (access count 930914)

Cache	Misses	Hit Rate	Stall Cycles	Area	Power
ARM	109538	88.23%	1643070	3.69mm^2	206mW
Custom1	26	99.99%	702	0.93mm^2	90mW
Custom2	20	99.99%	540	1.14mm^2	97mW
Custom3	16	**99.99%**	**432**	1.47mm^2	104mW

Our architecture with a combination of one window cache and one static cache performs very well with only 6K total cache size. Further small improvements are possible with optimal performance realized at 12K total cache size. The access patterns of this application suit both a static and multi-bank window cache being implemented. Our tool performs allocation to the available caches and banks, and allows the user to decide the area/performance tradeoff between the possible solutions.

4.3 FIR Filter

Finally, we implement an integer FIR filter with 6 taps and supply a 20Kb input data stream. Signal processing places a high demand on the memory subsystem, therefore good cache performance is reflected in good overall performance for these applications. Architectures chosen for the FIR Filter application are as follows:

- Custom1 – 2k Window cache, 1 bank
- Custom2 – 2k Static cache; 2k Window cache, 1 bank
- Custom3 – 2k Window cache, 2 1k banks

Results for this application are shown in Table 3.

Table 3. Results for FIR Filter (access count 458718)

Cache	Misses	Hit Rate	Stall Cycles	Area	Power
ARM	67580	85.27%	1013700	3.69mm^2	206mW
Custom1	65536	85.71%	1769472	0.39mm^2	49mW
Custom2	263	99.94%	2104	0.64mm^2	83mW
Custom3	10	**99.99%**	**270**	0.40mm^2	49mW

The FIR filter test clearly shows the benefit of utilizing the flexibility of our architecture and the effectiveness of our

allocation mechanism. The 2K single-bank window cache results in a considerable number of miss cycles, but adding a 2K static cache shows a vast improvement. Going back to a single 2K cache, but with two banks produces the optimal result while maintaining a low area requirement.

5. CONCLUSION AND FUTURE WORK

In this paper, we have presented a software tool for guiding the creation of a cache configuration for application-specific VLIW architectures and automating data placement into that cache. Our experimental results show that this approach provides significant improvements over what would be possible using a conventional cache with placement performed at run-time, while at the same time keeping area and energy requirements low. Using window caches allows tag overhead to be eliminated and coherency issues are greatly reduced, but maintaining performance requires careful selection of the architecture and effective placement of data. The problem of effectively utilizing tailored cache architectures is solved by our automated solution that analyzes the code and performs allocation with the aim of optimizing cache efficiency.

Our allocation algorithm is being continually evolved. Particular effort is being focused at identifying more accurately the suitability of allocating ranges to window cache banks. We have not yet integrated energy optimization into our algorithm; currently our approach aims to lower system energy by reducing cache misses thus minimizing costly bus accesses [13]. A more detailed energy analysis and optimization is a prime interest in our ongoing research since the cache subsystem can account for up to 50% of the energy consumption in typical embedded processors such as the ARM920T [9]. As part of the continuing development of our tool, we are currently integrating data cache energy analysis as part of the automated flow, and plan to provide optimizations that may be traded off against performance and/or area criteria at the user's prerogative. Additionally, although we have focused on the data cache for performance optimization, the wide instruction cache in a VLIW processor provides great scope for energy savings; therefore we are in the process of exploring energy optimizations for the instruction cache with a view to integrating this functionality into the tool.

6. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers who provided constructive comments. Paul Morgan's contribution to this work is partially funded by EPSRC.

7. REFERENCES

[1] ARM Limited. AMBA Specification Rev. 2.0, 1999, pp 3-1 – 3-58.

[2] Chaitin, G. Register Allocation and Spilling via Graph Coloring. Proceedings of the 1982 Symposium on Compiler Construction, June 1982, pp. 98-105.

[3] Givargis, T. Improved Indexing for Cache Miss Reduction in Embedded Systems. Proceedings of Design Automation Conference, 2003, pp. 875 – 880.

[4] Gordon-Ross, A., Vahid, F., Dutt, N. Automatic Tuning of Two-Level Caches to Embedded Applications. Proceedings of Design, Automation and Test in Europe Conference, Volume 1, February 2004, pp. 208 – 213.

[5] Grun, P., Dutt, N., Nicolau, A. Access Pattern-Based Memory and Connectivity Architecture Exploration. ACM Transactions on Embedded Computing Systems, Vol. 2, No. 1, February 2003, pp 33–73.

[6] Kimmel, R. Demosaicing: Image Reconstruction from Color CCD Samples, IEEE Transactions on Image Processing, Volume 8, Issue 9, September 1999, pp 1221 – 1228.

[7] Megiddo, N., Modha, D.S. Outperforming LRU with an Adaptive Replacement Cache Algorithm. Computer, Volume 37, Issue 4, April 2004, pp. 58 – 65.

[8] Panda, P.R., Dutt, N.D., Nicolau, A., Catthoor, F., Vandecappelle, A., Brockmeyer, E., Kulkarni, C., De Greef, E. Data Memory Organization and Optimizations in Application-Specific Systems, IEEE Design & Test of Computers, Volume 18, Issue 3, May-June 2001 , pp. 56 – 68.

[9] Segars, S. Low Power Design Techniques for Microprocessors, International Solid State Conference, February 2001, pp 34 – 35.

[10] Shivakumar, P., Jouppi, N. CACTI 3.0: An Integrated Cache Timing, Power and Area Model. Compaq Western Research Laboratory report, August 2001.

[11] Sloss, A., Symes, D., Wright, C. ARM System Developer's Guide - Designing and Optimizing System Software, Morgan Kaufmann publishers, 2004, pp 403 – 457.

[12] Sudarsanam, A., Malik, S. Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs. ACM Transactions on Design Automation of Electronic Systems, April 2000, pp 242–264.

[13] Verma, M., Wehmeyer, L., Marwedel P. Efficient Scratchpad Allocation Algorithms for Energy Constrained Embedded Systems. Workshop on Power-Aware Computer Systems, December 2003.

[14] Zhang, C., Vahid, F., Najjar, W. A Highly Configurable Cache Architecture for Embedded Systems. Proceedings of the 30th Annual International Symposium on Computer Architecture, June 2003, pp 136 – 146.

[15] Zhang, C., Vahid, F., Najjar, W. Energy Benefits of a Configurable Line Size Cache for Embedded Systems. Proceedings of the IEEE Computer Society Annual Symposium on VLSI, February 2003, pp. 87 – 91.

[16] Zhang, M., Asanovic, K. Highly-Associative Caches for Low-Power Processors. Kool Chips Workshop, 33rd International Symposium on Microarchitecture, December 2000.

L. Design Automation Conference 2007 Paper

ASIP Instruction Encoding for Energy and Area Reduction

Paul Morgan[1], Richard Taylor
Critical Blue
San Jose, CA 95131
+1 408 467 5091
{paulm, richardt} @criticalblue.com

ABSTRACT
Application-specific VLIW processors provide an energy and area efficient solution for high-performance embedded applications. One significant design issue is that the long instruction word required to express the instruction parallelism represents a significant cause of energy dissipation. We present an application-tailored instruction encoding solution that modifies the instruction architecture to minimize the instruction word width. We demonstrate the effectiveness of our solution across a range of benchmarks, resulting in average energy savings of 20% and an average area reduction of 18%, with no performance penalty.

Categories and Subject Descriptors
B.3.2 [**Memory Structures**]: Design Styles – *Cache memories;* C.1.1 [**Processor Architectures**]: Single Data Stream Architectures – *VLIW Architectures*

General Terms
Algorithms, Design, Theory

Keywords
Cache, cache optimization, embedded applications, energy, ASIP.

1. INTRODUCTION
Embedded systems often employ application-specific instruction processors (ASIPs) that have been tailored to the domain in which they will be employed. In the interests of maximizing performance and minimizing energy consumption it is desirable to exploit instruction level parallelism inherent in the code. Employing a VLIW processor provides an ideal mechanism for extracting this parallelism with minimal additional overhead penalty. A key decision in the architecting of an application-specific VLIW processor is selecting the instruction word layout; a wider word allows more parallel instruction issues to be made but at the cost of memory energy dissipation and area associated with the instruction cache. Many VLIW implementations' instruction format leans towards allowing maximal parallelism extraction, avoiding a performance bottleneck. As a result, the average entropy of the instruction word tends to be poor, meaning that the processor instruction cache and instruction fetch mechanism are both area and energy inefficient.

DAC 2007, June 4–8, 2007, San Diego, California, USA.

We propose a solution that allows for a significantly reduced instruction word width, while retaining the full performance of the underlying architecture. This is achieved by analyzing the nature of the application-specific code executed on the processor, and modifying the instruction set to make efficient use of commonly utilized instructions by means of short opcodes substituted for full opcodes (which include register operands and literal fields) within the VLIW instruction. These shortened opcodes may be easily decoded to the processor's native microcode without impacting the overall critical path timing of a typical ASIP design.

This paper is presented as follows. We examine a selection of related work in Section 2. In Section 3 we detail the algorithm used to encode an optimal instruction set for a specific application domain. In Section 4 we demonstrate the area and power benefits of our solution. Finally we present our conclusions and suggest future work that could further our research in Section 5.

2. RELATED WORKS
Code compression has been extensively examined in previous research, particularly in the area sensitive embedded domain. In this paper we concentrate on compression of both main memory and instruction cache. Wolfe and Chanin [7] demonstrated a Huffman-based code compression scheme for RISC processors that formed the basis for many subsequent studies. Lekatsas and Wolf [4] propose an arithmetic coding and Markov-model based instruction compression framework, again based on RISC embedded processors. ARM's Thumb encodes 32-bit instructions into a 16-bit format, providing an average of 35% smaller code; however this approach also carries a performance penalty of around 15-20% [3]. In general, VLIW processors have lower code density than RISC, providing an opportunity for more specialized compression techniques. Xie et al. [8] demonstrate a method that can be used on flexible VLIW architectures. A similar approach that achieves higher compression by using LZW-based compression is proposed by Lin et al. [5].

Other than ARM's Thumb instruction set, which carries an associated performance penalty, the aforementioned techniques do not attempt to reduce the instruction width by encoding microcode directly; rather they aim to improve the density of useful information in instructions while retaining the same ISA. Our approach differs in that we modify the instruction set in an application-specific manner. Additional decode logic is synthesized, enabling functionality equivalent to the original instruction format. To the best of our knowledge no previous work has explored such a fine-grained mechanism for generating an application-tailored instruction set encoding as proposed here.

[1] Paul Morgan is also based at the Institute for System Level Integration (Livingston, UK), and the University of Edinburgh (Edinburgh, UK).

3. OUR APPROACH

Application-specific processors have their instruction set tailored for high performance with low area and energy requirements when executing software from the targeted domain. Often a greater performance/area ratio can be achieved by targeting a narrower range of applications; our approach targets the more specialized end of the ASIP spectrum. We make changes to the instruction set, with the aim of improving the area and energy efficiency of both external memory (and associated bus transfers) and the on-core instruction cache.

Rather than examining only the VLIW instructions and attempting to perform compression on complete words, we observe how opcodes within each instruction effect operations at a deeper level within the processor. During software analysis, we measure the opcodes dispatched to individual functional units for repetition that may enable efficient short opcode substitutions to be made. We then implement a dictionary-like encoding scheme, similar to that described in [6], but at a more fine-grained level within the target architecture. The algorithm we propose targets a Harvard architecture processor, therefore we do not concern ourselves with operands which are stored in the data cache.

Assuming that each slot of the VLIW architecture is capable of issuing an opcode to any functional unit, for each functional unit that is amenable to opcode encoding we can create effective redundancy in the instruction word as we reduce that unit's average bandwidth requirement through our encoding scheme. As redundancy is created, we can reduce the width of the instruction word, adjusting the instruction decode mechanism as appropriate, while still retaining the same level of throughput. In practice we have found that many applications permit the instruction word to be reduced to 50% or less of its original width, with no adverse impact on throughput.

To ensure that the size and energy consumption of the short opcode decode look-up table (LUT) logic remains reasonable, and that the number of bits required for encoded opcodes is kept small, we do not attempt to encode all possible opcodes for each functional unit. Rather, a profile-based analysis is performed that results in infrequently-used opcodes being identified and removed from the LUT. In order that these opcodes can still be executed, we implement several escape codes at the processor instruction level that allow short opcodes to be bypassed and the full opcode be passed directly from other bits in the instruction word. Due to the reduced instruction word width, only a small number of full opcodes can be issued simultaneously; consequently use of this facility can have a detrimental effect on performance if it results in an instruction fetch bottleneck. It is therefore one of the key decisions of our approach with regards to the trade-off between reducing area and energy, and maintaining performance.

We begin by explaining the base VLIW architecture to which we apply our algorithm. This architecture will have various combinations of Functional Units (FUs) dependent upon the target application. Each slot in the instruction word can dispatch a short opcode to any FU. Our algorithm sweeps across the range of possible issue slots when performing allocation, ensuring effective utilization of the instruction word. For each candidate a trial schedule is performed to determine the impact on overall performance. We aim to optimize utilization of available slots on a per-cycle basis, reducing redundancy and overall cycle count, improving performance while minimizing energy consumption.

To create a framework for our encoding algorithm, we modify the instruction word as follows: A count field indicates how many short (i.e. encoded) opcodes are present in that particular instruction word, counting each slot from the most significant bits in the instruction word. In the case that the full opcode escape mechanism is required, the required number of bits will be made available from the least significant bits in the instruction word. Depending on the full opcode width, any number of short opcode slots may overlap with the full opcode. Thus the corresponding short opcode bits are not used during that cycle by the algorithm, which limits the short opcode count value for that cycle. Our instruction scheduling algorithm is aware of these restrictions and assigns the layout of the instruction word appropriately.

Each short opcode within the instruction word is itself split into two sections: a selector "address" indicating for which FU the short opcode is intended, and the encoded instruction. We allocate variable-width addresses in priority order of FU usage using a Huffman-type encoding so that heavily used FUs require fewer address bits and thus have more active opcode bits. The instruction part of the short opcode is then translated to the target FU's microcode by look-up table decode logic within the FU, with decode mappings unique to each FU. There is one escape code instruction for each FU that indicates no entry is available for the desired setup pattern in the look-up table; in this case that FU will fetch the full opcode from the instruction word, bypassing the decode mechanism. Figure 1 shows three FUs executing decoded short opcodes and one FU bypassing the decode logic, executing a full opcode from the instruction word.

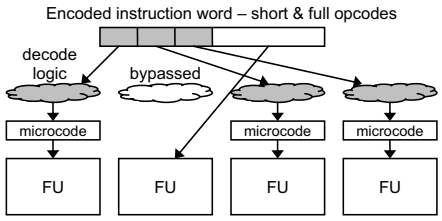


Figure 1. Comparison of instruction word formats

The key to ensuring that our approach achieves the desired goals is effective selection of full opcodes to be assigned to short opcodes. The number of opcodes can be varied for each individual FU, but the opcode value range is always aligned on a power of 2 boundary. The reason for this restriction is to allow the use of simple equality comparison hardware with the short opcode to select the correct individual FU, improving area and energy efficiency of the decode logic.

We implement our algorithm as follows: First we generate an execution trace by simulating the compiled application, driven by a typical stimulus and thus providing a representative profile. The trace contains a list of opcodes for each functional unit, which is then arranged into priority order based on occurrence frequency. Previous experimentation over a wide range of processors and input stimulus has led us to the conclusion that a short opcode width in the range of 8 to 11 bits wide tends to be optimal – any fewer is too restrictive on the number of short opcodes giving little benefit; any greater results in the decode hardware becoming too large and inefficient. Therefore we have a choice of between 256 and 2048 short opcodes that can be decoded to microcode.

The algorithm then proceeds to iterate through a loop. Initially, short opcodes are 8 bits wide, giving 256 slots available. At least one short opcode is used as an escape pattern to allow a non-encoded full opcode to be executed, and the remaining short opcodes are assigned to full opcodes in priority order of the aforementioned list. Assignment of opcodes progresses until either the list is exhausted, meaning all full opcodes present in the list have been assigned or all available short opcodes have been utilized. In the latter case, the algorithm has to decide whether to increase the FU short opcode width by a single bit, doubling the number of short opcodes available. This is a crucial decision in the algorithm, and one of the key factors taken into consideration is how many long opcodes will be efficiently encoded should the available number of short opcodes be doubled by increasing the width by one bit. The aim is to expand the number of encoded opcodes only when it will result in greater energy savings from the reduction in use of full opcodes, compared with the additional energy utilized by the larger decode logic.

If the decision is taken to not increase the short opcode width, the algorithm is complete. Otherwise the algorithm proceeds to assign full opcodes to the newly created short opcodes in priority order as before, until either all full opcodes have been assigned or all short opcodes have again been utilized and another decision to extend the short opcode width is taken. If the short opcode reaches 11 bits wide then it is not possible to increase the width any further, and the algorithm automatically completes.

When the opcode mapping is complete, we create hardware decode logic for each functional unit, incorporating the bypass mechanism for non-encoded full opcodes. We then automatically generate the application-specific processor RTL integrating our new instruction format and decode logic, and recompile the executable using our opcode mapping logic which is integrated with our targeted ASIP compiler. The result is a VLIW ASIP with a narrower instruction path that is functionally equivalent to the original processor from a high-level software design perspective.

4. EXPERIMENTAL RESULTS

To demonstrate the viability of our approach we evaluate the performance, area and energy consumption of both the instruction cache and complete core for ASIPs created with encoded and non-encoded instruction formats. Our experiments are based on benchmarks present in the Mediabench suite [2]. We compile the benchmarks for the ARM9 processor using arm-gcc, and profile using gprof. For each benchmark we select the key function(s) in terms of processor utilization, for offloading to an application-specific VLIW processor aimed at extracting maximal performance from the function(s) and therefore the overall application. Due to arm-gcc tool chain build issues with ghostscript, mesa and rasta benchmarks, these are excluded from our experiments but we evaluate all other Mediabench benchmarks.

To create each application-specific VLIW processor, we use Cascade, a tool we previously developed that generates such processors by analyzing target functions creating an ASIP that extracts maximum performance from those functions within user-defined area constraints. The instruction and data caches are also automatically generated as part of this process. To ensure a fair comparison, we leave the area constraint, cache size restrictions, and effort levels at their defaults throughout all tests with the only change is the application of our instruction encoding algorithm.

We first run the benchmarks using Cascade with no encoding of the instruction format. In this case, the word width is effectively unconstrained other than as part of overall processor area constraints. This approach results in large variations in the word width as the tool tries to optimize for performance within an area limit, meaning that the instruction word layout is very dependent on the peak level of parallelism extracted from the target function.

Our instruction encoding algorithm is then enabled and we create new ASIPs. We run cycle-accurate simulations of before and after processors to get the number of cycles taken to complete the benchmark using the supplied Mediabench data sets. Instruction cache stalls are taken into account, with estimated cache fill times based upon a typical external memory connected to an AMBA AHB bus [1]. Each processor is synthesized to obtain area estimates using Synopsys Design Compiler on a TSMC 0.13µm process. We run gate-level simulations on Synopsys VCS to obtain switching activity information before performing gate-level power and energy analysis using Synopsys Power Compiler.

With our instruction encoding algorithm enabled, cache area falls considerably in all except one test: JPEG encode. Further investigation reveals the reason that the instruction width doesn't fall significantly in this test – the original design has a narrow width of 128 bits and the instruction trace doesn't lend itself well to our encoding algorithm, leaving us little room for improvement. However this appears to be an unusual exception. The largest saving in instruction cache size was achieved for MPEG2 decode, falling from 320 bits to 104 bits wide, both with a depth of 256 words. The average instruction width over all benchmarks before implementation of our algorithm was 231 bits; with our encoding scheme enabled that drops significantly to an average of 94.5 bits. Cache depth increases in some cases to compensate for the additional instructions required when the bypass mechanism is used for instructions that have not been encoded. The average instruction cache depth before our algorithm is 384 words; afterwards it rises to 496 words. Overall the reduction in width is much more sizeable than the increase in depth; total cache memory size drops from an average of 92.5K bits to 49.25K bits.

Overall area falls to a lesser degree than cache area in all cases due to the additional decode logic – average gate count rises from 93.75K gates to 102.56K gates, an increase of just under 10%. Figure 2 shows the area of both the instruction cache and total processor area, after application of our algorithm, relative to the non-encoded case. In all cases overall synthesized area is lower after the application of our algorithm, as the saving in cache area more than compensates for the additional decode logic.

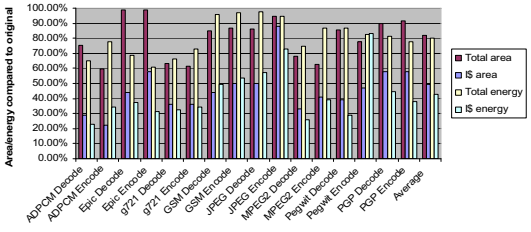


Figure 2. Area and energy compared to non-encoded case

Perhaps somewhat unexpectedly, performance also improves slightly in all cases even though this is not a primary goal of the algorithm. The reduced cycle count is a welcome side-effect created by two factors: more efficient use of the instruction cache resulting in fewer cache capacity stalls; and when a stall is encountered it takes fewer cycles to fetch a narrower VLIW word from 32-bit main memory than it does to fetch a wider word, resulting in reduced stall cycles. The average cycle count across all tests shows a drop of around 8% with our algorithm enabled.

We observe overall energy reduction in all examined tests, driven by the substantial savings in energy consumed by the instruction cache, as seen in Figure 2. We observe the average energy consumed over the tests is just over 80% of the original energy, a saving of almost 20%. Energy consumption of the instruction cache itself drops to 42.8% of original energy using our algorithm. The overall energy reduction is lower than that saved in the instruction cache because our algorithm introduces an additional energy consumption element in the look-up tables required to decode the encoded opcodes.

Leakage power also fell in all tests as a consequence of the smaller instruction cache. The average leakage power before applying our algorithm was 878.16 μW, dropping to 758.15 μW after the application of our encoding scheme, a reduction of 14%.

Overall system energy consumption is likely to be further improved beyond that observed in the processor itself, as our technique reduces the amount of system memory required to store instructions, and similarly a corresponding reduction in memory bus traffic will be observed. We summed the total microcode size for all MediaBench tests both before and after applying our algorithm, the results being 274,056 bytes and 161,476 bytes respectively – an average reduction of 41%. We observed a similar reduction in bus traffic due to instruction transfers.

5. CONCLUSION AND FUTURE WORK

In this paper, we have presented a method of reducing both area and energy consumption of on-chip instruction caches for application-specific VLIW processors. We achieve this by targeting the instruction word format more aggressively towards the application for which the processor is designed, removing some of the flexibility from the instruction format that is seldom necessary in highly tailored application-specific processors.

Our experimental results show that this approach provides significant savings, averaging 18% smaller area and 20% lower energy consumption across a range of benchmarks, while at the same time having no detrimental effect on performance. Indeed, all benchmarks show an improvement in performance due to the increased instruction cache utilization efficiency; average cycle count over all benchmarks reduced by 8%. Our automated solution analyzes the existing code and performs fine-grained encoding at the functional unit level, optimizing code density and instruction cache utilization without sacrificing performance.

We plan to continue development of our algorithm to improve the results in several ways. The short opcode selection mechanism is at present quite effective, but we expect there are improvements available particularly with regards to energy efficiency. At the moment the algorithm considers only the area and performance trade-off when deciding whether to increase short opcode width, on the basis that decode logic energy is largely correlated with area. While this naïve assumption holds true in most tests, there may be cases it does not. We are therefore developing the algorithm to explicitly consider the energy likely to be consumed by each candidate, and weigh that factor into the decision.

In the longer term, we are considering the viability of extending our approach to a more sophisticated technique where fixed decode logic is replaced by a small dynamically-allocated buffer. This would allow software instructions to load full opcodes to a slot that can later be referenced using a short opcode, allowing dynamic short opcode mapping under compiler control. It would be possible to work around potential bottlenecks before they occur by loading full opcodes during periods of redundancy in the instruction word. However, this approach introduces an extra dimension of complexity making it difficult to utilize effectively. We continue to investigate the practicality issues surrounding the implementation of such an extension to our algorithm.

6. ACKNOWLEDGMENTS

Paul Morgan's contribution to this work is partly funded by the Engineering and Physical Sciences Research Council (EPSRC).

7. REFERENCES

[1] ARM, "Amba Specification Rev. 2.0", 1999, pp. 3-1 - 3-58.

[2] Chunho, L., Potkonjak, M., and Mangione-Smith, W. H., "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", *Proceedings of the 13th International Symposium on Microarchitecture*, 1997, pp 330-335.

[3] Goudge, L. and Segars, S., "Thumb: Reducing the Cost of 32-Bit RISC Performance in Portable and Consumer Applications", *Compcon '96. 'Technologies for the Information Superhighway' Digest of Papers*, 1996, pp 176-181.

[4] Lekatsas, H. and Wolf, W., "SAMC: A Code Compression Algorithm for Embedded Processors", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 1689-1701, 1999.

[5] Lin, C. H., Xie, Y., and Wolf, W., "LZW-Based Code Compression for VLIW Embedded Systems", *Proceedings of Design, Automation and Test in Europe Conference*, 2004, pp 76-81 Vol.73.

[6] Piguet, C., Volet, P., et al., "Code Memory Compression with Online Decompression", *Proceedings of the 27th European Solid-State Circuits Conference*, 2001, pp 389-392.

[7] Wolfe, A. and Chanin, A., "Executing Compressed Programs on an Embedded RISC Architecture", *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 1992, pp 81-91.

[8] Xie, Y., Wolf, W., and Lekatsas, H., "A Code Decompression Architecture for VLIW Processors", *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture*, 2001, pp 66-75.

References

[1] P. Magarshack and P. Paulin, "System-on-chip beyond the nanometer wall," in *Proceedings of the 40th Design Automation Conference*, pp. 419–424, 2-6 June 2003.

[2] J. Bourgoin, "The coming reality for SOC designers," tech. rep., MIPS Technologies, Inc, 2002.

[3] T. Claasen, "An industry perspective on current and future state of the art in system-on-chip (SoC) technology," *Proceedings of the IEEE*, vol. 94, pp. 1121–1137, June 2006.

[4] K. Keutzer, S. Malik, and A. Newton, "From ASIC to ASIP: the next design discontinuity," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 84–90, 16-18 Sept. 2002.

[5] Semiconductor Industry Association, *International Technology Roadmap for Semiconductors (ITRS)*, 2005.

[6] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria, "Power exploration for embedded VLIW architectures," in *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pp. 498–503, 5–9 Nov. 2000.

[7] C. Shekhar, R. Singh, A. Mandal, S. Bose, R. Saini, and P. Tanwar, "Application Specific Instruction set Processors: redefining hardware-software boundary," in *Proceedings of the 17th International Conference on VLSI Design*, pp. 915–918, 2004.

[8] Semiconductor Industry Association, *International Technology Roadmap for Semiconductors (ITRS)*, 2003.

[9] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, pp. 437–445, Dec. 1994.

[10] F. Najm, "A survey of power estimation techniques in VLSI circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, pp. 446–455, Dec. 1994.

[11] P. Landman, "High-level power estimation," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 29–35, 12-14 Aug. 1996.

[12] M. Schmitz, B. Al-Hashimi, and P. Eles, *System-level Design Techniques for Energy-efficient Embedded Systems*. Springer, 2004.

[13] H. Veendrick, "Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits," *IEEE Journal of Solid-State Circuits*, vol. 19, pp. 468–473, Aug 1984.

[14] J. Kao, S. Narendra, and A. Chandrakasan, "Subthreshold leakage modeling and reduction techniques," in *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pp. 141–148, 10-14 Nov. 2002.

[15] A. Chandrakasan, S. Sheng, and R. Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 473–484, April 1992.

[16] B. Davari, R. Dennard, and G. Shahidi, "CMOS technology for low voltage/low power applications," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 3–10, May 1994.

[17] A. Chandrakasan, S. Sheng, and R. Brodersen, *Low Power Digital CMOS Design*. Kluwer Academic Publishers, 1995.

[18] A. Abnous and J. Rabaey, "Ultra-low-power domain-specific multimedia processors," in *IX Workshop on VLSI Signal Processing*, pp. 461–470, 30 Oct.-1 Nov. 1996.

[19] V. Ramakrishna, R. Kumar, and A. Basu, "Switching activity minimization by efficient instruction set architecture design," in *Proceedings of the 45th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 485–488, 4-7 Aug. 2002.

[20] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis, "Architectural and compiler techniques for energy reduction in high-performance microprocessors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, pp. 317–326, Jun 2000.

[21] A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi, "An ASIP instruction set optimization algorithm with functional module sharing constraint," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 526–532, 7-11 Nov. 1993.

[22] A. Alomary, T. Nakata, Y. Honma, J. Sato, N. Hikichi, and M. Imai, "PEAS-I: a hardware/software co-design system for ASIPs," in *Proceedings of the European Design Automation Conference*, pp. 2–7, 20-24 Sept. 1993.

[23] N. Binh, M. Imai, and A. Shiomi, "A new HW/SW partitioning algorithm for synthesizing the highest performance pipelined ASIPs with multiple identical FUs," in *Proceedings of the European Design Automation Conference*, pp. 126–131, 16-20 Sept. 1996.

[24] N. N. Binh, M. Imai, A. Shiomi, and N. Hikichi, "A hardware/software partitioning algorithm for designing pipelined ASIPs with least gate counts," in *Proceedings of the 33rd Design Automation Conference*, pp. 527–532, 3-7 June 1996.

[25] W. Dougherty, D. Pursley, and D. Thomas, "Instruction subsetting: trading power for programmability," in *Proceedings of the IEEE Computer Society Workshop on System Level Design*, pp. 42–47, 16-17 April 1998.

[26] S. Gupta and F. Najm, "Power macromodeling for high level power estimation," in *Proceedings of the 34th Design Automation Conference*, pp. 365–370, 9–13 June 1997.

[27] M. Barocci, L. Benini, A. Bogliolo, B. Ricco, and G. De Micheli, "Lookup table power macro-models for behavioral library components," in *Proceedings of the IEEE Alessandro Volta Memorial Workshop on Low-Power Design*, pp. 173–181, 4-5 March 1999.

[28] F. Vermeulen, F. Catthoor, L. Nachtergaele, D. Verkest, and H. De Man, "Power-efficient flexible processor architecture for embedded applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, pp. 376–385, June 2003.

[29] S. Yang, W. Wolf, and N. Vijaykrishnan, "Power and performance analysis of motion estimation based on hardware and software realizations," *IEEE Transactions on Computers*, vol. 54, pp. 714–726, Jun 2005.

[30] J.-H. Jeng, F. Lai, E. Naroska, and U. Schwiegelshohn, "Multimedia ASIP SoC codesign based on multicriteria optimization," in *Proceedings of the International Conference on Consumer Electronics*, pp. 342–343, 19-21 June 2001.

[31] L. Wehmeyer, M. Jain, S. Steinke, P. Marwedel, and M. Balakrishnan, "Analysis of the influence of register file size on energy consumption, code size, and execution time," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 1329–1337, Nov. 2001.

[32] M. K. Jain, M. Balakrishnan, and A. Kumar, "Integrated on-chip storage evaluation in ASIP synthesis," in *Proceedings of the 18th International Conference on VLSI Design*, pp. 274–279, 3-7 Jan. 2005.

[33] K. Karuri, C. Huben, R. Leupers, G. Ascheid, and H. Meyr, "Memory access microprofiling for ASIP design," in *Proceedings of the Third IEEE International Workshop on Electronic Design, Test and Applications*, p. 6pp., 17-19 Jan. 2006.

[34] V. Kalyanaraman, M. Mueller, S. Simon, M. Steinert, and H. Gryska, "A power dissipation comparison of ALU-architectures for ASIPs," in *Proceedings of the European Conference on Circuit Theory and Design*, vol. 2, pp. II/217–II/220, 28 Aug.-2 Sept. 2005.

[35] B. Middha, V. Raj, A. Gangwar, A. Kumar, M. Balakrishnan, and P. Ienne, "A Trimaran based framework for exploring the design space of VLIW ASIPs with coarse grain functional units," in *Proceedings of the 15th International Symposium on System Synthesis*, pp. 2–7, 2002.

[36] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai, "PEAS-III: an ASIP design environment," in *Proceedings of the International Conference on Computer Design*, pp. 430–436, 17-20 Sept. 2000.

[37] A. Kitajima, T. Sasaki, Y. Takeuchi, and M. Imai, "Design of application specific CISC using PEAS-III," in *Proceedings of the 13th IEEE International Workshop on Rapid System Prototyping*, pp. 12–18, 1-3 July 2002.

[38] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 1338–1354, Nov. 2001.

[39] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen, and H. Meyr, "A methodology for the design of application specific instruction set processors (ASIP) using the machine description language LISA," in *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pp. 625–630, 4-8 Nov. 2001.

[40] M. Mostafizur, R. Mozumdar, K. Karuri, A. Chattopadhyay, S. Kraemer, H. Scharwaechter, H. Meyr, G. Ascheid, and R. Leupers, "Instruction set customization of application specific processors for network processing: a case study," in *16th IEEE International Conference on Application-Specific Systems, Architecture Processors*, pp. 154–160, 23-25 July 2005.

[41] G. Ascia, V. Catania, M. Palesi, and D. Patti, "A system-level framework for evaluating area/performance/power trade-offs of VLIW-based embedded systems," in *Proceedings of the Asia and South Pacific Design Automation Conference*, vol. 2, pp. 940–943Vol.2, 18-21 Jan. 2005.

[42] D. Kudithipudi, S. Petko, and E. John, "Cache leakage power analysis in embedded applications," in *Proceedings of the 47th Midwest Symposium on Circuits and Systems*, vol. 2, pp. II–517–II–520vol.2, 25-28 July 2004.

[43] H. Hanson, M. Hrishikesh, V. Agarwal, S. Keckler, and D. Burger, "Static energy reduction techniques for microprocessor caches," in *Proceedings of the International Conference on Computer Design*, pp. 276–283, 23-26 Sept. 2001.

[44] M. Mamidipaka, K. Khouri, N. Dutt, and M. Abadir, "Analytical models for leakage power estimation of memory array structures," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pp. 146–151, 2004.

[45] H. Zhou, M. Toburen, E. Rotenberg, and T. Conte, "Adaptive mode control: a static-power-efficient cache design," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 61–70, 8-12 Sept. 2001.

[46] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 148–157, 25-29 May 2002.

[47] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge, "Drowsy instruction caches. Leakage power reduction using dynamic voltage scaling and cache sub-bank prediction," in *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 219–230, 18-22 Nov. 2002.

[48] N. Kim, K. Flautner, D. Blaauw, and T. Mudge, "Single-V/sub DD/ and single-V/sub T/ super-drowsy techniques for low-leakage high-performance instruction caches," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 54–57, 9-11 Aug. 2004.

[49] J.-J. Li and Y.-S. Hwang, "Snug set-associative caches. Reducing leakage power while improving performance," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 345–350, 8-10 Aug. 2005.

[50] Y. Guo, S. Chheda, I. Koren, M. Krishna, and C. Moritz, "Energy characterization of hardware-based data prefetching," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 518–523, 11-13 Oct. 2004.

[51] J. Halter and F. Najm, "A gate-level leakage power reduction method for ultra-low-power CMOS circuits," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 475–478, 5-8 May 1997.

[52] A. Abdollahi, F. Fallah, and M. Pedram, "Leakage current reduction in CMOS VLSI circuits by input vector control," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, pp. 140–154, Feb. 2004.

[53] X. Chang, D. Fan, Y. Han, and Z. Zhang, "SoC leakage power reduction algorithm by input vector control," in *Proceedings of the International Symposium on System-on-Chip*, pp. 86–89, 15-17 Nov. 2005.

[54] S. Mutoh, T. Douseki, Y. Matsuya, T. Aoki, S. Shigematsu, and J. Yamada, "1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS," *IEEE Journal of Solid-State Circuits*, vol. 30, pp. 847–854, Aug. 1995.

[55] W. Zhang, N. Vijaykrishnan, M. Kandemir, M. Irwin, D. Duarte, and Y.-F. Tsai, "Exploiting VLIW schedule slacks for dynamic and leakage energy reduction," in *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, pp. 102–113, 1-5 Dec. 2001.

[56] W. Zhang, Y.-F. Tsai, M. Kandemir, N. Vijaykrishnan, M. Irwin, and V. De, "Leakage-aware compilation for VLIW architectures," in *IEE Proceedings of Computers and Digital Techniques*, vol. 152, pp. 251–260, Mar 2005.

[57] Y. Cao and H. Yasuura, "Reducing dynamic power and leakage power for embedded systems," in *Proceedings of the 15th Annual IEEE International ASIC/SOC Conference*, pp. 291–295, 25-28 Sept. 2002.

[58] W. Liao, J. Basile, and L. He, "Leakage power modeling and reduction with data retention," in *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pp. 714–719, 10-14 Nov. 2002.

[59] W. Elgharbawy and M. Bayoumi, "Leakage sources and possible solutions in nanometer CMOS technologies," *IEEE Circuits and Systems Magazine*, vol. 5, pp. 6–17, Fourth Quarter 2005.

[60] Artisan Components, Inc., *TSMC 0.13 μm (CL013G) Process 1.2-Volt SAGE-X™ Standard Cell Library Databook*, 2.6 ed., January 2004.

[61] Synopsys, Inc., *Using Tcl With Synopsys Tools*, version V-2004.06 ed., June 2004.

[62] Synopsys, Inc., *HDL Compiler for VHDL Reference Manual*, version W-2004.12 ed., December 2004.

[63] J. Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC v8 architecture," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 409–415, 23-26 June 2002.

[64] ARM, Inc., *AMBA specification rev. 2.0*, 1999.

[65] SPARC International, *The SPARC Architecture Manual Version 8*. Prentice Hall, 1992.

[66] J. Gaisler, *The LEON-2 Processor User's Manual*. Gaisler Research, version 1.0.30 ed., May 2004.

[67] J. Grad, "The SOCks design platform," tech. rep., Illinois Institute of Technology, 2004.

[68] J. Grad, "System-On-Chip design with the Leon CPU," tech. rep., Illinois Institute of Technology, 2004.

[69] Synopsys, Inc., *Power Compiler User Guide*, version Z-2007.03 ed., March 2007.

[70] Artisan Components, Inc., *TSMC 0.13 μm Process High-Speed Synchronous Single-Port SRAM Memory Generator User Manual*, release 1.0 ed., April 2002.

[71] OpenCores.org, *OpenRISC 1000 Architecture Manual*, rev. 1.3 ed., April 2006.

[72] OpenCores.org, *OpenRISC 1200 IP Core Specification*, rev. 0.7 ed., September 2001.

[73] OpenCores.org, *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, revision b.3 ed., September 2002.

[74] L. Chunho, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the 13th International Symposium on Microarchitecture*, pp. 330–335, 1997.

[75] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization*, pp. 3–14, 2 Dec. 2001.

[76] CodeSourcery, Inc., *Sourcery G++ Lite ARM EABI*, version 2006q3-27 ed., October 2006.

[77] ARM, Inc., *Application Binary Interface for the ARM Architecture*, version 2.0 ed., March 2005.

[78] Synopsys, Inc., *DW_mult_pipe designware stallable pipelined multiplier datasheet*, dwf_0206 ed., June 2006.

[79] Synopsys, Inc., *DesignWare IP family reference guide*, March 2007.

[80] J. Stokes, *Inside The Machine: An Illustrated Introduction to Microprocessors and Computer Architecture*. No Starch Press, Inc., 2006.

[81] M. Donno, A. Ivaldi, L. Benini, and E. Macii, "Clock-tree power optimization based on rtl clock-gating," in *Proceedings of the 40th Design Automation Conference*, pp. 622–627, 2-6 June 2003.

[82] Synopsys, Inc., *Design Compiler User Guide*, version Z-2007.03 ed., March 2007.

[83] J. Schutz, "A 3.3V 0.6-μm & BiCMOS superscalar microprocessor," in *Digest of Technical Papers of the 41st IEEE International Solid-State Circuits Conference*, pp. 202–203, 16-18 Feb. 1994.

[84] T. Yamada, M. Abe, Y. Nitta, K. Ogura, M. Kusaoke, M. Ishikawa, M. Ozawa, K. Takada, F. Arakawa, O. Nishii, and T. Hattori, "Low-power design of 90-nm SuperH™ processor core," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 258–263, 2-5 Oct. 2005.

[85] L. Benini, P. Siegel, and G. De Micheli, "Saving power by synthesizing gated clocks for sequential circuits," *IEEE Design & Test of Computers*, vol. 11, pp. 32–41, Winter 1994.

[86] L. Benini and G. De Micheli, "Automatic synthesis of low-power gated-clock finite-state machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, pp. 630–643, June 1996.

[87] Artisan Components, Inc., *TSMC 90nm (CL013G) Process SAGE-X™ v3.0 Standard Cell Library Databook*, 1.1 ed., March 2005.

[88] K. C. Pokhrel, "Physical and silicon measures of low power clock gating success: an apple to apple case study," in *Synopsys Users Group Forum*, 2007.

[89] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi, *Low Power Methodology Manual for System-on-Chip Design*. Springer, 2007.

[90] A. Chandrakasan, W. Bowhil, and F. Fox, *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.

[91] L. Wei, Z. Chen, M. Johnson, K. Roy, and V. De, "Design and optimization of low voltage high performance dual threshold CMOS circuits," in *Proceedings of the 35th Design Automation Conference*, pp. 489–494, 15–19 Jun 1998.

[92] S. Narendra, S. Borkar, V. De, D. Antoniadis, and A. Chandrakasan, "Scaling of stack effect and its application for leakage reduction," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 195–200, 2001.

[93] A. Orpaz and S. Weiss, "A study of CodePack: optimizing embedded code space," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pp. 103–108, 2002.

[94] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 81–91, 1992.

[95] S.-J. Nam, I.-C. Park, and C.-M. Kyung, "Improving dictionary-based code compression in VLIW architectures," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E82-A, no. 11, pp. 2318–2324, 1999.

[96] H. Lekatsas and W. Wolf, "SAMC: a code compression algorithm for embedded processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 12, pp. 1689–1701, 1999. 0278-0070.

[97] M. Schmitz, B. Al-Hashimi, and P. Eles, "A co-design methodology for energy-efficient multi-mode embedded systems with consideration of mode execution probabilities," in *Design, Automation and Test in Europe Conference and Exhibition*, pp. 960–965, 2003.

[98] C. Piguet, P. Volet, J. M. Masgonty, F. Rampogna, and P. Marchal, "Code memory compression with online decompression," in *Proceedings of the 27th European Solid-State Circuits Conference*, pp. 389–392, 2001.

[99] Synopsys, Inc., *Recommended Astro Script-Based Methodology Application Note*, version X-2005.09 ed., December 2005.

[100] Synopsys, Inc., *JupiterXT Virtual Flat Flow User Guide*, version Y-2006.06 ed., December 2006.

[101] Synopsys, Inc., *Milkyway Environment Data Preparation User Guide*, version Y-2006.06 ed., June 2006.

[102] Synopsys, Inc., *Astro User Guide*, version Z-2007.03 ed., March 2007.

[103] Cadence Design Systems, Inc., *Using Encounter RTL Compiler*, version 6.2.2 ed., March 2007.

[104] Cadence Design Systems, Inc., *Encounter User Guide*, version 5.2 ed., December 2005.

[105] H. Nakashima, J. Inoue, K. Okada, and K. Masu, "ULSI interconnect length distribution model considering core utilization," in *Proceedings of the Design, Automation and Test in Europe Conference*, vol. 2, pp. 1210–1215, February 2004.