Norwegian University of
Science and Technology

Low-power microcontroller core

Stein Ove Eriksen

Master of Science in Electronics
Submission date: June 2009
Supervisor: Einar Johan Aas, IET
Co-supervisor: Øyvind Janbu, Energy Micro

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

Low Power Microcontroller Core
Today's "System on Chip" microcontroller solutions are mostly implemented as an RTL (typically VHDL or Verilog) description that is run through synthesis and lay-out. The design must be able to meet tight constraints on power consumption, chip area and timing. It is of major importance that the entire design flow is optimized for low-power to ensure the best results on the final microcontroller system. It is equally important that the microcontroller core itself is optimized with respect to the design flow, to meet all constraints.

This theses focuses on optimizing the dynamic power consumption for an available microcontroller core, the ZPU:

http://www.opencores.org/projects.cgi/web/zpu/overview

The ZPU does not have a low power design focus (but is rather focused on small size in an FPGA), and should hence have room for improvements when it comes to power consumption. Additionally, the ZPU design is relatively small and it should therefore be possible to get a good overview over the microcontroller system during the work of this thesis.

The work can be split into three phases:
Literature study to evaluate existing low-power design techniques for a digital design flow, for example
Design partitioning / system design
Clock gating / data gating
Power consumption evaluation
Run the microcontroller core through synthesis for a selected silicon process
Evaluate the power consumption of the selected microcontroller core both before synthesis (at RTL) and after synthesis.
Power consumption improvements
Evaluate the microcontroller design versus the design techniques found during the literature study
Make improvements to the microcontroller design and the design flow with respect to power consumption, and show how these improvements reduce the dynamic power consumption.
Evaluate the correctness of the power consumption estimates. How close can we expect the results to be compared with the actual silicon?
It is expected that the ZPU core will have weaknesses with respect to power consumption. Describe these fundamental weaknesses, and suggest architectural improvements to ZPU to improve this.
Veileder: Øyvind Janbu (o.janbu@energymicro.com)
Faglærer: Einar Aas (einar.aas@iet.ntnu.no)

Low-power Microcontroller Core

Stein Ove Eriksen

Master's thesis
Department of Electronics and Telecommunications, NTNU

June 23, 2009

Abstract

Energy efficiency in embedded processors is of major importance in order to achieve longer operating time for battery operated devices. In this thesis the energy efficiency of a microcontroller based on the open source ZPU microprocessor is evaluated and improved. The ZPU microprocessor is a zero-operand stack machine originally designed for small size FPGA implementation, but in this thesis the core is synthesized for implementation with a 180nm technology library. Power estimation of the design is done both before and after synthesis in the design flow, and it is shown that power estimates based on RTL simulations (before synthesis) are 35x faster to obtain than power estimates based on gate-level simulations (after synthesis). The RTL estimates deviate from the gate-level estimates by only 15% and can provide faster design cycle iterations without sacrificing too much accuracy. The energy consumption of the ZPU microcontroller is reduced by implementing clock gating in the ZPU core and also implementing a tiny stack cache to reduce stack activity energy consumption. The result of these improvements show a 46% reduction in average power consumption. The ZPU architecture is also compared to the more common MIPS architecture, and the Plasma CPU of MIPS architecture is synthesized and simulated to serve as comparison to the ZPU microcontroller. The results of the comparison with the MIPS architecture shows that the ZPU needs on average 15x as many cycles and 3x as many memory accesses to complete the benchmark programs as the MIPS does.

ENERGY micro

Low Power Microcontroller Core
Today's "System on Chip" microcontroller solutions are mostly implemented as an RTL (typically VHDL or Verilog) description that is run through synthesis and lay-out. The design must be able to meet tight constraints on power consumption, chip area and timing. It is of major importance that the entire design flow is optimized for low-power to ensure the best results on the final microcontroller system. It is equally important that the microcontroller core itself is optimized with respect to the design flow, to meet all constraints.

This theses focuses on optimizing the dynamic power consumption for an available microcontroller core, the ZPU:

http://www.opencores.org/projects.cgi/web/zpu/overview

The ZPU does not have a low power design focus (but is rather focused on small size in an FPGA), and should hence have room for improvements when it comes to power consumption. Additionally, the ZPU design is relatively small and it should therefore be possible to get a good overview over the microcontroller system during the work of this thesis.

The work can be split into three phases:

- Literature study to evaluate existing low-power design techniques for a digital design flow, for example
  - Design partitioning / system design
  - Clock gating / data gating
- Power consumption evaluation
  - Run the microcontroller core through synthesis for a selected silicon process
  - Evaluate the power consumption of the selected microcontroller core both before synthesis (at RTL) and after synthesis.
- Power consumption improvements
  - Evaluate the microcontroller design versus the design techniques found during the literature study
  - Make improvements to the microcontroller design and the design flow with respect to power consumption, and show how these improvements reduce the dynamic power consumption.
  - Evaluate the correctness of the power consumption estimates. How close can we expect the results to be compared with the actual silicon?
  - It is expected that the ZPU core will have weaknesses with respect to power consumption. Describe these fundamental weaknesses, and suggest architectural improvements to ZPU to improve this.

Veileder: Øyvind Janbu (o.janbu@energymicro.com)
Faglærer: Einar Aas (einar.aas@iet.ntnu.no)

Preface

This Master's thesis in electrical engineering has been written at NTNU spring/-summer 2009 as a continuation of my project work autumn 2008. The assignment was given by Energy Micro in Oslo and involves power estimation and energy consumption improvements in a microcontroller system.

During this work I have spent much time on studying microprocessor design and low power design methodology. All the small steps and processes needed to achieve the end results and conclusions have also been time consuming, for instance configuring cross compilers, or getting design tools by different manufacturers to cooperate with each other. These kinds of problems are seldom described in detail or solved in user guides or literature, and they often stall work progress when they occur. Making the workflow of a design process mostly glitch free and automated has also been something I have striven for throughout this work. Scripting and automation is time consuming at first, but the pay-off is tremendous in the long run, and it boosts designer productivity.

I would like to thank Professor Einar Aas at NTNU and Øyvind Janbu at Energy Micro for invaluable guidance during this whole last year. I would also like to thank my fellow students for great collaboration and lots of fun. And last but not least thanks to my family and Mari for everything else.

-Stein Ove Eriksen
Trondheim, 23.06.2009

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem description . . . 1
1.3 Report structure . . . 2

2 Embedded systems power consumption **3**
2.1 Programmable processors vs hardwired ASICs . . . 3
2.2 Energy efficiency in programmable processors . . . 4

3 Low power design theory **5**
3.1 Power dissipation in CMOS technology . . . 5
3.2 Power and energy definitions . . . 5
3.3 Dynamic power consumption . . . 6
3.4 Static Power consumption . . . 8

4 Low power design techniques **9**
4.1 Clock gating . . . 9
4.2 Power gating . . . 9
4.3 Multi-Voltage design . . . 12
4.4 Multi-V_t design . . . 13

5 Microprocessor architecture **15**
5.1 General computer organization theory . . . 15
5.1.1 The Processor . . . 16
5.1.2 Pipelining . . . 16
5.2 The ZPU microprocessor . . . 17
5.3 The Plasma MIPS microprocessor . . . 18

6 Microcontroller configurations **21**
6.1 Configuration 1: ZPU original microcontroller . . . 21
6.2 Configuration 2: Plasma MIPS microcontroller . . . 22
6.3 Configuration 3: Improved ZPU microcontroller . . . 22

7 Synthesis **25**
7.1 Artisan Sage-X 0.180μm technology library . . . 25
7.2 Synopsys synthesis tools . . . 26
7.2.1 Library Compiler . . . 26
7.2.2 Design Compiler . . . 27
7.2.3 Power Compiler . . . 28
7.3 Configuration 1 core synthesis results . . . 29
7.3.1 VHDL-code modifications . . . 29
7.4 Configuration 2 core synthesis results . . . 31
7.5 Configuration 3 core synthesis results . . . 32
7.6 SRAM memory synthesis results . . . 33

8 Simulation and power estimation **35**
8.1 Benchmarks . . . 35
8.1.1 Dhrystone . . . 35
8.1.2 AES-128 . . . 36
8.1.3 Pi approximation . . . 36
8.1.4 While(1) spinlock . . . 36
8.2 Compilers . . . 36
8.2.1 GCC for ZPU . . . 37
8.2.2 GCC for MIPS . . . 37
8.3 Simulation and power estimation . . . 38
8.3.1 Configuration 1 simulation results . . . 38
8.3.2 Configuration 2 simulation results . . . 45
8.3.3 Configuration 3 simulation results . . . 48

9 Estimation method evaluation **53**
9.1 RTL vs gate-level power estimation accuracy . . . 53
9.2 RTL vs gate-level power estimation speed . . . 56
9.3 Gate-level power estimation vs actual silicon chip power consumption 56

10 Evaluation of ZPU design **59**
10.1 Comparison with MIPS architecture . . . 59
10.2 ZPU power weaknesses . . . 59

11 Energy consumption improvements to ZPU microcontroller design **63**
11.1 Memory improvements . . . 64
11.2 Control . . . 66

12 Discussion **69**
12.1 Power estimation accuracy and speed . . . 69
12.2 ZPU architectural improvement potential . . . 69
12.3 Compiler considerations . . . 70
12.4 Implemented ZPU microcontroller improvements . . . 70

13 Conclusion **71**

14 Further work 73

A Design-flow scripts and programs 77
A.1 Modelsim simulation scripts . . . 77
A.1.1 ZPU simulation script . . . 77
A.1.2 Plasma CPU simulation script . . . 77
A.2 Synopsys synthesis and power estimation scripts . . . 78
A.2.1 Library Compiler script . . . 78
A.2.2 Synthesis with Design Compiler scripts . . . 78
A.2.3 Power estimation with Power Compiler scripts . . . 79
A.3 Stack cache memory iterator . . . 79

B VHDL code 81
B.1 ZPU core . . . 81
B.2 ZPU memory module . . . 92
B.3 ZPU testbench . . . 93
B.4 Plasma CPU core . . . 95
B.5 Plasma memory module . . . 99
B.6 Plasma test bench . . . 102

List of Abbreviations

.saif	switching activity interchange format
AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
CMOS	Complementary metal–oxide–semiconductor
CPU	Central Processing Unit
DMIPS	Dhrystone loops per second divided by 1757
FPGA	Field-Programmable Gate Array
GCC	GNU Compiler Collection
I/O	Input/Output
ISA	Instruction Set Architecture
MCU	Microcontroller Unit
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SRAM	Static Random Access Memory
UART	Asynchronous Reciever/Transmitter

Chapter 1

Introduction

1.1 Motivation

A microcontroller unit (MCU) is a system-on-chip with a processing unit (CPU), memory (RAM) and peripheral devices (I/O) all in the same IC-package. The market for microcontrollers is huge and they are found in everything everywhere from cars and industrial motor controllers to pacemakers and smoke detectors. In many of these applications the microcontroller system is battery powered. The operating time of the system then relies on the battery capacity and energy consumption of the MCU. It is of major importance to minimize the energy consumption of the MCU in such a system to improve the life-time for a given battery capacity.

1.2 Problem description

In this Master's thesis an open-source microcontroller core, the ZPU, is synthesized and evaluated with respect to dynamic power consumption. Improvements to lower the energy consumption of the ZPU microcontroller are also implemented and shown to actually reduce the energy consumption of the system. Another main goal of this work has been to establish an efficient and automated low-power oriented design flow to improve designer productivity.

Three microcontroller configurations are described in this thesis: Configuration 1 is the original ZPU microcontroller, configuration 2 is a MIPS-based microcontroller and configuration 3 is the improved ZPU microcontroller. Configuration 1 and 3 implementations, simulations and power estimations are direct answers to the assignment given for this thesis. Configuration 2 with its MIPS architecture is introduced to serve as a comparison to the ZPU microcontrollers.

1.3 Report structure

A brief overview of this thesis can be given as:

- In chapter 2 the energy consumption distribution of embedded systems are described.
- Chapter 3 contains key concepts for CMOS technology.
- In chapter 4 low-power design techniques are described.
- Chapter 5 gives an overview of general microprocessor design and presents the two architectures used in this thesis.
- Chapter 6 presents the three microcontroller configurations that are synthesized and simulated in this thesis; the ZPU and the Plasma.
- In chapter 7 the synthesis process and results of the three microcontroller configurations are presented.
- In chapter 8 the results of simulation an power estimation of the three microcontroller configurations are presented.
- In chapter 9 the power estimation methods are evaluated.
- In chapter 10 the ZPU architecture is discussed and compared with MIPS architecture.
- In chapter 11 improvements are implemented to the ZPU microcontroller to reduce the energy consumption of the system.
- Chapter 12 contains a discussion of the main results in this thesis.
- Chapter 13 concludes this thesis.
- Chapter 14 contains suggestions to further work.

Chapter 2

Embedded systems power consumption

2.1 Programmable processors vs hardwired ASICs

General programmable processors (CPUs) are due to their programmability far more flexible than application specific integrated circuit (ASIC) implementations, though this flexibility comes with a cost. An ASIC spends all of its total energy on the specific arithmetics or algorithms it is hardwired to perform, making it highly energy efficient within its limited capabilities. CPUs on the other hand, can perform virtually any algorithm with its instruction set architecture (ISA) once the algorithm is compiled to a sequence of data and instructions. This sequence of data and instructions is stored in system memory, and needs to be fetched from memory to be executed by the ISA in the processor core. This implies that to perform the same algorithms an ASIC is hardwired to do, a CPU has a large energy consumption overhead caused by fetching instructions and data from memory [1].

Other comparable metrics are development cost and time. ASICs meet the energy demands of embedded applications, but they typically have two year design schedules with a typical cost of 20 million dollars or more. This makes them economically feasible only for high volume applications. During this relatively long development period it is also a challenge to keep up with the development on the algorithms, protocols and codecs the ASIC needs to work with when it is to be released into market [1].

Closing the gap between ASIC and CPU energy efficiency is therefore highly favorable when developing embedded systems for an ever changing market. In the next section it is described where the energy is spent during execution of software in programmable processors, and suggestions on how energy consumption can be reduced is given.

2.2 Energy efficiency in programmable processors

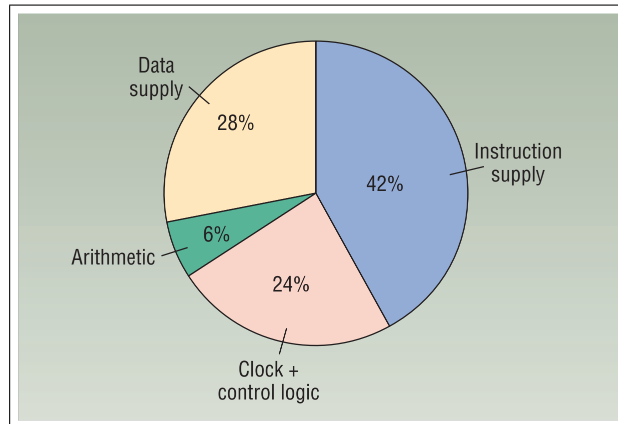


Figure 2.1: Embedded processor efficiency. 70 percent of the processor's energy is consumed by instruction and data supply. Control and arithmetics makes up for the remaining 30 percent [1].

Embedded processors spend most of their energy on fetching instructions and data from memory as shown in figure 2.1. Dally et. al. [1] shows that 70 percent of the energy is consumed by memory access, with the remaining 30 percent consumed by control logic, clock distribution and arithmetics. Because of additional overhead on the arithmetics such as updating loop indices and calculating memory addresses, only 59 percent of the total arithmetic energy is spent on useful arithmetics. The energy consumption by the useful arithmetics in the CPU is comparable to the ASIC hardware implementation, but only makes up 3.5 percent of the total CPU energy. This large energy overhead by performing arithmetics in CPUs is caused by the the way it supplies data and instructions to the arithmetic units in the ISA. The processor needs to spend 119pJ on instruction and data fetching to control a 10pJ arithmetic operation, and only 59 percent of these operations are useful ones [1].

The energy consumed by instruction and data supply from memory ranges from 15 to 50 times the energy of actually performing the arithmetic instruction in the ISA. Reducing the energy consumed by the memory subsystem is because of this of great importance in order to close the gap between ASIC and CPU energy efficiency.

Chapter 3

Low power design theory

3.1 Power dissipation in CMOS technology

Power dissipation in CMOS technology has three sources: $P_{switching}$, $P_{short-circuit}$ and P_{static}. The total power dissipation P_{avg} is the sum of these three components as stated in the equation:

$$\begin{aligned} P_{avg} &= P_{switching} + P_{short-circuit} + P_{static} \\ &= (\alpha_{0\to1} f_{clock} C_L V_{dd}^2) + (t_{sc} I_{max} f_{clk} V_{dd}) + (I_{leakage} V_{dd}), \end{aligned} \tag{3.1}$$

where V_{dd} is the supply voltage [2] [3]. $P_{switching}$ is the power required to charge and recharge the output capacitance on a gate. $P_{short-circuit}$ is caused by short-circuit currents when signal transitions occur. P_{static} is the power contribution from leakage currents in the transistors. The power components are further explained in sections 3.3 and 3.4.

The *delay* through a CMOS gate can be modeled with a first-order derivation given by:

$$T_d = \frac{C_L V_{dd}}{I} = \frac{C_L V_{dd}}{k(V_{dd} - V_t)^2}, \tag{3.2}$$

where T_d is the delay and k is a technology dependent constant [2].

It follows from 3.1 and 3.2 that lowering the supply voltage reduces the total power consumption and also increases transition delay through the gates in a circuit. This implies that there has to be a trade-off between power consumption and performance for a given design in general.

3.2 Power and energy definitions

Power and energy are interconnected terms. *Power* is the instantaneous power dissipated in a circuit. The *energy* required to complete a certain task is the integral

of the power function over time:

$$Energy = \int Power\,dt \tag{3.3}$$

Figure 3.1 shows the same task running at two different power levels, where the low-power approach requires longer time to complete, but consumes the same amount of energy as the high-power approach.

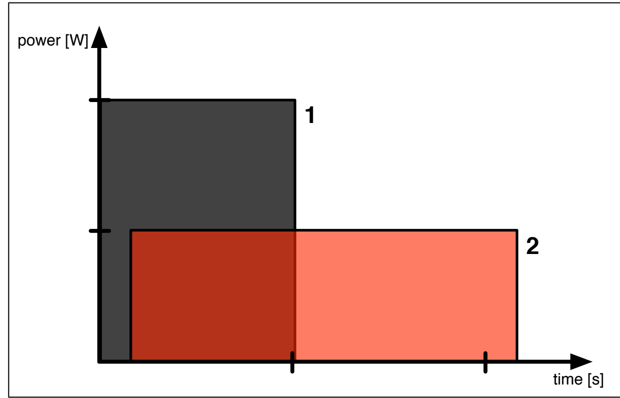


Figure 3.1: The low-power (2) approach could just be slower than the high-power approach (1) while both requires the the same amount of energy.

The correlation between power consumption and computation speed indicates a relation between low-power and high-performance design techniques. The work in this report is concentrated on using the techniques available to save energy. For battery operated devices, the battery-life is directly connected to the energy consumption as batteries holds only a finite amount of energy.

3.3 Dynamic power consumption

The dynamic power consumption is the power dissipated in a circuit when it is in an active state, which means internal signals are changing values. Dynamic power consists of the two components $P_{switching}$ and $P_{short-circuit}$ from equation 3.1.

Switching power

The switching power component $P_{switching}$ is the power required to charge and recharge the output capacitance on a gate during signal transitions as shown in figure 3.2. It is the main contributor to the dynamic power and is given by the following equation:

$$P_{switching} = \alpha_{0\rightarrow 1} f_{clock} C_L V_{dd}^2, \tag{3.4}$$

where $\alpha_{0\rightarrow 1}$ is the number of transitions from 0 to 1 per clock cycle, f_{clock} is the clock frequency, C_L is the output capacitance and V_{dd} is the supply voltage [2].

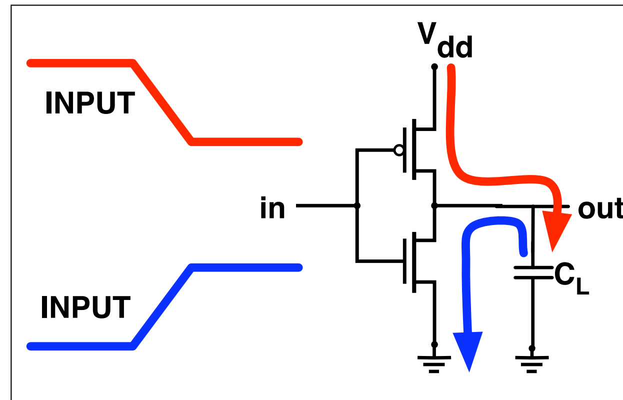


Figure 3.2: Switching power in a CMOS inverter.

Short-circuit power

The short-circuit power component $P_{short-circuit}$ is caused by the NMOS and PMOS transistors both conducting a current through the channels during signal transitions as shown in figure 3.3. Because of this a temporary current $I_{short-circuit}$ flows from V_{dd} to ground during input transitions. The expression for $P_{short-circuit}$ can be written as:

$$P_{short-circuit} = t_{sc} I_{max} f_{clk} V_{dd}, \tag{3.5}$$

where t_{sc} is the time duration of the short-circuit current, I_{max} is max internal switching current, f_{clk} is clock frequency and V_{dd} is the supply voltage. The duration t_{sc} will depend on the transition time of the input signals, and as long as the transition time is kept low, the dynamic power dissipation will be dominated by the switching power $P_{switching}$ [3].

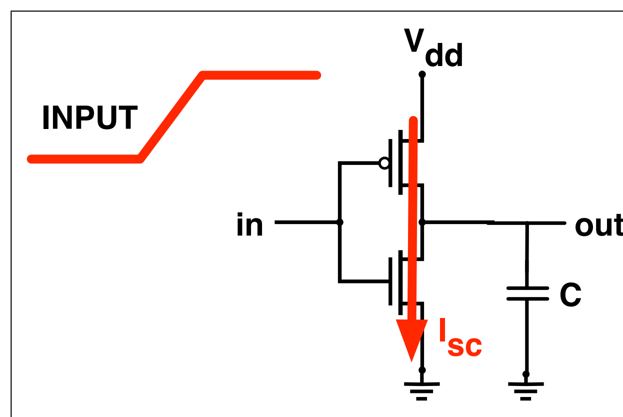


Figure 3.3: Short-circuit power in CMOS, caused by the NMOS and PMOS transistor both being partially open during signal transition.

3.4 Static Power consumption

The static power dissipation is due to leakage currents in CMOS transistors, and increases as the transistor dimension and threshold voltage decreases. The three main components are sub-threshold leakage, gate tunneling leakage, and drain diode leakage currents as shown in figure 3.4 [4].

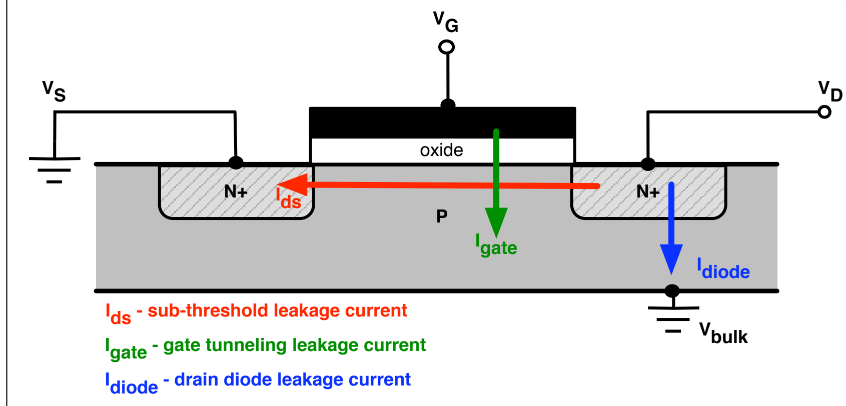


Figure 3.4: Three sources of static power consumption in CMOS

.

The *sub-threshold leakage current* can be approximated by

$$I_{sub} = \mu C_{ox} V_{th}^2 \frac{W}{L} e^{\frac{V_{GS}-V_T}{nV_{th}}}, \tag{3.6}$$

where W and L are transistor dimensions, V_T is threshold voltage, V_{th} is the thermal voltage and μ, C_{ox} and n are fabrication process parameters.

This means that scaling V_{dd} and V_T down to limit dynamic power will make leakage power exponentially worse, as the leakage current I_{sub} depends exponentially on the difference between V_{GS} and V_T.

The oxide thickness for 90nm technology is so small that the *gate tunneling leakage current* is nearly 1/3 of I_{sub}, but the sub-threshold current remains the main contributer to static power dissipation [3].

Chapter 4

Low power design techniques

4.1 Clock gating

As 50% or more of the total dynamic power can be dissipated in clock buffers, an effective method for decreasing the dynamic power consumption is to disable the clock of a system component when not in use [3]. Clock gating decreases the switching activity in the *clock tree*, *flip-flops* and in the *fanout gates* of the flip-flops in a circuit, and decreases by equation 3.4 the switching power $P_{switching}$. This way clock gating also serves as data gating for clouds of combinatorial logic with registers at the input [5].

Clock gating is implemented with the synthesis tool as shown in figure 4.1, where the HDL code description is compiled with and without clock gating cells. This approach requires no change to the HDL code. In [3], it is referred to a power reduction project by K. Pokhrel in 180nm technology where power savings of 34% to 43% are achieved on parts of a system with clock gating. Pokhrel finds that clock gating on one-bit registers is not power efficient and uses clock gating only on registers with a bit-width of three or more.

Clock gating theory is further described in [5], chapter 13, and implementation with Synopsys tools is described in [6].

4.2 Power gating

The static power component P_{static} contributes more and more to the total power dissipation of a circuit for each generation of CMOS technology. *Power gating techniques* are about powering down blocks of the design when not in use, and by this reducing the static power dissipation caused by leakage currents. The use of different power modes such as an active mode and a sleep mode is essential for battery operated devices [3].

A sleep mode implementation as in figure 4.2 has SLEEP and WAKE events to initiate entry to sleep mode S and return to active mode A. The figure shows the power consumption for a sub-system with the sleep mode implemented in different ways. First, (1) has clock gating only, where the leakage power is equal for both

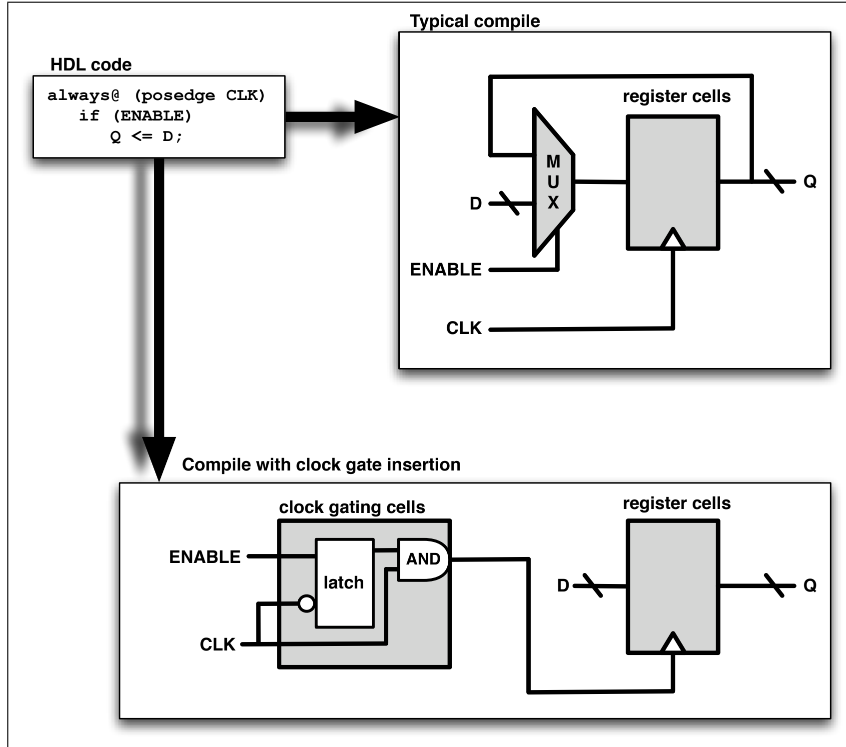


Figure 4.1: Compile of design without and with clock gate insertion.

active mode and sleep mode. In (2) a simplyfied model for sleep mode entry/return is used, with instant leakage reduction on the SLEEP event and a time penalty only on the WAKE event. (3) has the most realistic model, showing that the leakage power decreases over some time, not instantly, to reach power saving levels after a SLEEP event.

In addition to the sleep mode entry/exit penalties shown in figure 4.2 (3), one needs to take into account the extra energy required to enter and exit sleep mode, mainly the energy used to store the system register contents on SLEEP and load back the system register contents on WAKE.

Based on the energy and time penalties mentioned above and in figure 4.2, it is possible write an expression for the minimum sleep time to conserve energy. T_{min} is the minimum time the system must stay in sleep mode to save energy on an active→sleep→active mode transition and is given by:

$$T_{min} = \frac{E_{A\to S} + E_{S\to A} - P_A(T_{A\to S} + T_{S\to A})}{P_A - P_s} \tag{4.1}$$

where $E_{A\to S}$ and $E_{S\to A}$ are energy amounts used to store/load on SLEEP and WAKE events, $T_{A\to S}$ and $T_{S\to A}$ are the transition times between active and sleep mode and P_A and P_s are total power dissipated in active and sleep state.

Physical implementation of power gating has two approaches in general, fine

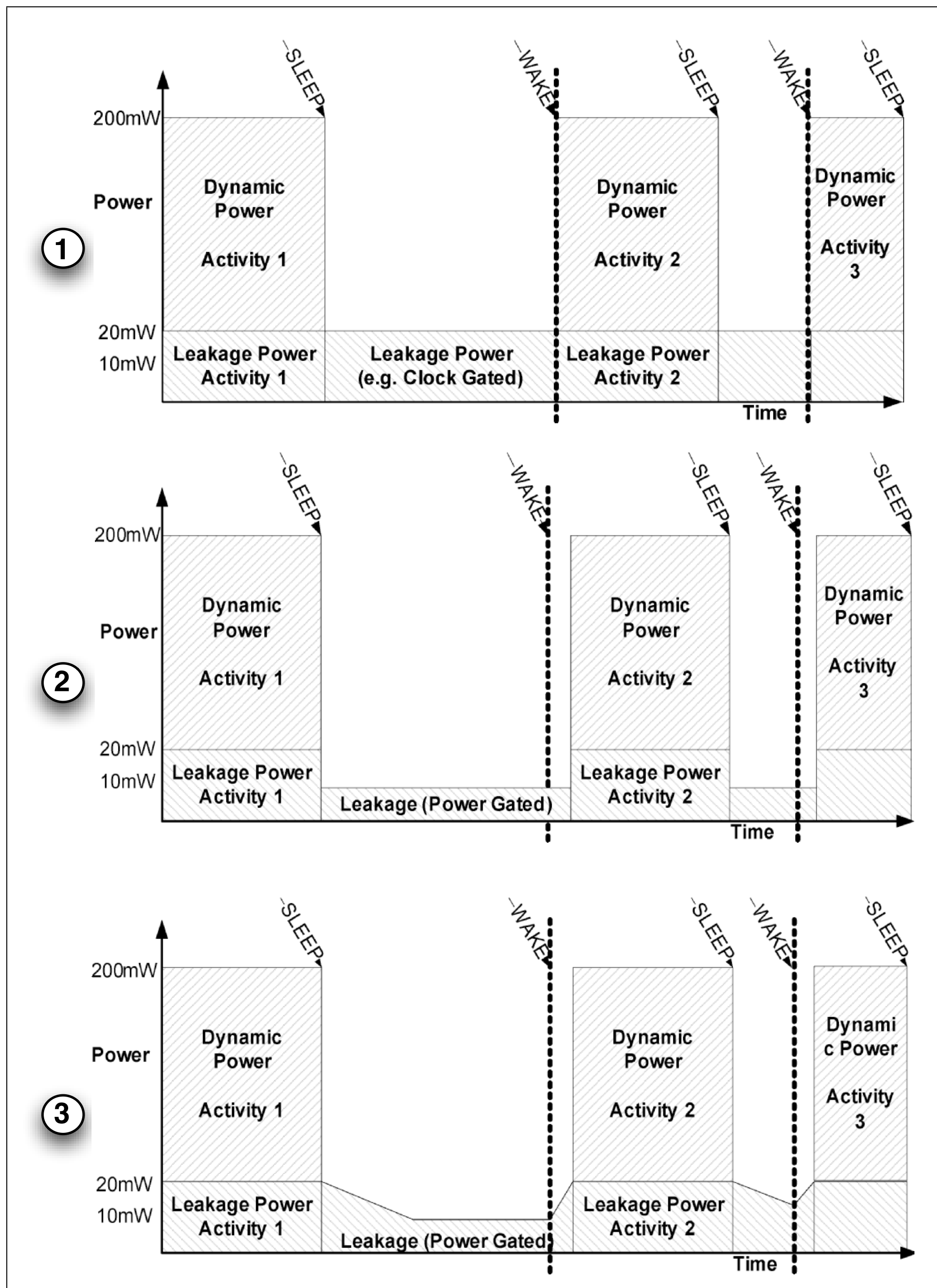


Figure 4.2: Comparison of entry/exit sleep mode in a system with (1): no power gating, (2): ideal power gating and (3): realistic power gating. [3].

grain and coarse grain power gating. These approaches are explained in the two following subsections.

Fine grain power gating

In fine grain power gating power switching transistors are placed inside each cell in the technology library as shown in figure 4.3. High V_T sleep transistors are used to create virtual V_{dd} and virtual ground to the lower V_T main logic transistors [5]. Fine grain power gating can be implemented by changing just the cell library in a traditional design flow, but the approach has a large area overhead of 2x-4x the original cell size. Because the area penalty has not proven worth the savings in design effort, most designs today uses coarse grain power gating [3].

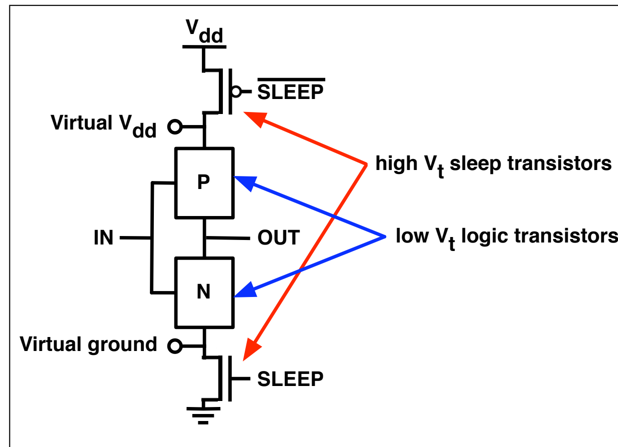


Figure 4.3: Fine grain power gating.

Coarse grain power gating

The coarse grain power gating approach uses a collection of switch cells to switch off the power for a whole block of the system as shown in figure 4.4. The coarse grain switching network is more difficult to implement than the transparent fine grain from a designers point of view, but coarse grain gating has less area overhead and is most frequently used today [3]. Detailed explanation of power gating implementation is given in [3], chapter 5.

4.3 Multi-Voltage design

Different blocks in a SoC design has different performance constraints, and exploiting this by partitioning the design into multiple power domains is called *Multi-voltage design*. Lowering V_{dd} on design blocks that are not critical to performance as in figure 4.5 can decrease total power dissipation without decreasing performance of the system. In figure 4.5, the cache RAM runs at 1.2V, the highest voltage possible because it is the most critical component to performance. The CPU can then run at at 1V while the cache at 1.2V is still the limiting factor, and the rest of the chip can run at 0.9V without impacting the system performance.

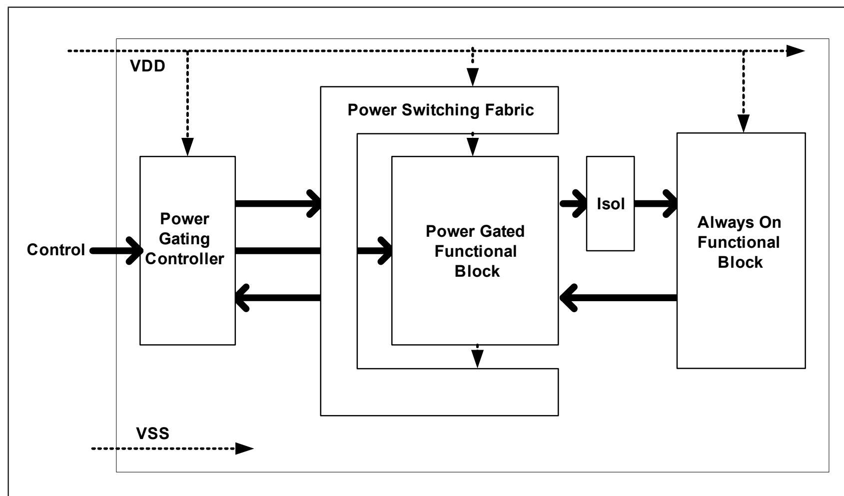


Figure 4.4: Coarse grain power gating. [3]

Multi-voltage designs needs level shifter interfaces between the power domains that causes an area overhead. The level shifters makes timing analysis and floor planning more complex than for a single power domain. Each power domain also needs its own power supply rail, and so the SoC will have to contain voltage regulators[3].

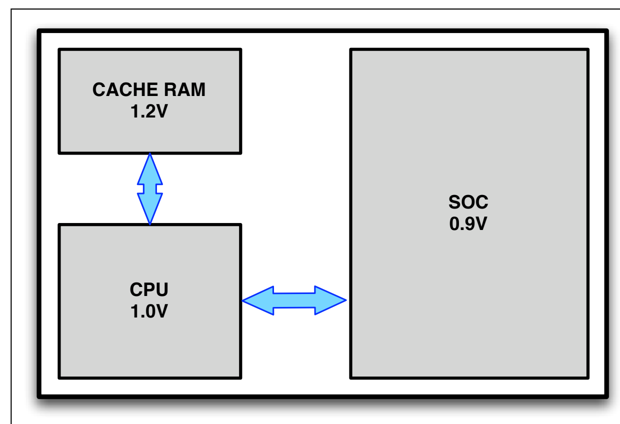


Figure 4.5: Multi-voltage design. [3]

4.4 Multi-V_t design

In CMOS technology sizes 130nm and downward, static power dissipation caused by leakage currents becomes a main contributor to the total power consumption. From equation 3.4 it follows that increasing V_T reduces the sub-threshold leakage, and

from equation 3.2 that gate delay increases with increasing V_T. Figure 4.6 shows the leakage current as a function of gate delay for a 90nm process. Many libraries contain three different cell versions: low-V_T, standard-V_T and high-V_T. In Figure 4.7 the leakage and delay for these three cell types are plotted relative to each other.

Libraries with multiple threshold voltages V_T can be used to optimize timing and reduce static power dissipation for a design. System components with low timing constraints can be implemented with the slower high-V_T cells, while timing critical components can be implemented with the faster low-V_T cells. By using cells with appropriate V_T to the given timing constraint for a system component, static power dissipation can be significally reduced [3].

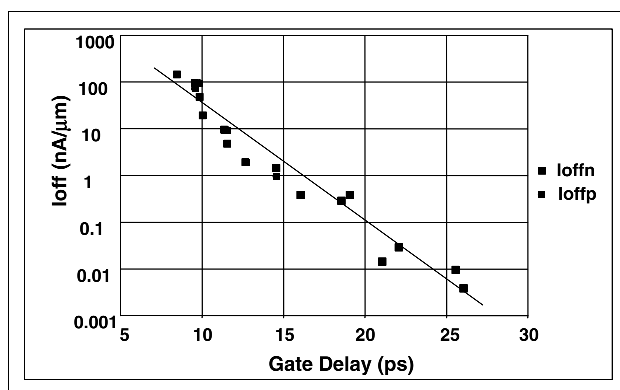


Figure 4.6: Leakage current as a function of gate delay for a 90nm process. [3]

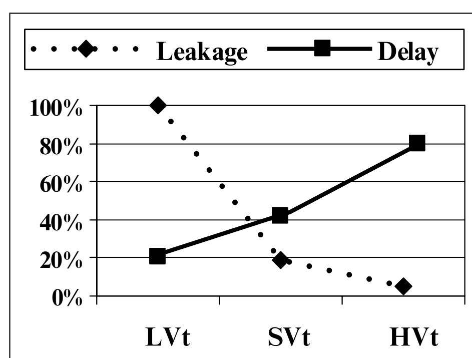


Figure 4.7: Leakage and delay for three different V_T cell types: low-V_T, standard-V_T and high-V_T. [3]

Chapter 5

Microprocessor architecture

5.1 General computer organization theory

Patterson & Hennessy [7] states five classical components of a computer as shown in figure 5.1. Data to be processed and processed data is put into memory by I/O. The control unit controls the data processing in the datapath. Both instructions and data are fetched from memory to the processor.

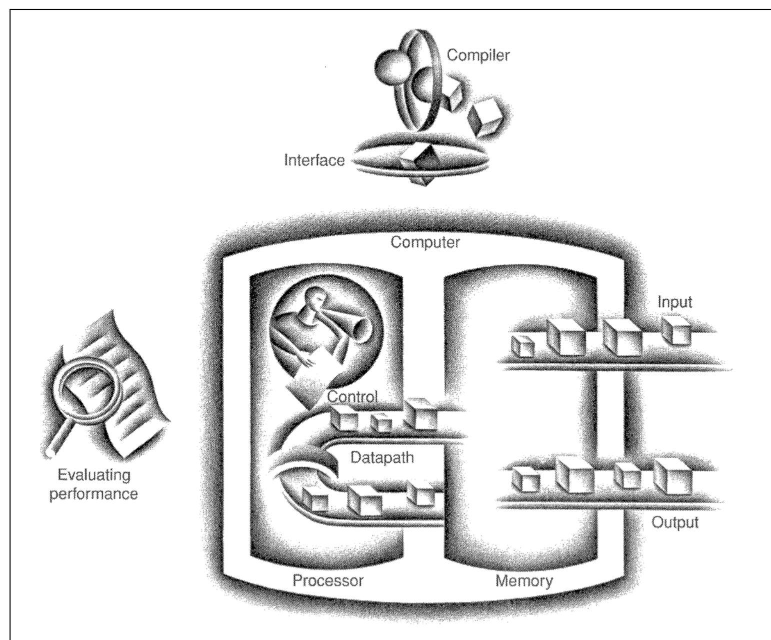


Figure 5.1: The five classic components of a computer [7].

5.1.1 The Processor

This section describes the basic function of the MIPS processor, a common microprocessor architecture. The processor executes the instructions it has implemented in its ISA. Instructions start by supplying the instruction address to the instruction memory from the program counter (PC). After fetching the instruction, the register operands to be used by are specified by decoding the instruction word. Now the register operands can be operated on to compute a memory address (load/store instructions), compute an arithmetic result, or do a compare (for a branch). If the instruction is an arithmetic instruction, the data is fed to the ALU from registers and the result is written back to a register in the register file. The function of the MIPS processor can thus be summarized in five steps:

1. Fetch instruction from memory
2. Read registers while decoding instruction
3. Execute the operation or calculate an address
4. Access an operand in data memory
5. Write the results to a register

For more information see [7], chapter 5.

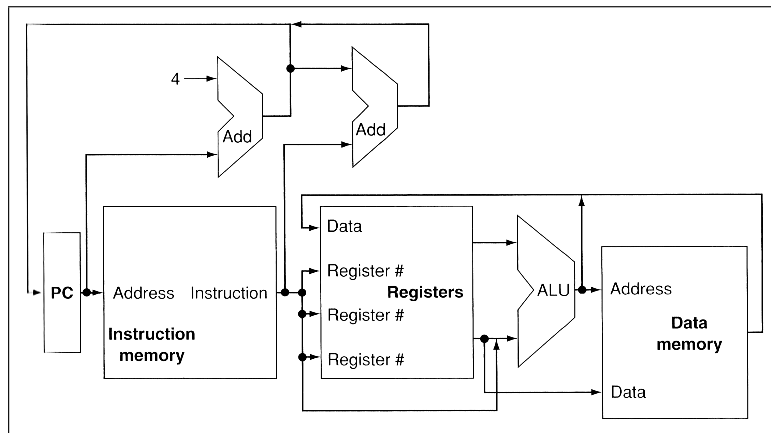


Figure 5.2: Abstract view of the MIPS processor [7].

5.1.2 Pipelining

Pipelining is an implementation technique in which instructions are overlapped in execution. The five-step execution of each instruction described in the previous section is plotted in the top part of figure 5.3. A pipelined version the five-step execution is shown at the bottom of figure 5.3. The pipelining technique provides

performance and energy consumption improvements as the instruction rate is increased by a factor 4 for the example in figure 5.3. Examples of how to improve performance and energy efficiency with pipelining are described in [7] chapter 6.

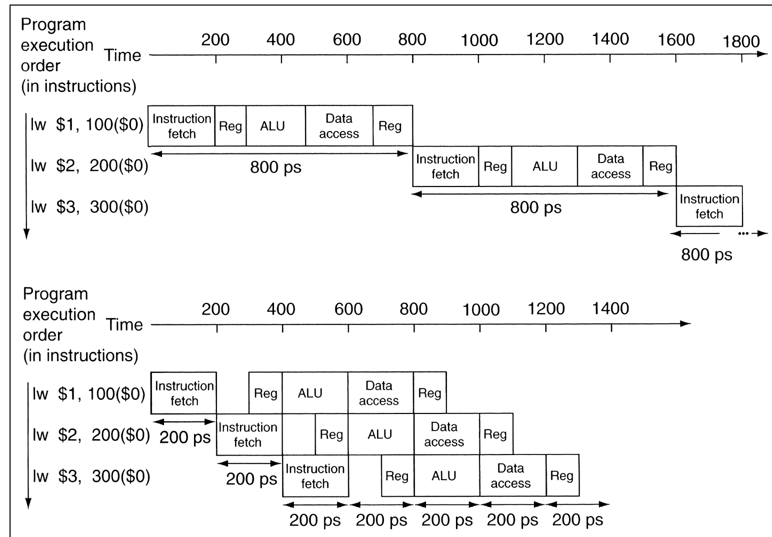


Figure 5.3: Comparing nonpipelined (top) and pipelined (bottom) execution of three load word instructions.

5.2 The ZPU microprocessor

Overview

The ZPU is an open source 32-bit RISC CPU developed by Zylin AS based in Stavanger, Norway, and it also has a development project at OpenCores.org [8]. Originally designed for FPGA implementation, the philosophy behind the ZPU is to take up as little FPGA resources as possible, and leave as much FPGA real estate as possible for other hardware modules [9]. The ZPU is a zero operand CPU with stack computer architecture and no register file. The stack is allocated in main memory and all instructions are performed on the top-of-stack. The ZPU is distributed with a GCC tool-chain, and the source code is written in VHDL. Figure 5.4 shows a block diagram of the ZPU core. The stack-machine architecture has no register file. Further information about the ZPU is found on the ZPU developement web pages [9].

Instruction set architecture

The ZPU instruction set is configurable. A base set of the instructions must be implemented in the ISA, but the rest can be implemented either in the ISA or as microcode (emulated instructions). This is to allow a trade-off of core size vs. code size and performance.

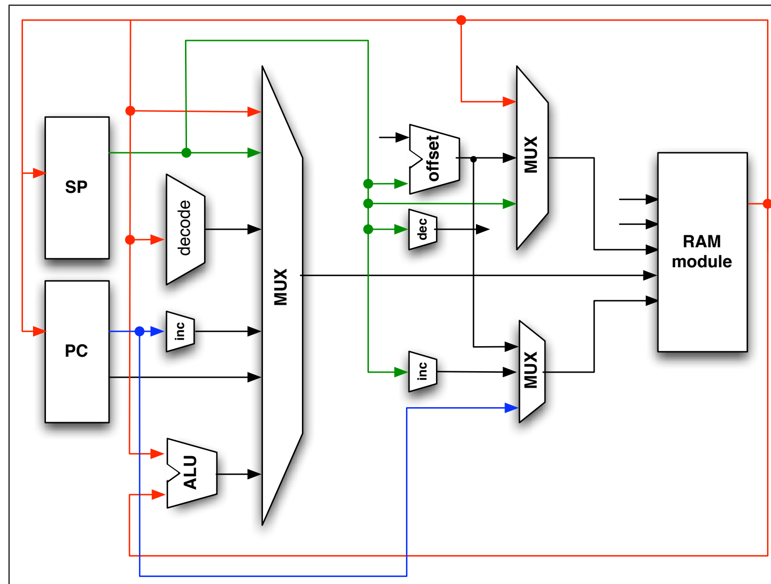


Figure 5.4: ZPU processor block diagram. All instructions are performed on top-of-stack, the stack is located in memory, and the processor has no register file.

5.3 The Plasma MIPS microprocessor

Overview

The Plasma CPU is an open source synthesizable 32-bit RISC microprocessor developed at OpenCores.org. It has MIPS architecture and is serving as a comparison for the ZPU in this thesis. The block diagram of the Plasma core is shown in figure 5.5. Further information about the Plasma processor is found on the Plasma OpenCores.org project web page [13].

Instruction set architecture

The Plasma CPU implements all MIPS-I instructions except unaligned load and store operations. This is due to the patent issued on these instructions.

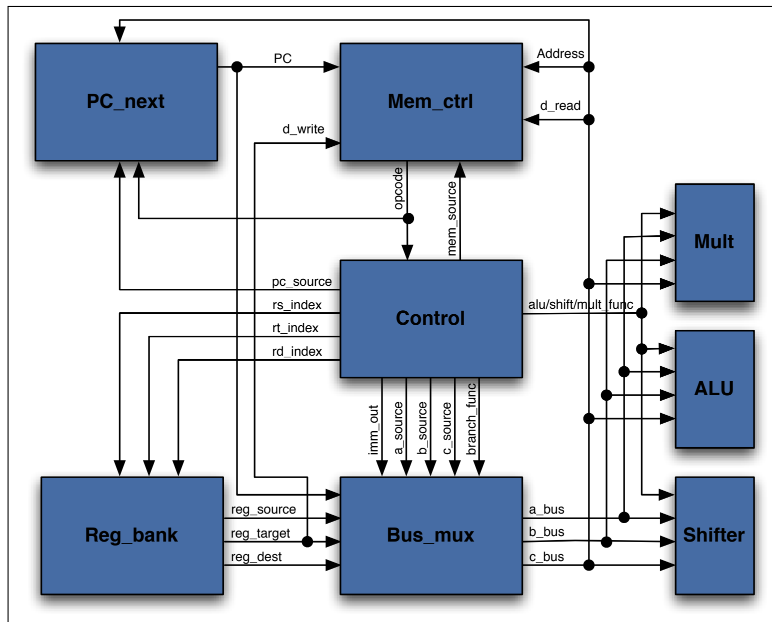


Figure 5.5: Plasma processor block diagram.

Chapter 6

Microcontroller configurations

Three microcontroller configurations will be synthesized and simulated in this thesis: *Configuration 1* is the ZPU core with an on-chip 32kB SRAM memory. A part of the assignment given is to improve the energy efficiency of this design. *Configuration 2* is another CPU core called Plasma with an on-chip 32kB SRAM memory. The Plasma CPU is of MIPS architecture and will serve as a reference versus the ZPU architecture. *Configuration 3* is an improved version of the ZPU microcontroller in Configuration 1, and this configuration will further discussed in chapter 11.

6.1 Configuration 1: ZPU original microcontroller

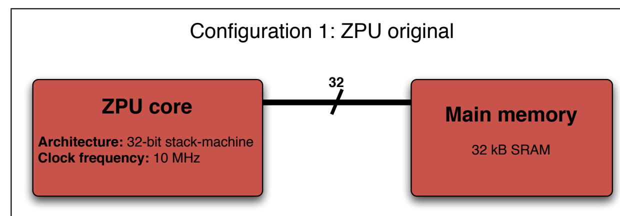


Figure 6.1: Configuration 1: microcontroller with ZPU core and 32 kB SRAM main memory.

Processor core

To trade chip area for low power, the ZPU core in configuration 1 is configured to have the entire instruction set implemented in the ISA.

Memory

Configuration 1 has 32kB of on-chip SRAM memory synthesized by the *TSMC 0.18um High Speed/Density Single-Port SRAM Generator*. Read and write currents for the 32kB SRAM are given in table 7.6.

6.2 Configuration 2: Plasma MIPS microcontroller

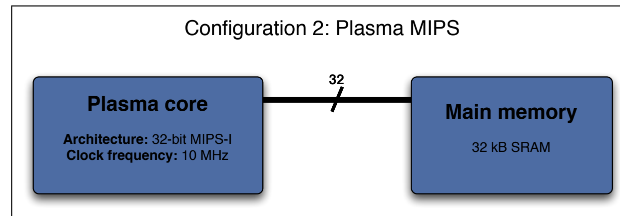


Figure 6.2: Configuration 2: microcontroller with Plasma MIPS core and 32 kB SRAM main memory.

Processor core

Configuration 2 has the Plasma 32-bit MIPS processor implemented as its core.

Memory

Configuration 2 has 32kB of on-chip SRAM memory synthesized by the *TSMC 0.18μm High Speed/Density Single-Port SRAM Generator*. Read and write currents for the 32kB SRAM are given in table 7.6.

6.3 Configuration 3: Improved ZPU microcontroller

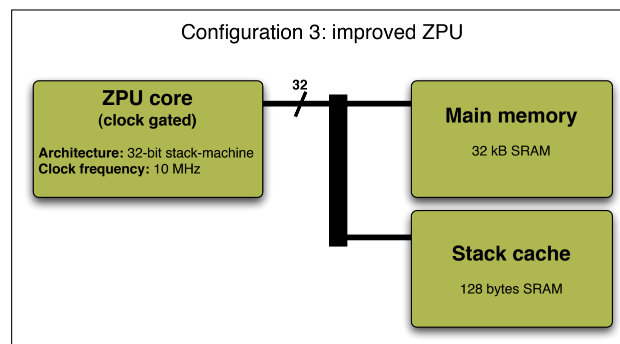


Figure 6.3: Configuration 3: microcontroller with clock gated ZPU core, 32 kB SRAM main memory and 128 bytes SRAM stack cache.

Processor core

To trade chip area for low power, the ZPU core in configuration 3 is configured to have the entire instruction set implemented in the ISA.

Memory

Configuration 3 has 32kB of on-chip SRAM memory and another 128 bytes of on-chip SRAM for stack cache. Both SRAMs are synthesized with the *TSMC 0.18mum High Speed/Density Single-Port SRAM Generator*. Read and write currents for the 32kB SRAM and the 128 bytes SRAM are given in table 7.6.

Chapter 7

Synthesis

The processor core of the three configurations are synthesized with *Synopsys synthesis tools* and the *Artisan Sage-X 0.180μm process for TSMC*. The SRAM elements are synthesized with the *TSMC 0.18μm High Speed/Density Single-Port SRAM Generator*.

7.1 Artisan Sage-X 0.180μm technology library

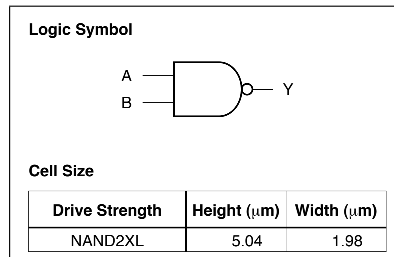
The technology library used for synthesis is the Artisan Sage-X 0.180μm process for TSMC. Only the best case and worst case libraries were available for use in this thesis, the typical library was not available. The worst case library was chosen for synthesis because this ensures that timing constraints will hold for all chips in a production batch and thus provide higher production yield. While the worst case library is worst case for timing, it will give lower power estimates than when using the typical library, as the worst case library is operating at a lower voltage. The worst case voltage and temperature is given as the minimum voltage and maximum temperature in figure 7.1.

Parameter	Minimum	Maximum
DC Supply Voltage (Vdd)	1.62 V	1.98 V
Junction Temperature	-40°C	125°C

Figure 7.1: Minimum and maximum operating voltage and temperature for the *Artisan Sage-X 0.180 μm cell library* [10].

To compute the NAND2 gate equivalent area for the synthesized designs, the NAND2 data from the library documentation [10] is shown in figure 7.2. NAND2 gate area for the Sage-X library can then be calculated to be:

$$Area_{NAND2} = 5.04\mu m \cdot 1.98\mu m = 9.9792\mu m^2$$



Drive Strength	Height (μm)	Width (μm)
NAND2XL	5.04	1.98

Figure 7.2: NAND2 gate information from the *Artisan Sage-X 0.180 μm cell library* documentation [10].

7.2 Synopsys synthesis tools

Three Synopsys tools are used for the synthesis process: *Library Compiler*, *Design Compiler* and *Power Compiler*. These tools are accessible from the `dc_shell` in a terminal window or from the command prompt window in Design Vision. Design Vision is a graphical user interface for the Synopsys tools, and it outputs every command executed with mouse-clicks in the GUI as text-based commands in a log window. This makes it possible to explore the tools trough the GUI interface as well as the user manuals, and then use the text-commands to write work flow scripts. Once the work flow in a design project is established, tcl-scripts can be written to speed up productivity. The scripts are executable from either the `dc_shell` or from within *Synopsys Design Vision*.

7.2.1 Library Compiler

Library Compiler is used to read the technology library in .lib format and compile the library to Synopsys database .db format. This is done in the `dc_shell` with the commands `read_lib` and `write_lib` as shown in the script `compile_library.tcl` below. The .db libraries are used in Synopsys Design Compiler synthesis and the .vhdl libraries are used in MentorGraphics Modelsim gate-level simulations. Further information can be found in the Library Compiler Reference Manual [11].

From `compile_library.tcl`

```
1 # READ .LIB LIBRARY TO MEMORY
2 read_lib slow.lib
3
4 # COMPILE HUMAN READABLE .LIB LIBRARY TO SYNOPSYS .DB
5 write_lib slow -format db -output slow.db
6
7 # COMPILE COMPONENT, VITAL AND FUNC .VHDL MODELS
8 # FOR USE IN MODELSIM SIMULATION
9 write_lib slow -format vhdl
```

Description of the `compile_library.tcl` script: Library Compiler first reads the human readable .lib technology library. The library is compiled and a Synopsys database .db format is written. Finally, VHDL model libraries are output for use in gate-level simulations in *MentorGraphics Modelsim*.

7.2.2 Design Compiler

Design Compiler is used for synthesis, which is converting the design description written in VHDL into an optimized gate-level netlist mapped to the spescific technology library. Figure 7.3 shows the Design Compiler design flow. The synthesis workflow used in this thesis is shown in the `synthesize.tcl`-script below

In general, the steps in the synthesis process are:

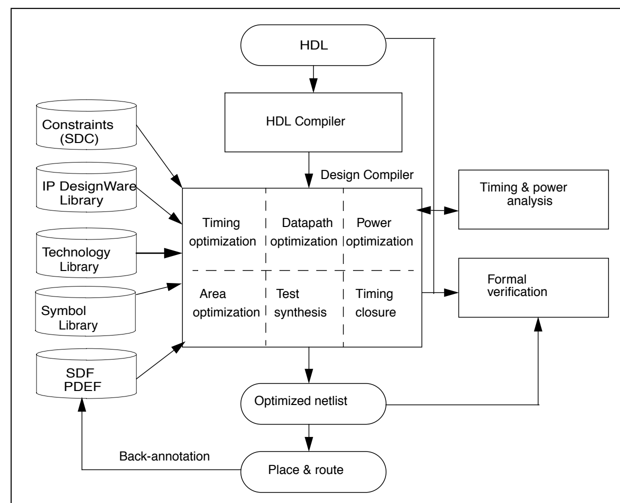


Figure 7.3: Design flow using Synopsys Design Compiler [12].

1. Input design files are read by Design Compiler.
2. Design Compiler uses technology libraries and DesignWare libraries to implement the design.
3. The synthesized gate-level design is optimized to the constraints set by the designer based on specifications.
4. The optimized gate-level netlist is output and is ready for gate-level simulation or place-and-route.

From `synthesize.tcl`:

```
1  # CLEAR MEMORY
2  remove_design -all
3
4  # SET CELL LIBRARY
5  set target_library {/home/steinoe/CELL_LIB/sc/synopsys/slow/slow.db}
```

```
6  set link_library {/home/steinoe/CELL_LIB/sc/synopsys/slow/slow.db}
7
8  lappend search_path {.}
9
10 # READ FILES
11 analyze -library WORK -format vhdl \
12  { /home/steinoe/zpu/zpu/hdl/example_medium/zpu_config_trace.vhd \
13   /home/steinoe/zpu/zpu/hdl/zpu4/core/zpupkg.vhd \
14   /home/steinoe/zpu/zpu/hdl/zpu4/core/zpu_core.vhd
15  }
16
17 # ELABORATE
18 elaborate ZPU_CORE -architecture BEHAVE -library WORK
19
20 # CONSTRAINTS
21 create_clock clk -name clock -period 100
22
23 # CLOCK GATING
24 # insert_clock_gating -global
25
26 # MINIMIZING POWER
27 set_max_dynamic_power 0
28 set_max_total_power 0
29
30 # COMPILE
31 compile -map_effort medium -area_effort medium
32
33 # WRITE NETLIST
34 change_names -rules vhdl -hierarchy
35 set power_preserve_rtl_hier_names TRUE
36 write -hierarchy -format vhdl -output ../zpu_core_reference_netlist.vhdl
```

Description of the `synthesize.tcl` script The target library is first set. Then the .vhdl files for the module to be synthesized is read with the `analyze` command. The top entity is then elaborated using the `elaborate` command. Then the design constraints are set with the `create_clock` and `set_max_dynamic_power` commands. Clock gating for configuration 3 is set with the `insert_clock_gating` command. Finally the design is compiled with the `compile` command and a netlist .vhdl file is output with the `write` command.

7.2.3 Power Compiler

Power Compiler analyzes switching information and propagates the switching information through the synthesized design. The switching information is generated while running simulations of the design in MentorGraphics Modelsim as described in chapter 8. The workflow for reading switching activity and reporting power consumption is shown in the script below and is graphically presented in figure 8.1 and figure 8.2.

From `analyze_and_report.tcl`

```
1  # READ SWITCHING ACTIVITY FROM MODELSIM SIMULATION
2  set find_ignore_case TRUE
3  read_saif -verbose -input ../simulate_aes_encrypt_RTL/simulation_aes_encrypt_RTL.saif -
       instance fpga_top/zpu
4  set find_ignore_case FALSE
5
6  # WRITE POWER REPORT
7  report_power -analysis_effort high >> ../simulate_aes_encrypt_RTL/
       ReportPower_aes_encrypt_RTL.txt
8
9  # WRITE CELL, TIMING, AREA AND SAIF REPORTS
10 report_cell >> ../simulate_aes_encrypt_RTL/ReportCell_aes_encrypt_RTL.txt
11 report_timing -path full -delay max -nworst 1 -max_paths 1 -significant_digits 2 -sort_by
       group >> ../simulate_aes_encrypt_RTL/ReportTiming_aes_encrypt_RTL.txt
12 report_area -nosplit >> ../simulate_aes_encrypt_RTL/ReportArea_aes_encrypt_RTL.txt
13 report_saif >> ../simulate_aes_encrypt_RTL/ReportSaif_aes_encrypt_RTL.txt
```

Description of the `analyze_and_report.tcl` script: The `.saif` file containing switching activity from *Modelsim* simulation is read with the `read_saif` command. Power compiler then propagates the switching activity in the synthesized design and outputs a power report. Reports are also written on cells, timing, area and the switching activity annotation in the synthesized design.

7.3 Configuration 1 core synthesis results

Configuration 1 is a microcontroller with a ZPU core and 32kB SRAM main memory as described in section 6.1. A screenshot of Design Vision with the synthesized configuration 1 core is found in figure 7.4 and the synthesis area results are found in table 7.1.

7.3.1 VHDL-code modifications

The behavioral description of the two ZPU instructions *Loadb2* and *Storeb2* in the original zpu_core.vhd code are not accepted by Design Compiler when running synthesis. The reason for this is that the ZPU is developed on an FPGA platform with Xilinx synthesis tools. These tools uses a different sub-set of the VHDL standard than the Synopsys tools and accepts the more compact description in the original code. The one-line compact code of the original zpu_core.vhd was re-written with case structures to satisfy the standards of the Synopsys tools. The re-written code is shown below.

Code rewritten to be synthesizable with Synopsys tools:

```
1      when State_Loadb2 =>
2       if in_mem_busy='0' then
3        stackA <= (others => '0');
4        --stackA(7 downto 0) <= unsigned(mem_read(((wordBytes-1-to_integer(stackA(byteBits-1
             downto 0)))*8+7) downto (wordBytes-1-to_integer(stackA(byteBits-1 downto 0)))
             *8));
5        --HAVE TO WRITE LINE ABOVE AS CASE STRUCTURE TO COMPILE SUCCESSFULLY
6        case stackA(byteBits-1 downto 0) is
7         when "00" => stackA(7 downto 0) <= unsigned(mem_read(((wordBytes-1-0)*8+7) downto (
             wordBytes-1-0)*8));
8         when "01" => stackA(7 downto 0) <= unsigned(mem_read(((wordBytes-1-1)*8+7) downto (
             wordBytes-1-1)*8));
9         when "10" => stackA(7 downto 0) <= unsigned(mem_read(((wordBytes-1-2)*8+7) downto (
             wordBytes-1-2)*8));
10        when "11" => stackA(7 downto 0) <= unsigned(mem_read(((wordBytes-1-3)*8+7) downto (
             wordBytes-1-3)*8));
11        when others => null;
12       end case;
13       --CASE RE-WRITE ENDS HERE
14       pc <= pc + 1;
15       state <= State_Execute;
16      end if;
17     when State_Storeb2 =>
18      if in_mem_busy='0' then
19       mem_addr <= std_logic_vector(stackA(maxAddrBitIncIO downto minAddrBit));
20       mem_write <= mem_read;
21       --mem_write(((wordBytes-1-to_integer(stackA(byteBits-1 downto 0)))*8+7) downto (
             wordBytes-1-to_integer(stackA(byteBits-1 downto 0)))*8) <= std_logic_vector(
             stackB(7 downto 0));
22       --HAVE TO WRITE LINE ABOVE AS CASE STRUCTURE TO COMPILE SUCCESSFULLY
23       case stackA(byteBits-1 downto 0) is
24        when "00" => mem_write(((wordBytes-1-0)*8+7) downto (wordBytes-1-0)*8) <=
             std_logic_vector(stackB(7 downto 0));
25        when "01" => mem_write(((wordBytes-1-1)*8+7) downto (wordBytes-1-1)*8) <=
             std_logic_vector(stackB(7 downto 0));
26        when "10" => mem_write(((wordBytes-1-2)*8+7) downto (wordBytes-1-2)*8) <=
             std_logic_vector(stackB(7 downto 0));
```

```
27         when "11" => mem_write(((wordBytes-1-3)*8+7) downto (wordBytes-1-3)*8) <=
               std_logic_vector(stackB(7 downto 0));
28         when others => null;
29       end case;
30       --CASE RE-WRITE ENDS HERE
31       mem_writeEnable <= '1';
32       pc <= pc + 1;
33       sp <= incIncSp;
34       state <= State_Resync;
35     end if;
```

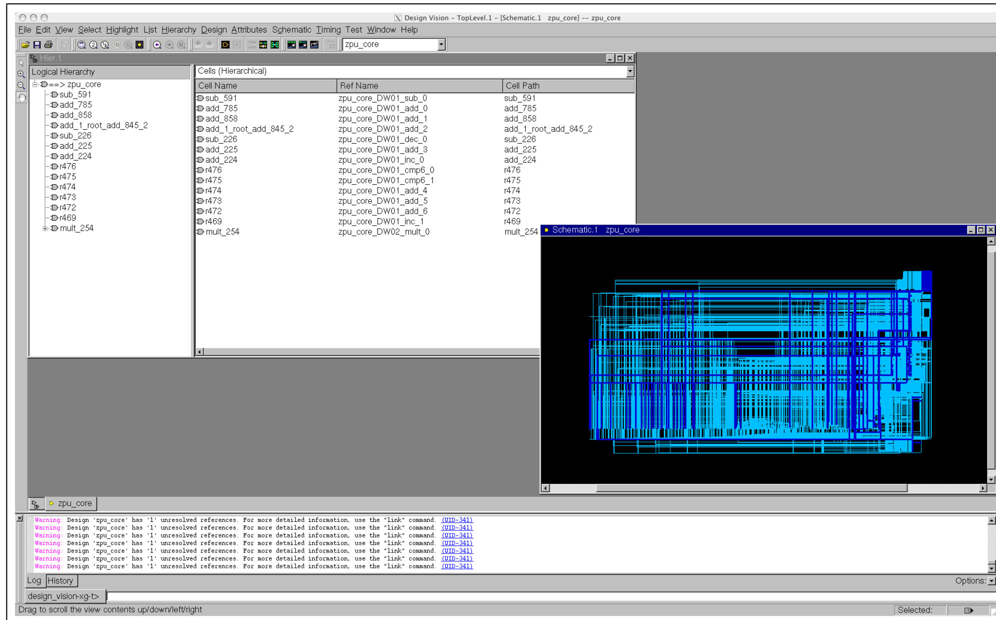


Figure 7.4: Screenshot of Synopsys Design Vision showing the ZPU core of Configuration 1 synthesized with the *Artisan Sage-X 0.180 μm cell library*.

Table 7.1: Configuration 1 core synthesis results

Architecture	**Area**	μm^2	**NAND2 gate equivalent**	**% of total area**
ZPU original	Combinational	127291	12756	77.6
	Sequential	36760	3684	22.4
	Total	164055	16440	100

7.4 Configuration 2 core synthesis results

Configuration 2 is a microcontroller with the Plasma CPU core and 32kB of SRAM main memory as described in section 6.2. A screenshot of Design Vision with the synthesized configuration 2 core is found in figure 7.5 and the synthesis area results are found in table 7.2.

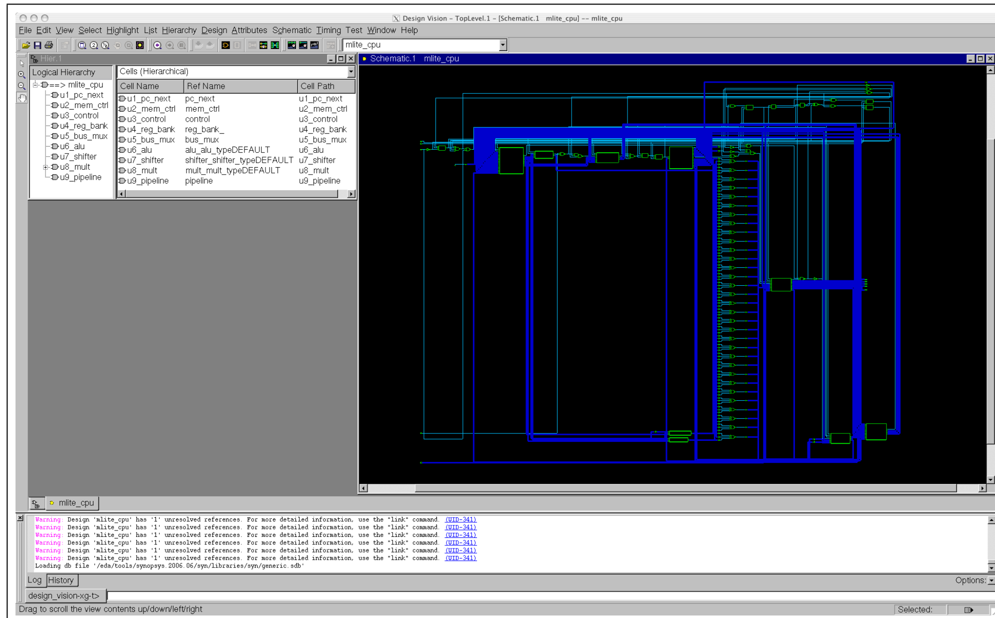


Figure 7.5: Screenshot of Synopsys Design Vision showing the Plasma MIPS core of Configuration 2 synthesized with the *Artisan Sage-X 0.180 µm cell library*.

Table 7.2: Configuration 2 core synthesis results

Architecture	**Area**	μm^2	**NAND2 gate equivalent**	**% of total area**
Plasma MIPS	Combinational	49340	4944	72.1
	Sequential	19070	1911	27.9
	Total	68411	6855	100

7.5 Configuration 3 core synthesis results

Configuration 3 is a microcontroller with a clock gated implementation of the ZPU core, 32kB SRAM main memory and a 128 bytes SRAM cache memory as described in section 6.3. The total area is about 900 gates smaller than for the original ZPU. A screenshot of Design Vision with the synthesized configuration 3 core is found in figure 7.6 and the synthesis area results are found in table 7.3.

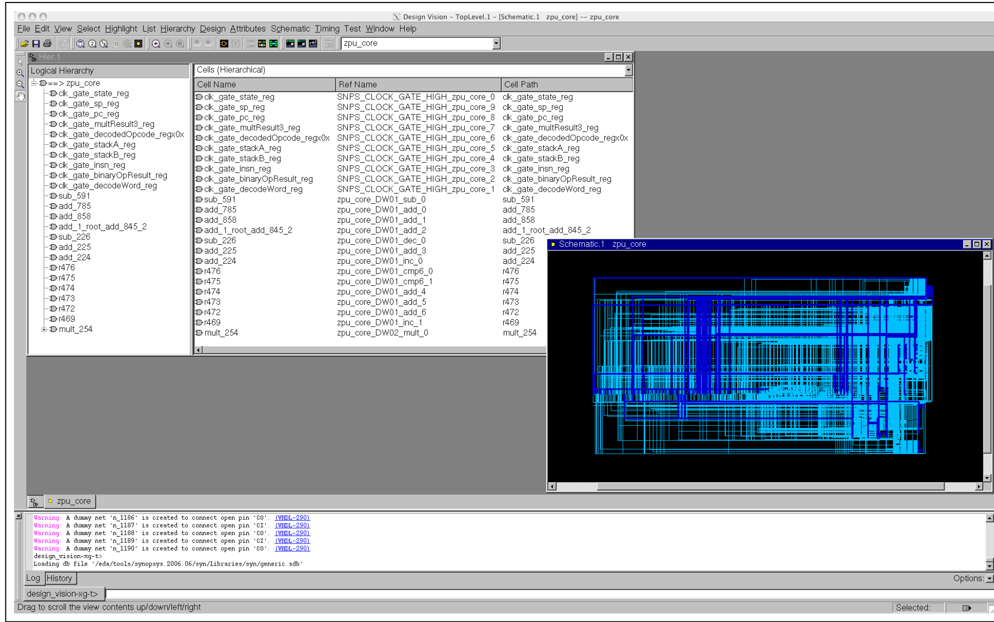


Figure 7.6: Screenshot of Synopsys Design Vision showing the clock gated ZPU core of Configuration 3 synthesized with the *Artisan Sage-X 0.180 μm cell library.*

Table 7.3: Configuration 3 core synthesis results

Architecture	**Area**	μm^2	**NAND2 gate equivalent**	**% of total area**
Clock gated ZPU	Combinational	126253	12652	81.5
	Sequential	28743	2880	18.5
	Total	155000	15532	100

7.6 SRAM memory synthesis results

Synthesis of on-chip SRAM memory is done with *TSMC 0.18um High Speed/Density Single-Port SRAM Generator*. Two SRAM array sizes are compiled, a 32kB SRAM to be used as main memory in configuration 1, 2 and 3, and a 128 byte SRAM to be used as a stack cache in the improved ZPU configuration 3. The SRAM synthesis results are provided by Øyvind Janbu at Energy Micro. The read and write energy per word is estimated on the assumption that the average memory access lasts one clock cycle with equation 7.1, and the values are found in table 7.6.

$$E_{read} = I_{read} \cdot V_{global} \cdot \frac{1}{f_{clk}} \tag{7.1}$$

Table 7.4: Read and write currents for a 32kB SRAM and a 128byte SRAM generated by *TSMC 0.18µm High Speed/Density Single-Port SRAM Generator*.

	32kB SRAM	**128byte SRAM**
Read Current @ 10MHz [mA]	1.57	0.155
Write Current @ 10MHz [mA]	2.07	0.200
Read energy pr word @ 10MHz [pJ]	254.3	25.1
Write energy pr word @ 10MHz [pJ]	335.3	32.4

Chapter 8

Simulation and power estimation

Simulation of the three microcontroller configurations in this thesis are done with Mentor Graphics Modelsim. The switching activity from the simulations are used by Synopsys Power Compiler to estimate the power consumption of the respective synthesized core. The memory subsystem energy consumption is calculated by logging memory read and write signals during simulation of the benchmarks programs, and combined with the memory read and write energy provided by the SRAM synthesis tool in section 7.6.

8.1 Benchmarks

8.1.1 Dhrystone

Dhrystone is a benchmark program developed in 1984 by Reinhold P. Weicker to measure the performance of a computer system. It was written on the basis of a study of different programs, and contains procedure calls, pointer indirections and assignments. Dhrystone version 2.1 was written in 1989 and is the version used in this thesis. The DMIPS value is the number of Dhrystone main loops the processor can execute per second divided by the reference value 1757. The reference value originates from the VAX11/78 computer that could execute 1757 main loops per second, and was used as a reference 1 DMIPS machine.

The benchmark is a measure of compiler and CPU efficiency combined, and many microprocessor manufacturers optimize their compiler to achieve a higher score in the Dhrystone benchmark. Although it is an old benchmark and gives a very narrow performance measurement, Dhrystone is still widely used as a performance indicator for microprocessors, and DMIPS/MHz is often stated in data sheets.

The Dhrystone 2.1 source code has been slightly modified for use in this thesis. The ZPU is simulated while running 100 main Dhrystone loops, and the `printf()` calls in the beginning and end of the benchmark measurement has been commented

out. The reason for this is that the text written by `printf()` to the ZPU UART (Universal Asynchronous Reciever/Transmitter) has almost as long runtime as the 100 main loops themselves. In real-time on a physical CPU the Dhrystone benchmark would run millions of loops, and the `printf()` runtime would only be a small fraction of the total runtime. Such a Modelsim simulation would take weeks, and so to isolate only the benchmark loops and get the most accurate performance measurement, the `printf()` parts have been commented out.

8.1.2 AES-128

AES (Advanced Encryption Standard) is a symmetrical block sipher crypto algorithm. AES encrypts 128 bits of data at a time with either 128, 192 or 256 bit key size, and is one of the most widely used algorithms used in symmetric key cryptography.

In this thesis AES encryption and decryption is performed with a 128-bit key size. The algorithm is also compiled to run on the Plasma MIPS CPU and serves as a comparison between ZPU and MIPS architectures on this kind of algorithm. The software implementation of AES used in this thesis is written by Øivind Ekelund, a fellow student at NTNU who is comparing AES hardware and software implementations in his Master's thesis spring 2009.

8.1.3 Pi approximation

To compare another program execution between the ZPU and MIPS architecture, a numerical approximation to π is used. The program uses the James Gregory approximation given by

$$\pi = 4(1 - 1/3 + 1/5 - 1/7 + 1/9... + \frac{(-1)^n}{2^{k+1}}...)$$

The series is calculated with 100 iterations which gives the value $\pi = 3.151493$.

8.1.4 While(1) spinlock

The `while(1)` spinlock is used to compare how much power the ZPU and MIPS microcontroller configurations consumes while busy waiting. The `while(1)` program is run 10ms on each configuration.

8.2 Compilers

The ZPU and Plasma microprosessors are simulated in Modelsim while running the benchmarks in section 8.1. The Benchmark programs are written in C and compiled to binaries for the respective architecture. These binaries are inserted into the RAM module .vhdl file of the microcontroller. The ZPU is distributed with a complete Linux tool chain which is used in this thesis to compile the programs for the ZPU. The tool chain distributed by the Plasma Project is incomplete with

no support for standard C libraries. Because of this a GCC cross compiler for the MIPS architechture is configured and used in this thesis to compile the benchmark programs for the Plasma CPU.

8.2.1 GCC for ZPU

The ZPU distribution from OpenCores.org comes with a complete Linux tool chain, including a compiled version of GCC cross compiler for the ZPU. The GCC-ZPU includes Newlib and Libstdc++ libraries which means many C/C++ programs can be compiled without modifications.

8.2.2 GCC for MIPS

The Plasma distribution from OpenCores.org comes with a GCC-MIPS compiler with no library support, and with a cumbersome tool chain based on Linux emulation in Windows with Cygwin. Because of this a new GCC for MIPS is compiled and configured for use in this thesis. This self compiled GCC cross compiler for MIPS includes the Newlib libraries to widely extend program compatibility.

Building and configuring GCC as a cross compiler for MIPS

To be able to compile programs that includes the basic C libraries for the Plasma MIPS architecture, it is needed to build GCC as a cross compiler for the MIPS architecture. Cross compiling means making GCC able to run on an x86 Linux machine and compile binaries for the MIPS architecture. The configuration of GCC as a cross compiler for MIPS is basically done in four steps:

1. Build **GNU Binutils**.
2. Build **GNU MPFR (Multiple-Precision Floating-point with Rounding)**.
3. Build **Newlib**.
4. Build **GNU GCC**.

Steps 1, 2 and 4 are explained i further detail on the GNU GCC tab on the Plasma CPUs OpenCores.org webpage [13]. Step 3 is explained in the DOCS section of the Newlib webpage [14]. A short explanation of the tools listed in steps 1 to 4 are given below.

GNU Binutils The GNU binutils is a collection of binary tools for the manipulation of object code in object file formats. Binutils include `ld`, the GNU linker, and `as`, the GNU assembler. For more information, see the GNU binutils webpage [15].

GNU MPFR (Multiple-Precision Floating-point with Rounding) The GNU MPFR is a C library for for binary floating-point computation with correct rounding. For more information see the MPFR library web page [16].

Newlib Newlib is a C standard library intended for us on embedded systems, and can be compiled for a wide array of processors. Newlib is written by the Redhat Project. For more information see the Newlib webpage [14].

GNU GCC The GNU GCC (compiler collection) is a compiler system written by the GNU Project and supports many programming languages. It can be configured as a cross compiler, which means it can run on one system architecture while compiling for another architecture. For more information see the GNU GCC webpage [17].

8.3 Simulation and power estimation

The three microcontroller configurations described in chapter 6 are simulated in *MentorGraphics Modelsim*. The simulations are done on RTL and gate-level with **configuration 1**, on RTL level with **configuration 2** and on gate-level with **configuration 3**. The RTL and gate-level simulation and power estimation workflow are shown in figure 8.2 and figure 8.1. The compiled benchmark programs described in section 8.1 are compiled with the compilers described in section 8.2 and the binary codes are inserted into memory module `.vhdl` files.

The switching activity during simulation is captured by Modelsim and written to a `.saif` file (switching activity interchange format). The `.saif` file is read by *Synopsys Power Compiler* and is used to estimate the power consumption of the respective microprocessor while running a benchmark program.

8.3.1 Configuration 1 simulation results

Microcontroller configuration 1 as described in section 6.1 has a ZPU core and 32kB of SRAM main memory. This configuration is simulated both at RTL and gate-level to compare the power consumption estimates before and after synthesis. The total microcontroller energy consumption and distribution is found in table 8.4 and in figures 8.4 and 8.5.

Core simulation

The ZPU core in configuration 1 is simulated both at RTL and gate-level. The core energy consumption for each benchmark is calculated as shown in equation 8.1. The power estimation results based on RTL and gate-level simulation are found in table 8.1 and figure 8.3. Core energy consumption per benchmark for configuration 1 is found in table 8.2.

$$E_{core} = P_{core} \cdot t_{execution} \quad (8.1)$$

Memory simulation

The Modelsim simulation output `.saif` file for each benchmark includes the internal `read` and `write` signal toggle count. This toggle count is divided by 2 to get the

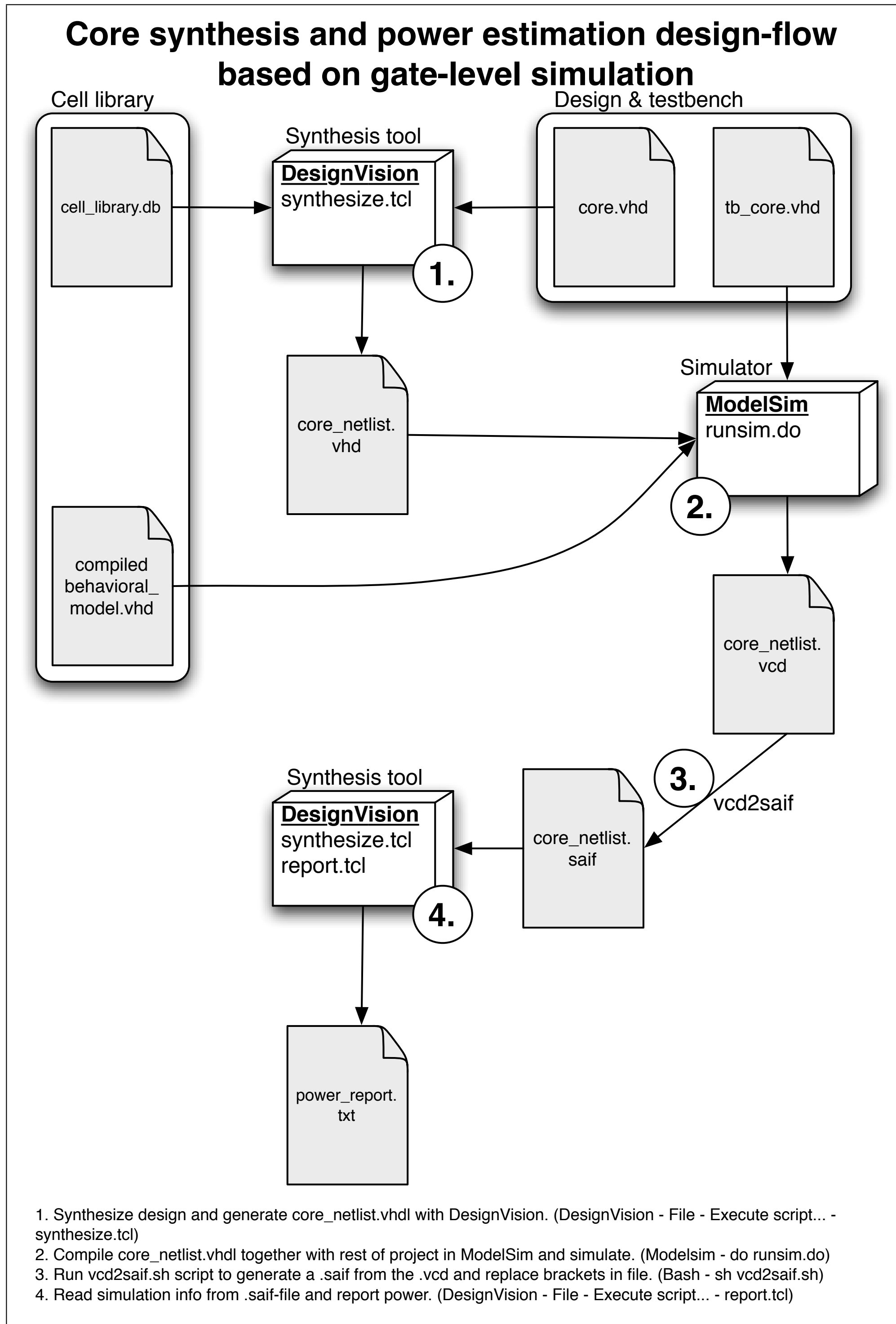


Figure 8.1: Synthesis and power estimation design flow with gate-level simulation.

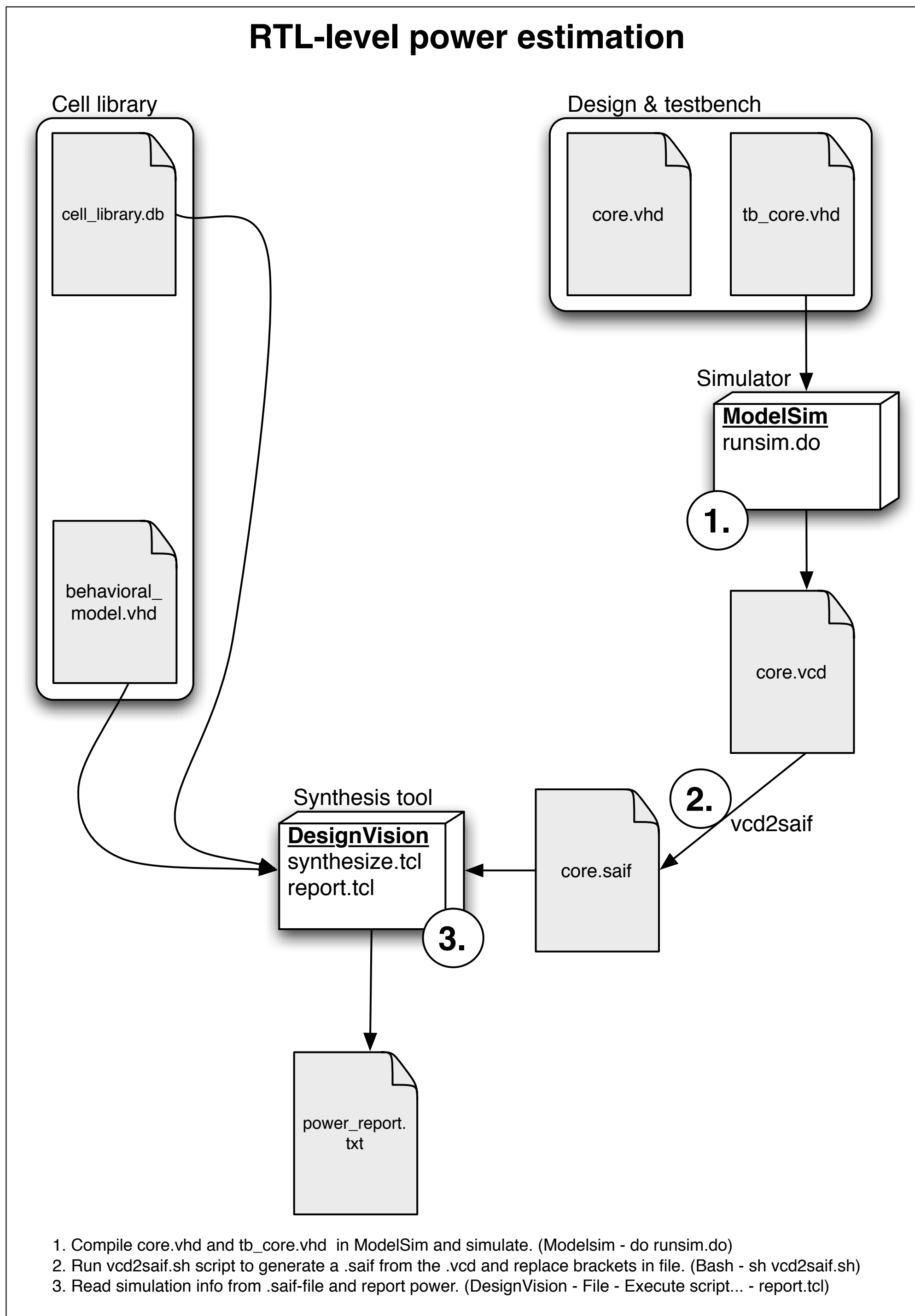


Figure 8.2: RTL-level power estimation flow.

number of reads/writes, and then multiplied with the 32kB SRAM read and write energy values to get the total memory energy consumption for configuration 1 as shown in equation 8.2. The energy per read/write is described in section 7.6 and found in table 7.4. Memory energy consumption for each benchmark is found in table 8.3. The stack pointer trace for each benchmark is shown in figure 8.6 and the stack looping depth and max depth are found in table 8.5.

$$E_{memory} = E_{R(32kB)} \cdot (memReads) + E_{W(32kB)} \cdot (memWrites) \tag{8.2}$$

Table 8.1: Configuration 1 core power estimates based on gate-level and RTL simulation.

Benchmark	Gate-level [μW]	RTL [μW]	Deviation [%]
AES encrypt	390	334	14.4
AES decrypt	389	331	14.9
Dhrystone	386	325	15.8
Pi approximation	380	322	15.6
While(1)	431	362	16.0
Average	395	335	15.3

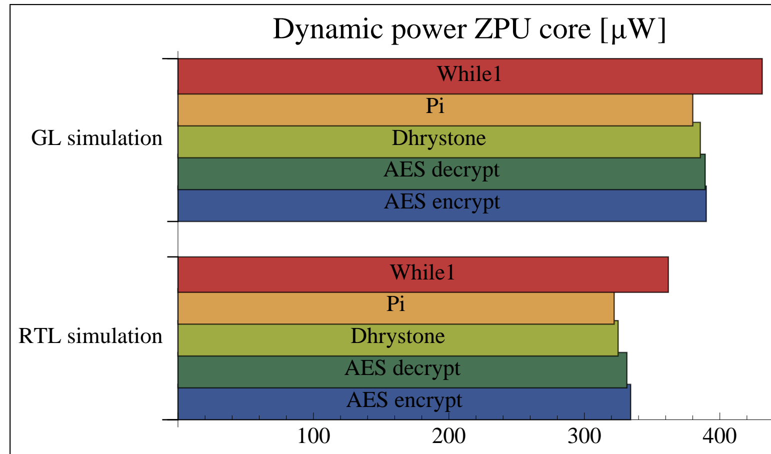


Figure 8.3: Configuration 1 core power estimates based on gate-level and RTL simulation.

Table 8.2: Configuration 1 core total energy for each benchmark.

Benchmark	Core power [μW]	Runtime [ms]	Total core energy [μJ]
AES encrypt	334	7.12	2.38
AES decrypt	331	9.16	3.03
Dhrystone (1 loop)	325	9.42	0.306
Pi approximation	322	0.957	0.308
While(1) (1 ms)	362	1.00	0.362

Table 8.3: Configuration 1 total memory energy consumed for each benchmark.

Benchmark	Memory reads	Memory writes	Total memory energy [μJ]
AES encrypt	15734	7786	6.61
AES decrypt	20619	10195	8.66
Dhrystone (1 loop)	2269	1217	0.985
Pi approximation	2373	1194	1.00
While(1) (1 ms)	2724	1363	1.15

Table 8.4: Configuration 1 total memory energy consumed for each benchmark.

Benchmark	Total energy [μJ]	Core energy [%]	Memory energy [%]
AES encrypt	8.99	26.5	73.5
AES decrypt	11.7	25.9	74.1
Dhrystone (1 loop)	1.29	23.7	76.3
Pi approximation	1.31	23.5	76.4
While(1) (1 ms)	1.51	23.9	76.1
Average percentage		24.7	75.3

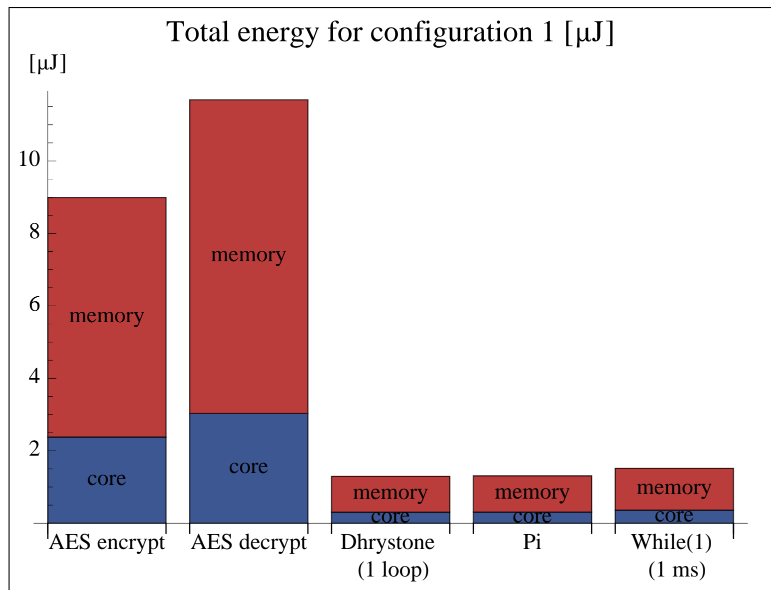


Figure 8.4: Configuration 1 energy distribution.

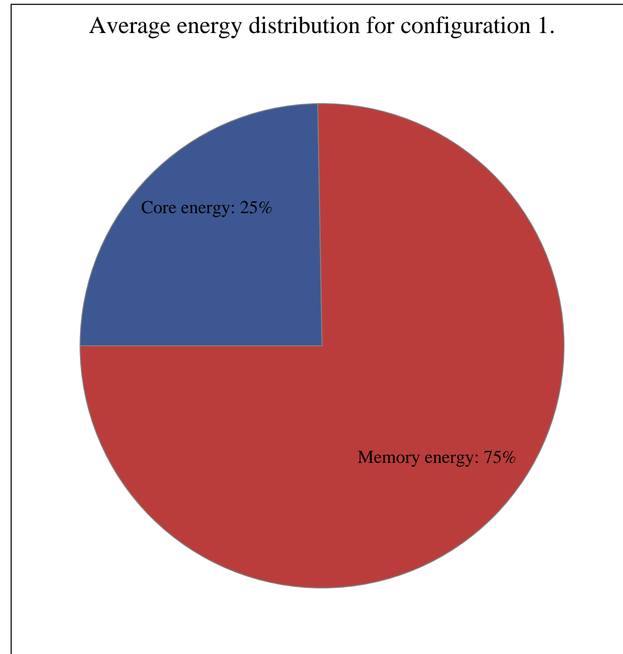


Figure 8.5: Configuration 1 average energy distribution.

Table 8.5: Stack pointer trace for configuration 1.

Benchmark	**Stack max depth** [words]	**Loop depth** [words]
AES encrypt	168	73
AES decrypt	169	75
Dhrystone (1 loop)	127	25
Pi approximation	21	6
While(1) (1 ms)	10	2

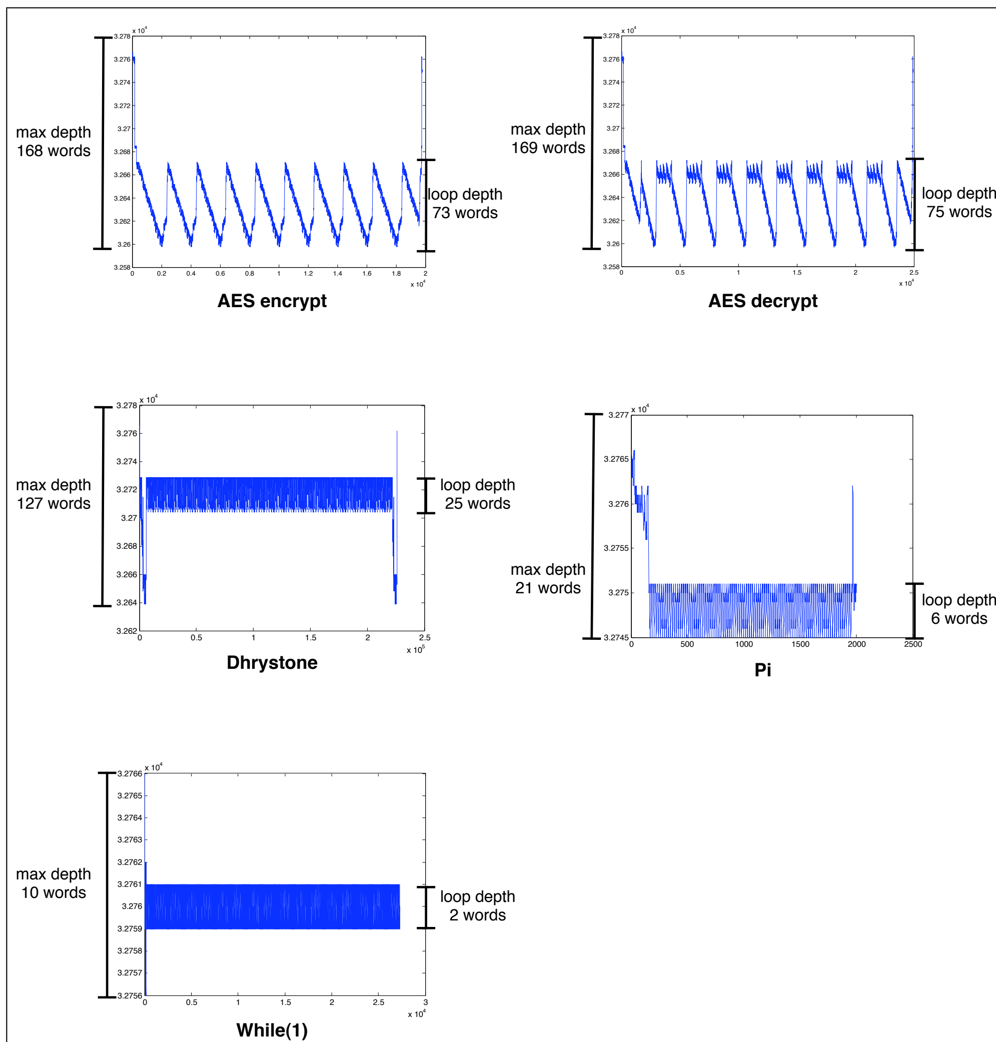


Figure 8.6: Plot of stack pointer trace for the benchmarks executed on configuration 1.

8.3.2 Configuration 2 simulation results

Microcontroller configuration 2 as described in section 6.2 has a Plasma MIPS core and 32kB of SRAM main memory. This configuration is simulated at RTL level. The total microcontroller energy consumption and distribution is found in table 8.10, and shown in figure 8.8 and figure 8.9.

Core simulation

The Plasma MIPS CPU core in configuration 2 is simulated at RTL level. The core power estimation for each benchmark is found in table 8.6 and figure 8.7, and the total energy for each benchmark is calculated with equation 8.3 and found in table 8.7.

$$E_{core} = P_{core} \cdot t_{execution} \tag{8.3}$$

Memory simulation

The memory controller module in the Plasma CPU core, `mem_ctrl`, is modified with three additional signals: one signal toggles every time the CPU reads from memory, another signal toggles every time the CPU writes to memory, and a third signal toggles every time the CPU fetches an instruction. The Modelsim simulation output `.saif` file includes the toggle count of these signals. This toggle count is multiplied with the 32kB SRAM read and write energy values to get the total memory energy consumption for configuration 2. The energy per read/write is described in section 7.6 and found in table 7.4. The number of data memory read and writes and the number of instruction fetches are found in table 8.8. The data memory and instruction memory energy consumption for each benchmark is found in table 8.9.

$$\begin{aligned} E_{dataMem} &= E_{R(32kB)} \cdot (dataMemReads) + \\ &\quad E_{W(32kB)} \cdot (dataMemWrites) \\ E_{instructionMem} &= E_{R(32kB)} \cdot (instructionMemReads) \\ E_{memory} &= E_{dataMem} + E_{instructionMem} \end{aligned} \tag{8.4}$$

Table 8.6: Configuration 2 core power estimates based on RTL simulation.

Benchmark	RTL [μW]
AES encrypt	262
AES decrypt	265
Pi approximation	266
While(1)	167
Average	240

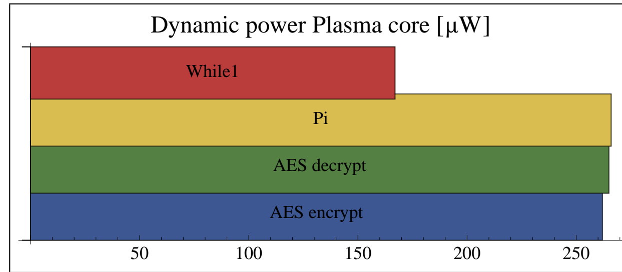


Figure 8.7: Configuration 2 core power estimates based on RTL simulation.

Table 8.7: Configuration 2 core total energy for each benchmark.

Benchmark	Core power [μW]	Runtime [ms]	Core energy [μJ]
AES encrypt	262	0.465	0.122
AES decrypt	265	0.510	0.135
Pi approximation	266	0.080	0.0213
While(1) (1 ms)	167	1.00	0.167

Table 8.8: Configuration 2 data memory reads/writes and instruction fetches.

Benchmark	Data memory reads	Data memory writes	Instruction fetches
AES encrypt	1613	1434	4604
AES decrypt	1600	1409	5530
Pi approximation	53	551	804
While(1) (1 ms)	0	519	149646

Table 8.9: Configuration 2 data memory and instruction memory energy consumption for each benchmark.

Benchmark	Data memory energy [μJ]	Instruction memory energy [μJ]
AES encrypt	0.891	1.17
AES decrypt	0.879	1.40
Pi approximation	0.198	0.204
While(1) (1 ms)	0.0174	3.81

Table 8.10: Configuration 2 total memory energy consumed for each benchmark.

Benchmark	Total energy [μJ]	Core energy [%]	Data energy [%]	Instruction energy [%]
AES encrypt	2.18	5.6	40.8	53.6
AES decrypt	2.42	5.6	36.3	58.1
Pi approximation	0.424	5.0	46.8	48.2
While(1) (1 ms)	3.99	4.2	0.4	95.4
Average percentage		5.1	31.1	63.8

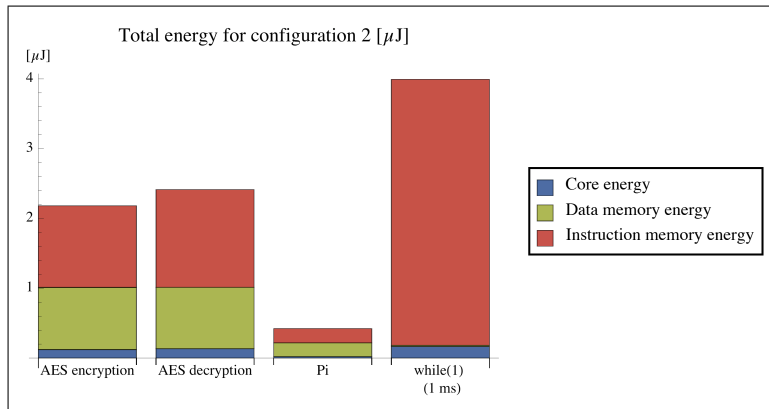


Figure 8.8: Configuration 2 energy distribution.

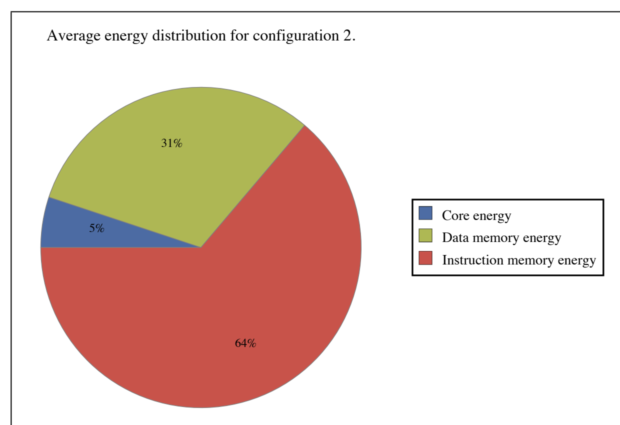


Figure 8.9: Configuration 2 average energy distribution.

8.3.3 Configuration 3 simulation results

Microcontroller configuration 2 as described in section 6.3 has a clock gated ZPU core, 32kB of SRAM main memory and a 128 bytes SRAM stack cache. This configuration is simulated at gate-level. The total microcontroller energy consumption and distribution is found in table 8.15, and shown in figure 8.11 and figure 8.12.

Core simulation

The ZPU core in configuration 3 has implemented clock gating in the synthesis process, and therefore simulation is done at gate-level (after synthesis). The core power estimation results for each benchmark are found in table 8.11. The core energy consumption is calculated with equation 8.5 and found in table 8.12.

$$E_{core} = P_{core} \cdot t_{execution} \tag{8.5}$$

Memory simulation

The memory subsystem of configuration 3 is improved with a small stack cache based on the findings in simulation of configuration 1. The stack cache controller is described in section 11.1. The stack pointer output from simulations is run through a Python script which simulates the stack cache controller. This script is found in Appendix A.3. Simulation output from this script includes how many writes/reads that has occurred to the stack cache, and also how many read/write-backs to main memory has occurred during the execution of a benchmark. The number of reads/writes for main memory is multiplied with the 32kB SRAM read/write energy values to get the main memory energy consumption for configuration 3. The reads/writes for the stack cache is multiplied with the 128 bytes SRAM read/write energy values to get the stack cache memory consumption for configuration 3. The energy per read/write for both SRAM sizes are described in section 7.6 and found in table 7.4. The main memory and stack memory reads and writes are found in table 8.13. The main memory and stack memory energy consumption is calculated with equation 8.6 and the results are found in table 8.14.

$$\begin{aligned}
E_{mainMem} &= E_{R(32kB)} \cdot (mainMemReads - stackReads + readBacks) + \\
&\quad E_{W(32kB)} \cdot (mainMemWrites - stackWrites + writeBacks) \\
\\
E_{stackCache} &= E_{R(128B)} \cdot (stackReads + writeBacks) + \\
&\quad E_{W(128B)} \cdot (stackWrites + readBacks) \\
\\
E_{memory} &= E_{mainMem} + E_{stackCache}
\end{aligned} \tag{8.6}$$

Table 8.11: Configuration 3 core power estimates based on gate-level simulation.

Benchmark	**Core power** [μW]
AES encrypt	246
AES decrypt	245
Dhrystone	236
Pi approximation	228
While(1)	273
Average	246

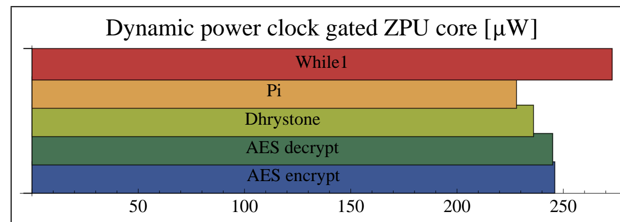


Figure 8.10: Configuration 3 core power estimates based on gate-level simulation.

Table 8.12: Configuration 3 core total energy for each benchmark.

Benchmark	**Core power** [μW]	**Runtime** [ms]	**Total core energy** [μJ]
AES encrypt	246	7.12	1.75
AES decrypt	245	9.16	2.24
Dhrystone (1 loop)	236	9.42	0.222
Pi approximation	228	0.957	0.218
While(1) (1 ms)	273	1.00	0.273

Table 8.13: Configuration 3 memory subsystem reads/writes for each benchmark.

Benchmark	**Main memory reads**	**Main memory writes**	**Stack cache reads**	**Stack cache writes**
AES encrypt	10340	2493	6431	6330
AES decrypt	13268	2947	8454	8351
Dhrystone (1 loop)	1252	201	1022	1021
Pi approximation	1386	223	987	971
While(1) (1 ms)	1363	2	1361	1360

Table 8.14: Configuration 3 main memory and stack cache energy consumed for each benchmark.

Benchmark	**Main memory energy** [μJ]	**Stack cache energy** [μJ]	**Total memory energy** [μJ]
AES encrypt	3.77	0.367	4.14
AES decrypt	4.69	0.483	5.17
Dhrystone (1 loop)	0.388	0.0588	0.446
Pi approximation	0.427	0.0562	0.483
While(1) (1 ms)	0.347	0.0782	0.426

Table 8.15: Configuration 3 total energy consumed for each benchmark.

Benchmark	**Total energy** [μJ]	**Core energy** [%]	**Memory energy** [%]
AES encrypt	5.89	29.7	70.3
AES decrypt	7.41	30.3	69.7
Dhrystone (1 loop)	0.669	33.2	66.8
Pi approximation	0.702	31.1	68.9
While(1) (1 ms)	0.699	39.1	60.9
Average percentage		32.7	67.3

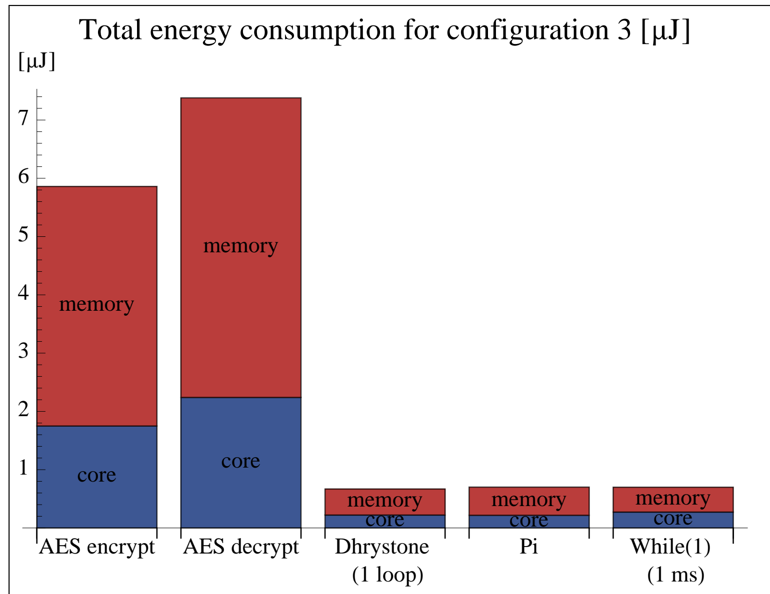


Figure 8.11: Configuration 3 energy consumption distribution.

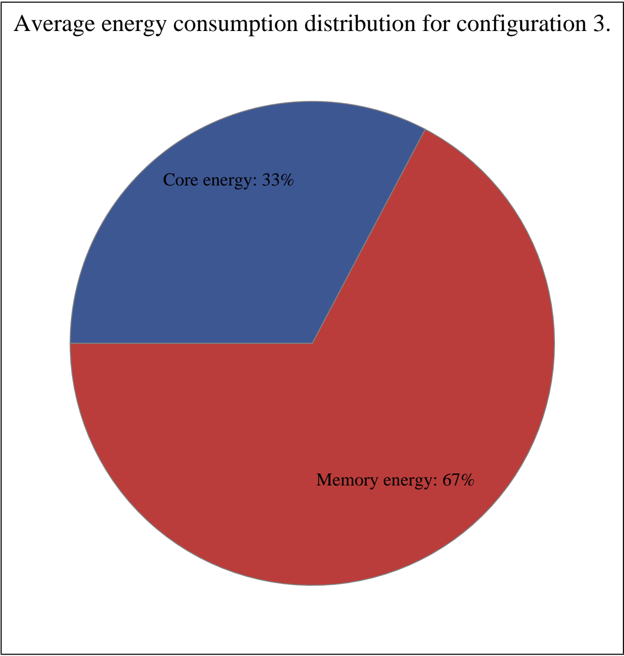


Figure 8.12: Configuration 3 average energy consumption distribution.

Chapter 9

Estimation method evaluation

9.1 RTL vs gate-level power estimation accuracy

Synopsys Power Compiler calculates power consumption in a design based on switching activity from simulations. The switching activity `.saif` file output from RTL simulations contains information only about the signals in the RTL description of the design. This means that Power Compiler only gets to know the switching activity for the flip-flops in the synthesized design. All the other cells cells in the synthesized design are called unannotated cells, as they are not part of the RTL simulation, and hence have unknown switching activity. The switching activity of these unannotated cells are calculated by Power Compiler by propagating the switching activity through the circuitry. Figure 9.1 shows graphically the information that the power estimate is based on when doing RTL estimation.

Simulation at gate-level captures switching activity for all cells in the synthesized design. Power Compiler uses this information to calculate a more accurate power consumption estimate than the RTL estimate. Figure 9.2 shows graphically the information that the power estimate is based on when doing gate-level estimation.

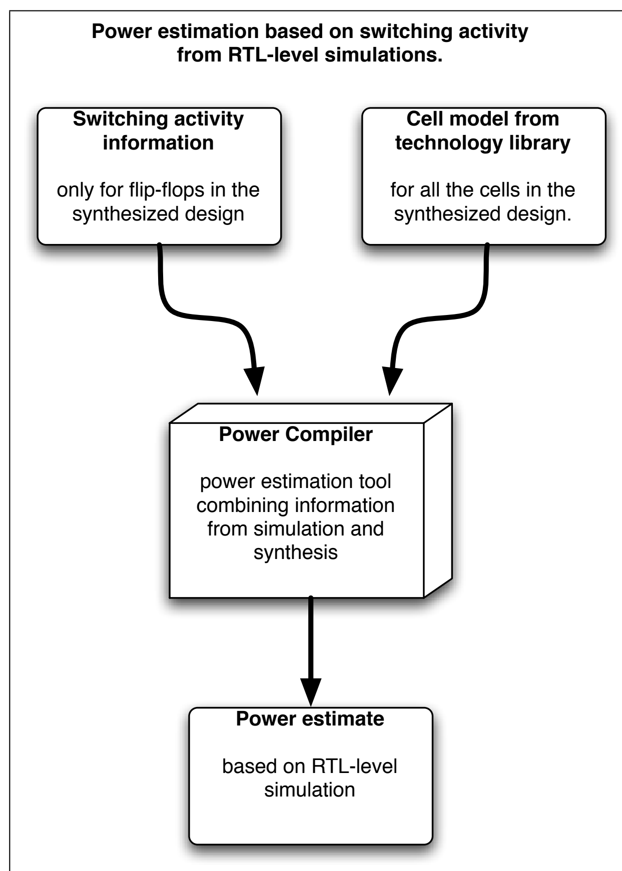


Figure 9.1: Power estimate based on switching activity for the flip-flops only in the synthesized design.

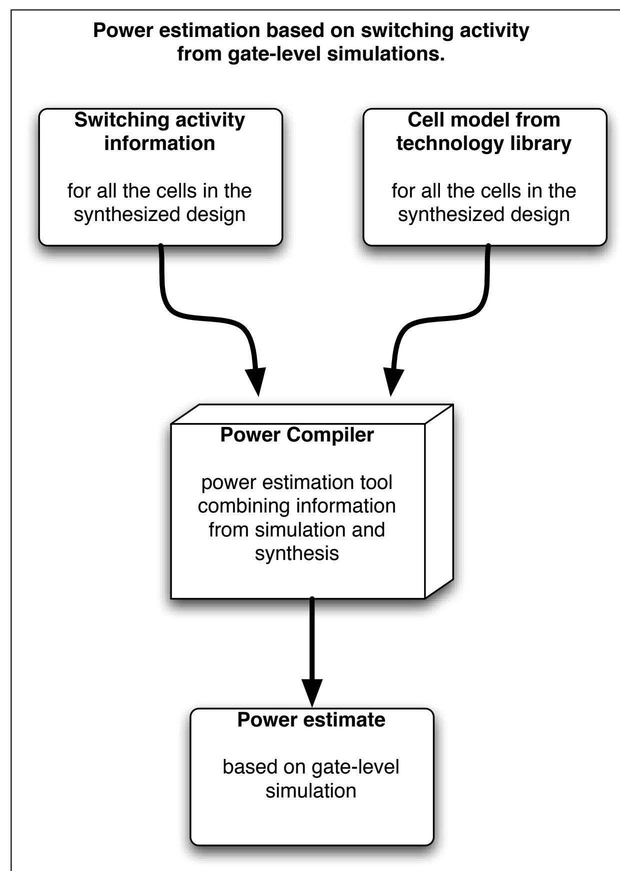


Figure 9.2: Power estimate based on switching activity for all the cells in the synthesized design.

9.2 RTL vs gate-level power estimation speed

Gate-level simulations are a lot more time consuming than RTL simulations. This is because the gate-level simulation is done with the synthesized netlist containing every cell of the design. RTL simulation is done with the HDL description of the design, and so no cells are simulated. Comparing the simulation runtime in table 9.2 and figure 9.3, shows that RTL simulations are 35x faster than gate-level simulations for the ZPU.

Table 9.1: Simulation runtime for RTL and gate-level simulation of the Dhrystone benchmark executing on the ZPU.

Simulation level	Runtime [s]
RTL simulation	41
Gate-level simulation	1464

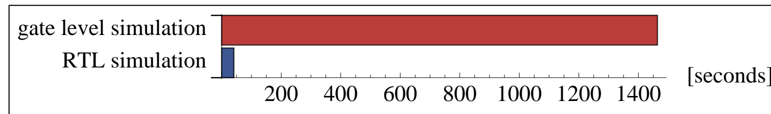


Figure 9.3: Comparison of simulation runtime for the Dhrystone benchmark on configuration 1.

9.3 Gate-level power estimation vs actual silicon chip power consumption

The gate-level power estimation is more accurate than the RTL power estimation because the whole netlist is simulated and switching information on every gate is obtained during simulation. However, at gate-level, the design has not been through the layout step, and the power estimation tools have no information about how much metal interconnect there is between the gates in the synthesized design. The metal interconnect capacitance adds to the total gate input capacitance that the output of every gate is connected to as equation 9.1 shows.

The total output capacitance of a gate on a real silicon chip is given by:

$$C_{gate\ output} = C_{fanout\ input} + C_{metal\ interconnect} \tag{9.1}$$

Where $C_{fanoutinput}$ is the total input capacitances of the connected gates, and $C_{metalinterconnect}$ is the total capacitance of the metal interconnect of the fanout.

Combining the switching power equation 3.4 with the real silicon chip capacitance

equation 9.1 leads to:

$$P_{switching} = (C_{fanout\ input} + C_{metal\ interconnect}) \cdot V_{dd}^2 \tag{9.2}$$

Equation 9.2 shows that the dynamic power consumption of the design implemented on an actual silicon chip will be higher than the power estimate done at gate-level. How much higher depends on the total area of metal interconnect, and this area will be obtained in the layout step of the design process. The layout process is outside the scope of this work, and therefore an accurate measure of how close the power consumption estimate at gate-level is to the actual silicon chip power consumption cannot be derived from the simulations in this thesis.

Chapter 10

Evaluation of ZPU design

10.1 Comparison with MIPS architecture

Table 10.1 and figure 10.1 shows how many cycles the ZPU and the Plasma microprocessor needs to execute some of the benchmark programs. The ZPU uses on an average 15x as many cycles on executing the benchmarks as the Plasma CPU. Table 10.1 shows how many memory accesses is done during execution of the benchmark programs on the ZPU and the Plasma. The ZPU has an average of 3.1x as many memory accesses as the Plasma CPU.

Table 10.1: Number of cycles needed to finish the benchmark programs for the ZPU and the Plasma microprocessor.

Benchmark	ZPU cycles	Plasma cycles	Difference factor
AES encrypt	71200	4650	15.3
AES decrypt	91600	5100	18.0
Pi approximation	9570	800	12.0
Average			15.1

Table 10.2: Number of memory accesses while running the benchmark programs for the ZPU and the Plasma microprocessor.

Benchmark	ZPU memory accesses	Plasma memory accesses	Difference factor
AES encrypt	23520	7651	3.1
AES decrypt	30814	8531	3.6
Pi approximation	4087	1408	2.9
Average			3.2

10.2 ZPU power weaknesses

The ZPU has three major power consumption weaknesses.

1. Memory access
2. Cycle efficiency
3. Stack-machines cannot be pipelined

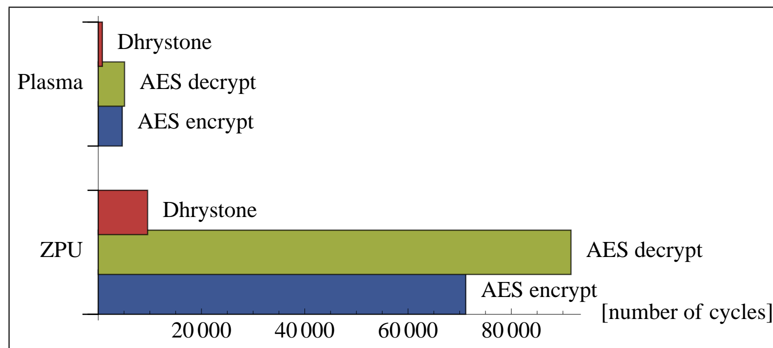


Figure 10.1: Execution cycles for benchmark programs on ZPU and Plasma.

All three weaknesses are interconnected and caused by the fact that the ZPU has a stack-machine architecture. In order to execute a simple `add` instruction with to variables from memory and store the result back to memory, the ZPU needs to:

1. read data from memory and write to top of stack (fetch variable1)
2. read data from memory and write it to top of stack (fetch variable2)
3. read instruction from memory and write it to top of stack (fetch `add` instruction)
4. read from stack (decode instruction)
5. read from stack (feed variable1 to ALU)
6. read from stack (feed variable2 to ALU)
7. write to stack (`add` result)
8. write to memory (store result in memory)

As the stack is also located in main memory, the execution of the `add` instruction needs 6 reads and 5 writes to memory. The Plasma MIPS would need to fetch the instruction word and the two variables stored in memory, do the arithmetics and then store result to memory. This gives the total of 4 memory accesses for an isolated `add` instruction on the Plasma as opposed to 11 memory accesses for the isolated `add` instruction on the ZPU.

The register file in the Plasma also gives room to reuse of variables stored in registers. The Plasma CPU register file can also be used to store temporary values. All instructions in the ZPU are done on top of stack and temporary values needs to be stored in memory with by pushing a `store` instruction to stack, and reading it back from memory with a `load instruction`. Because of this, the ZPU has an a memory access overhead by a factor of 3 when compared to the Plasma, as shown in table 10.1. Table 10.1 shows how many more cycles it takes to execute a benchmark

program on the ZPU versus the Plasma. On average the ZPU requires 15x as many cycles. This is a combination of compiler efficiency and architecture efficiency. The MIPS architecture has been a very popular architecture over the last twenty years, and so the The MIPS GCC compiler used to compile programs for the Plasma probably has many more man-hours into its development than the ZPU GCC compiler used for the ZPU.

A major improvement that can be done with conventional RISC microprocessors with respect to energy consumption is *pipelining*, as shown in section 5.1.2. This cannot be done with single-stack stack-machines, as instruction fetching, data fetching and the arithmetics results needs the top of stack for one instruction before the next instruction can be fetched.

Chapter 11

Energy consumption improvements to ZPU microcontroller design

The microcontroller configuration 1 is used as the reference ZPU microcontroller implementation in this thesis. Microcontroller configuration 3 is the improved microcontroller. Based on the observations in chapter 2 and the simulation and power estimation results of chapter 8, the main efforts to reduce energy consumption have been on reducing memory access energy consumption and the energy consumption of the control logic in the processor core.

The two main implementation changes to achieve lower energy consumption are:

1. a 128 bytes SRAM stack cache memory
2. clock gating implementation of the ZPU core

The implementation details of these two improvements are described in the two following subsections. Table 11.1 shows the original energy consumption per benchmark of configuration 1. Table 11.2 shows the energy consumption of the improved configuration 3, and table 11.3 shows the reduction in percent for each benchmark. Figure 11.1 compares the energy consumption of configuration 1 and configuration 3.

Table 11.1: Energy consumption of microcontroller configuration 1, the original ZPU microcontroller.

Benchmark	**Config. 1 core energy** $[\mu J]$	**Config. 1 memory energy** $[\mu J]$	**Config. 1 total energy** $[\mu J]$
AES encrypt	2.78	6.61	9.39
AES decrypt	3.56	8.66	12.2
Dhrystone (1 loop)	0.364	0.985	1.35
Pi approximation	0.363	1.00	1.37
While(1) (1ms)	0.431	1.15	1.58

Table 11.2: Energy consumption of microcontroller configuration 3, the improved ZPU microcontroller.

Benchmark	**Config. 3 core energy** [μJ]	**Config. 3 memory energy** [μJ]	**Config. 3 total energy** [μJ]
AES encrypt	1.75	4.14	5.89
AES decrypt	2.24	5.17	7.42
Dhrystone (1 loop)	0.222	0.446	0.669
Pi approximation	0.218	0.483	0.702
While(1) (1ms)	0.273	0.426	0.699

Table 11.3: Energy consumption improvements in percent, difference between configuration 1 and configuration 3.

Benchmark	**Core energy reduction** [%]	**Memory energy reduction** [%]	**Total energy reduction** [%]
AES encrypt	36.9	37.4	37.2
AES decrypt	37.0	40.3	39.3
Dhrystone (1 loop)	38.9	54.7	50.4
Pi approximation	40.0	51.8	48.7
While(1) (1ms)	36.7	63.0	55.8
Average	37.9	49.4	46.3

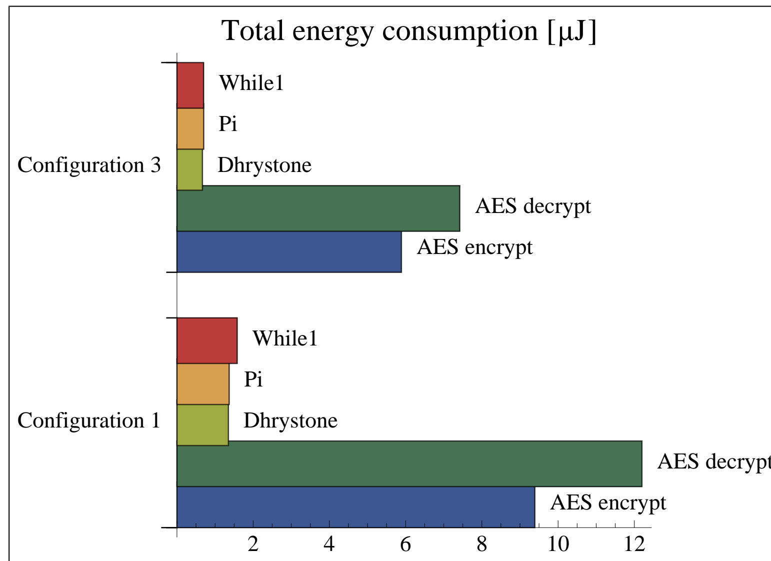


Figure 11.1: Energy consumption of microcontroller configurations 1 and 3 based on gate-level simulation.

11.1 Memory improvements

The stack in configuration 1 is located in main memory. Simulations results in figure 8.6 shows the stackpointer address for the benchmarks executed. During most of the execution time, the stack is looping around a given depth. A 128 bytes SRAM

has only about $\frac{1}{10}$th of the energy consumption per read/write as a 32kB SRAM as shown in the SRAM synthesis results in section 7.6. Configuration 3 therefore has implemented an additional 128 bytes of SRAM as a top of stack cache to reduce power consumption for the stack reads and writes. This 32-word stack cache is implemented as a circular buffer that writes the oldest 16 words back to main memory if it becomes full, and reads the 16 words on top of stack in main memory if it becomes empty. The state diagram of the stack cache is shown in figure 11.2. A python script that simulates the behavior of the stack cache memory module is used to iterate through simulation output and determine how many readbacks and writebacks that occurs for each benchmark program. The memory energy consumption for configuration 3 then can be calculated with equation 8.6. The stack cache memory module python script is found in appendix A.3. On average the memory energy consumption is reduced with 49 % with the stack cache implementation.

Figure 11.3 and 11.4 shows how the stack cache module behaves when it becomes full or empty. The `start` pointer is used to implement the SRAM memory addresses as a circular buffer.

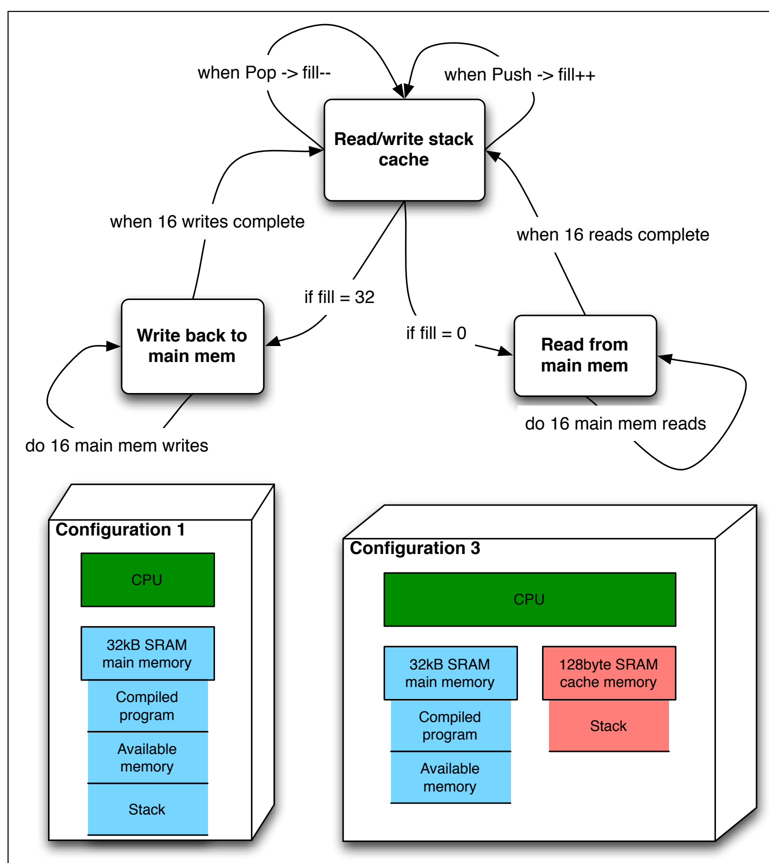


Figure 11.2: State machine of the stack cache controller and memory space of configuration 1 and configuration 3.

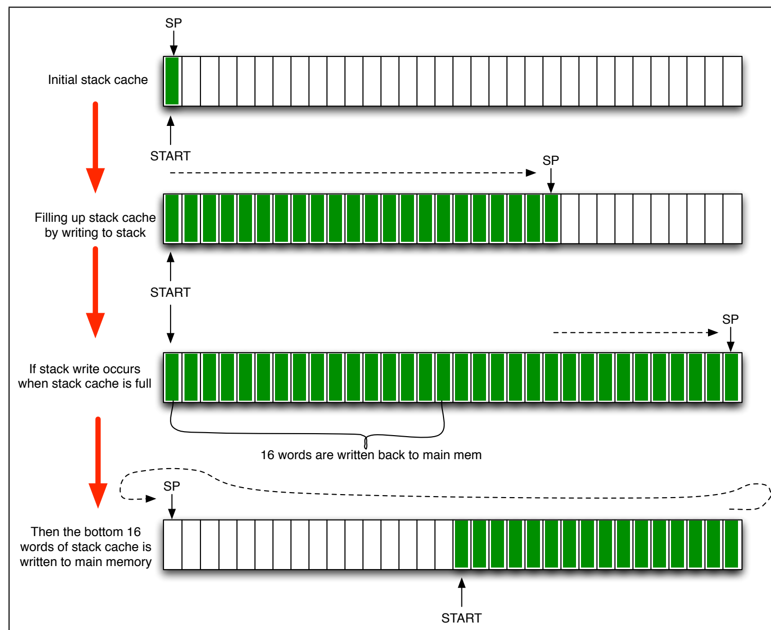


Figure 11.3: Writeback.

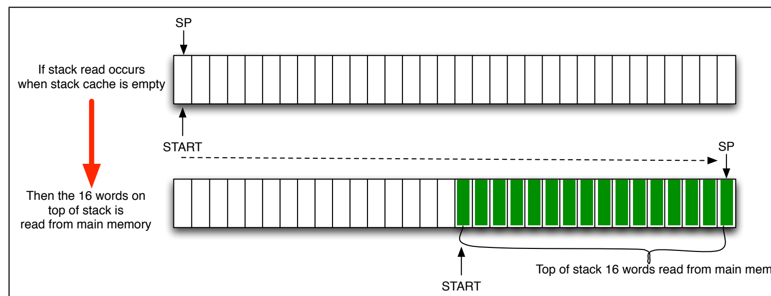


Figure 11.4: Readback.

11.2 Control

The ZPU core in configuration 3 is synthesized with global clock gating to reduce the energy consumption. The clock gating is implemented with Synopsys Design Compiler as shown in the `synthesize.tcl` script in section 7.2.2. On average the core energy consumption is reduced with 38% with clock gating implementation. Figure 11.5 shows compares the power dissipation in the original ZPU microcontroller, the MIPS microcontroller, and the improved ZPU microcontroller.

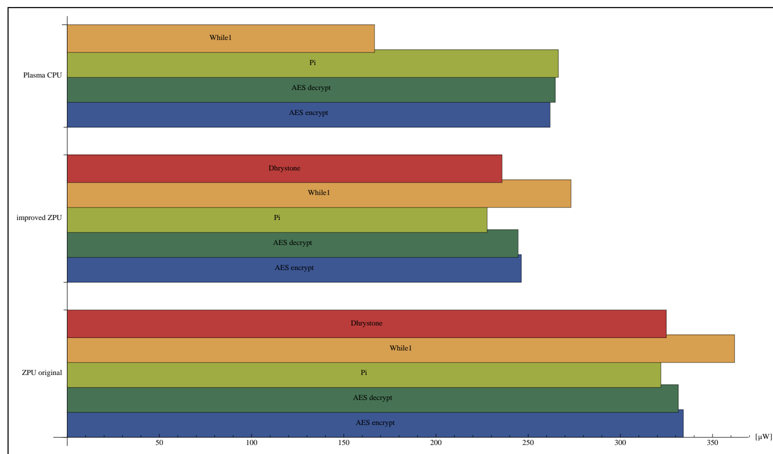


Figure 11.5: Comparison of the core dynamic power dissipation of the three different configurations.

Chapter 12

Discussion

12.1 Power estimation accuracy and speed

When comparing the power estimation results in table 8.3.1 it is shown that the gate level power estimations are at an average 15% higher than the RTL power estimations for the ZPU core. The simulation speed presented in section 9.2 shows that gate level simulation of the ZPU core lasts over 24 minutes while RTL simulation lasts 41 seconds, making RTL simulation 35x faster than gate level simulation. While gate level simulations are more accurate, they makes design flow iterations far slower.

To compare the power estimate based on gate-level simulations with actual silicon chip power consumption one would have to do layout of the design in order to get the value of the interconnect capacitance as described in section 9.3. Taking the design through the layout process was not in the scope of this thesis, and reliable sources on how much the metal interconnect adds to the dynamic power consumption has not been found during an extensive article search. It is relatively certain however, that the gate-level power estimate is an under-estimate as equation 9.2 shows.

12.2 ZPU architectural improvement potential

The ZPU has a stack machine architecture that uses on average 15x as many clock cycles and 3x as many memory accesses as a MIPS processor to complete the benchmark programs, as shown in chapter 10. Also, the single-stack ZPU cannot be pipelined as conventional RISCs to improve throughput and energy efficiency. Addressing the clock cycle and memory access overhead is probably the way to go when attempting to improve the energy efficiency of the ZPU. Two architectural proposals can thus be suggested for future work: implementing multiple stacks to enable pipelining and re-organizing the memory space into a Harvard architecture.

Multiple stack machines are discussed in [18] chapter 3. A plausible architectural

exploration would be to implement the ZPU with two stacks and a 2-stage pipeline. Energy consumption estimation would then show if the multiple stack implementation improves energy efficiency.

Splitting up the memory space into separate instruction and data memory spaces will make it easier to implement traditional caching and loop caching to further reduce memory energy consumption. In [19] it is shown that loop caching the instruction memory reduce energy consumption of instruction fetches.

12.3 Compiler considerations

When comparing the ZPU and the Plasma MIPS in number of cycles needed to execute the benchmark programs, a very important factor is the compiler used for each processor. The MIPS GCC compiler has been under development for decades by a large community, as opposed to the ZPU GCC compiler mostly created by a single developer over a shorter timespan.

The ZPU spends on average 15x as many cycles on executing the benchmark programs as the MIPS. If the ZPU compiler is rewritten and optimized for certain coding styles such as recursive algorithms, the compiler can possibly greatly increase the energy efficiency of the processor. Recursive code is highly optimizable for a stack machine, as nesting in recursive code is compatible with the stack machine way of executing code.

12.4 Implemented ZPU microcontroller improvements

The energy distribution for embedded processors is shown in chapter 2, and memory access, clock distribution and control logic are identified as the main contributers to the energy consumption of an embedded system. The simulations of configuration 1 in chapter 8 confirms these observations. During the planning and the implementation of the improvements in configuration 3, stack memory access and the core control logic were identified as the best places to attack in order to reduce the energy consumption of a ZPU microcontroller. As shown in chapter 11, memory access energy consumption is reduced with 49% by stack caching, and core energy consumption is reduced with 38% by clock gating. This adds to a total average energy consumption reduction of 46% for the ZPU microcontroller.

Chapter 13

Conclusion

A ZPU microcontroller has been synthesized and evaluated with respect to energy consumption. A workflow for power estimation during low-power design has been established to assist decision making and provide faster design cycle iterations during a low-power design process. The ZPU microprocessor has also been compared with a MIPS microprocessor, and the energy consumption weaknesses of the ZPU architecture have been described an discussed. Improvements to the ZPU microcontroller are implemented with the end result of a 46% reduction of the total energy consumption compared to the original microcontroller. Key results of this thesis are:

- Power estimates based on RTL simulations are 35x faster to produce than power estimates based on gate-level simulation, and the RTL estimates deviates by only 15% from the gate-level estimates. Thus RTL power estimates provides faster design cycle iterations without sacrificing too much accuracy.
- The ZPU processor needs 15x as many cycles as the Plasma MIPS processor to execute the benchmark programs used in this thesis. The ZPU also does 3x as many memory accesses as the Plasma MIPS while running the benchmarks.
- The ZPU microcontroller is improved with respect to energy consumption by implementing stack memory caching and processor core clock gating. These improvement measures attack the major energy consumption contributors of embedded systems and reduces total energy consumption with 46%.
- The power consumption estimates produced in this thesis are most likely under-estimates compared to the actual silicon chip implementation power consumption. This is because the metal interconnect capacitance between all the gates in the design is not yet modeled at the gate-level stage in the design flow.

Chapter 14

Further work

Implementing multiple stacks

In order to reduce the energy consumption of the ZPU processor, implementation of a pipelined design should be investigated in future work. This can possibly be done with multiple stacks, and a place to start research on multiple stacks is chapter 3 in [18].

Reorganizing memory space as in Harvard architecture

The current ZPU design is of Von Neumann architecture as the instructions and data are stored in the same memory space. The implementation of separate memory space for instructions and data should be investigated in future work, as this can make implementation of other energy reduction measures possible, such as instruction caching. A place to start research on instruction caching after the memory space is split up into separate instruction and data memory would be [19].

Rewriting ZPU compiler

The current ZPU GCC compiler is not optimized for recursive code. Recursive code is highly optimizable for stack machine architecture. In future work on making the ZPU more energy efficient it should be looked into rewriting the ZPU compiler. It may also be possible to define some coding style that can be recognized by the compiler to produce highly efficient stack-machine code.

References

[1] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield, "Efficient embedded computing," *IEEE Computer Magazine*, vol. 41, pp. 27–32, July 2008.

[2] A. P. Chandrakasan and R. Brodersen, *Low Power Digital CMOS Design.* Springer, 1995.

[3] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi, *Low Power Methodology Manual.* Springer, 2007.

[4] A. Chandrakasan, *Leakage in Nanometer CMOS Technologies.* Springer, 2006.

[5] M. Pedram and J. M. Rabaey, *Power Aware Design Methodologies.* Springer, 1st edition ed., 2002.

[6] Synopsys, *Synopsys Power Compiler User Guide*, version y-2006.06 ed., 2006.

[7] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design.* Morgan Kaufmann, third edition ed., 2007.

[8] www.opencores.org, "Opencores.org web page." 13.06.09.

[9] http://opensource.zylin.com/zpudocs.html, "Zpu processor documentation web page." 13.06.09, 2009.

[10] Artisan Components, *TSMC 0.18um Process 1.8-Volt Sage-X Standard Cell Library Databook*, 2003.

[11] Synopsys, *Synopsys Library Compiler Reference Manual*, version y-2006.06 ed., 2006.

[12] Synopsys, *Synopsys Design Compiler User Guide.* Version Y-2006.06, 2006.

[13] http://www.opencores.org/?do=projectwho=plasma, "Plasma processor opencores.org documentationn page." 13.06.09, 2009.

[14] http://sourceware.org/newlib/, "Newlib library web page." 13.06.09, 2009.

[15] http://www.gnu.org/software/binutils/, "Gnu binutils library web page." 13.06.09, 2009.

[16] http://www.mpfr.org/, “Mpfr library web page.” 13.06.09, 2009.

[17] http://gcc.gnu.org/, “Gnu gcc web page.” 13.06.09, 2009.

[18] J. Philip Koopman, *Stack computers - the new wave.* Ellis Horwood, 1989.

[19] A. Gordon-Ross, S. Cotterell, and F. Vahid, “Tiny instruction caches for low power embedded systems,”

Appendix A

Design-flow scripts and programs

A.1 Modelsim simulation scripts

A.1.1 ZPU simulation script

simulate_zpu.do

```
# SET TO BREAK WHEN DONE
set BreakOnAssertion 1

# SET LIBRARY
vlib work

# COMPILE ZPU TO WORK
vcom -93 -explicit  /home/steinoe/zpu/zpu/hdl/example_medium/zpu_config_trace.vhd
vcom -93 -explicit  /home/steinoe/zpu/zpu/hdl/zpu4/core/zpupkg.vhd
vcom -93 -explicit  /home/steinoe/zpu/zpu/hdl/zpu4/src/txt_util.vhd
vcom -93 -explicit  /home/steinoe/zpu/zpu/hdl/zpu4/core/zpu_core.vhd
vcom -93 -explicit  /home/steinoe/zpu/zpu/hdl/example_medium/sim_fpga_top.vhd
vcom -93 -explicit  /home/steinoe/zpu/sim/roms/V09/dhrystone/dram_dhrystone_noprintf.vhd
vcom -93 -explicit  /home/steinoe/zpu/zpu/hdl/zpu4/src/timer.vhd
vcom -93 -explicit  /home/steinoe/zpu/zpu/hdl/zpu4/src/io.vhd
vcom -93 -explicit  /home/steinoe/zpu/zpu/hdl/zpu4/src/trace.vhd

# RUN ZPU SIM
vsim fpga_top
view wave
add wave -recursive fpga_top/zpu/*
#add wave -recursive fpga_top/*
view structure
#view signals

# WRITE SWITCHING ACTIVITY TO .VCD FILE
vcd file simulation.vcd
vcd add -r sim:fpga_top/zpu/*
power add -r sim:fpga_top/zpu/*

# RUN SIMULATION
run 1000 ms
```

A.1.2 Plasma CPU simulation script

simulate_plasma.do

```
# SET LIBRARY
vlib work

# COMPILE PACKAGE USED IN ALL .VHDL FILES
vcom -93 -explicit  mlite_pack.vhd

```

```
7  # COMPILE SUBMODULES OF MLITE_CPU
8  vcom -93 -explicit  pc_next.vhd
9  vcom -93 -explicit  mem_ctrl.vhd
10 vcom -93 -explicit  control.vhd
11 vcom -93 -explicit  reg_bank.vhd
12 vcom -93 -explicit  bus_mux.vhd
13 vcom -93 -explicit  alu.vhd
14 vcom -93 -explicit  shifter.vhd
15 vcom -93 -explicit  mult.vhd
16 vcom -93 -explicit  pipeline.vhd
17 vcom -93 -explicit  mlite_cpu.vhd
18
19 # COMPILE SUBMODULES OF PLASMA
20 vcom -93 -explicit  ram.vhd
21 vcom -93 -explicit  uart.vhd
22 vcom -93 -explicit  eth_dma.vhd
23 vcom -93 -explicit  plasma.vhd
24
25 # COMPILE TESTBENCH
26 vcom -93 -explicit  tbench.vhd
27
28 # RUN SIMULATION
29 vsim tbench
30 view wave
31 add wave -recursive tbench/u1_plasma/*
32 view structure
33
34 # WRITE SWITCHING ACTIVITY TO .VCD FILE
35 vcd file simulation.vcd
36 vcd add -r sim:tbench/u1_plasma/u1_cpu/*
37
38 #RUNTIME FOR BENCHMARKS
39 # AES endecrypt
40 # run 1050000ns
41
42 # AES key expansion
43 # run 480000ns
44
45 # AES encrypt
46 # run 465000ns
47
48 # AES decrypt
49 # run 510000ns
50
51 # while1
52 # run 10ms
53
54 # pi
55 run 80000ns
56
57 quit -sim
```

A.2 Synopsys synthesis and power estimation scripts

A.2.1 Library Compiler script

A.2.2 Synthesis with Design Compiler scripts

```
1  # CLEAR MEMORY
2  remove_design -all
3
4  # SET CELL LIBRARY
5  set target_library {/home/steinoe/CELL_LIB/sc/synopsys/slow/slow.db}
6  set link_library  {/home/steinoe/CELL_LIB/sc/synopsys/slow/slow.db}
7
8  lappend search_path {.}
9
10 # READ FILES
11 analyze -library WORK -format vhdl \
12  { /home/steinoe/zpu/zpu/hdl/example_medium/zpu_config_trace.vhd \
13   /home/steinoe/zpu/zpu/hdl/zpu4/core/zpupkg.vhd \
14   /home/steinoe/zpu/zpu/hdl/zpu4/core/zpu_core.vhd
15  }
16
17 # ELABORATE
18 elaborate ZPU_CORE -architecture BEHAVE -library WORK
19
20 # CONSTRAINTS
21 create_clock clk -name clock -period 100
```

```
22
23 # CLOCK GATING
24 # insert_clock_gating -global
25
26 # MINIMIZING POWER
27 set_max_dynamic_power 0
28 set_max_total_power 0
29
30 # COMPILE
31 compile -map_effort medium -area_effort medium
32
33 # WRITE NETLIST
34 change_names -rules vhdl -hierarchy
35 set power_preserve_rtl_hier_names TRUE
36 write -hierarchy -format vhdl -output ../zpu_core_reference_netlist.vhdl
```

A.2.3 Power estimation with Power Compiler scripts

VCD to .saif conversion script

```
1  #!/bin/bash
2
3  #convert from vcd to saif
4  vcd2saif -format vhdl -input simulation.vcd -output simulation.saif
5
6  #replace square brackets in saif-file
7  sed -i 's/\[/(/g' simulation.saif
8  sed -i 's/\]/)/g' simulation.saif
9
10 rm simulation.vcd
```

Analyze and report

```
1  # READ SWITCHING ACTIVITY FROM MODELSIM SIMULATION
2  set find_ignore_case TRUE
3  read_saif -verbose -input ../simulate_aes_encrypt_RTL/simulation_aes_encrypt_RTL.saif -
       instance fpga_top/zpu
4  set find_ignore_case FALSE
5
6  # WRITE POWER REPORT
7  report_power -analysis_effort high >> ../simulate_aes_encrypt_RTL/
       ReportPower_aes_encrypt_RTL.txt
8
9  # WRITE CELL, TIMING, AREA AND SAIF REPORTS
10 report_cell >> ../simulate_aes_encrypt_RTL/ReportCell_aes_encrypt_RTL.txt
11 report_timing -path full -delay max -nworst 1 -max_paths 1 -significant_digits 2 -sort_by
       group >> ../simulate_aes_encrypt_RTL/ReportTiming_aes_encrypt_RTL.txt
12 report_area -nosplit >> ../simulate_aes_encrypt_RTL/ReportArea_aes_encrypt_RTL.txt
13 report_saif >> ../simulate_aes_encrypt_RTL/ReportSaif_aes_encrypt_RTL.txt
```

A.3 Stack cache memory iterator

```
1  #!/usr/bin/env
2  #init values
3  toc=131064
4  spinc = 0
5  spdec = 0
6  cachelevel = 1
7  writebacks = 0
8  readbacks = 0
9
10 #cache size
11 maxcachelevel = 1024
12 mincachelevel = 0
13
14 #cache tuning parameters
15 readbacksize = 16
16 writebacksize = 16
17
18 writebackthreshold = maxcachelevel-writebacksize
19 readbackthreshold = mincachelevel+readbacksize
```

```
20
21
22  for l in open('sp_aes_encrypt.txt').read().split('\n'):
23   try:
24    newtoc = int(l)
25    jumps=((toc-newtoc)/4)
26    absjumps = abs(jumps)
27    #print newtoc
28    if (cachelevel < maxcachelevel) & (cachelevel > mincachelevel):
29     if jumps < 0:
30      spdec=spdec+absjumps
31      cachelevel=cachelevel+absjumps
32     elif jumps > 0:
33      spinc=spinc+absjumps
34      cachelevel=cachelevel-absjumps
35
36    elif (cachelevel <= mincachelevel):
37     cachelevel = readbackthreshold
38     readbacks=readbacks+readbacksize+absjumps
39     if jumps < 0:
40      spdec=spdec+absjumps
41      #cachelevel=cachelevel+absjumps
42     elif jumps > 0:
43      spinc=spinc+absjumps
44      #cachelevel=cachelevel-absjumps
45    elif (cachelevel >= maxcachelevel):
46     cachelevel = writebackthreshold
47     writebacks=writebacks+writebacksize+absjumps
48     if jumps < 0:
49      spdec=spdec+absjumps
50      #cachelevel=cachelevel+absjumps
51     elif jumps > 0:
52      spinc=spinc+absjumps
53      #cachelevel=cachelevel-absjumps
54    toc = newtoc
55   except: pass
56  readbacks=readbacks-119
57  sp=open('cache.txt','r+')
58  print >> sp, "SPdec_count_", spdec
59  print >> sp, "SPinc_count_", spinc
60  print >> sp, "Number_of_writebacks_", writebacks
61  print >> sp, "Number_of_readbacks_", readbacks
```

Appendix B

VHDL code

B.1 ZPU core

zpucore.vhdl

```
-- ZPU
--
-- Copyright 2004-2008 oharboe - yvind Harboe - oyvind.harboe@zylin.com
--
-- The FreeBSD license
--
-- Redistribution and use in source and binary forms, with or without
-- modification, are permitted provided that the following conditions
-- are met:
--
-- 1. Redistributions of source code must retain the above copyright
--    notice, this list of conditions and the following disclaimer.
-- 2. Redistributions in binary form must reproduce the above
--    copyright notice, this list of conditions and the following
--    disclaimer in the documentation and/or other materials
--    provided with the distribution.
--
-- THIS SOFTWARE IS PROVIDED BY THE ZPU PROJECT ``AS IS'' AND ANY
-- EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
-- THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
-- PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
-- ZPU PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
-- INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
-- (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
-- OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
-- HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
-- STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
-- ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
-- ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
--
-- The views and conclusions contained in the software and documentation
-- are those of the authors and should not be interpreted as representing
-- official policies, either expressed or implied, of the ZPU Project.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

library work;
use work.zpu_config.all;
use work.zpupkg.all;


-- mem_writeEnable - set to '1' for a single cycle to send off a write request.
--                   mem_write is valid only while mem_writeEnable='1'.
-- mem_readEnable - set to '1' for a single cycle to send off a read request.
--
-- mem_busy - It is illegal to send off a read/write request when mem_busy='1'.
--            Set to '0' when mem_read is valid after a read request.
--            If it goes to '1'(busy), it is on the cycle after mem_read/writeEnable
--            is '1'.
-- mem_addr - address for read/write request
-- mem_read - read data. Valid only on the cycle after mem_busy='0' after
--            mem_readEnable='1' for a single cycle.
-- mem_write - data to write
```

```
56  -- mem_writeMask - set to '1' for those bits that are to be written to memory upon
57  --                 write request
58  -- break - set to '1' when CPU hits break instruction
59  -- interrupt - set to '1' until interrupts are cleared by CPU.
60
61
62
63
64  entity zpu_core is
65      Port ( clk : in std_logic;
66        areset : in std_logic;
67        enable : in std_logic;
68        in_mem_busy : in std_logic;
69        mem_read : in std_logic_vector(wordSize-1 downto 0);
70        mem_write : out std_logic_vector(wordSize-1 downto 0);
71       out_mem_addr : out std_logic_vector(maxAddrBitIncIO downto 0);
72       out_mem_writeEnable : out std_logic;
73       out_mem_readEnable : out std_logic;
74        mem_writeMask: out std_logic_vector(wordBytes-1 downto 0);
75        interrupt : in std_logic;
76        break : out std_logic);
77  end zpu_core;
78
79  architecture behave of zpu_core is
80
81  type InsnType is
82  (
83  State_AddTop,
84  State_Dup,
85  State_DupStackB,
86  State_Pop,
87  State_Popdown,
88  State_Add,
89  State_Or,
90  State_And,
91  State_Store,
92  State_AddSP,
93  State_Shift,
94  State_Nop,
95  State_Im,
96  State_LoadSP,
97  State_StoreSP,
98  State_Emulate,
99  State_Load,
100 State_PushPC,
101 State_PushSP,
102 State_PopPC,
103 State_PopPCRel,
104 State_Not,
105 State_Flip,
106 State_PopSP,
107 State_Neqbranch,
108 State_Eq,
109 State_Loadb,
110 State_Mult,
111 State_Lessthan,
112 State_Lessthanorequal,
113 State_Ulessthanorequal,
114 State_Ulessthan,
115 State_Pushspadd,
116 State_Call,
117 State_Callpcrel,
118 State_Sub,
119 State_Break,
120 State_Storeb,
121 State_InsnFetch
122 );
123
124 type StateType is
125 (
126 State_Load2,
127 State_Popped,
128 State_LoadSP2,
129 State_LoadSP3,
130 State_AddSP2,
131 State_Fetch,
132 State_Execute,
133 State_Decode,
134 State_Decode2,
135 State_Resync,
136
137 State_StoreSP2,
138 State_Resync2,
139 State_Resync3,
140 State_Loadb2,
141 State_Storeb2,
142 State_Mult2,
```

```
143 State_Mult3 ,
144 State_Mult5 ,
145 State_Mult4 ,
146 State_BinaryOpResult2 ,
147 State_BinaryOpResult ,
148 State_Idle
149 );
150
151
152 signal pc     : unsigned(maxAddrBitIncIO downto 0);
153 signal sp     : unsigned(maxAddrBitIncIO downto minAddrBit);
154 signal incSp    : unsigned(maxAddrBitIncIO downto minAddrBit);
155 signal incIncSp    : unsigned(maxAddrBitIncIO downto minAddrBit);
156 signal decSp    : unsigned(maxAddrBitIncIO downto minAddrBit);
157 signal  stackA      : unsigned(wordSize-1 downto 0);
158 signal  binaryOpResult   : unsigned(wordSize-1 downto 0);
159 signal  binaryOpResult2   : unsigned(wordSize-1 downto 0);
160 signal  multResult2     : unsigned(wordSize-1 downto 0);
161 signal  multResult3     : unsigned(wordSize-1 downto 0);
162 signal  multResult     : unsigned(wordSize-1 downto 0);
163 signal  multA    : unsigned(wordSize-1 downto 0);
164 signal  multB     : unsigned(wordSize-1 downto 0);
165 signal  stackB     : unsigned(wordSize-1 downto 0);
166 signal idim_flag    : std_logic;
167 signal busy      : std_logic;
168 signal mem_writeEnable : std_logic;
169 signal mem_readEnable : std_logic;
170 signal mem_addr : std_logic_vector(maxAddrBitIncIO downto minAddrBit);
171 signal mem_delayAddr : std_logic_vector(maxAddrBitIncIO downto minAddrBit);
172 signal mem_delayReadEnable : std_logic;
173
174 signal decodeWord : std_logic_vector(wordSize-1 downto 0);
175
176
177 signal state : StateType;
178 signal insn : InsnType;
179 type InsnArray is array(0 to wordBytes-1) of InsnType;
180 signal decodedOpcode : InsnArray;
181
182 type OpcodeArray is array(0 to wordBytes-1) of std_logic_vector(7 downto 0);
183
184 signal opcode : OpcodeArray;
185
186
187
188
189 signal begin_inst    : std_logic;
190 signal trace_opcode   : std_logic_vector(7 downto 0);
191 signal trace_pc     : std_logic_vector(maxAddrBitIncIO downto 0);
192 signal trace_sp     : std_logic_vector(maxAddrBitIncIO downto minAddrBit);
193 signal trace_topOfStack     : std_logic_vector(wordSize-1 downto 0);
194 signal trace_topOfStackB     : std_logic_vector(wordSize-1 downto 0);
195
196 -- state machine.
197
198 begin
199
200
201  traceFileGenerate:
202    if Generate_Trace generate
203  trace_file: trace port map (
204         clk => clk,
205         begin_inst => begin_inst,
206         pc => trace_pc,
207   opcode => trace_opcode,
208   sp => trace_sp,
209   memA => trace_topOfStack,
210   memB => trace_topOfStackB,
211   busy => busy,
212   intsp => (others => 'U')
213         );
214  end generate;
215
216
217  -- the memory subsystem will tell us one cycle later whether or
218  -- not it is busy
219  out_mem_writeEnable <= mem_writeEnable;
220  out_mem_readEnable <= mem_readEnable;
221  out_mem_addr(maxAddrBitIncIO downto minAddrBit) <= mem_addr;
222  out_mem_addr(minAddrBit-1 downto 0) <= (others => '0');
223
224  incSp <= sp + 1;
225  incIncSp <= sp + 2;
226  decSp <= sp - 1;
227
228
229  opcodeControl:
```

```
230  process(clk, areset)
231   variable tOpcode : std_logic_vector(OpCode_Size-1 downto 0);
232   variable spOffset : unsigned(4 downto 0);
233   variable tSpOffset : unsigned(4 downto 0);
234   variable nextPC : unsigned(maxAddrBitIncIO downto 0);
235   variable tNextState : InsnType;
236   variable tDecodedOpcode : InsnArray;
237   variable tMultResult : unsigned(wordSize*2-1 downto 0);
238  begin
239   if areset = '1' then
240    state <= State_Idle;
241    break <= '0';
242    sp <= unsigned(spStart(maxAddrBitIncIO downto minAddrBit));
243
244    pc <= (others => '0');
245    idim_flag <= '0';
246    begin_inst <= '0';
247    mem_writeEnable <= '0';
248    mem_readEnable <= '0';
249    multA <= (others => '0');
250    multB <= (others => '0');
251    mem_writeMask <= (others => '1');
252   elsif (clk'event and clk = '1') then
253    -- we must multiply unconditionally to get pipelined multiplication
254          tMultResult := multA * multB;
255       multResult3 <= multResult2;
256       multResult2 <= multResult;
257       multResult <= tMultResult(wordSize-1 downto 0);
258
259
260    binaryOpResult2 <= binaryOpResult; -- pipeline a bit.
261
262
263    multA <= (others => DontCareValue);
264    multB <= (others => DontCareValue);
265
266
267    mem_addr <= (others => DontCareValue);
268    mem_readEnable <='0';
269    mem_writeEnable <='0';
270    mem_write <= (others => DontCareValue);
271
272    if (mem_writeEnable = '1') and (mem_readEnable = '1') then
273     report "read/write_collision" severity failure;
274    end if;
275
276
277
278
279    spOffset(4):=not opcode(to_integer(pc(byteBits-1 downto 0)))(4);
280    spOffset(3 downto 0):=unsigned(opcode(to_integer(pc(byteBits-1 downto 0)))(3 downto 0))
         ;
281    nextPC := pc + 1;
282
283    -- prepare trace snapshot
284    trace_opcode <= opcode(to_integer(pc(byteBits-1 downto 0)));
285    trace_pc <= std_logic_vector(pc);
286    trace_sp <= std_logic_vector(sp);
287    trace_topOfStack <= std_logic_vector(stackA);
288    trace_topOfStackB <= std_logic_vector(stackB);
289    begin_inst <= '0';
290
291
292    case state is
293     when State_Idle =>
294      if enable='1' then
295       state <= State_Resync;
296      end if;
297     -- Initial state of ZPU, fetch top of stack + first instruction
298     when State_Resync =>
299      if in_mem_busy='0' then
300       mem_addr <= std_logic_vector(sp);
301       mem_readEnable <= '1';
302       state <= State_Resync2;
303      end if;
304     when State_Resync2 =>
305      if in_mem_busy='0' then
306       stackA <= unsigned(mem_read);
307       mem_addr <= std_logic_vector(incSp);
308       mem_readEnable <= '1';
309       state <= State_Resync3;
310      end if;
311     when State_Resync3 =>
312      if in_mem_busy='0' then
313       stackB <= unsigned(mem_read);
314       mem_addr <= std_logic_vector(pc(maxAddrBitIncIO downto minAddrBit));
315       mem_readEnable <= '1';
```

```
316          state <= State_Decode;
317        end if;
318      when State_Decode =>
319        if in_mem_busy='0' then
320          decodeWord <= mem_read;
321          state <= State_Decode2;
322        end if;
323      when State_Decode2 =>
324        -- decode 4 instructions in parallel
325        for i in 0 to wordBytes-1 loop
326          tOpcode := decodeWord((wordBytes-1-i+1)*8-1 downto (wordBytes-1-i)*8);
327
328          tSpOffset(4):=not tOpcode(4);
329          tSpOffset(3 downto 0):=unsigned(tOpcode(3 downto 0));
330
331          opcode(i) <= tOpcode;
332          if (tOpcode(7 downto 7)=OpCode_Im) then
333            tNextState:=State_Im;
334          elsif (tOpcode(7 downto 5)=OpCode_StoreSP) then
335            if tSpOffset = 0 then
336              tNextState := State_Pop;
337            elsif tSpOffset=1 then
338              tNextState := State_PopDown;
339            else
340              tNextState :=State_StoreSP;
341            end if;
342          elsif (tOpcode(7 downto 5)=OpCode_LoadSP) then
343            if tSpOffset = 0 then
344              tNextState :=State_Dup;
345            elsif tSpOffset = 1 then
346              tNextState :=State_DupStackB;
347            else
348              tNextState :=State_LoadSP;
349            end if;
350          elsif (tOpcode(7 downto 5)=OpCode_Emulate) then
351            tNextState :=State_Emulate;
352            if tOpcode(5 downto 0)=OpCode_Neqbranch then
353              tNextState :=State_Neqbranch;
354            elsif tOpcode(5 downto 0)=OpCode_Eq then
355              tNextState :=State_Eq;
356            elsif tOpcode(5 downto 0)=OpCode_Lessthan then
357              tNextState :=State_Lessthan;
358            elsif tOpcode(5 downto 0)=OpCode_Lessthanorequal then
359              --tNextState :=State_Lessthanorequal;
360            elsif tOpcode(5 downto 0)=OpCode_Ulessthan then
361              tNextState :=State_Ulessthan;
362            elsif tOpcode(5 downto 0)=OpCode_Ulessthanorequal then
363              --tNextState :=State_Ulessthanorequal;
364            elsif tOpcode(5 downto 0)=OpCode_Loadb then
365              tNextState :=State_Loadb;
366            elsif tOpcode(5 downto 0)=OpCode_Mult then
367              tNextState :=State_Mult;
368            elsif tOpcode(5 downto 0)=OpCode_Storeb then
369              tNextState :=State_Storeb;
370            elsif tOpcode(5 downto 0)=OpCode_Pushspadd then
371              tNextState :=State_Pushspadd;
372            elsif tOpcode(5 downto 0)=OpCode_Callpcrel then
373              tNextState :=State_Callpcrel;
374            elsif tOpcode(5 downto 0)=OpCode_Call then
375              --tNextState :=State_Call;
376            elsif tOpcode(5 downto 0)=OpCode_Sub then
377              tNextState :=State_Sub;
378            elsif tOpcode(5 downto 0)=OpCode_PopPCRel then
379              --tNextState :=State_PopPCRel;
380            end if;
381          elsif (tOpcode(7 downto 4)=OpCode_AddSP) then
382            if tSpOffset = 0 then
383              tNextState := State_Shift;
384            elsif tSpOffset = 1 then
385              tNextState := State_AddTop;
386            else
387              tNextState :=State_AddSP;
388            end if;
389          else
390            case tOpcode(3 downto 0) is
391              when OpCode_Nop =>
392                tNextState :=State_Nop;
393              when OpCode_PushSP =>
394                tNextState :=State_PushSP;
395              when OpCode_PopPC =>
396                tNextState :=State_PopPC;
397              when OpCode_Add =>
398                tNextState :=State_Add;
399              when OpCode_Or =>
400                tNextState :=State_Or;
401              when OpCode_And =>
402                tNextState :=State_And;
```

```
403                when OpCode_Load =>
404                 tNextState :=State_Load;
405                when OpCode_Not =>
406                 tNextState :=State_Not;
407                when OpCode_Flip =>
408                 tNextState :=State_Flip;
409                when OpCode_Store =>
410                 tNextState :=State_Store;
411                when OpCode_PopSP =>
412                 tNextState :=State_PopSP;
413                when others =>
414                 tNextState := State_Break;
415
416               end case;
417              end if;
418              tDecodedOpcode(i) := tNextState;
419
420            end loop;
421
422            insn <= tDecodedOpcode(to_integer(pc(byteBits-1 downto 0)));
423
424            -- once we wrap, we need to fetch
425            tDecodedOpcode(0) := State_InsnFetch;
426
427            decodedOpcode <= tDecodedOpcode;
428            state <= State_Execute;
429
430
431
432            -- Each instruction must:
433            --
434            -- 1. set idim_flag
435            -- 2. increase pc if applicable
436            -- 3. set next state if appliable
437            -- 4. do it's operation
438
439          when State_Execute =>
440           insn <= decodedOpcode(to_integer(nextPC(byteBits-1 downto 0)));
441
442           case insn is
443            when State_InsnFetch =>
444             state <= State_Fetch;
445            when State_Im =>
446             if in_mem_busy='0' then
447              begin_inst <= '1';
448              idim_flag <= '1';
449              pc <= pc + 1;
450
451              if idim_flag='1' then
452               stackA(wordSize-1 downto 7) <= stackA(wordSize-8 downto 0);
453               stackA(6 downto 0) <= unsigned(opcode(to_integer(pc(byteBits-1 downto 0)))(6
                    downto 0));
454              else
455               mem_writeEnable <= '1';
456               mem_addr <= std_logic_vector(incSp);
457               mem_write <= std_logic_vector(stackB);
458               stackB <= stackA;
459               sp <= decSp;
460               for i in wordSize-1 downto 7 loop
461                stackA(i) <= opcode(to_integer(pc(byteBits-1 downto 0)))(6);
462               end loop;
463               stackA(6 downto 0) <= unsigned(opcode(to_integer(pc(byteBits-1 downto 0)))(6
                    downto 0));
464              end if;
465             end if;
466            when State_StoreSP =>
467             if in_mem_busy='0' then
468              begin_inst <= '1';
469              idim_flag <= '0';
470              state <= State_StoreSP2;
471
472              mem_writeEnable <= '1';
473              mem_addr <= std_logic_vector(sp+spOffset);
474              mem_write <= std_logic_vector(stackA);
475              stackA <= stackB;
476              sp <= incSp;
477             end if;
478
479
480            when State_LoadSP =>
481             if in_mem_busy='0' then
482              begin_inst <= '1';
483              idim_flag <= '0';
484              state <= State_LoadSP2;
485
486              sp <= decSp;
487              mem_writeEnable <= '1';
```

```
488        mem_addr <= std_logic_vector(incSp);
489        mem_write <= std_logic_vector(stackB);
490      end if;
491    when State_Emulate =>
492      if in_mem_busy='0' then
493        begin_inst <= '1';
494        idim_flag <= '0';
495        sp <= decSp;
496        mem_writeEnable <= '1';
497        mem_addr <= std_logic_vector(incSp);
498        mem_write <= std_logic_vector(stackB);
499        stackA <= (others => DontCareValue);
500        stackA(maxAddrBitIncIO downto 0) <= pc + 1;
501        stackB <= stackA;
502
503        -- The emulate address is:
504        --        98 7654 3210
505        -- 0000 00aa aaa0 0000
506        pc <= (others => '0');
507        pc(9 downto 5) <= unsigned(opcode(to_integer(pc(byteBits-1 downto 0)))(4 downto 0)
             );
508        state <= State_Fetch;
509      end if;
510    when State_Callpcrel =>
511      if in_mem_busy='0' then
512        begin_inst <= '1';
513        idim_flag <= '0';
514        stackA <= (others => DontCareValue);
515        stackA(maxAddrBitIncIO downto 0) <= pc + 1;
516
517        pc <= pc + stackA(maxAddrBitIncIO downto 0);
518        state <= State_Fetch;
519      end if;
520    when State_Call =>
521      if in_mem_busy='0' then
522        begin_inst <= '1';
523        idim_flag <= '0';
524        stackA <= (others => DontCareValue);
525        stackA(maxAddrBitIncIO downto 0) <= pc + 1;
526        pc <= stackA(maxAddrBitIncIO downto 0);
527        state <= State_Fetch;
528      end if;
529    when State_AddSP =>
530      if in_mem_busy='0' then
531        begin_inst <= '1';
532        idim_flag <= '0';
533        state <= State_AddSP2;
534
535        mem_readEnable <= '1';
536        mem_addr <= std_logic_vector(sp+spOffset);
537      end if;
538    when State_PushSP =>
539      if in_mem_busy='0' then
540        begin_inst <= '1';
541        idim_flag <= '0';
542        pc <= pc + 1;
543
544        sp <= decSp;
545        stackA <= (others => '0');
546        stackA(maxAddrBitIncIO downto minAddrBit) <= sp;
547        stackB <= stackA;
548        mem_writeEnable <= '1';
549        mem_addr <= std_logic_vector(incSp);
550        mem_write <= std_logic_vector(stackB);
551      end if;
552    when State_PopPC =>
553      if in_mem_busy='0' then
554        begin_inst <= '1';
555        idim_flag <= '0';
556        pc <= stackA(maxAddrBitIncIO downto 0);
557        sp <= incSp;
558
559        mem_writeEnable <= '1';
560        mem_addr <= std_logic_vector(incSp);
561        mem_write <= std_logic_vector(stackB);
562        state <= State_Resync;
563      end if;
564    when State_PopPCRel =>
565      if in_mem_busy='0' then
566        begin_inst <= '1';
567        idim_flag <= '0';
568        pc <= stackA(maxAddrBitIncIO downto 0) + pc;
569        sp <= incSp;
570
571        mem_writeEnable <= '1';
572        mem_addr <= std_logic_vector(incSp);
573        mem_write <= std_logic_vector(stackB);
```

```
574           state <= State_Resync;
575         end if;
576       when State_Add =>
577         if in_mem_busy='0' then
578          begin_inst <= '1';
579          idim_flag <= '0';
580          stackA <= stackA + stackB;
581
582          mem_readEnable <= '1';
583          mem_addr <= std_logic_vector(incIncSp);
584          sp <= incSp;
585          state <= State_Popped;
586         end if;
587       when State_Sub =>
588         if in_mem_busy='0' then
589          begin_inst <= '1';
590          idim_flag <= '0';
591          binaryOpResult <= stackB - stackA;
592          state <= State_BinaryOpResult;
593         end if;
594       when State_Pop =>
595         if in_mem_busy='0' then
596          begin_inst <= '1';
597          idim_flag <= '0';
598          mem_addr <= std_logic_vector(incIncSp);
599          mem_readEnable <= '1';
600          sp <= incSp;
601          stackA <= stackB;
602          state <= State_Popped;
603         end if;
604       when State_PopDown =>
605         if in_mem_busy='0' then
606          -- PopDown leaves top of stack unchanged
607          begin_inst <= '1';
608          idim_flag <= '0';
609          mem_addr <= std_logic_vector(incIncSp);
610          mem_readEnable <= '1';
611          sp <= incSp;
612          state <= State_Popped;
613         end if;
614       when State_Or =>
615         if in_mem_busy='0' then
616          begin_inst <= '1';
617          idim_flag <= '0';
618          stackA <= stackA or stackB;
619          mem_readEnable <= '1';
620          mem_addr <= std_logic_vector(incIncSp);
621          sp <= incSp;
622          state <= State_Popped;
623         end if;
624       when State_And =>
625         if in_mem_busy='0' then
626          begin_inst <= '1';
627          idim_flag <= '0';
628
629          stackA <= stackA and stackB;
630          mem_readEnable <= '1';
631          mem_addr <= std_logic_vector(incIncSp);
632          sp <= incSp;
633          state <= State_Popped;
634         end if;
635       when State_Eq =>
636         if in_mem_busy='0' then
637          begin_inst <= '1';
638          idim_flag <= '0';
639
640                   binaryOpResult <= (others => '0');
641                       if (stackA=stackB) then
642                        binaryOpResult(0) <= '1';
643                       end if;
644          state <= State_BinaryOpResult;
645         end if;
646                    when State_Ulessthan =>
647         if in_mem_busy='0' then
648          begin_inst <= '1';
649          idim_flag <= '0';
650
651                   binaryOpResult <= (others => '0');
652                       if (stackA<stackB) then
653                        binaryOpResult(0) <= '1';
654                       end if;
655          state <= State_BinaryOpResult;
656         end if;
657                    when State_Ulessthanorequal =>
658         if in_mem_busy='0' then
659          begin_inst <= '1';
660          idim_flag <= '0';
```

```
661
662                     binaryOpResult <= (others => '0');
663                        if (stackA<=stackB) then
664                         binaryOpResult(0) <= '1';
665                        end if;
666           state <= State_BinaryOpResult;
667          end if;
668                      when State_Lessthan =>
669          if in_mem_busy='0' then
670           begin_inst <= '1';
671           idim_flag <= '0';
672
673                     binaryOpResult <= (others => '0');
674                        if (signed(stackA)<signed(stackB)) then
675                         binaryOpResult(0) <= '1';
676                        end if;
677           state <= State_BinaryOpResult;
678          end if;
679                      when State_Lessthanorequal =>
680          if in_mem_busy='0' then
681           begin_inst <= '1';
682           idim_flag <= '0';
683
684                     binaryOpResult <= (others => '0');
685                        if (signed(stackA)<=signed(stackB)) then
686                         binaryOpResult(0) <= '1';
687                        end if;
688           state <= State_BinaryOpResult;
689          end if;
690         when State_Load =>
691          if in_mem_busy='0' then
692           begin_inst <= '1';
693           idim_flag <= '0';
694           state <= State_Load2;
695
696           mem_addr <= std_logic_vector(stackA(maxAddrBitIncIO downto minAddrBit));
697           mem_readEnable <= '1';
698          end if;
699
700         when State_Dup =>
701          if in_mem_busy='0' then
702           begin_inst <= '1';
703           idim_flag <= '0';
704           pc <= pc + 1;
705
706           sp <= decSp;
707           stackB <= stackA;
708           mem_write <= std_logic_vector(stackB);
709           mem_addr <= std_logic_vector(incSp);
710           mem_writeEnable <= '1';
711          end if;
712         when State_DupStackB =>
713          if in_mem_busy='0' then
714           begin_inst <= '1';
715           idim_flag <= '0';
716           pc <= pc + 1;
717
718           sp <= decSp;
719           stackA <= stackB;
720           stackB <= stackA;
721           mem_write <= std_logic_vector(stackB);
722           mem_addr <= std_logic_vector(incSp);
723           mem_writeEnable <= '1';
724          end if;
725         when State_Store =>
726          if in_mem_busy='0' then
727           begin_inst <= '1';
728           idim_flag <= '0';
729           pc <= pc + 1;
730           mem_addr <= std_logic_vector(stackA(maxAddrBitIncIO downto minAddrBit));
731           mem_write <= std_logic_vector(stackB);
732           mem_writeEnable <= '1';
733           sp <= incIncSp;
734           state <= State_Resync;
735          end if;
736         when State_PopSP =>
737          if in_mem_busy='0' then
738           begin_inst <= '1';
739           idim_flag <= '0';
740           pc <= pc + 1;
741
742           mem_write <= std_logic_vector(stackB);
743           mem_addr <= std_logic_vector(incSp);
744           mem_writeEnable <= '1';
745           sp <= stackA(maxAddrBitIncIO downto minAddrBit);
746           state <= State_Resync;
747          end if;
```

```
748         when State_Nop =>
749          begin_inst <= '1';
750          idim_flag <= '0';
751          pc <= pc + 1;
752         when State_Not =>
753          begin_inst <= '1';
754          idim_flag <= '0';
755          pc <= pc + 1;
756
757          stackA <= not stackA;
758         when State_Flip =>
759          begin_inst <= '1';
760          idim_flag <= '0';
761          pc <= pc + 1;
762
763          for i in 0 to wordSize-1 loop
764           stackA(i) <= stackA(wordSize-1-i);
765            end loop;
766         when State_AddTop =>
767          begin_inst <= '1';
768          idim_flag <= '0';
769          pc <= pc + 1;
770
771          stackA <= stackA + stackB;
772         when State_Shift =>
773          begin_inst <= '1';
774          idim_flag <= '0';
775          pc <= pc + 1;
776
777          stackA(wordSize-1 downto 1) <= stackA(wordSize-2 downto 0);
778          stackA(0) <= '0';
779         when State_Pushspadd =>
780          begin_inst <= '1';
781          idim_flag <= '0';
782          pc <= pc + 1;
783
784          stackA <= (others => '0');
785          stackA(maxAddrBitIncIO downto minAddrBit) <= stackA(maxAddrBitIncIO-minAddrBit
                 downto 0)+sp;
786         when State_Neqbranch =>
787          -- branches are almost always taken as they form loops
788          begin_inst <= '1';
789          idim_flag <= '0';
790          sp <= incIncSp;
791                       if (stackB/=0) then
792                        pc <= stackA(maxAddrBitIncIO downto 0) + pc;
793                       else
794                        pc <= pc + 1;
795                       end if;
796                       -- need to fetch stack again.
797          state <= State_Resync;
798                      when State_Mult =>
799          begin_inst <= '1';
800          idim_flag <= '0';
801
802          multA <= stackA;
803          multB <= stackB;
804          state <= State_Mult2;
805         when State_Break =>
806          report "Break_instruction_encountered" severity failure;
807          break <= '1';
808
809         when State_Loadb =>
810          if in_mem_busy='0' then
811           begin_inst <= '1';
812           idim_flag <= '0';
813           state <= State_Loadb2;
814
815           mem_addr <= std_logic_vector(stackA(maxAddrBitIncIO downto minAddrBit));
816           mem_readEnable <= '1';
817          end if;
818         when State_Storeb =>
819          if in_mem_busy='0' then
820           begin_inst <= '1';
821           idim_flag <= '0';
822           state <= State_Storeb2;
823
824           mem_addr <= std_logic_vector(stackA(maxAddrBitIncIO downto minAddrBit));
825           mem_readEnable <= '1';
826          end if;
827
828         when others =>
829          sp <= (others => DontCareValue);
830          report "Illegal_instruction" severity failure;
831          break <= '1';
832        end case;
833
```

```
834
835         when State_StoreSP2 =>
836          if in_mem_busy='0' then
837           mem_addr <= std_logic_vector(incSp);
838           mem_readEnable <= '1';
839           state <= State_Popped;
840          end if;
841         when State_LoadSP2 =>
842          if in_mem_busy='0' then
843           state <= State_LoadSP3;
844           mem_readEnable <= '1';
845           mem_addr <= std_logic_vector(sp+spOffset+1);
846          end if;
847         when State_LoadSP3 =>
848          if in_mem_busy='0' then
849           pc <= pc + 1;
850           state <= State_Execute;
851           stackB <= stackA;
852           stackA <= unsigned(mem_read);
853          end if;
854         when State_AddSP2 =>
855          if in_mem_busy='0' then
856           pc <= pc + 1;
857           state <= State_Execute;
858           stackA <= stackA + unsigned(mem_read);
859          end if;
860         when State_Load2 =>
861          if in_mem_busy='0' then
862           stackA <= unsigned(mem_read);
863           pc <= pc + 1;
864           state <= State_Execute;
865          end if;
866         when State_Loadb2 =>
867          if in_mem_busy='0' then
868           stackA <= (others => '0');
869           --stackA(7 downto 0) <= unsigned(mem_read(((wordBytes-1-to_integer(stackA(byteBits-1
                  downto 0)))*8+7) downto (wordBytes-1-to_integer(stackA(byteBits-1 downto 0)))
                  *8));
870           --HAVE TO WRITE LINE ABOVE AS CASE STRUCTURE TO COMPILE SUCCESSFULLY
871           case stackA(byteBits-1 downto 0) is
872            when "00" => stackA(7 downto 0) <= unsigned(mem_read(((wordBytes-1-0)*8+7) downto (
                  wordBytes-1-0)*8));
873            when "01" => stackA(7 downto 0) <= unsigned(mem_read(((wordBytes-1-1)*8+7) downto (
                  wordBytes-1-1)*8));
874            when "10" => stackA(7 downto 0) <= unsigned(mem_read(((wordBytes-1-2)*8+7) downto (
                  wordBytes-1-2)*8));
875            when "11" => stackA(7 downto 0) <= unsigned(mem_read(((wordBytes-1-3)*8+7) downto (
                  wordBytes-1-3)*8));
876            when others => null;
877           end case;
878           --CASE RE-WRITE ENDS HERE
879           pc <= pc + 1;
880           state <= State_Execute;
881          end if;
882         when State_Storeb2 =>
883          if in_mem_busy='0' then
884           mem_addr <= std_logic_vector(stackA(maxAddrBitIncIO downto minAddrBit));
885           mem_write <= mem_read;
886           --mem_write(((wordBytes-1-to_integer(stackA(byteBits-1 downto 0)))*8+7) downto (
                 wordBytes-1-to_integer(stackA(byteBits-1 downto 0)))*8) <= std_logic_vector(
                 stackB(7 downto 0));
887           --HAVE TO WRITE LINE ABOVE AS CASE STRUCTURE TO COMPILE SUCCESSFULLY
888           case stackA(byteBits-1 downto 0) is
889            when "00" => mem_write(((wordBytes-1-0)*8+7) downto (wordBytes-1-0)*8) <=
                  std_logic_vector(stackB(7 downto 0));
890            when "01" => mem_write(((wordBytes-1-1)*8+7) downto (wordBytes-1-1)*8) <=
                  std_logic_vector(stackB(7 downto 0));
891            when "10" => mem_write(((wordBytes-1-2)*8+7) downto (wordBytes-1-2)*8) <=
                  std_logic_vector(stackB(7 downto 0));
892            when "11" => mem_write(((wordBytes-1-3)*8+7) downto (wordBytes-1-3)*8) <=
                  std_logic_vector(stackB(7 downto 0));
893            when others => null;
894           end case;
895           --CASE RE-WRITE ENDS HERE
896           mem_writeEnable <= '1';
897           pc <= pc + 1;
898           sp <= incIncSp;
899           state <= State_Resync;
900          end if;
901         when State_Fetch =>
902          if in_mem_busy='0' then
903           mem_addr <= std_logic_vector(pc(maxAddrBitIncIO downto minAddrBit));
904           mem_readEnable <= '1';
905           state <= State_Decode;
906          end if;
907                     when State_Mult2 =>
908          state <= State_Mult3;
```

```
909                     when State_Mult3 =>
910         state <= State_Mult4;
911                     when State_Mult4 =>
912         state <= State_Mult5;
913                     when State_Mult5 =>
914         if in_mem_busy='0' then
915                    stackA <= multResult3;
916          mem_readEnable <= '1';
917          mem_addr <= std_logic_vector(incIncSp);
918          sp <= incSp;
919          state <= State_Popped;
920         end if;
921       when State_BinaryOpResult =>
922         state <= State_BinaryOpResult2;
923       when State_BinaryOpResult2 =>
924         mem_readEnable <= '1';
925         mem_addr <= std_logic_vector(incIncSp);
926         sp <= incSp;
927         stackA <= binaryOpResult2;
928         state <= State_Popped;
929       when State_Popped =>
930         if in_mem_busy='0' then
931          pc <= pc + 1;
932          stackB <= unsigned(mem_read);
933          state <= State_Execute;
934         end if;
935       when others =>
936         sp <= (others => DontCareValue);
937         report "Illegal_state" severity failure;
938         break <= '1';
939     end case;
940    end if;
941  end process;
942
943
944
945 end behave;
```

B.2 ZPU memory module

dram.vhdl

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5
6  library work;
7  use work.zpu_config.all;
8  use work.zpupkg.all;
9
10 entity dram is
11 port (clk : in std_logic;
12 areset : std_logic;
13   mem_writeEnable : in std_logic;
14   mem_readEnable : in std_logic;
15   mem_addr : in std_logic_vector(maxAddrBit downto 0);
16   mem_write : in std_logic_vector(wordSize-1 downto 0);
17   mem_read : out std_logic_vector(wordSize-1 downto 0);
18   mem_busy : out std_logic;
19   mem_writeMask : in std_logic_vector(wordBytes-1 downto 0));
20 end dram;
21
22 architecture dram_arch of dram is
23
24
25 type ram_type is array(natural range 0 to ((2**(maxAddrBitDRAM+1))/4)-1) of
       std_logic_vector(wordSize-1 downto 0);
26
27 shared variable ram : ram_type :=
28 (
29    0 => x"0b0b0b0b",
30    1 => x"82700b0b",
31    2 => x"80f8e40c",
32    3 => x"3a0b0b80",
33    4 => x"e7e20400",
34    5 => x"00000000",
35    6 => x"00000000",
36    7 => x"00000000",
37    8 => x"80088408",
38    9 => x"88080b0b",
```

```
39    10 => x"80e8af2d",
40
41  and so on down to...
42
43  4368 => x"ffffffff",
44  4369 => x"00000000",
45  4370 => x"ffffffff",
46  4371 => x"00000000",
47  4372 => x"00000000",
48   others => x"00000000"
49  );
50
51  begin
52
53  mem_busy<=mem_readEnable; -- we're done on the cycle after we serve the read request
54
55  process (clk, areset)
56  begin
57    if areset = '1' then
58    elsif (clk'event and clk = '1') then
59     if (mem_writeEnable = '1') then
60      ram(to_integer(unsigned(mem_addr(maxAddrBit downto minAddrBit)))) := mem_write;
61     end if;
62    if (mem_readEnable = '1') then
63     mem_read <= ram(to_integer(unsigned(mem_addr(maxAddrBit downto minAddrBit))));
64    end if;
65   end if;
66  end process;
67
68
69
70
71  end dram_arch;
```

B.3 ZPU testbench

sim_fpga_top.vhdl

```
1   ----------------------------------------------------------------------------------
2   -- Company:
3   -- Engineer:
4   --
5   -- Create Date:    20:15:31 04/14/05
6   -- Design Name:
7   -- Module Name:    fpga_top - behave
8   -- Project Name:
9   -- Target Device:
10  -- Tool versions:
11  -- Description:
12  --
13  -- Dependencies:
14  --
15  -- Revision:
16  -- Revision 0.01 - File Created
17  -- Additional Comments:
18  --
19  ----------------------------------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22
23  ---- Uncomment the following library declaration if instantiating
24  ---- any Xilinx primitives in this code.
25  --library UNISIM;
26  --use UNISIM.VComponents.all;
27
28  library work;
29  use work.zpu_config.all;
30
31  entity fpga_top is
32  end fpga_top;
33
34  use work.zpupkg.all;
35
36  architecture behave of fpga_top is
37
38
39  signal clk : std_logic;
40
41  signal areset   : std_logic := '1';
42
43
```

```
44  component zpu_io is
45    generic (
46            log_file:        string  := "log.txt"
47           );
48    port(
49          clk         : in std_logic;
50          areset        : in std_logic;
51    busy : out std_logic;
52    writeEnable : in std_logic;
53    readEnable : in std_logic;
54    write : in std_logic_vector(wordSize-1 downto 0);
55    read : out std_logic_vector(wordSize-1 downto 0);
56    addr : in std_logic_vector(maxAddrBit downto minAddrBit)
57    );
58  end component;
59
60
61
62
63
64  signal      mem_busy : std_logic;
65  signal      mem_read : std_logic_vector(wordSize-1 downto 0);
66  signal      mem_write : std_logic_vector(wordSize-1 downto 0);
67  signal      mem_addr : std_logic_vector(maxAddrBitIncIO downto 0);
68  signal      mem_writeEnable : std_logic;
69  signal      mem_readEnable : std_logic;
70  signal      mem_writeMask: std_logic_vector(wordBytes-1 downto 0);
71
72  signal      enable : std_logic;
73
74  signal      dram_mem_busy : std_logic;
75  signal      dram_mem_read : std_logic_vector(wordSize-1 downto 0);
76  signal      dram_mem_write : std_logic_vector(wordSize-1 downto 0);
77  signal      dram_mem_writeEnable : std_logic;
78  signal      dram_mem_readEnable : std_logic;
79  signal      dram_mem_writeMask: std_logic_vector(wordBytes-1 downto 0);
80
81
82  signal       io_busy : std_logic;
83
84  signal      io_mem_read : std_logic_vector(wordSize-1 downto 0);
85  signal      io_mem_writeEnable : std_logic;
86  signal      io_mem_readEnable : std_logic;
87
88
89  signal      dram_ready : std_logic;
90  signal      io_ready : std_logic;
91  signal      io_reading : std_logic;
92
93
94  signal break : std_logic;
95
96  begin
97   zpu: zpu_core port map (
98    clk => clk ,
99     areset => areset,
100    enable => enable,
101    in_mem_busy => mem_busy,
102    mem_read => mem_read,
103    mem_write => mem_write,
104   out_mem_addr => mem_addr,
105   out_mem_writeEnable => mem_writeEnable,
106   out_mem_readEnable => mem_readEnable,
107    mem_writeMask => mem_writeMask,
108    interrupt => '0',
109    break  => break);
110
111  dram_imp: dram port map (
112   clk => clk ,
113   areset => areset,
114    mem_busy => dram_mem_busy,
115    mem_read => dram_mem_read,
116    mem_write => mem_write,
117   mem_addr => mem_addr(maxAddrBit downto 0),
118   mem_writeEnable => dram_mem_writeEnable,
119   mem_readEnable => dram_mem_readEnable,
120    mem_writeMask => mem_writeMask);
121
122
123  ioMap: zpu_io port map (
124         clk => clk,
125    areset => areset,
126   busy => io_busy,
127   writeEnable => io_mem_writeEnable,
128   readEnable => io_mem_readEnable,
129   write => mem_write(wordSize-1 downto 0),
130   read => io_mem_read,
```

```
131      addr => mem_addr(maxAddrBit downto minAddrBit)
132    );
133
134    dram_mem_writeEnable <= mem_writeEnable and not mem_addr(ioBit);
135    dram_mem_readEnable <= mem_readEnable and not mem_addr(ioBit);
136    io_mem_writeEnable <= mem_writeEnable and mem_addr(ioBit);
137    io_mem_readEnable <= mem_readEnable and mem_addr(ioBit);
138    mem_busy <= io_busy or dram_mem_busy or io_busy;
139
140
141
142    -- Memory reads either come from IO or DRAM. We need to pick the right one.
143    memorycontrol:
144    process(dram_mem_read, dram_ready, io_ready, io_mem_read)
145    begin
146      mem_read <= (others => 'U');
147      if dram_ready='1' then
148        mem_read <= dram_mem_read;
149      end if;
150
151      if io_ready='1' then
152        mem_read <= io_mem_read;
153      end if;
154    end process;
155
156
157    io_ready <= (io_reading or io_mem_readEnable) and not io_busy;
158
159    memoryControlSync:
160    process(clk, areset)
161    begin
162      if areset = '1' then
163        enable <= '0';
164        io_reading <= '0';
165        dram_ready <= '0';
166      elsif (clk'event and clk = '1') then
167        enable <= '1';
168        io_reading <= io_busy or io_mem_readEnable;
169        dram_ready<=dram_mem_readEnable;
170
171      end if;
172    end process;
173
174    -- wiggle the clock @ 100MHz
175    clock : PROCESS
176        begin
177          clk <= '0';
178          wait for 5 ns;
179          clk <= '1';
180          wait for 5 ns;
181          areset <= '0';
182    end PROCESS clock;
183
184
185  end behave;
186
187  configuration CfgTestBench of fpga_top is
188    for behave
189      for zpu: zpu_core
190        use entity work.zpu_core(SYN_behave);
191      end for;
192    end for;
193  end CfgTestBench;
```

B.4 Plasma CPU core

mlite_cpu.vhd

```
1  ---------------------------------------------------------------------
2  -- TITLE: Plasma CPU core
3  -- AUTHOR: Steve Rhoads (rhoadss@yahoo.com)
4  -- DATE CREATED: 2/15/01
5  -- FILENAME: mlite_cpu.vhd
6  -- PROJECT: Plasma CPU core
7  -- COPYRIGHT: Software placed into the public domain by the author.
8  --     Software 'as is' without warranty. Author liable for nothing.
9  -- NOTE: MIPS(tm) and MIPS I(tm) are registered trademarks of MIPS
10 --     Technologies. MIPS Technologies does not endorse and is not
11 --     associated with this project.
12 -- DESCRIPTION:
13 --     Top level VHDL document that ties the nine other entities together.
```

```
14  --
15  -- Executes all MIPS I(tm) opcodes but exceptions and non-aligned
16  -- memory accesses.  Based on information found in:
17  --    "MIPS RISC Architecture" by Gerry Kane and Joe Heinrich
18  --    and "The Designer's Guide to VHDL" by Peter J. Ashenden
19  --
20  -- The CPU is implemented as a two or three stage pipeline.
21  -- An add instruction would take the following steps (see cpu.gif):
22  -- Stage #0:
23  --    1.  The "pc_next" entity passes the program counter (PC) to the
24  --        "mem_ctrl" entity which fetches the opcode from memory.
25  -- Stage #1:
26  --    2.  The memory returns the opcode.
27  -- Stage #2:
28  --    3.  "Mem_ctrl" passes the opcode to the "control" entity.
29  --    4.  "Control" converts the 32-bit opcode to a 60-bit VLWI opcode
30  --        and sends control signals to the other entities.
31  --    5.  Based on the rs_index and rt_index control signals, "reg_bank"
32  --        sends the 32-bit reg_source and reg_target to "bus_mux".
33  --    6.  Based on the a_source and b_source control signals, "bus_mux"
34  --        multiplexes reg_source onto a_bus and reg_target onto b_bus.
35  -- Stage #3 (part of stage #2 if using two stage pipeline):
36  --    7.  Based on the alu_func control signals, "alu" adds the values
37  --        from a_bus and b_bus and places the result on c_bus.
38  --    8.  Based on the c_source control signals, "bus_bux" multiplexes
39  --        c_bus onto reg_dest.
40  --    9.  Based on the rd_index control signal, "reg_bank" saves
41  --        reg_dest into the correct register.
42  -- Stage #3b:
43  --   10.  Read or write memory if needed.
44  --
45  -- All signals are active high.
46  -- Here are the signals for writing a character to address 0xffff
47  -- when using a two stage pipeline:
48  --
49  -- Program:
50  -- addr     value  opcode
51  -- =============================
52  --   3c: 00000000  nop
53  --   40: 34040041  li $a0,0x41
54  --   44: 3405ffff  li $a1,0xffff
55  --   48: a0a40000  sb $a0,0($a1)
56  --   4c: 00000000  nop
57  --   50: 00000000  nop
58  --
59  --        intr_in                             mem_pause
60  --  reset_in                                byte_we     Stages
61  --      ns          address   data_w     data_r        40 44 48 4c 50
62  --   3600  0  0  00000040   00000000   34040041  0  0   1
63  --   3700  0  0  00000044   00000000   3405FFFF  0  0   2  1
64  --   3800  0  0  00000048   00000000   A0A40000  0  0      2  1
65  --   3900  0  0  0000004C   41414141   00000000  0  0         2  1
66  --   4000  0  0  0000FFFC   41414141   XXXXXX41  1  0         3  2
67  --   4100  0  0  00000050   00000000   00000000  0  0               1
68  ---------------------------------------------------------------------
69  library ieee;
70  use work.mlite_pack.all;
71  use ieee.std_logic_1164.all;
72  use ieee.std_logic_unsigned.all;
73
74  entity mlite_cpu is
75     generic(memory_type     : string  := "XILINX_16X"; --ALTERA_LPM, or DUAL_PORT_
76             mult_type       : string  := "DEFAULT"; --AREA_OPTIMIZED
77             shifter_type    : string  := "DEFAULT"; --AREA_OPTIMIZED
78             alu_type        : string  := "DEFAULT"; --AREA_OPTIMIZED
79             pipeline_stages : natural := 3); --2 or 3
80     port(clk          : in std_logic;
81          reset_in     : in std_logic;
82          intr_in      : in std_logic;
83
84          address_next : out std_logic_vector(31 downto 2); --for synch ram
85          byte_we_next : out std_logic_vector(3 downto 0);
86
87          address      : out std_logic_vector(31 downto 2);
88          byte_we      : out std_logic_vector(3 downto 0);
89          data_w       : out std_logic_vector(31 downto 0);
90          data_r       : in std_logic_vector(31 downto 0);
91          mem_pause    : in std_logic);
92  end; --entity mlite_cpu
93
94  architecture logic of mlite_cpu is
95     --When using a two stage pipeline "sigD <= sig".
96     --When using a three stage pipeline "sigD <= sig when rising_edge(clk)",
97     --  so sigD is delayed by one clock cycle.
98     signal opcode         : std_logic_vector(31 downto 0);
99     signal rs_index       : std_logic_vector(5 downto 0);
100    signal rt_index       : std_logic_vector(5 downto 0);
```

```
101     signal rd_index         : std_logic_vector(5 downto 0);
102     signal rd_indexD        : std_logic_vector(5 downto 0);
103     signal reg_source       : std_logic_vector(31 downto 0);
104     signal reg_target       : std_logic_vector(31 downto 0);
105     signal reg_dest         : std_logic_vector(31 downto 0);
106     signal reg_destD        : std_logic_vector(31 downto 0);
107     signal a_bus            : std_logic_vector(31 downto 0);
108     signal a_busD           : std_logic_vector(31 downto 0);
109     signal b_bus            : std_logic_vector(31 downto 0);
110     signal b_busD           : std_logic_vector(31 downto 0);
111     signal c_bus            : std_logic_vector(31 downto 0);
112     signal c_alu            : std_logic_vector(31 downto 0);
113     signal c_shift          : std_logic_vector(31 downto 0);
114     signal c_mult           : std_logic_vector(31 downto 0);
115     signal c_memory         : std_logic_vector(31 downto 0);
116     signal imm              : std_logic_vector(15 downto 0);
117     signal pc_future        : std_logic_vector(31 downto 2);
118     signal pc_current       : std_logic_vector(31 downto 2);
119     signal pc_plus4         : std_logic_vector(31 downto 2);
120     signal alu_func         : alu_function_type;
121     signal alu_funcD        : alu_function_type;
122     signal shift_func       : shift_function_type;
123     signal shift_funcD      : shift_function_type;
124     signal mult_func        : mult_function_type;
125     signal mult_funcD       : mult_function_type;
126     signal branch_func      : branch_function_type;
127     signal take_branch      : std_logic;
128     signal a_source         : a_source_type;
129     signal b_source         : b_source_type;
130     signal c_source         : c_source_type;
131     signal pc_source        : pc_source_type;
132     signal mem_source       : mem_source_type;
133     signal pause_mult       : std_logic;
134     signal pause_ctrl       : std_logic;
135     signal pause_pipeline   : std_logic;
136     signal pause_any        : std_logic;
137     signal pause_non_ctrl   : std_logic;
138     signal pause_bank       : std_logic;
139     signal nullify_op       : std_logic;
140     signal intr_enable      : std_logic;
141     signal intr_signal      : std_logic;
142     signal exception_sig    : std_logic;
143     signal reset_reg        : std_logic_vector(3 downto 0);
144     signal reset            : std_logic;
145  begin  --architecture
146
147     pause_any <= (mem_pause or pause_ctrl) or (pause_mult or pause_pipeline);
148     pause_non_ctrl <= (mem_pause or pause_mult) or pause_pipeline;
149     pause_bank <= (mem_pause or pause_ctrl or pause_mult) and not pause_pipeline;
150     nullify_op <= '1' when (pc_source = FROM_LBRANCH and take_branch = '0')
151                            or intr_signal = '1' or exception_sig = '1'
152                            else '0';
153     c_bus <= c_alu or c_shift or c_mult;
154     reset <= '1' when reset_in = '1' or reset_reg /= "1111" else '0';
155
156     --synchronize reset and interrupt pins
157     intr_proc: process(clk, reset_in, reset_reg, intr_in, intr_enable,
158        pc_source, pc_current, pause_any)
159     begin
160        if reset_in = '1' then
161           reset_reg <= "0000";
162           intr_signal <= '0';
163        elsif rising_edge(clk) then
164           if reset_reg /= "1111" then
165              reset_reg <= reset_reg + 1;
166           end if;
167
168           --don't try to interrupt a multi-cycle instruction
169           if pause_any = '0' then
170              if intr_in = '1' and intr_enable = '1' and
171                    pc_source = FROM_INC4 then
172                 --the epc will contain pc+4
173                 intr_signal <= '1';
174              else
175                 intr_signal <= '0';
176              end if;
177           end if;
178
179        end if;
180     end process;
181
182     u1_pc_next: pc_next PORT MAP (
183          clk          => clk,
184          reset_in     => reset,
185          take_branch  => take_branch,
186          pause_in     => pause_any,
187          pc_new       => c_bus(31 downto 2),
```

```
188          opcode25_0    => opcode(25 downto 0),
189          pc_source     => pc_source,
190          pc_future     => pc_future,
191          pc_current    => pc_current,
192          pc_plus4      => pc_plus4);
193
194     u2_mem_ctrl: mem_ctrl
195        PORT MAP (
196          clk           => clk,
197          reset_in      => reset,
198          pause_in      => pause_non_ctrl,
199          nullify_op    => nullify_op,
200          address_pc    => pc_future,
201          opcode_out    => opcode,
202
203          address_in    => c_bus,
204          mem_source    => mem_source,
205          data_write    => reg_target,
206          data_read     => c_memory,
207          pause_out     => pause_ctrl,
208
209          address_next => address_next,
210          byte_we_next => byte_we_next,
211
212          address       => address,
213          byte_we       => byte_we,
214          data_w        => data_w,
215          data_r        => data_r);
216
217     u3_control: control PORT MAP (
218          opcode        => opcode,
219          intr_signal   => intr_signal,
220          rs_index      => rs_index,
221          rt_index      => rt_index,
222          rd_index      => rd_index,
223          imm_out       => imm,
224          alu_func      => alu_func,
225          shift_func    => shift_func,
226          mult_func     => mult_func,
227          branch_func   => branch_func,
228          a_source_out => a_source,
229          b_source_out => b_source,
230          c_source_out => c_source,
231          pc_source_out=> pc_source,
232          mem_source_out=> mem_source,
233          exception_out=> exception_sig);
234
235     u4_reg_bank: reg_bank
236        generic map(memory_type => memory_type)
237        port map (
238          clk              => clk,
239          reset_in         => reset,
240          pause            => pause_bank,
241          rs_index         => rs_index,
242          rt_index         => rt_index,
243          rd_index         => rd_indexD,
244          reg_source_out => reg_source,
245          reg_target_out => reg_target,
246          reg_dest_new     => reg_destD,
247          intr_enable      => intr_enable);
248
249     u5_bus_mux: bus_mux port map (
250          imm_in        => imm,
251          reg_source    => reg_source,
252          a_mux         => a_source,
253          a_out         => a_bus,
254
255          reg_target    => reg_target,
256          b_mux         => b_source,
257          b_out         => b_bus,
258
259          c_bus         => c_bus,
260          c_memory      => c_memory,
261          c_pc          => pc_current,
262          c_pc_plus4    => pc_plus4,
263          c_mux         => c_source,
264          reg_dest_out => reg_dest,
265
266          branch_func   => branch_func,
267          take_branch   => take_branch);
268
269     u6_alu: alu
270        generic map (alu_type => alu_type)
271        port map (
272          a_in          => a_busD,
273          b_in          => b_busD,
274          alu_function => alu_funcD,
```

```
275          c_alu          => c_alu);
276
277     u7_shifter: shifter
278        generic map (shifter_type => shifter_type)
279        port map (
280          value         => b_busD,
281          shift_amount => a_busD(4 downto 0),
282          shift_func    => shift_funcD,
283          c_shift       => c_shift);
284
285     u8_mult: mult
286        generic map (mult_type => mult_type)
287        port map (
288          clk        => clk,
289          reset_in   => reset,
290          a          => a_busD,
291          b          => b_busD,
292          mult_func => mult_funcD,
293          c_mult     => c_mult,
294          pause_out => pause_mult);
295
296     pipeline2: if pipeline_stages <= 2 generate
297        a_busD <= a_bus;
298        b_busD <= b_bus;
299        alu_funcD <= alu_func;
300        shift_funcD <= shift_func;
301        mult_funcD <= mult_func;
302        rd_indexD <= rd_index;
303        reg_destD <= reg_dest;
304        pause_pipeline <= '0';
305     end generate; --pipeline2
306
307     pipeline3: if pipeline_stages > 2 generate
308        --When operating in three stage pipeline mode, the following signals
309        --are delayed by one clock cycle:  a_bus, b_bus, alu/shift/mult_func,
310        --c_source, and rd_index.
311     u9_pipeline: pipeline port map (
312          clk              => clk,
313          reset            => reset,
314          a_bus            => a_bus,
315          a_busD           => a_busD,
316          b_bus            => b_bus,
317          b_busD           => b_busD,
318          alu_func         => alu_func,
319          alu_funcD        => alu_funcD,
320          shift_func       => shift_func,
321          shift_funcD      => shift_funcD,
322          mult_func        => mult_func,
323          mult_funcD       => mult_funcD,
324          reg_dest         => reg_dest,
325          reg_destD        => reg_destD,
326          rd_index         => rd_index,
327          rd_indexD        => rd_indexD,
328
329          rs_index         => rs_index,
330          rt_index         => rt_index,
331          pc_source        => pc_source,
332          mem_source       => mem_source,
333          a_source         => a_source,
334          b_source         => b_source,
335          c_source         => c_source,
336          c_bus            => c_bus,
337          pause_any        => pause_any,
338          pause_pipeline => pause_pipeline);
339
340     end generate; --pipeline3
341
342  end; --architecture logic
```

B.5 Plasma memory module

ram.vhd

```
1  ---------------------------------------------------------------------
2  -- TITLE: Random Access Memory
3  -- AUTHOR: Steve Rhoads (rhoadss@yahoo.com)
4  -- DATE CREATED: 4/21/01
5  -- FILENAME: ram.vhd
6  -- PROJECT: Plasma CPU core
7  -- COPYRIGHT: Software placed into the public domain by the author.
8  --     Software 'as is' without warranty.  Author liable for nothing.
```

```
9   -- DESCRIPTION:
10  --    Implements the RAM, reads the executable from either "code.txt",
11  --    or for Altera "code[0-3].hex".
12  --    Modified from "The Designer's Guide to VHDL" by Peter J. Ashenden
13  ---------------------------------------------------------------------
14  library ieee;
15  use ieee.std_logic_1164.all;
16  use ieee.std_logic_misc.all;
17  use ieee.std_logic_arith.all;
18  use ieee.std_logic_unsigned.all;
19  use ieee.std_logic_textio.all;
20  use std.textio.all;
21  use work.mlite_pack.all;
22
23  entity ram is
24     generic(memory_type : string := "DEFAULT");
25     port(clk               : in std_logic;
26          enable            : in std_logic;
27          write_byte_enable : in std_logic_vector(3 downto 0);
28          address           : in std_logic_vector(31 downto 2);
29          data_write        : in std_logic_vector(31 downto 0);
30          data_read         : out std_logic_vector(31 downto 0));
31  end; --entity ram
32
33  architecture logic of ram is
34     constant ADDRESS_WIDTH   : natural := 13;
35  begin
36
37     generic_ram:
38     if memory_type /= "ALTERA_LPM" generate
39     begin
40     --Simulate a synchronous RAM
41     ram_proc: process(clk, enable, write_byte_enable,
42           address, data_write) --mem_write, mem_sel
43        variable mem_size : natural := 2 ** ADDRESS_WIDTH;
44        variable data : std_logic_vector(31 downto 0);
45        subtype word is std_logic_vector(data_write'length-1 downto 0);
46        type storage_array is
47           array(natural range 0 to mem_size/4 - 1) of word;
48        variable storage : storage_array;
49        variable index : natural := 0;
50        file load_file : text open read_mode is "code.txt";
51        variable hex_file_line : line;
52     begin
53
54        --Load in the ram executable image
55        if index = 0 then
56           while not endfile(load_file) loop
57  --The following two lines had to be commented out for synthesis
58              readline(load_file, hex_file_line);
59              hread(hex_file_line, data);
60              storage(index) := data;
61              index := index + 1;
62           end loop;
63        end if;
64
65        if rising_edge(clk) then
66           index := conv_integer(address(ADDRESS_WIDTH-1 downto 2));
67           data := storage(index);
68
69           if enable = '1' then
70              if write_byte_enable(0) = '1' then
71                 data(7 downto 0) := data_write(7 downto 0);
72              end if;
73              if write_byte_enable(1) = '1' then
74                 data(15 downto 8) := data_write(15 downto 8);
75              end if;
76              if write_byte_enable(2) = '1' then
77                 data(23 downto 16) := data_write(23 downto 16);
78              end if;
79              if write_byte_enable(3) = '1' then
80                 data(31 downto 24) := data_write(31 downto 24);
81              end if;
82           end if;
83
84           if write_byte_enable /= "0000" then
85              storage(index) := data;
86           end if;
87        end if;
88
89        data_read <= data;
90     end process;
91     end generate; --generic_ram
92
93
94     altera_ram:
95     if memory_type = "ALTERA_LPM" generate
```

```
96        signal byte_we : std_logic_vector(3 downto 0);
97     begin
98        byte_we <= write_byte_enable when enable = '1' else "0000";
99        lpm_ram_io_component0 : lpm_ram_dq
100          GENERIC MAP (
101             intended_device_family => "UNUSED",
102             lpm_width => 8,
103             lpm_widthad => ADDRESS_WIDTH-2,
104             lpm_indata => "REGISTERED",
105             lpm_address_control => "REGISTERED",
106             lpm_outdata => "UNREGISTERED",
107             lpm_file => "code0.hex",
108             use_eab => "ON",
109             lpm_type => "LPM_RAM_DQ")
110          PORT MAP (
111             data    => data_write(31 downto 24),
112             address => address(ADDRESS_WIDTH-1 downto 2),
113             inclock => clk,
114             we      => byte_we(3),
115             q       => data_read(31 downto 24));
116
117       lpm_ram_io_component1 : lpm_ram_dq
118          GENERIC MAP (
119             intended_device_family => "UNUSED",
120             lpm_width => 8,
121             lpm_widthad => ADDRESS_WIDTH-2,
122             lpm_indata => "REGISTERED",
123             lpm_address_control => "REGISTERED",
124             lpm_outdata => "UNREGISTERED",
125             lpm_file => "code1.hex",
126             use_eab => "ON",
127             lpm_type => "LPM_RAM_DQ")
128          PORT MAP (
129             data    => data_write(23 downto 16),
130             address => address(ADDRESS_WIDTH-1 downto 2),
131             inclock => clk,
132             we      => byte_we(2),
133             q       => data_read(23 downto 16));
134
135       lpm_ram_io_component2 : lpm_ram_dq
136          GENERIC MAP (
137             intended_device_family => "UNUSED",
138             lpm_width => 8,
139             lpm_widthad => ADDRESS_WIDTH-2,
140             lpm_indata => "REGISTERED",
141             lpm_address_control => "REGISTERED",
142             lpm_outdata => "UNREGISTERED",
143             lpm_file => "code2.hex",
144             use_eab => "ON",
145             lpm_type => "LPM_RAM_DQ")
146          PORT MAP (
147             data    => data_write(15 downto 8),
148             address => address(ADDRESS_WIDTH-1 downto 2),
149             inclock => clk,
150             we      => byte_we(1),
151             q       => data_read(15 downto 8));
152
153       lpm_ram_io_component3 : lpm_ram_dq
154          GENERIC MAP (
155             intended_device_family => "UNUSED",
156             lpm_width => 8,
157             lpm_widthad => ADDRESS_WIDTH-2,
158             lpm_indata => "REGISTERED",
159             lpm_address_control => "REGISTERED",
160             lpm_outdata => "UNREGISTERED",
161             lpm_file => "code3.hex",
162             use_eab => "ON",
163             lpm_type => "LPM_RAM_DQ")
164          PORT MAP (
165             data    => data_write(7 downto 0),
166             address => address(ADDRESS_WIDTH-1 downto 2),
167             inclock => clk,
168             we      => byte_we(0),
169             q       => data_read(7 downto 0));
170
171    end generate; --altera_ram
172
173
174    --For XILINX see ram_xilinx.vhd
175
176 end; --architecture logic
```

B.6 Plasma test bench

`tbench.vhd`

```
1  ---------------------------------------------------------------------
2  -- TITLE: Test Bench
3  -- AUTHOR: Steve Rhoads (rhoadss@yahoo.com)
4  -- DATE CREATED: 4/21/01
5  -- FILENAME: tbench.vhd
6  -- PROJECT: Plasma CPU core
7  -- COPYRIGHT: Software placed into the public domain by the author.
8  --    Software 'as is' without warranty. Author liable for nothing.
9  -- DESCRIPTION:
10 --    This entity provides a test bench for testing the Plasma CPU core.
11 ---------------------------------------------------------------------
12 library ieee;
13 use ieee.std_logic_1164.all;
14 use work.mlite_pack.all;
15 use ieee.std_logic_unsigned.all;
16 
17 entity tbench is
18 end; --entity tbench
19 
20 architecture logic of tbench is
21    constant memory_type : string :=
22    "TRI_PORT_X";
23 --    "DUAL_PORT_";
24 --    "ALTERA_LPM";
25 --    "XILINX_16X";
26 
27    constant log_file  : string :=
28 --    "UNUSED";
29    "output.txt";
30 
31  --magical memory signals
32  --signal mem_fetching : std_logic := '0';
33  --signal mem_reading  : std_logic := '0';
34  --signal mem_writing  : std_logic := '0';
35 
36    signal clk         : std_logic := '1';
37    signal reset       : std_logic := '1';
38    signal interrupt   : std_logic := '0';
39    signal mem_write   : std_logic;
40    signal address     : std_logic_vector(31 downto 2);
41    signal data_write  : std_logic_vector(31 downto 0);
42    signal data_read   : std_logic_vector(31 downto 0);
43    signal pause1      : std_logic := '0';
44    signal pause2      : std_logic := '0';
45    signal pause       : std_logic;
46    signal no_ddr_start: std_logic;
47    signal no_ddr_stop : std_logic;
48    signal byte_we     : std_logic_vector(3 downto 0);
49    signal uart_write  : std_logic;
50    signal gpioA_in    : std_logic_vector(31 downto 0) := (others => '0');
51 begin  --architecture
52    --Uncomment the line below to test interrupts
53    interrupt <= '1' after 20 us when interrupt = '0' else '0' after 445 ns;
54 
55    clk   <= not clk after 50 ns;
56    reset <= '0' after 500 ns;
57    pause1 <= '1' after 700 ns when pause1 = '0' else '0' after 200 ns;
58    pause2 <= '1' after 300 ns when pause2 = '0' else '0' after 200 ns;
59    pause <= pause1 or pause2;
60    gpioA_in(20) <= not gpioA_in(20) after 200 ns; --E_RX_CLK
61    gpioA_in(19) <= not gpioA_in(19) after 20 us;  --E_RX_DV
62    gpioA_in(18 downto 15) <= gpioA_in(18 downto 15) + 1 after 400 ns; --E_RX_RXD
63    gpioA_in(14) <= not gpioA_in(14) after 200 ns; --E_TX_CLK
64 
65    u1_plasma: plasma
66       generic map (memory_type => memory_type,
67                    ethernet    => '1',
68                    use_cache   => '0',
69                    log_file    => log_file)
70       PORT MAP (
71          clk               => clk,
72          reset             => reset,
73          uart_read         => uart_write,
74          uart_write        => uart_write,
75 
76          address           => address,
77          byte_we           => byte_we,
78          data_write        => data_write,
79          data_read         => data_read,
80          mem_pause_in      => pause,
81          no_ddr_start      => no_ddr_start,
```

```
 82           no_ddr_stop         => no_ddr_stop,
 83
 84           gpio0_out           => open,
 85           gpioA_in            => gpioA_in);
 86    --introducing mem counter signals to keep track of how many instruction fetches,
 87    --mem reads and mem writes that has occured
 88    --mem_fetching => mem_fetching,
 89    --mem_reading => mem_reading,
 90     --mem_writing => mem_writing);
 91
 92     dram_proc: process(clk, address, byte_we, data_write, pause)
 93        constant ADDRESS_WIDTH : natural := 16;
 94        type storage_array is
 95           array(natural range 0 to (2 ** ADDRESS_WIDTH) / 4 - 1) of
 96           std_logic_vector(31 downto 0);
 97        variable storage : storage_array;
 98        variable data    : std_logic_vector(31 downto 0);
 99        variable index   : natural := 0;
100     begin
101        index := conv_integer(address(ADDRESS_WIDTH-1 downto 2));
102        data := storage(index);
103
104        if byte_we(0) = '1' then
105           data(7 downto 0) := data_write(7 downto 0);
106        end if;
107        if byte_we(1) = '1' then
108           data(15 downto 8) := data_write(15 downto 8);
109        end if;
110        if byte_we(2) = '1' then
111           data(23 downto 16) := data_write(23 downto 16);
112        end if;
113        if byte_we(3) = '1' then
114           data(31 downto 24) := data_write(31 downto 24);
115        end if;
116
117        if rising_edge(clk) then
118           if address(30 downto 28) = "001" and byte_we /= "0000" then
119              storage(index) := data;
120           end if;
121        end if;
122
123        if pause = '0' then
124           data_read <= data;
125        end if;
126     end process;
127
128
129  end; --architecture logic
```