Masters thesis
Design of a low-power platform for running an embedded operating system

Nicolai Ascanius Jørgensen
s971889

30th November 2003

Foreword

This is the masters thesis of Nicolai Ascanius Jørgensen.

The thesis was written at the Institute for Informatics and Mathematical Modeling (IMM) at the Technical University of Denmark (DTU). The thesis was supervised by Jan Madsen, IMM.

This report is accompanied by a CD with source code, data sheets and bibliography. If the CD is not present, please contact the author via e-mail at s971889@student.dtu.dk for any files that might be needed.

Sammenfatning

To eksisterende processorer er anvendt som grundlag for en lav-effekt platform til afvikling af det indlejrede operativsystem, TinyOS.

Den første processor er en MIPS processor skrevet i SystemC. Processoren blev udvidet med en coprocessor til afvikling af timer interrupts.
Den anden processor er en AVR processor oprindeligt skrevet i VHDL. Processoren er oversat til SystemC og nogle entiteter er blevet omskrevet.

Til verifikation, er et hamming kodning program og en TinyOS applikation blevet afviklet på begge platforme. Strømforbruget under afvikling er målt med Synopsys Power Compiler. Til afvikling af TinyOS på MIPS platformen, er en del af TinyOS blevet oversat fra AVR assembler til MIPS assembler.

MIPS platformen er syntetiseret til en FPGA og verificeret.

Begge platforme er clock gated med Synopsys Power Compiler.

Alt RTL design er skrevet med SystemC som HDL.

Abstract

Two existing processor cores have been used as the foundation of a low-power platform for running the embedded operating system, TinyOS.

The first core is a MIPS core written in SystemC. The processor was extended with a coprocessor for executing timer interrupts.
The second core is an AVR core written originally in VHDL. The core was translated to SystemC and some entities were re-written.

For verification, a hamming encoder program and a TinyOS application was executed on the platforms. The power consumption of the executions was measured using the Synopsys Power Compiler. For the MIPS core, a subsection of TinyOS was translated from AVR assembly to MIPS assembly.

The MIPS platform was synthesized to an FPGA and verified.

Both platforms were clock gated using the Synopsys Power Compiler.

All RTL design is written with the SystemC HDL.

8

Contents

Chapter 1

Word list

- **ABI** Application Binary Interface. A standard for compilers, linkers and assemblers to follow. This could be what registers to use for function parameters. A compiler conforms to a certain ABI standard when creating binaries.
- **APM** Advanced Power Management. A system for shutting down various parts of a personal computer after being idle for a pre-defined amount of time.
- **ACPI** Advanced Configuration and Power Interface. The next-generation APM system. An interface where halt and sleep modes are controlled by the operating system.
- **AVR** Advanced Virtual RISC (could also be **A**lf Egil Bogen, **V**egard Wollan **R**ISC, the creators of the AVR architecture). An 8-bit architecture.
- **Architecture** Architectures are processor designs, e.g. MIPS, 486, AVR, Itanium and G5.
- **CPI** Clock cycles Per Instruction.
- **CPLD** Complex Programmable Logic Device. A device similar to an FPGA but the content is retained when power is lost. The device is programmed in VHDL.
- **DBT** Dual Bit Type. An algorithm for high-level power estimation of a design. The name refers to the fact that bits are divided into two categories.
- **Data Hazard** A hazard that occurs in a pipelined processor when a value is read from memory and used in the following clock cycle. Because of the pipeline, the processor must wait for one clock cycle before using a register, if its value is read from memory.
- **DVS** Dynamic Voltage Scaling. The ability to use lower voltages to power a circuit at the cost of performance.
- **EDA** Electronic Design Automation. A generic term for programs that are used to create ICs and circuits.
- **ELF** Executable and Linkable Format. A standard for binary assembly files in UNIX.
- **FPGA** Field Programmable Gate Array. A device that is programmed using VHDL. On most FPGAs the programming is lost when power is lost.

- **FSM** Finite State Machine. A set of states with logic that triggers state transitions on e.g. rising edge of certain signals.
- **GCC** GNU Compiler Collection. An open-source compiler collection that compiles C-code and other languages on many platforms.
- **Glitching** When a primary input to a combinatorical entity change, signals through to the outputs change accordingly. When more primary (or secondary) inputs change, internal signal may go high/low for a small fraction of a clock cycle until the entity has stabilized. The "spikes" that appear in a waveform are glitches.
- **GPL** GNU Public License. An open source standard license from the GNU project.
- **Grey encoding** An encoding scheme for a binary number sequence where only one bit alters pr. transition, i.e. the hamming distance is 1.
- **Hamming distance** The hamming distance of two binary numbers is the number of bits that are different.
- **Hamming encoding** Hamming encoding is a Dual-error-detecting Single-error-correcting (DED/SEC) algorithm that is used in digital logic when reading and writing data through a noisy source, e.g. an Ethernet connection or a CD-ROM.
- **HDL** Hardware Description Language. A generic term for languages like VHDL, Verilog and SystemC.
- **ISA** Instruction Set Architecture. ISA is the instructions and registers that are available.
- **ISDB** Integrated Signal Data Base. A file format that can be used to create waveforms.
- **ISE** Integrated Software Environment. A Xilinx program for synthesis running on several platforms.
- **JTAG** Joint Test Access Group. A protocol for doing scan-chain analysis of a device. JTAG can also be used for programming FPGAs and CPLDs. JTAG is also known as IEEE standard 1149.1
- **LUT** Look-up table. A programmable gate with N inputs and usually one or two outputs.
- **Markov Diagram** A state diagram with probabilities listed for each transition from one state to another.
- **MIPS** Millions of Instructions Per Second.
- **MIPS32$^{\text{TM}}$** A 32-bit RISC architecture from MIPS Technologies.
- **NEST** Network Embedded Systems Technology. A programming platform that is a part of TinyOS.
- **One-hot encoding** An encoding scheme where only one bit at a time is set in a N-bit binary signal. N equals the number of possible values in the encoding.

- **OpenCores.org** A website where engineers can publish hobby projects for everyone to use. License agreements are included in most projects but non-commercial use is always "free".
- **Parasitic current** Parasitic currents run everywhere else but through planned wires in an IC. Parasitic currents are unwanted currents.
- **PIC** Position Independent Code. A method for organizing code in ELF file without using actual/virtual addresses, i.e. all function calls are done as relative jumps.
- **Place-and-route** A technique for analyzing a design for FPGA. The process involves assigning design elements to resources on the FPGA and doing the routing between them.
- **Platform** A platform is the entire embedded system, i.e the processor, the I/O, memory and sensors.
- **Process (1)** A process in any HDL is a set of logic expressions that will be executed when any signal in the process' sensitivity list changes.
- **Process (2)** A process is the production process of a chip manufacturer. Processes are defined by a set of minimum sizes for the transistors and the metals connecting them. The smallest size wire decides the short name of the process, e.g .25 μm and .13 μm.
- **RISC** Reduced Instruction Set Computer.
- **RTL** Register Transfer Level. A level of observation where only the values transferred and their destinations are looked at.
- **SAIF** Switching Activity Interchange Format. A file format containing information about activity in ports and registers.
- **Sensor network** A set of sensors that communicate with each other to gather the sensed data in one end of the network for read-out of the data.
- **Synthesis** A method for converting HDL code to a logic data-path.
- **Toggle rate** Toggle rate is a sum of rising and falling edges and glitches on a port or register.
- **VCD** Value Change Dump. A file format that can be used to create waveforms.
- **VHDL** VHSIC Hardware Description Language, where VHSIC is an acronym for Very High Speed Integrated Circuit. VHDL is a language for describing logic.
- **VLSI** Very Large Scale Integration. Integrated circuits with millions of gates.
- **WIF** Wave Interchange Format. A file format that can be used to create waveforms.

Chapter 2

Introduction

This project aims to implement a platform for running an embedded operating system. A platform consists of a processor and peripheral modules such as memory, displays and sensors.
Low-power is important in embedded devices as they are often battery powered and hence their lifespan is proportional to how effectively the use power.
One form of embedded devices is sensor networks. Sensor networks are composed of many, cheap sensors. To keep the battery size and cost to a minimum, the battery power must be used optimally.
In this project, special attention will be given to the TinyOS operating system (OS). TinyOS has been called a *truly minimalistic operating system*, in the sense that TinyOS provides a kernel that an application is compiled into to become a self-contained embedded OS in stead of being an OS that executes an application on top of the kernel.

2.1 Motivation

Sensor networks are in need of low-power operation as they are often installed non-permanently in high quantities to e.g. measure temperatures in a certain area. To avoid high-cost power-wiring these sensors are often battery or solar powered. If the sensors are battery powered, the battery is dimensioned according to the requested lifetime of the sensor network.

Low-power design is motivated by 3 problems:

- Battery capacity
- Heat-dissipation
- Environment

The first concern is that battery capacity is only slowly increasing, while System-on-Chip (SoC) techniques are adding more features to embedded devices, hence increasing power consumption.
Heat-dissipation is a problem since a device that uses power, gets hotter. If the temperature gets too high, then the device will be destroyed, unless measures are taken. Embedded devices, e.g. a laptop computer, are often small and therefore have limited options for cooling.
The environment is often an issue when dealing with electronics. Using power pollutes in one way or another so using less power is considered better for the environment.

2.2 Background

This project will base the work on two donor processor cores. The first is a MIPS r2000 processor [36] designed in SystemC. The other is an AVR micro-controller [31] designed in VHDL and released as an OpenCores project.
Both processors are known to be working with smaller programs, but extensions will have to be made before running TinyOS. These extensions will be covered in the implementation chapter.

2.3 Project goals

This project focus on the best utilization of power and reduction of wasted power in hardware design. For this, the two platforms will be evaluated, MIPS and AVR. The MIPS core already exists in a synthesizable SystemC version while the AVR core must be translated from VHDL. The power consumption of the two platforms will be evaluated for different applications, including the embedded operating system, TinyOS.
For the implementation of the cores, SystemC will be used. SystemC can be combined with other C programs, which is of great value when running applications in RTL simulation.

2.4 How to read this report

The report contains 8 major chapters:
Chapter 2 is this chapter where an introduction is given to the work being presented in the following chapters.
Chapter 3 contains theory of low-power design explaining what causes power consumption and how to reduce it.
Chapter 4 describes how power is measured using high-level analysis programs and activity information.
Chapter 5 describes the work done to create two processors in SystemC. The implementation and testing will be covered for the SystemC version and the synthesized version in VHDL.
Chapter 6 covers the embedded operating system, TinyOS, and describes the effort to convert TinyOS to the MIPS architecture.
Chapter 7 explain how to use the power compiler and how to use the output from the power compiler properly.
Chapter 8 is the summary of the report where results are presented and commented.
Chapter 9 sums up the presented work and achievements. Finally, a recommendation for future work is given.

To assist the reader, a word list is provided in chapter 1.
To ease reading, all function names, signal names, file names, data types and entity names are written with a mono-spaced font, e.g. `output_port`.

Tasks that do not fit in the above chapters are described in the appendices.

Chapter 3

Low power optimizations

As mentioned in the introduction, this project focuses on getting the most out of a given power source. Power sources are limited for various reasons and the power source chosen for embedded purposes is always a compromise in one way or the other.
The most common non-outlet power source is batteries. They are getting ever better but the weight/size of the battery is still a dominant factor in many embedded designs.
An overview of current battery technologies is given in appendix E

3.1 Theory

To understand the techniques for low-power design, a brief introduction of power-consumption is given. The basic theory says that every time an output of a gate rises, the capacitance, of the wire that carries the signal, is charged. The charge results in a voltage on the wire and a tiny increase in temperature occurs due to the resistance of the wire.

3.1.1 Definitions

Design is the connection of blocks, i.e. registers, adders, gates and the data path.
Noise margins are the range of voltages where a signal is considered a certain logic value, 1 or 0. A process has different noise margins for inputs and outputs.
Parasitic currents are unwanted current cause by the process and the way transistors are constructed.
Supply voltage is called V_{DD} and ground is called V_{SS}. The supply voltage can be from 1.8 - 5 V depending on the process.
Threshold voltage is called V_t. This voltage value is process dependent and the value sets the source-gate voltage where the transistor cuts off. This value decides the input noise margin of a gate.

3.1.2 A generic equation for power

The total power consumption for an integrated circuit can be described with formula 3.1 from [11]. The terms in the formula are either process or design dependent as listed in table 3.1.

$$P_{total} = p_t \cdot (C_{load} \cdot (V_{DD})^2 \cdot f_{clk}) + I_{sc} \cdot V_{DD} + I_{lk} \cdot V_{DD} \tag{3.1}$$

Term	Description	Depends on
p_t	Transition probability	design
f_{clk}	Frequency	process+design
C_{load}	Load capacitance	process
V_{DD}	Supply voltage	process (see below)
I_{sc}	Short-circuit current	process
I_{lk}	Leakage current	process

Table 3.1: The terms in formula 3.1

The transition probability is the probability that a power-consuming, i.e. rising, transition occurs. Frequency depends on the process and the design. The maximum frequency is decided by the path with the longest latency in the design and the latency is decided by the process.
Though V_{DD} is process-dependent, this term can be lowered at the cost of higher latency [11, 12], hence lower maximum f_{clk}. This technique is used in designs with Dynamic Voltage Scaling (DVS), which is covered in this chapter.
The load capacitance and its significance is explained in chapter 4.

3.1.3 Currents in integrated circuits

There are two major types of currents in integrated circuits. Parasitic and non-parasitic. The parasitic currents are unwanted but accepted as they appear as a compromise of other factors, e.g. low threshold voltage, V_t. Low V_t gives is necessary when using a low supply voltage as the noise margins needs to be adjusted. When V_t is low, the transistor is harder to turn completely off, resulting in a larger short-circuit current.
Non-parasitic currents will be explained in the following sections.

3.1.4 Short-circuit current

Short-circuit (I_{sc}) current occurs the N-transistor and the P-transistor are both in their saturated states, i.e. not completely turned on or off. This scenario draws a current from V_{DD} to V_{SS}, hence the name short-circuit current.

Short-circuit current is also known as dynamic current as it is only present when the circuit changes state.
For fast circuit operation, short-circuit currents are of little significance but in slower circuits, the loss can be as much as 30 % of the total power dissipated. Since this is the only power consumed in idle circuits, this is also called internal power when reported by the Power Compiler.

In figure 3.1 from [19], the non-parasitic currents in an inverter are shown.

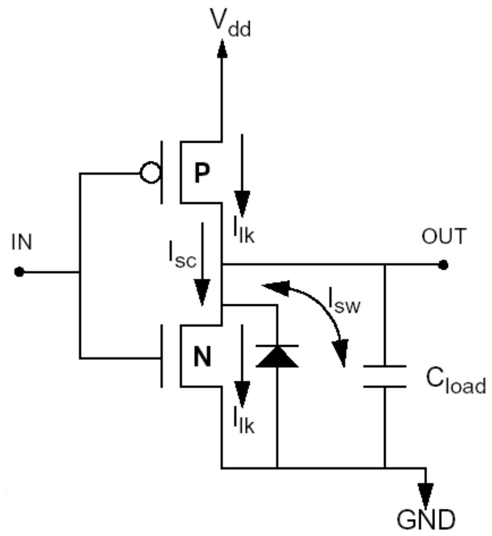

Figure 3.1: Dynamic and static currents in an inverter

3.1.5 Leakage current

Leakage-current (I_{lk}) is caused by the substrate and diffusion areas acting as a parasitic diode in a pnp-transistor (see figure 3.2b). The current in these diodes adds a power consumption to the design which can be in the mW range for a VLSI design.
In figure 3.2a from [15], a row of inverters are shown along with a cross-section of the inverters in an IC. In the cross-section the pnp-transistors are illustrated as well as the leakage currents.
I_{lk} is known as static current since it is present at all times no matter what the activity of the circuit is. Other parasitic current exist such as source-to-drain leakage, but substrate leakage is the most dominant source for power consumption when the circuit is idle.

3.1.6 Switching current

The inverter in figure 3.1 also shows the switching current, I_{sw}, which is the power used to charge the load capacitance to V_{DD}. The resistance of the transistor in the non-saturated (on) state is very small - very few ohms as opposed to the cutoff state where the resistance can be several mega-ohms.

Another term in power consumption is dynamic current which is the sum of the switching currents and short-circuit currents.

3.2 Methods

Many practical solutions exist to reduce power. Some are very hard to implement, but this section will focus on the solutions that are well-founded in the industry and, when possible, examples will be given of the technology in real-life products.

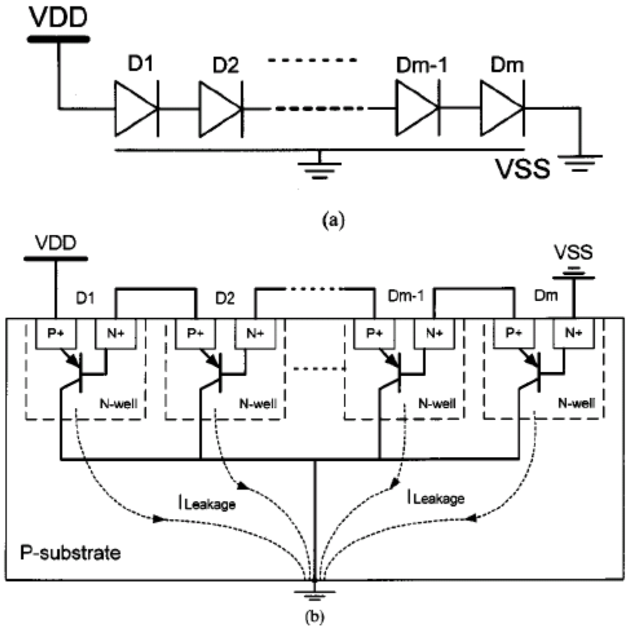

Figure 3.2: Parasitic transistors in a row of inverters

3.2.1 Clock gating

Clock gating is a (usually automated) method to control the enable input of a flip-flop to prevent loading values that are not used further in the data path. A similar technique is operand isolation where the current value is kept in the register instead of disabling the register. Only when needed, a new value is loaded and hence dynamic power consumed.
Using LPSDFF[1] flip-flops, a dramatic power reduction of 48% was achieved [6], when compared to a similar design with un-gated D-flip-flops.
Clock gating can also stop an ALU from operating in a certain clock cycle, e.g. while executing a `nop` instruction in a MIPS processor, where no calculations are needed in the ALU.

3.2.2 Sleep-halt modes

Most processors have sleep/wait/halt instructions that place the processor in an idle state where less power is consumed. The processor resumes from the idle state when an interrupt occurs.
A technique used in halt-modes is to save the processor state somewhere in the memory, allowing for the processor to be completely turned off. Reading the precise state of a processor involves reading out the contents of all important registers. This can be done via the scan-chain in the processor, if a scan-chain is included for testing. A scan-chain can be read via the JTAG protocol.

The Centrino platform [27, 28] is a commercial platform from Intel with a low-power processor and a chipset to control the processor. The Pentium-M processor is characterized by long range of different low-power modes, i.e. being able to run at multiple frequencies and core voltages. It

[1]Low Power Sleep mode with output pull Down Flip-Flip

also has a very large Level 2 (L2) cache - 1 MB. In comparison the Pentium 4 (HT[2]) has 512KB L2 cache. The extra cache improves speed when running with lower clock frequencies because of fewer cache misses.

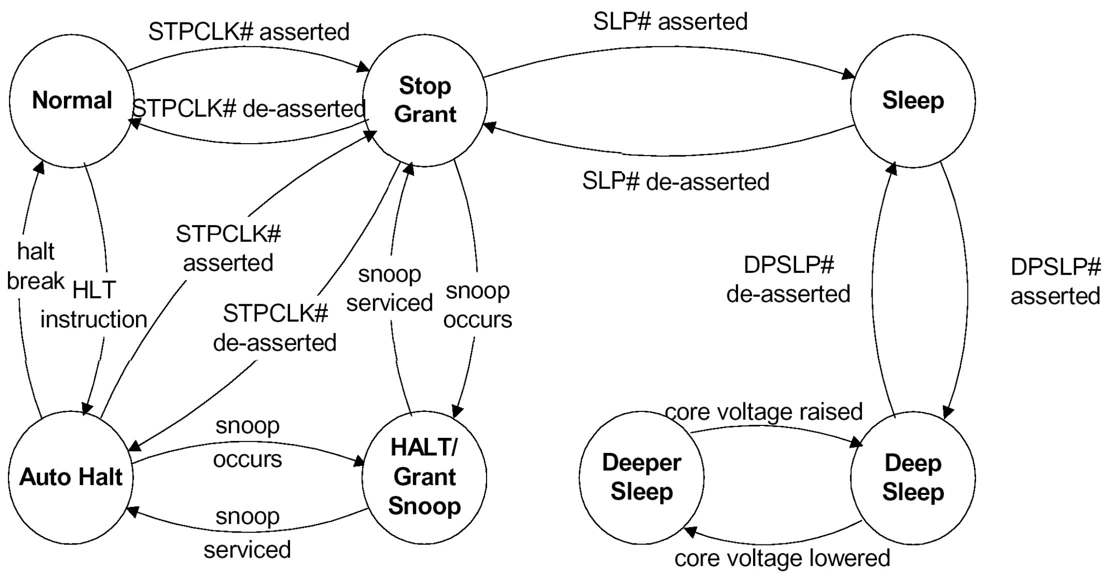

Figure 3.3: Control states for the Pentium-M processor from [27]

3.2.3 SpeedStepping^{TM}

Intel has designed a method for decreasing the clock frequency of a processor when power needs to be saved or the processor overheats. The first implementation of SpeedStepping^{TM}only allowed for a full-speed and a half-speed mode. This feature is present in Pentium-III and Pentium 4 mobile editions.
As described in the previous section, the Pentium-M processor allows numerous speed modes.

3.2.4 Dynamic Voltage Scaling

By compromising speed, the supply voltage, V_{DD} can be lowered. Unfortunately, V_{DD} has a quadratic dependence on the latency.
Formula 3.2 is a simple method for calculating dynamic power consumption in a design. This formula shows along with formula 3.1 that V_{DD} also has a quadratic dependence on the power consumption. These dependencies has lead to the adoption of the power-delay product [6, 11, 16]. The power-delay product shows a exponential relationship with V_{DD}, hence illustrating both trade-offs when reducing V_{DD}.

$$P_{total} \propto f_{clk} \cdot C_{load} \cdot (V_{DD})^2 \tag{3.2}$$

[2]Hyper Threading technology

A problem with this technique in embedded devices is that batteries have a certain voltage so extra circuitry must be introduced to scale the voltage down from the original value. This circuitry may take a while to settle at the desired voltage, hence adding a time-cost in the μs range on reducing the supply voltage.

3.2.5 Asynchronous designs

An important detail to notice is, that only rising signals consume power. This fact gives even more reason to reduce glitching. One method is asynchronous logic where any entity of the design implements a handshaking protocol, hence only doing work when requested and returning the result upon completion rather than following the timing of a clock. Asynchronous design promises speed and good power efficiency but the complexity of such designs are preventing them from wide acceptance.
In stead, engineers are making minor parts of a design asynchronous while still maintaining a synchronous structure.

3.2.6 FSM State Encoding for low power

Finite State Machines (FSMs) are one the most common parts of digital logic. In essence, every design is a state machine with the state defined by the current values of every register in the design. A more common implementation of a state machine is shown in figure 3.4. The state vectors are a list of bits that tell the output logic which state the state machine is in. The state bits can be encoded with many different enumeration algorithms:

- Forward encoding: 1, 2, 3,...
- Grey encoding: An encoding where only 1 bit changes each state.
- One-hot encoding: Only one bit is active pr. state.

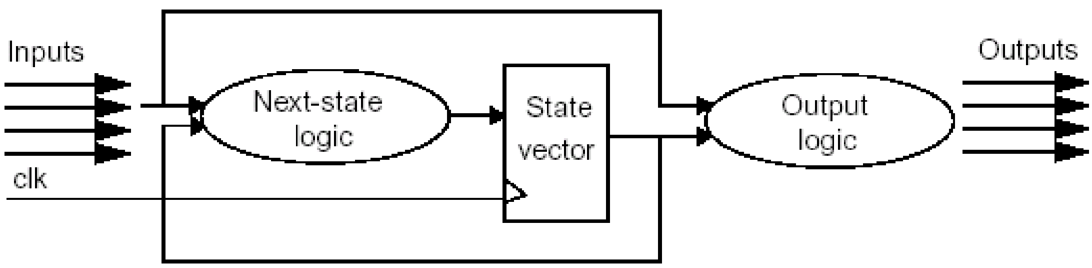

Figure 3.4: A finite state machine from [20]

Power-wise, the optimal state machine has only one transition pr. state change (Hamming distance = 1). So either a grey-encoding or a one-hot encoding should be chosen. Grey is slightly more power efficient as some state transitions have only one falling bit. One-hot encoding has one falling and one rising bit for each state change (Hamming distance = 2).

The challenge is to find an order and encoding of states in such a way that the most common sequence of states has the lowest sum of hamming distances.

The results of optimizing encoding can vary depending on the application being executed. Experimental results [17] show that the value of a cost function can improve from 4.12 % to 29.88 % when encoding state values from a probabilistic description of the FSM. The measurements were done with 16 different algorithms. The algorithms had from 4 to 14 states.

3.3 Discussion

Performing any form of calculation consumes power. The amount of power used for actual calculations compared to the amount of wasted power, could be called the efficiency rate, E_r, shown in equation 3.3:

$$E_r = \frac{P_{active}}{P_{total}} \tag{3.3}$$

The ideal efficiency is 1, where $P_{active} = P_{total}$, but this value is impossible to achieve. There will always be process variations that cause parasitic currents and the extra logic of creating a 100% glitch-free design would not be worth the extra static power.

Dynamic power can be reduced with very simple techniques but there will still be static power. The problem is that operating systems take time to start so there is little motivation for shutting down an embedded device to save power once the operating system is running and ready.
This problem has motivated sleep and halt techniques alongside SpeedStepping and DVS. These techniques reduces power consumption in e.g. a personal computer while still being ready for use.

An example is given in [8] where a Pentium III processor with the following characteristics is analyzed:

- Frequency = 600 MHz
- P_{100} at 100 % workload = 26.7 W
- P_0 at 0 % workload = 5 W

Comparing the uptime and the idle process time, often reveals that processors spend more than 95% of their life in the idle state.
For an 8 hour period, the mentioned processor would consume:

$$P_{active} = P_{100} \cdot (8h \cdot 0.05) = 10.68Wh \tag{3.4}$$

$$P_{idle} = P_0 \cdot (8h \cdot 0.95) = 38.00Wh \tag{3.5}$$

$$P_{total} = P_{active} + P_{idle} = 10.68Wh + 38.00Wh = 48.68Wh \tag{3.6}$$

The processor does not have any low power functionality and hence the efficiency rate is:

$$E_r = \frac{10.68Wh}{48.68Wh} = 0.219 \tag{3.7}$$

If the processor could go into sleep mode when idle and consume half the power, then P_{total} would be 29.68 Wh and the efficiency rate would be:

$$E_r = \frac{10.68Wh}{29.68Wh} = 0.360 \tag{3.8}$$

3.4 Summary

This chapter explained what options there are for power sources in embedded devices. An description of how power is consumed in integrated circuits was given. Methods for reducing power consumption were described along with their costs.
An overview of hardware techniques was given along with examples of real-life use and results of the techniques.

Chapter 4

High-level power estimation

When designing with any HDL, it is often necessary to estimate the resources that the design will use if implemented. The simplest technique is to run a synthesis program to get information on the area resources and then use port and register input activity for getting the power consumption. The problem with this technique is, that it is very time consuming to run a synthesis program.
This problem has motivated many techniques that promise to give power/area/timing estimates within some percentage of the actual value, but in much less time.

For this project, the Synopsys Power Compiler will be used for power estimation.

4.1 Theory

Power is consumed whenever a capacitance is charged to a high value. From [2, 37] we have the following relation:

$$P \propto C_{load} \cdot (V_{DD})^2 \cdot f_{clk} \tag{4.1}$$

Equation 4.1 states that power is proportional to the total capacitance in the circuit, but the capacitance being charged is more accurately described in equation 4.2. This equation states that the total charged capacitance is the sum of all capacitances multiplied by the number of times each capacitance is charged.

$$C_{total} = \sum_n C_n \cdot N_n \tag{4.2}$$

As explained in appendix D, some glitches are power consuming while others are not. A major challenge in the gathering of switching information is to locate all power consuming transitions and glitches. The design has to be analyzed on a gate-level basis to be sure to catch all data and even then, some data might not get captured accurately. An example is an AND-gate constructed from a NAND gate and an inverter. When the AND-gate shows a falling transition at the output, the internal NAND-gate will have a rising transition. A vast knowledge of the technology library must be obtained to avoid such short-comings.

4.2 Methods

Synthesis software analyzes a design and based on switching activity and a technology library, a value for power consumption can be found. One method of analysis, adds probes to each gate and gather all probes as a sampling-bus where the rising transitions can be counted. The cost of a transition is available in the technology library and hence the power consumption can be calculated. The runtime is proportional to the amount of switching data and the size of the design.

4.3 Current techniques

One often quoted technique in the high-level area is Paul E. Landmans paper [18] on the DBT (Dual Bit Type) method. This method gives an estimate within 10-15% accuracy [37] which is considered very good for RTL power estimation.
The model uses statistical parameters of the inputs to do an estimate on the data-path level. Each part of the data-path must be analyzed for this model but this only has to be done once. The name refers to the two types of bits that the algorithm recognizes: sign-bits and "noise" bits, referred to as UWN (Uniform White Noise). Sign bits have great significance for power consumption compared to the remaining bits in a data input. This distinction gives the algorithm extra accuracy.

An even more high-level technique looks at the application being executed, hence the instructions being fed to a processor. Before analysis, the processor has to be analyzed to learn the power consumption of each instruction. This can be done for several processors to find the most power efficient processor for a certain software algorithm.
The problem gets much harder if the design is pipelined. Then many decisions have to be made on how to do a fair evaluation of one instruction running through the pipeline.
No matter what, the pre-analysis is very time-consuming and has to be re-done if the processor design is altered or implemented with another technology library.
The advantage, however, is a very fast and very accurate estimation which allow for immediate optimizations.

4.4 Summary

This chapter covered high-level estimation of power consumption. The theory was covered and some current techniques were mentioned with their advantages and disadvantages.

Chapter 5

Implementation

This chapter explains the implementation details of the two processors from start to finish, including test and verification.
In this chapter, no distinction is made between micro-controllers and micro-processors. They are both referred to as processors. The AVR core and MIPS core are written by their entity names `avr_core` and `sc_risc`.

5.1 AVR vs. MIPS

The AVR core is designed to be compact to allow applications to be executed with the least amount of overhead. The instructions are all 16 bit and registers are 8 bit. AVR-based processors are used for real-time purposes and minor tasks where processor speed is less vital.
The MIPS is first and foremost designed for speed. The design is pipelined to maximize throughput and minimize latency. Each pipeline stage does a small, fast task to allow maximum speed. The MIPS design is available in 16, 32 and 64 bit architectures with 32 being the most common. The large instruction size gives some overhead that consumes unnecessary power.

So what are the similarities? They can both be programmed with GCC, allowing for any algorithm to be described with C/C++ as long as it fits within the available instruction memory.
They are both general-purpose processors with separate modules for handling interrupts. They are also both dual-memory architectures with separate memory for instructions and data.
Table 5.1 sums up the most relevant specifications.

5.2 AVR Core

The donor-core is called AVR_core. It was published by Ruslan Lepetenok to the website OpenCores.org. It is an implementation of the ATMEL 103 micro-controller. AVR_core supports all normal AVR instructions. The design is fitted with a timer, also published by Ruslan Lepetenok. Besides the timer, other modules were included. All external modules are controllable via I/O registers.

	AVR	MIPS
Architecture	8 bit RISC	32 bit RISC
Program counter	16 or 22 bit	32 bit
Instruction width	16 bit	32 bit
Pipelined	no	yes, 5-stage w. forwarding
CPI	1-4	1
Registers	32 8-bit	32 32-bit
Timer/Int. registers	64 8-bit	32 32-bit
Register bits total	736	2048
Instruction memory	2^{16} or 2^{22} instructions	2^{32} instructions

Table 5.1: Comparison of the two mentioned architectures

5.2.1 Design summary

The AVR core is a processor that executes 16-bit instructions in 1-4 clock cycles. The main processor is the entity `avr_core` that executes arithmetic functions and handles register loads and stores.
3 entities were added to the platform and others were replaced. This is accounted for in separate sections later.
After adding `halt_cpu`, `nj_mux` and `sleep_generator`, the design consists of the entities in table 5.2. For simplicity, a connection entity, `irq_io_module`, was created to connect all modules (except `avr_core`) into one entity.

Entity	Description
`avr_core`	The core processor
`nj_mux`	Select one of the many signals driving dbusin
`service_module`	Contains I/O registers for setup
`sleep_generator`	Stops the clock signal when `avr_core` executes `sleepi`
`halt_cpu`	Saves register and enables halt of `avr_core`
`timer_counter`	The timer, described in section 5.2.2

Table 5.2: Design entities in the AVR implementation

The `avr_core` entity consists of 6 entities:

- `pm_fetch_dec` set the program counter based on many inputs.
- `reg_file` is the register file with 32 8-bit registers.
- `bit_processor` does bitwise operation such as XOR.
- `io_adr_dec` controls the `dbusin` input to `pm_fetch_dec`.
- `io_reg_file` contains the status register, sreg, and the stack pointer.
- `alu_avr` is an ALU for addition, subtraction, etc...

Only half of the entities receive data from the outside `avr_core`. Internally, the `pm_fetch_dec` is the central entity, i.e. it handles communication with all the other entities. `pm_fetch_dec` also handles the communication with the memories, see figure 5.1.

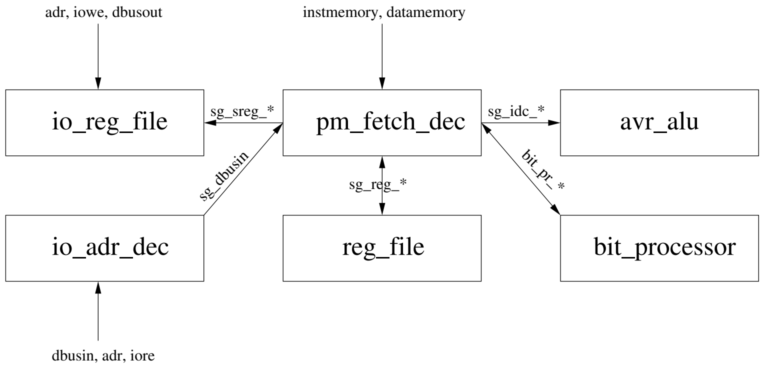

Figure 5.1: The interconnection of the entities in avr_core

5.2.2 The timer

The `avr_core` only executes the instructions. Connected to the `avr_core` are 64 8-bit registers that all have special purposes, e.g. programming the timer. The timer is an 8-bit timer/counter with 3 programmable triggers.

5.2.3 Extensions

The AVR core was missing an implementation of the `sleepi` instruction that causes the `avr_core` to stop until an interrupt line goes high. The output signals in table 5.3 where added and connected to the `sleep_generator` entity. These ports are part of the AVR specification [21] but were missing in the donor core.

Port	Description
`sleepi`	The processor tells that sleepi has been executed
`irqok`	The processor tells that the clock should be restarted
`globint`	A copy of the Interrupt flag (I) in SREG

Table 5.3: Outputs added to AVR core to enable sleep mode

5.2.4 Sleep-halt entities

In addition to the `sleep_generator`, an entity called `halt_module` was created, enabling the processor to completely shut off it's supply voltage. The `halt_module` halts the processor after a predefined number of clock cycles in the sleep state. Figure 5.2 shows the 3 major states for the design. The complete state machine has 3 and 5 states for sleep and halt entities respectively.

When the halt_module is activated, the entity saves the 32 general purpose registers and the program counter for the avr_core. These registers represent the current state of the avr_core. For reading out the registers, the ports in table 5.4 were added to avr_core.

Port	I/O	Description
pcwe	I	Program Counter write enable
pc_in<16>	O	A copy of the program counter
pc_out<16>	I	The Program Counter being written
regadr<5>	I	The register address for reading and writing
regre	I	Register read enable
regwe	I	Register write enable
reg_in<8>	O	The contents of the register specified with regadr
reg_out<8>	I	Input for writing the register specified with regadr

Table 5.4: Inputs/outputs added to AVR core to enable halt mode

In config.h in the AVR source directory, the parameter SLEEP2HALT can be defined to choose the number of clock cycles the processor will spend in the sleep state before entering the halt state. All behavior of the sleep_generator and halt_module has been verified to work in RTL simulation.

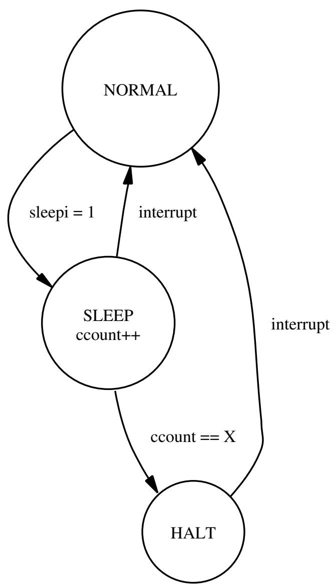

Figure 5.2: FSM for sleep/halt operation in AVR implementation

5.2.5 Discarded parts of the design

Some parts of the design were redesigned in SystemC in stead of converting from VHDL. This saved much logic in the finished design:

- Regfile. The register file contained many redundant registers. Replacement: A new entity, `regfile` with only one 32x8-bit structure.
- IO-registers in `external_mux` contained many redundant registers. Replacement: A new multiplexer, `nj_mux` with only one 64x8-bit structure.
- PROM. This entity was a blank RO-memory. Replacement: The program memory PROM was replaced with `instruction_memory`, which loads data from a stripped ELF-file with Position Independent Code (PIC).
- Dataram was a blank RW-memory. Replacement: A dataram module which fills the memory with the `.text`- and `.sdata`-segments of the ELF file.

5.2.6 Corrections to the AVR core

The design was converted to SystemC as described in appendix A. A few bugs were discovered in the design. The following were found and corrected:

- The timing for the `push` and `pop` instructions was wrong as some signals of the write operations were set too early. Only the `ramwr` (write enable) signal was set correctly, i.e. the second clock cycle of the 2-cycle instructions.
  Action taken: The signals needed were set in the second cycle of the instruction in stead of the first. The destination register had to be saved, so a register was added.
- `ld rD,Y` and `ld rD,Y` instructions were not detected, i.e. `idc_ld_y` and `idc_ld_z` signals were not set during these instructions. Instructions with pre-decrement (`ld rD,-Y`) and post-increment (`ld rD,Y+`) were detected correctly.
  Action taken: The method `instruction_decode()` in the entity `pm_fetch_dec` was corrected to recognize the instructions.
- As above, with `idc_st_y` and `idc_st_z`.
  Action taken: As above.
- Calculation of `postinc` and `predec` signals didn't include `idc_ld_x` in the expression.
  Action taken: Included `idc_ld_x` in expression and sensitivity list.
- The logic for `bst` (bit store to T flag) was mistaken for `bset` (bit set in SREG) in the `bit_processor` entity.
  Action taken: The `bset` signal was changed to `bst` in the expression for the `bit_pr_sreg_out` signal.

5.3 MIPS core

The MIPS core used in the project was designed as a Polytechnical Midterm Project in the fall of 2003 [36]. It is an implementation of the MIPS32 r2000 processor but without any coprocessors.

5.3.1 Design summary

MIPS is a well-documented pipelined processor which executes 32-bit instructions. The standard design is pipelined in 5 stages but other variants exist with more stages. The processor is well-suited for arithmetic calculations because of the 32-bit structure and the large available memory area of 4 Gb.
At the start of the project, the MIPS core had been synthesized to an FPGA, but correct functionality had only vaguely been confirmed. As simulation, the processor was able to execute most programs, including the Hamming-encoding program.

The MIPS implementation has fewer entities than AVR when comparing table 5.5 to 5.2. This is because more functionality is embedded in the MIPS processor core compared to the AVR core.

Entity	**Description**
`sc_risc`	The core processor, including `cp0`
`sc_bus`	Bus interface to connect the two memories

Table 5.5: Design entities in the MIPS implementation

`sc_risc` is a dummy entity that only contains `sc_cpu`, but the name `sc_risc` is used when referring to the MIPS core. `sc_bus` also a dummy entity containing the entity `sc_bus_interface`.

`sc_risc` contains the following entities:

- `pc_stage` is the program counter register and MUX.
- `if_stage` is the instruction fetch stage.
- `id_stage` is the instruction decode stage.
- `regfile` is the register file. It is connected in the ID stage.
- `cp0` is coprocessor 0. It is connected in the ID stage.
- `ex_stage` is the execution stage.
- `mem_stage` is the memory read/write stage.

The processor has 5 pipeline stages: IF, ID, EX, MEM and WB. The last stage is writeback. It is not implemented as a separate entity, but is located at the output registers of the MEM stage. For further information on the MIPS architecture, see [1].

5.3.2 Coprocessor 0

The MIPS core did not include an interrupt controller, so one was added. The functionality of the interrupt controller is to set the program counter when an interrupt/exception occurs. It is the task of the operating system to place code to handle the interrupt/exception. If an exception is unhandled the processor should reset.

Format	Description
`mfc0 rt, rd`	Move from coprocessor 0
`mtc0 rt, rd`	Move to coprocessor 0
`wait`	Place processor in wait state until interrupt

Table 5.6: The newly implemented instructions for operating on coprocessor 0.

Coprocessor 0 contains 32 32-bit registers that can be accessed with the two first instructions in table 5.6. The registers contains configuration bits for the processor.
For this project coprocessor 0 was implemented and the instructions `mtc0`, `mfc0` and `wait` were added to the processor core. Since the implementation of a coprocessor with a timer is a vast one, the implementation was simplified by connecting the `timer_counter` of the AVR design directly to the first 8 bits of the registers in table 5.7.

Register name	AVR name (no.)	MIPS name (no.)
Status register	SREG (63)	STATUS (12)
Control register	TCCR0 (51)	CONFIG (16)
Compare register	OCR0 (49)	COMPARE (11)
Counter register	TCNT0 (50)	COUNT (9)

Table 5.7: Registers in MIPS and AVR

The `wait` instruction is not a part of the MIPS32 (Also known as "mips2") ISA specification. However, the instruction is a part of the "mips3" ISA. When compiling TinyOS, the compiler must have the parameter `mips3`. The `mips3` parameter tells the compiler to use level 3 of the MIPS ISA, which actually means 64-bit MIPS. To use 32-bit MIPS, the compiler is given another parameter, `-mabi=32`, hence forcing ABI to 32 bit again, since the level 3 ISA contains both a 32-bit and a 64-bit ABI.
Unfortunately, using level 3 on a R2000 architecture results in code with data hazards as explained below. When using level 2, a `nop` instruction is inserted after every `lw` (load word) instruction to avoid hazards.

Coprocessor 0 is implemented according to the description in [3]

5.3.3 Missing tasks

At the time of writing, the design has yet some shortcomings that should be put first on a to-do list for further development.

- The `sra` (shift right arithmetic) is not implemented correctly as there is no known one-line algorithm for doing so and the instruction rarely is used.

- The `wait` instruction only works in simulation. The error when implemented on an FPGA should be located and corrected.

- As explained above, the code has data hazards. To avoid hazards, a hazard detection unit (HDU) should be implemented. A HDU sets the hold-signals in the pipeline registers and stalls the processor whenever there is a data hazard.

5.4 AVR analysis for improvements

The AVR architecture is often credited as a low-power architecture. This is due to the compact instruction set and the low clock frequencies in the current implementations.
However, an analysis can still be done to locate improvements on many levels. Such efforts have previously resulted in a new instruction set for the MIPS processor [7], but these attempts require a new implementation of the processor and a custom compiler for the new instruction set to work. The resulting instruction set saves power with methods similar to the operand isolation technique described in section 3.2.1. Operand isolation, however, only requires a hardware implementation to work.

5.4.1 Markov diagram for running the encoder program

A Markov diagram can be used for creating power-optimal FSM encoding. The diagram gives a good indication of the most common instruction sequences. The FSM for the processor can be optimized with grey-encoding for the most common sequences.
The diagram in figure 5.4 was drawn with data from the program `sequencegenerate` which was programmed for this project. The program converts the assembly lines of the debug output to lines of the format: "opcode -> opcode", e.g. "sbi -> cbi". `nop` instructions are not included in the diagram since nops are inserted whenever an instruction takes more than one clock cycle.
The Linux command tools, `sort` and `uniq`, are used to sort the output lines and count/group matching lines. The process is illustrated in figure 5.3.
The source code for `sequencegenerate` is shown in appendix H.8.1.

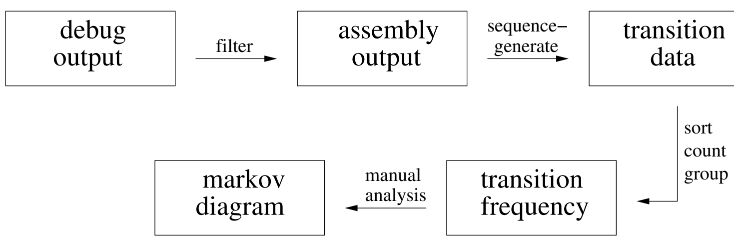

Figure 5.3: The creation of the Markov diagram using the sequencegenerate program

The Markov diagram seems to have dead ends but this is because the diagram only shows the 15 most common instructions. A Markov diagram for every instruction in the ISA would be impossible to show in 2D.
The Markov diagram should be evaluated with a listing of the instruction frequency. Such a list is shown in table 5.8.
The gathered data show that `sbi` (set bit in IO register) and `cbi` (clear bit in IO register) are the most common instructions in the analyzed program and they are two of the most likely instructions

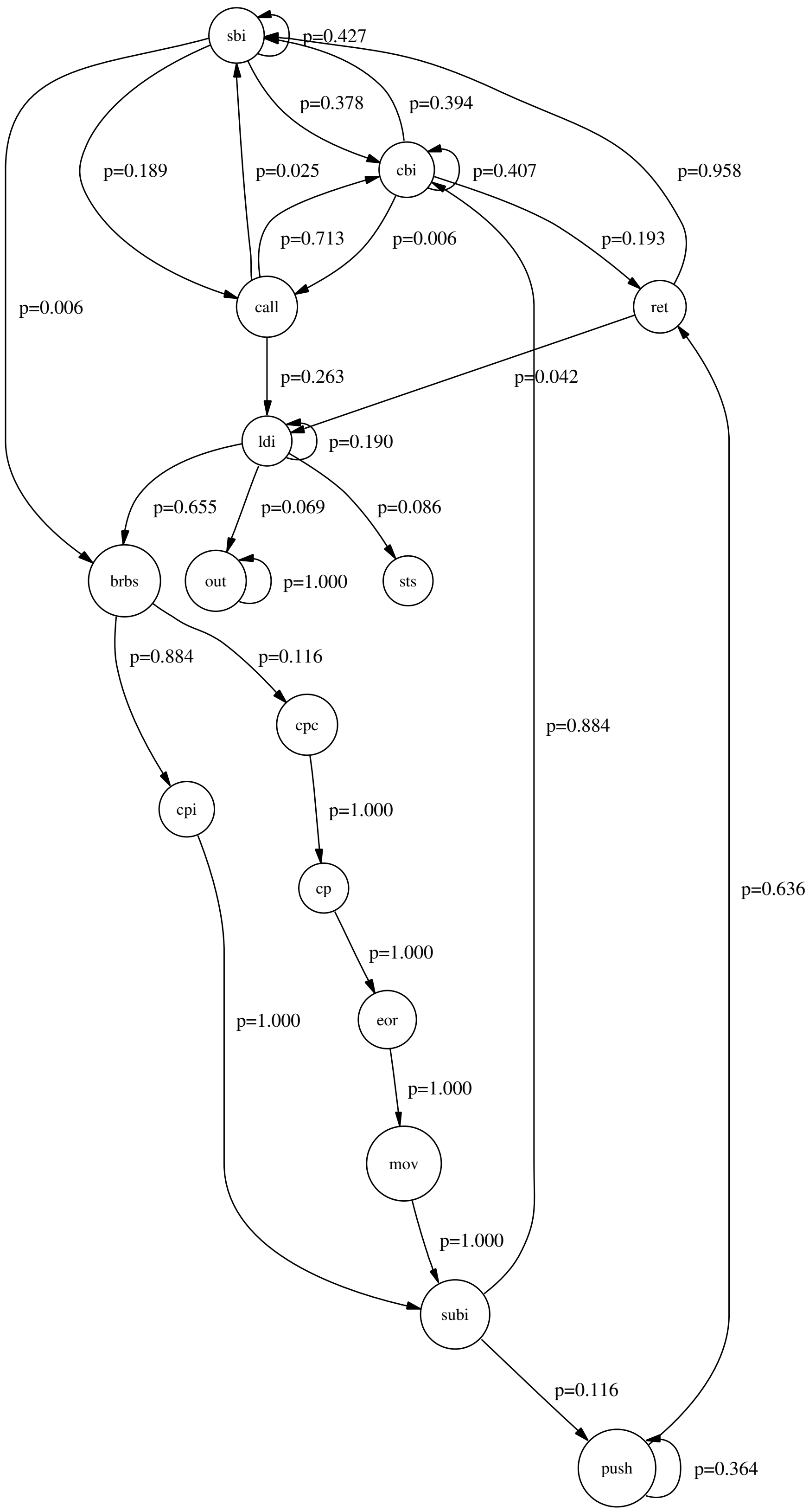

Figure 5.4: Markov diagram of the avr_core running the blink application

Instruction	Frequency
`sbi`	36.72 %
`cbi`	34.95 %
`ret`	7.41 %
`call`	7.41 %
`ldi`	3.05 %
`brbc`	2.07 %
`subi`	1.90 %
`cpi`	1.72 %
`out`	0.66 %
`push`	0.57 %
`lds`	0.57 %
`mov`	0.53 %
`sts`	0.31 %
`pop`	0.31 %
`cp`	0.31 %

Table 5.8: The 15 most common instructions when running the blink application

in a sequence. An optimized FSM should be based upon these observations.

5.4.2 Input activity measurements

When doing high-level analysis, the input activity of every input is of interest. The activities can be used when creating SAIF files or for annotating multi-bit signals individually as SAIF only looks at the switching activity of the entire signal.

Figure 5.5 shows how to gather the information using the program `transitioncount` which was programmed for this project. The program is a console program which outputs the values for each bit as a percentage value. The values can be plotted as a graph using existing programs.

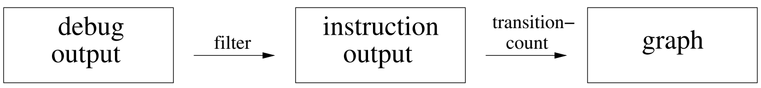

Figure 5.5: How to find input activity information from debug output

The bit activities for the blink application and the hamming encoder programs is shown in figure 5.6 and figure 5.7.
AVR instructions are not as easy to decode as MIPS instructions. The bits have different meaning for each instruction. The DBT algorithm would be a lengthy task to use for analysis as each instruction would have to be analyzed very thoroughly.

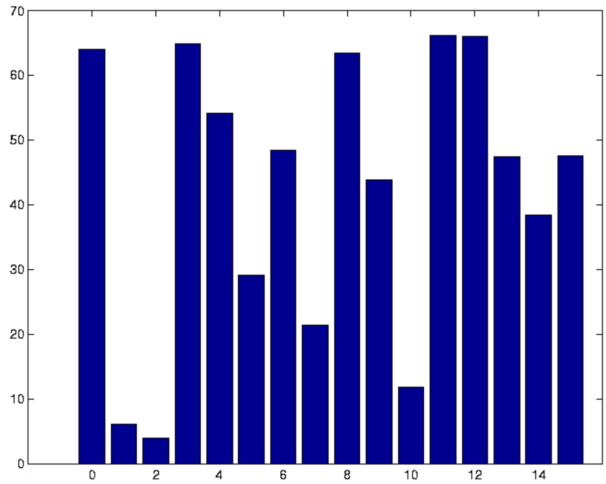

Figure 5.6: Bit activity for the instruction data when running the blink application

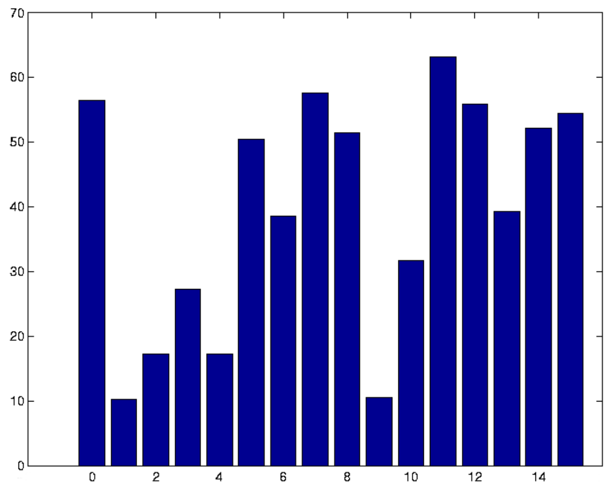

Figure 5.7: Bit activity for the instruction data when running the encoder program

5.5 SystemC, RTL implementation

To verify the functional correctness of the two designs, a small set of programs were executed and the outputs verified. The test programs are compiled and linked with a `Makefile` like the one in appendix H.9.1 (for the AVR implementation).
In [36], a set of test programs were created to test the MIPS processor in steps, where only a small set of instructions are tested per program. These programs were used to re-confirm the functionality of the MIPS core and to verify the functionality of the AVR core. The source code for the test programs is in appendix H.6.
All test programs have been compiled and executed on both platforms successfully.

5.5.1 Testing with applications

Two larger programs were tested to confirm functionality. The purpose of testing an application instead of each instruction separately is to test as many sequences as possible.
It is very time-consuming to test each instruction for every possible sequence that the instruction can enter. Since the objective of this project isn't 100% correctness, this hasn't been done. Instead, the processors have been verified for the sequence of instructions in the test programs, hence giving a rough verification.

5.5.2 Hamming encoding

Hamming encoding calculates redundant bits for a stream of data. The redundant bits work both as a checksum and for restoring bits that may have been destroyed since the bits were calculated.

E.g. an encoding of 4 data bits (referred to as b0 - b3) bits adds 4 redundant code bits (referred to as b4 - b7) for a total of 8 bits. This is one of the simplest encoding examples. In the encoded 8 bits, the 4 data bits are included as every other bit:

```
h1 = b0
h3 = b1
h5 = b2
h7 = b3
```

The code bits are calculated as XOR'ed combinations of the data bits:

```
h0 = (!b3) XOR b2 XOR b0
h2 = (!b3) XOR b1 XOR b0
h4 = (!b2) XOR b1 XOR b0
h6 = b3 XOR b2 XOR b1
```

When the 8 bits are received the parity can be calculated to verify that the data is OK:

```
p  = h7 XOR h6 XOR h5 XOR h4 XOR h3 XOR h2 XOR h1 XOR h0
```

If the parity, p, is 1 and the 3 following check expressions are 1, then the data is OK:

```
c0 = h7 XOR h5 XOR h1 XOR h0
c1 = h7 XOR h3 XOR h2 XOR h1
c2 = h5 XOR h4 XOR h3 XOR h1
```

If the parity is 0, then the binary number (c2, c1, c0) indicates the faulty bit.

More advanced hamming algorithms exists to add more bits for better error correction. In this example, only the minimum number of bits were used.
The Hamming encoder program runs error-correcting code on a small piece of text of 332 bytes. The hamming encoding is executed on packages of 11 data bits with 5 code bits. Because of the code structure and the register size of 8 bits in the AVR core, the data size can not exceed 16 bits.

The verification in both programs is done by writing the hamming encoded data to a specific address in memory that is detected by the implementation and written to a file. The filenames and the reserved addresses are shown in table 5.9.

Architecture	Entity	Reserved address	Output file
AVR	`dataram.cc`	0xFFF0	`avroutput.bin`
MIPS	`sc_bus_interface.cpp`	0x7FFFFFF8	`riscoutput.bin`

Table 5.9: Destination addresses for hamming encoded data

The source code for the encoder program can be seen in appendix H.6.2.

5.5.3 Timing details

The timing details of the Hamming encoder and the TinyOS BLINK example are shown in table 5.10. The numbers show how the two processors have very different execution time for different applications.

Architecture	Application	Execution time	Program size
AVR	Hamming encoder	2800 μs	1920 bytes
MIPS	Hamming encoder	980 μs	2384 bytes
AVR	TinyOS (BLINK)	65.8 + 1.5 μs	1096 bytes
MIPS	TinyOS (BLINK)	170.1 + 0.9 μs	3452 bytes

Table 5.10: Testing details of verified programs

5.5.4 Run-time measurements

The programs would only run for a few second if implemented in full-speed, but the runtime of the simulation is several minutes as shown in table 5.11.
There are two simulation times in Table 5.11. The first value is when sending debug output to console. The other is when sending debug output to `/dev/null`. The second value is the one to notice since it is (almost) independent of the amount of debug output the console. The debug output can be configured in `config.h` as explained below.

Architecture	# instructions	Simulation times	Inst/sec	Debug output
AVR	49000	18:45 / 5:16 Minutes	155	3690 bytes/cc
MIPS	140000	13:05 / 1:11 Minutes	1972	1160 bytes/cc

Table 5.11: Simulation details simulations running Hamming encoding on a 1.8 GHz Athlon

5.5.5 Debug output

One method of debugging is using print-statements that write the current values of some registers and/or inputs. In the setup file, `config.h`, a set of `#define` statements can be used to give more debug output. The output is displayed in the console while running the RTL simulation.
None of the print-statements are translated to VHDL.

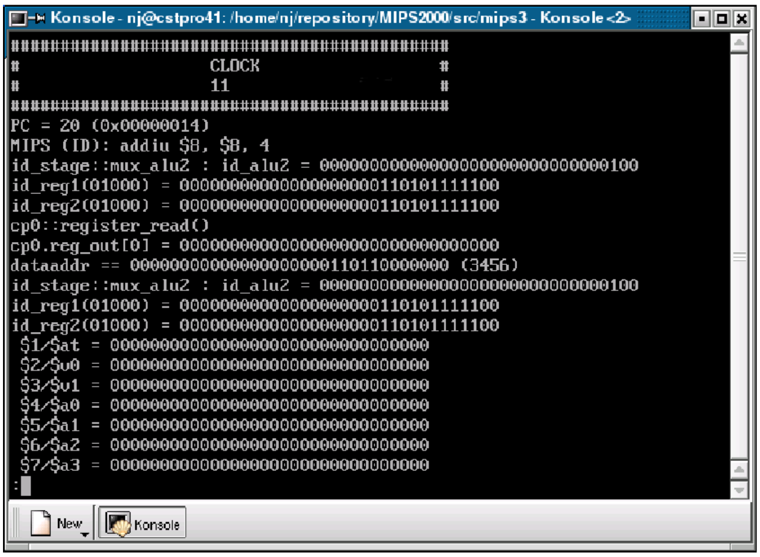

Figure 5.8: The Linux console when running the MIPS core

5.5.6 Waveform viewing

To look at timing and value changes of many signals, a waveform viewer can be used. The implementation in this project creates waveforms in the VCD format. VCD files can be displayed with the program GTKWave as shown in figure 5.9. GTKWave is part of the gEDA[1] project.
In `config.h`, the contents of the VCD file can be controlled and signals from each entity can be selected or deselected.

5.5.7 Memory models

The memory models from [36] have been simplified. The original memory models read an ELF file and located the `.text`- and `.sdata`-segments and placed in the memory.

[1]GPL Electronic Design Automation

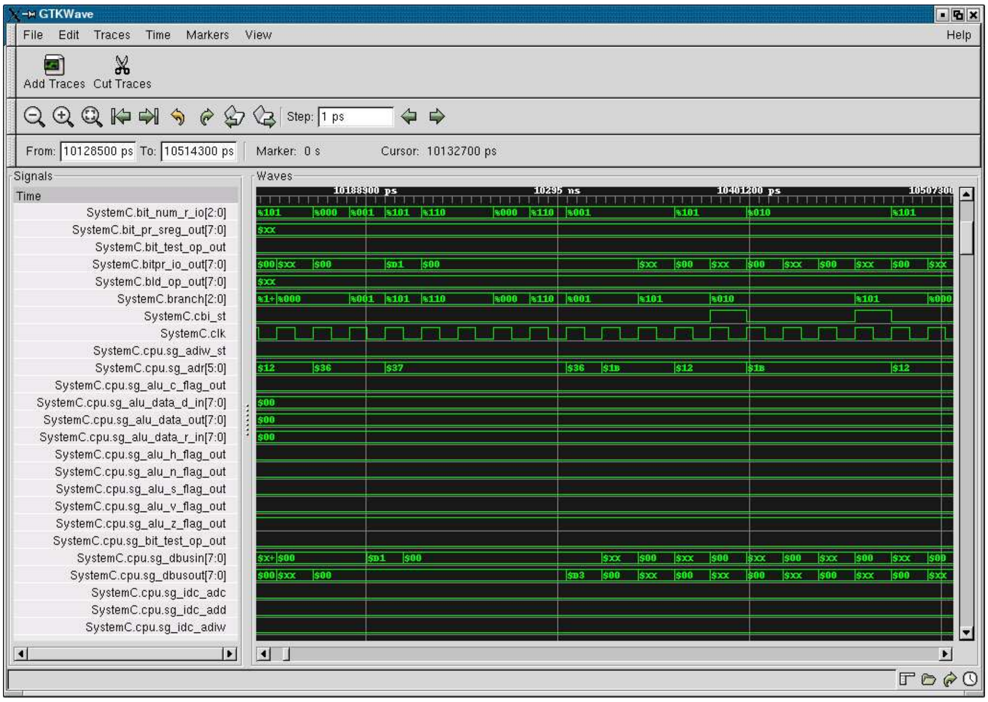

Figure 5.9: The GTKWave program displaying waveforms for the AVR core

The TinyOS distribution automatically creates a memory image that can be loaded directly in the memory. For the MIPS memory, a linker script created by Lars Munch Christensen was modified and used for compiling. The compiled application was afterward stripped of ELF headers with the program `objdump` from the GCC-distribution.

The two resulting memory images fit directly in the memory, starting at address 0.

5.6 FPGA implementation

Both cores were converted to VHDL with Synopsys programs (see appendix B) and synthesized with Xilinx ISE. The designs were implemented on a Xilinx Virtex FPGA based prototype board. The board and setup is described in details in appendix C.

5.6.1 Purpose

The idea of FPGA testing is to verify the timing of the design in real-life. Real-life testing can show problems that are not present in zero-delay simulation. Timing can also be confirmed by doing a back-annotated, simulation but in this project the setup of such a simulation would require that all memory entities were designed twice, so the work was in stead focused directly on FPGA implementation.

5.6.2 Setup of VHDL top-level entities

The setup of the top-level entities for synthesis was done entirely in VHDL. There would be no time saved if the top-level SystemC entity was converted as the setup of memories is very different.
The `mipssystem_sead` entity consists of a `sc_mips` entity, connected to with two 32-bit memories and the display driver for the SEAD board (appendix C.2). Figure 5.10 shows the setup.

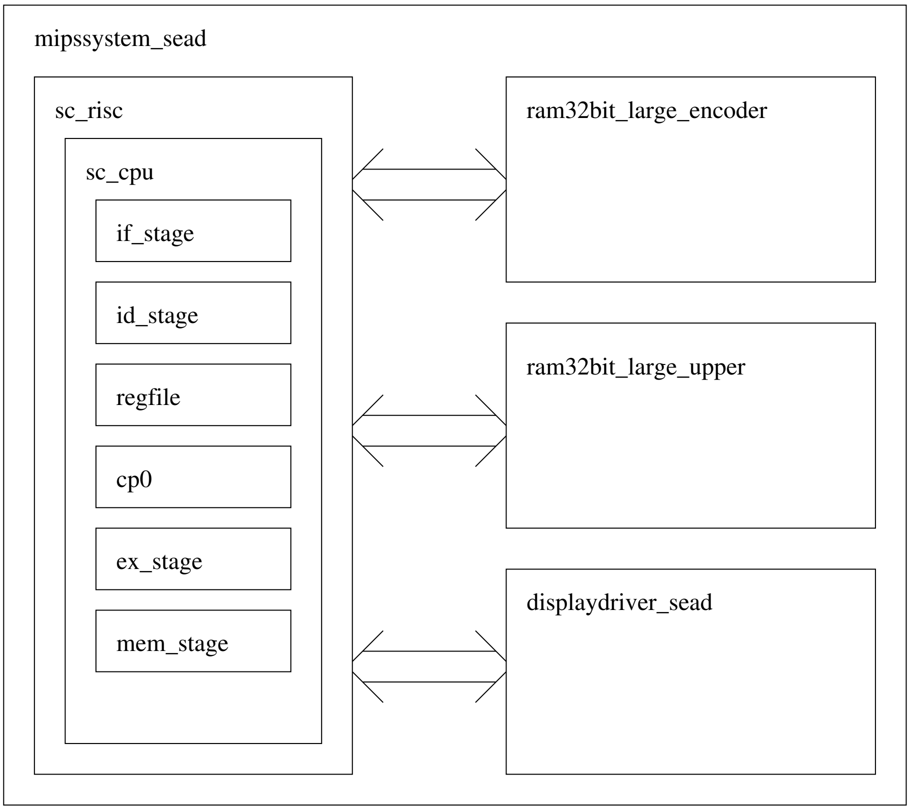

Figure 5.10: VHDL entity overview for MIPS platform

For the SEAD board, a constraints file was written to define the I/O pins to control the display, and to use one of the clock generators and the reset switch on the board.
The source code for the top-level entry, `mipssystem_sead.vhd`, is listed in appendix H.5.3.

The `avrcore_sead` consists of two memories (one 8-bit and one 16-bit) and a display driver as shown in figure 5.11.
The source code for the top-level entry, `testavr_sead.vhd`, is listed in appendix H.5.4.

More details on the setup are explained in appendix C.

5.6.3 Synthesis details

Table 5.13 shows how many slices and I/O pins are used by the two designs. The number of available Slice flip-flops in table 5.13 is twice the number of slices. This is because each slice consists

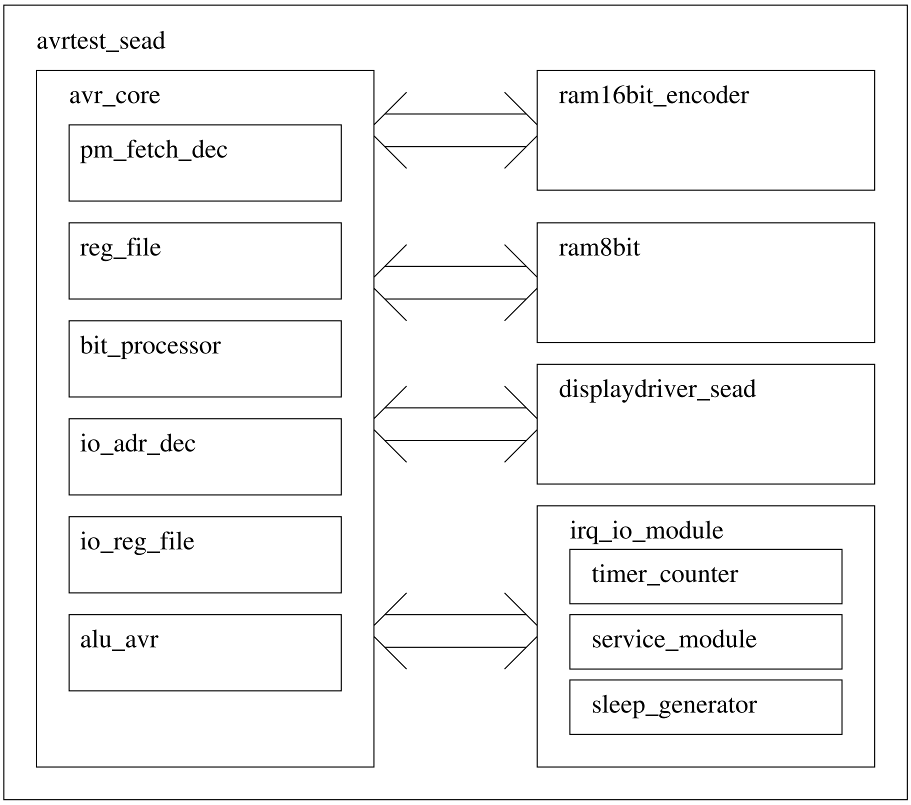

Figure 5.11: VHDL entity overview for AVR platform

of several types of logic including two flip-flops and a 4 input Look-Up Table.

The results of the synthesis with ISE is shown in table 5.12. For the two platforms, `mips_sead` and `avrtest_sead`, 3 values are shown. The first is for the entire platform including the processors, memories and display drivers. The second value is for the processor cores. The values for `sc_risc` also include the resources of `cp0`. The 7th value in the table, are for the `timer_counter` entity which is included in both `cp0` and `irq_io_module`.
Note that the slack increases for some sub-modules of the design when synthesizing separately. This is due to optimization of paths between entities.
Figures 5.11 and 5.10 show how the entities are related to each other. The memory entities do not use any slices in the implementation.

The details of the FPGA setup are shown in appendix C.

Entity	**I/O pins**	**Slices**	**Slice flip-flops**	**Longest path**
`mips_sead`	84	3821	2627	54.498 ns
`sc_risc` (incl. `cp0`)	202	3743	2519	56.998 ns
`cp0`	213	937	1127	21.594 ns
`avrtest_sead`	84	938	522	54.830 ns
`avr_core`	165	1049	570	56.403 ns
`irq_io_module`	68	566	670	20.394 ns
`timer_counter`	50	157	121	14.887 ns

Table 5.12: Synthesis details from the Xilinx ISE Synthesis Reports

	User I/O pins	**Slices**	**Slice flip-flops**
XCV1000	512	12288	24576

Table 5.13: Total number of resources in the Xilinx XCV1000 FPGA

The slack values can be used to give a clock frequency of the cores as in table 5.14. These values are useful if the processor is to be used in an FPGA-based prototype.

Entity	**Clock frequency**
`sc_risc`	17.544 MHz
`avr_core`	17.730 MHz

Table 5.14: Maximum clock frequencies for the FPGA processors

The reason for the relatively low clock frequency for the MIPS, is the addition of the coprocessor 0. Without cp0, the `sc_risc` runs at 22.493 MHz, i.e. 5 MHz faster.

5.7 Discussion

The starting point of both processors was pushed forward by using existing cores in stead of implementing from scratch. The process, however, is not always as big a short-cut as might be expected. It can be a lottery when using alien code as the foundation of a platform. Initial analysis is important to evaluate the credibility of the implementor. The danger of using half-completed cores is that the designs may be faulty or just poorly implemented. In many cases, every process of the design should be analyzed with regards to quality and efficiency.
In this project, quality assurance was achieved in the translation of the AVR core to SystemC. Of course, this process itself introduced some errors.
For the MIPS core, quality was assured by the fact that the author co-wrote [36] where the MIPS core was created from scratch.

It is essential to verify a design occasionally when doing optimizations and adding modules. An error must be detected as soon as possible even though it is very time consuming to verify correctness.

An 8-bit architecture might seem unfair to compare to a 32-bit architecture but both are rather ingenious in their own way. The 8-bit architecture is compact as it was meant to be. There really isn't much room for improvement. The 32-bit architecture has room for improvement. New instructions can be added as many opcodes are still unused. The pipelined design opens many new ideas for e.g. branch prediction. All these optimizations are for speed, though. The cost of the extra speed is more power. The basic philosophy is "Well, we might as well do something". A low-power approach would be to do as much possible with the power being used.

The Markov diagram could be used to alter the structure of the `avr_core` drastically. The current design has a seperate state machine for every instruction that used more than one clock cycle to execute. This could ideally be altered to just one state machine, optimized according to Markov diagrams made from executing various algorithms.

Looking at the input activity reveals the exact switching activity of the instruction signal. This information could be used for annotating inputs on bit-level in stead of annotating an entire signals switching activity.

As the next chapter will show, the two processors are very different when looking at speed and power consumption.

5.8 Summary

This chapter covered the implementation of the MIPS and AVR processors. The backgrounds for the implemented processors was explained and the work done one each processor was accounted for.
An analysis of the activity level in the AVR core was done and commented on.
Testing was also covered on two levels: High-level RTL simulation and FPGA implementation.

Chapter 6

Tiny OS

TinyOS is a project created by the Computer Science Department at University of California, Berkeley. TinyOS is primarily designed for programming sensor nodes, affectionately called motes, in sensor networks.

6.1 Concept of TinyOS

The concept of Tiny OS is to provide the basic surroundings (a.k.a. a kernel) for an application and thus allowing the application to be executed on an embedded platform as illustrated in figure 6.1. The application must be written in the NEST programming platform , which is a large set of C macros. A set of macros have been designed to use the hardware on the boards, e.g. timers, temperature-sensitive resistors, light-sensitive resistors, LEDs, speakers and microphones. Sensor networks are an essential part of the targeted embedded platforms for TinyOS.

TinyOS is only available for the AVR architecture. There are no known projects to use other platforms for TinyOS. In this project, an effort will be made to analyze and convert the TinyOS distribution to execute on a MIPS platform.

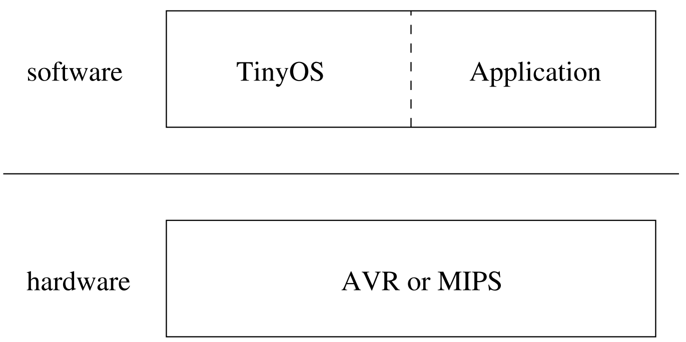

Figure 6.1: The TinyOS application becomes part of TinyOS

6.2 The sensors

To get a platform for running TinyOS, AVR-based Mica- or Rene-board specifications can be downloaded from the TinyOS website. The sensors can be also be purchased from external vendors. The sensors are designed to be operated by 2 AA NiMH batteries (see table E.1) which gives a voltage well within the AVR operating range of 1.6V - 3.6V [21].

Currently, a new prototype of the Smart-Dust specification is being tested. Smart-Dust is a single-IC sensor with integrated communication, sensing and computation. The challenge at the moment is to create good, accurate sensors on-Die.

6.3 The blink example

The blink example is located in the examples-folder of the TinyOS distribution. It is one of the smallest examples in the TinyOS distribution and the entire application is 4 small files:

- `BLINK.comp`
- `blink.desc`
- `BLINK.c`
- `BLINK.h`

The file that contains the actual code for the example is `BLINK.c`, while `BLINK.h` contains the prototypes for the functions.
The two remaining files, `BLINK.comp` and `blink.desc`, are include files which tell the compiler what elements should be linked together with the BLINK module. They also define what events to handle, in this case the event `BLINK_CLOCK_EVENT`.

The blink application initializes a timer to execute an interrupt every second. Upon every interrupt, the core flips the red LED on the Mica board.
The LEDs on the Mica board are located at address 0x0064 so a VHDL module was designed to detect writes on this address and transfer the data to LEDs on the FPGA board.

The details for the compiled application is shown in table 6.1. The two time values are for the initialization and the interrupt request, respectively.

	File size	**# instructions**	t_{INIT}	$t_{INT-REQ}$
AVR	1096	432	65.8 μs	1.5 μs
MIPS	3452	821	170.1 μs	0.9 μs
Diff	2356	389	104.3 μs	- 0.6 μs
Diff %	+ 214.0 %	+ 90.0 %	+ 158.5 %	- 40.0 %

Table 6.1: Summary of the blink example when compiled for the two platforms

6.4 Setting up the timer

As the setup of a timer is crucial to the TinyOS application in this project, a closer look will be taken. In the function `CLOCK_INIT_COMMAND`, timer0 is programmed. The timer is set in "Clear Timer on Compare match" (CTC) mode and the result is one tick pr. second.
The parameter values sent to the IO registers are shown in table 6.2.

Timer 0 has a built-in counter register, TCNT0 that is incremented each clock cycle. The timer also has a compare register, OCR0 that is compared to TCNT0 every clock cycle. When the registers are equal, another register is incremented, hence the design works as a programmable clock divider. When the register reaches the value set in TCCR0, a timer interrupt is executed.
The setup of the timer involves setting the interrupt enable bit in the status register, SREG, and setting the timer 0 enable signal in the TIMSK register.

Port name	**Register name**	**Value**	**Description**
`ASSR_Address`	Asynchronous Status Reg.	XXXX1XXX	Enable timer0 output
`TCCR0_Address`	Timer/Counter Control Reg.	00001110	Setup counter
`TCNT0_Address`	Timer/Counter Reg.	00000001	Reset counter
`OCR0_Address`	Output Compare Reg.	10000000	Set prescaler
`TIMSK_Address`	Timer Interrupt Mask Reg.	XXXXXX10	Enable timer
`SREG_Address`	Status Reg.	1XXXXXXX	Enable interrupt

Table 6.2: Configuration parameters for Timer/Counter 0 in BLINK example. X means that value is preserved.

6.5 Getting and running TinyOS

The TinyOS source code is released as an open source project and can be downloaded from [34]. To compile the project, two versions of GCC are needed: one for the AVR core and one for the local platform where the TinyOS application will be programmed and debugged.

GCC is available for many platforms, including AVR and MIPS, which are used in this project. ATMEL provides an AVR compiler as a precompiled distribution and Silicon Graphics (SGI) provides the MIPS-EL (endian little) version as both Debian and RedHat packages. The Debian packages seems to be flawless but the RedHat packages have problems linking programs.

6.6 Porting TinyOS to MIPS

The pre-analysis before starting such a task is very important. Decisions have to be made regarding I/O addresses and all peripheral items in general. Another option is to base the design on an already existing platform, such as the Mica platform.
The pre-analysis also involves locating all AVR assembly code in TinyOS to get an overview of the amount of code to translate.

Finally, the build system must be extended to compile with the MIPS version of GCC. The build system is naturally `make`.

6.6.1 Platform and in-line assembly

The platform is based upon the Mica platform as it seems to be the most common in the AVR based sensor nodes. The Mica platform has many variant but for now the focus will be to used the 3 different colored LEDs on the simplest Mica board.
To compile the blink example for the MIPS architecture, a subset of the AVR assembly macros must be translated to MIPS assembly. A set of conventions must be chosen first. In the AVR architecture, there are 64 I/O registers for configuring everything outside the `avr_core`. In MIPS the equivalent is the 32 registers in coprocessor 0. Table 5.7 lists the most relevant registers for both platforms.
The TinyOS distribution is divided into many files and folders but when looking for architecture-specific macros, there are 2 locations to consider: `tos/nest/tos/include/` and `tos/nest/tos/system/`

Other macros are design-specific. These are located in two other locations, here are the locations for the Mica boards: `tos/nest/tos/platform/mica/` and `tos/nest/tos/shared/`

The majority of the macros are located in the `include` folder and this is where most work must be done to translate from the AVR architecture to MIPS. Figure 6.2 shows the source code of the macro `__port_or` for the AVR architecture. In figure 6.3, the macro has been translated to the MIPS architecture.

```
#define __port_or(port, val)                \
        __asm__ __volatile__ (               \
                "in __tmp_reg__,%0" "\n\t"   \
                "or __tmp_reg__,%1" "\n\t"   \
                "out %0,__tmp_reg__"         \
                :                            \
                : "I" ((uint8_t)(port)),     \
                  "r" ((uint8_t)(val))       \
                : "r0"                       \
        )
```

Figure 6.2: A macro with in-line assembly for AVR

6.6.2 Application sizes

The values for number of instructions in table 6.1 are very different even though the application is the same. This is because the macros for the MIPS architecture have more instructions than the macros for the AVR architecture. In MIPS assembly, there are no instructions to e.g. set a certain bit in an IO register. In stead, the following steps are necessary:

```
#define __port_or(port, val)                    \
        __asm__ __volatile__ (                  \
                "mfc0 $8, $%0" "\n\t"           \
                "or $8, %1, $8" "\n\t"          \
                "mtc0 $8, $%0" "\n\t"           \
                :                               \
                : "I" ((uint8_t)(port)),        \
                  "r" ((uint8_t)(val))          \
                : "$8"                          \
        )
```

Figure 6.3: A macro with in-line assembly for MIPS

- Fetch register from `cp0`
- Create bitmask for the bit to be set
- OR bitmask with register
- Store register in `cp0`

This extra overhead of instructions also give longer execution times when simulating the two architectures at the same clock speed, 50 MHz.
When looking at the initialization, the MIPS has much longer initialization. This execution time can be reduced by removing a loop in the MIPS linker code that writes 500 zeros in the memory to clear the stack values.

6.6.3 The build system, make

Besides translating the many macros, the `Makefile` for the TinyOS applications must be altered to include "mips" as a platform to build binaries for.
The `Makefile` works as a recursive function as shown in figure 6.4. The Makefile executes itself 4 times after adding "mips".

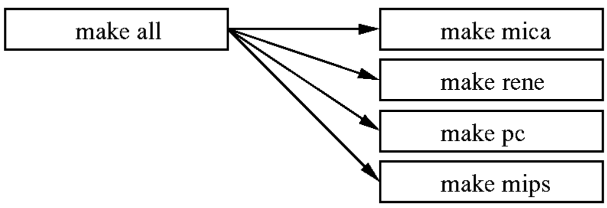

Figure 6.4: Recursive Makefile runs make 4 times

The `Makefile` was very straightforward to modify. In the header of the `Makefile`, a description of how to add hardware is given.

The `Makefile` and all other files required to compile TinyOS for MIPS, can be found on the CD. The files on the CD is simply copied to replace the ones in the TinyOS distribution.

6.7 Discussion

TinyOS is one of the many embedded operating systems available as open source. Other examples include embedded Linux and Windows CE/Mobile, which both requires more functionality from the processor.

TinyOS provides a simple and flexible platform for programming embedded systems and the resulting kernel is very compact and only contains the most necessary code for running the application.

The TinyOS modules provides simple configuration of many peripheral devices. The modules are all written with the set of basic C macros which naturally are architecture and platform-specific.

The conversion from AVR to MIPS was made possible by the modular structure of the distribution, where platform-wise source files are located in a single folder, kindly named `platform`. The only modules used in the example tested are CLOCK and LED. All other modules should be verified before TinyOS can be considered 100 % ported. This task involves connection many I/O modules that first have to be programmed in VHDL.

6.8 Summary

This chapter explained the concept and inner workings of the TinyOS project, especially the blink example. The task of porting TinyOS to the MIPS platform was documented.
A brief description of how to get and run TinyOS was given.
Finally, the effort of running the MIPS version of TinyOS on the MIPS core was described.

Chapter 7

Using Synopsys Power Compiler

The Synopsys Power Compiler provides several methods for estimating power within different accuracy levels. Figure 7.1 shows the data flow no matter what level of accuracy is desired. The switching information can be set for each pin or given as a SAIF-file (see appendix D).

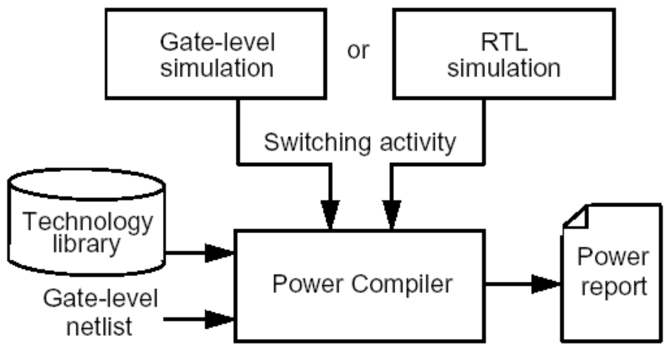

Figure 7.1: Information data-path for Synopsys Power Compiler (Figure from [19]).

7.1 Definitions

Also, see word list in chapter 1 for other terms.
RTL: RTL simulation only looks at the data being transferred between registers and not what timing is involved so all data changes immediately.
Zero-delay: Zero-delay simulation means that all gates stabilize their outputs immediately, hence no glitching.

7.2 Accuracy

Accuracy is proportional to the amount of switching data available and the runtime equally increases with the amount of data being analyzed.
PC provides 3 basic methods for power estimation, here listed with increasing accuracy:

- RTL simulation
- Zero-delay an unit-delay gate-level simulation
- Full-timing gate-level simulation

If less accuracy can be accepted, the power estimation can be done with zero-delay simulation. Zero-delay simulation does pessimistic calculations since toggle rates are set to default values meant to give an above-or-equal value for power consumption.
Glitching is only captured in full-timing simulation and this requires switching activity files based upon back-annotated simulation.

7.3 Requirements for running power compiler

Any level of analysis need a basic set of data to work on to get optimal results. The minimum requirement is naturally a set of VHDL files to work on. The power compiler does not analyze SystemC files directly (yet) but it's a one line command to do the conversion. The VHDL/SystemC files must be synthesizable.

Adding switching activity is not specifically required but without this information, every input and register is assumed to have a toggle rate of a default value:

$$TR = 2 \cdot SP \cdot (1 - SP) \cdot f_{clk} \tag{7.1}$$

SP is static probability and has a default value of 0.5, hence the default toggle rate:

$$TR = 2 \cdot 0.5 \cdot (1 - 0.5) \cdot f_{clk} = 0.5 \cdot f_{clk} \tag{7.2}$$

This is a very pessimistic value as the port or register is expected to shift every other clock cycle.

7.4 RTL Power Estimator

For fast power estimation, a program called `report_rtl_power` is available. The program gives a power report equal to `report_power` but only based on synthesis-invariant elements in the design.

> "...because the RTL Power Estimator is geared toward enabling users to make quick architectural trade offs based on power estimates rather than on power results from accurate load information." - from [19].

`report_rtl_power` should only be used for very rough estimated, e.g. to see if a certain change in the design will give extra power consumption compared to the previous version.

7.5 Power Compiler Analysis Engine

The Synopsys Power Compiler, `report_power`, gives sufficient accuracy for most tasks and the runtime is acceptable. The SAIF files can be created from VCD files (see appendix D) which is the preferred waveform file-format when simulating in SystemC.

An example script for doing power analysis in the AVR core can be seen in appendix H.9.4.

7.6 Clock gating with Power Compiler

The Power Compiler can do automated clock gating in VHDL files. A parameter i simply set in `dc_shell` to enable Clock gating:

```
set_clock_gating_style -sequential latch -control_point before
```

The clock gating is done when the file is analyzed. When reading a VHDL design, the command, `read` is usually used, but a two-step approach is necessary when clock gating:

```
analyze -format vhdl sc_cpu.vhd
elaborate sc_cpu -gate_clock
```

The above example clock gates an entity `sc_cpu` in the file `sc_cpu.vhd`. The result of the command can be shown with the `report_clock_gating` command:

```
report_clock_gating -gating_elements -gated -ungated -hier
```

A complete script for clock gating a processor is shown in appendix H.9.3.

7.7 Physical compiler

The most accurate power estimation tool is the physical compiler, `phys_opt`. The physical compiler analyzes a design with actual process values and hence gives a result equal to that of a real-life IC if the same process is used.
The physical compiler has not been used in this project, because IMM does not have the required license for running the `phys_opt` program.

7.8 Discussion

Before running a power estimation tool, the requested accuracy must be chosen. The accuracy must reflect the topic being investigated.
For this project, a low accuracy estimation was chosen with most focus on the switching activity of the designs. Leakage currents and short circuit currents can be reduced primarily by changing the production process, which is beyond the scope of this thesis.

Other companies specialize in power estimation and perhaps provides more accurate results than the Synopsys solution. At IMM, previous projects have tried to do precise power estimation but none have succeeded in a degree that has resulted in a practically usable program.

A major limitation in this project has been the libraries provided by Synopsys. The libraries do not contain information about leakage or internal power, leaving only analysis of dynamic power. The process is also quite old, .25 and 5 V supply voltage. The library folder does not contain other libraries that could be used in stead.

7.9 Summary

This chapter covered the Synopsys Power Compiler which has been used to find the results in the next chapter. The many levels of accuracy that can be obtained have been covered and commented on.
The dilemma of runtime vs. accuracy was covered as well as the types of measurements done by the Power Compiler. The physical compiler was mentioned.

Chapter 8

Results

To do an evaluation of the implementations and their power efficiency, the designs must be analyzed with a program that takes every gate into consideration based upon a measured switching activity. Synopsys provides these programs, but these programs have limitations in terms of run-time vs. accuracy as described in chapter 4 and chapter 7.

8.1 Definitions of outputs

Before looking at the results of the Synopsys Power Compiler, it is important to look at what is measured.
Switching power is the power consumed by the nets in the data path.
Net count is the number of interconnections (a.k.a. wires) between the cells. The value switching power refers to the power consumed when charging the nets.
Total dynamic power is the power used by the cells when generating output to gates.
Cell count is the number of basic building blocks in the data path. Approximately 2/3 of used area is used by the cells.

Besides the dynamic values, both nets and cells consume power during static operation as explained in chapter 3.

8.2 Setup of measurements

The power compiler is executed on the IMM Sunfire server running the SparcOS5 operating system. The Synopsys 2003.06 distribution is installed on the server with valid licenses for running power compiler and much more.
The power compiler uses a technology library and two design-ware libraries. Design-ware libraries contain technology-independent building blocks for synthesis.
Technology libraries contain information about a set of basic gates such as AND, NOR and XOR gates. The libraries also contain information about the internal capacitances in all the gates. There is also data for wires and tri-state busses. This information can be used with a synthesis tool to give a power estimate for an entire design.

A script for measuring power consumption can be seen in appendix H.9.4. This script analyzes the `avr_core` when running the hamming encoder program.

8.3 Power measurements

8.3.1 Elements not captured

The libraries are provided by Synopsys for very rough power measurements. The values for power consumption are based on a .25 process with a supply voltage of 5V. The values are not 100% accurate but they give a good idea of the improvements from version to version when comparing values.
The technology library does not contain information about "Cell internal power" or "Cell leakage power" as these values would require giving away the secrets of a process.

8.3.2 Pessimistic power measurements

If absolute no ports or registers are annotated with a switching activity, the power compiler assumes pessimistic default values and the results are as shown in table 8.1.

The values show how power consumption is measured as very high when using these pessimistic values. Note that all values are mW - not μW

Entity	Switching power	Net count	Total dynamic power	Cell count
`avr_core`	12.8562 mW	5421	5.934 mW	5278
`sc_risc`	133.4945 mW	13587	7.865 mW	12918

Table 8.1: Power consumption when not annotating inputs

8.3.3 Idle power measurements

Before doing any optimizations, the power consumption for the idle processors was measured. The measurements were performed by creating a SAIF file with all values set to 0 except the T0 value (the time spent with the value 0) which is set to maximum. The values for the measurements are shown in table 8.2.

The Idle power consumption tells how much power the processor uses when no program is running and no new instructions are fed to the processor. This power can be reduced to zero if the processor has the ability to do a halt of the processor. Idle power is considered wasted as no computations are gained from it.

The values for both power measurements in this case are equal due to the fact that no inputs change, and hence all power is consumed by internal cell outputs which affect switching power and dynamic power equally.

Entity	Switching power	Net count	Total dynamic power	Cell count
avr_core	5.756 μW	5421	5.756 μW	5278
sc_risc	4.561 μW	13576	4.561 μW	12841

Table 8.2: Idle power consumption

8.3.4 Power consumption for an application

When running the SystemC simulations, a VCD file is generated. This VCD file can be converted to an SAIF file as shown in appendix D. The SAIF file can then be used to annotate switching activity an average value for the power consumption can be found.
The results can be seen in table 8.4 and table 8.3. The power consumption is shown for each processor and for the entities that each processor contains.
The first tables show the execution of the Hamming encoder program and hence the interrupt handling entities are not measured.

An value to notice is the Switching power for the sc_risc entity in table 8.4. When running the analysis initially, a warning was given that the input signal, cp2 i.e. the clock signal, was a high-fanout and had be set at 1000 pF for easier analysis. To avoid this worst-case estimate, the parameter high_fanout_net_threshold can be set to 0. The value before setting the parameter was 1433.100 μW, where 1250 μW was the clock switching power. The high-fanout was caused by the pipelined design and in particular the ID stage where the register file is clocked.

Entity	Switching power	Net count	Total dynamic power	Cell count
pm_fetch_dec	52.432 μW	2355	27.758 μW	2181
reg_file	62.196 μW	2406	27.506 μW	2350
bit_processor	4.461 μW	202	2.092 μW	158
io_adr_reg	0.386 μW	87	0.083 μW	40
io_reg_file	4.442 μW	237	0.307 μW	188
alu_avr	8.386 μW	584	5.965 μW	539
Sum	132.303 μW	5871	63.711 μW	5456
avr_core	105.701 μW	5421	44.884 μW	5278
Difference	- 26.602 μW	- 450	- 18.827 μW	- 178
Difference (%)	- 20.1 %	- 7.66 %	- 29.55 %	- 3.26 %

Table 8.3: Power summary, AVR (encoder)

The sums of the cell and net count values for the sub-entities are smaller than the entire entity, since nets and cells are added when connecting the entities to each other. The power consumption, however, drops for the single analyzed entity. This is because switching activity always is annotated on the port of the top-level entity. For some reason it has not been possible to annotate internal registers with the measured values, hence leading to different values in the measurements.

In table 8.5, the same values are shown for both processors running the initialization part of the TinyOS blink example. The value for the avr_core was found by adding the measured value for the avr_core entity with the value for the timer_counter entity. The MIPS core is the

Entity	Switching power	Net count	Total dynamic power	Cell count
pc_stage	10.181 μW	158	0.000121 μW	122
if_stage	9.087 μW	462	0.717 μW	294
id_stage	269.80 μW	4716	98.430 μW	4480
ex_stage	152.340 μW	5052	125.311 μW	4355
mem_stage	8.796 μW	277	2.778 μW	165
Sum	450.204 μW	10665	227.236 μW	9416
sc_risc	339.50 μW	13549	181.991 μW	12880
Difference	- 110.704 μW	2884	-45.245 μW	3464
Difference (%)	- 24.59 %	27.04 %	-19.91 %	36.79 %

Table 8.4: Power summary, MIPS (encoder)

actual value of the sc_risc entity with coprocessor 0 enabled. As mentioned previously, the pipelines design must be analyzed with the high_fanout_net_threshold parameter set or very high values will be calculated.

Entity	Switching power	Net count	Total dynamic power	Cell count
avr_core (*)	108.404 μW	5421	47.673 μW	5278
sc_risc	544.726 μW	20196	246.376 μW	19497

Table 8.5: Power consumption running blink initialization, 80000 ns

8.3.5 Clock gated designs

As explained in chapter 3, clock gating reduces unnecessary transitions. Both processors were clock gated with the power compiler and the SAIF information from the Hamming encoder program. The results of the power analysis can be seen in table 8.6. The calculations took almost 21 hours for the sc_risc entity and 26 hours for the avr_core entity.

The power compiler uses switching activity to analyze what registers to clock gate. In the case of the MIPS processor, 100% of the registers where clock gated and 71.80 % were clock gated in the AVR processor, see details in table 8.7.

Looking at the switching power, the avr_core power consumption is reduced by 30.283 μW in the AVR processor and by 19.528 μW in the MIPS processor. The dynamic power, however increases as would be expected when the extra logic for clock gating is introduced. The increase in power is equivalent to the increase in areas listed in table 8.8.

Entity	Switching power	Net count	Total dynamic power	Cell count
avr_core	75.418 μW	13091	54.554 μW	12948
sc_risc	319.972 μW	22449	315.186 μW	21780

Table 8.6: Power summary with clock gating, (encoder)

Entity	Clock gating elements	Gated registers	Ungated registers
avr_core	18	410 (71.80%)	161 (28.20%)
sc_risc	6	1394 (100.00%)	0 (0.00 %)

Table 8.7: Details on the amount of registers being clock gated

8.4 Area and timing information

The technology libraries provide an area-value for every library component, allowing for a total area to be calculated. The areas are divided into combinatorical, non-combinatorical and interconnect areas. The values for the two processors are shown in table 8.8.
The area for the clock gated processors are also shown. The numbers show the increase in area by 100,50% (AVR) and 32.97 % (MIPS) even though more registers were clock gated in the MIPS core. This is probably due to the fact that the AVR core used 3 times as many clock gating elements than the MIPS, as seen in table 8.7.

Entity	Combinatorial area	Non-combinational area	Net Interconnect area	Total area
avr_core	8281.95	3253.93	11536.10	58524.16
avr_core (cg)	26082.17	3195.12	29276.18	117331.36
sc_risc	22583.33	8199.67	30783.71	144496.44
sc_risc (cg)	36412.71	7785.48	44204.14	190693.81

Table 8.8: Area information from the Design Compiler, all values are μm^2

The Power compiler can also be used to get an estimate of the maximum frequency of the processors if implemented in a 5 V .25 process. The values are shown below in table 8.9.

The values show that the pipelined MIPS processor runs more than twice the speed of the AVR core. Note that the data arrival time of the MIPS actually is a calculated value. The value given by the `report_timing` command was 110.39 but this value was based upon a wrongful assumption by the Design Compiler. When using the Design Compiler, a value for the clock can be given and the `report_timing` command will give a warning if the timing requirements aren't met. For this calculation, the clock period was set at 200 ns and hence the values of the data arrival time was calculated from the falling edge of the clock cycle which is 100 ns.
In stead of the value 110.39 ns, the value $2 \cdot 10.39$ns was used. 20.78 ns is close to what was expected when looking at the value for the non-pipelined AVR core.
The Design Compiler somehow has a problem when calculating the data arrival time for the clock gated processor. The `report_timing` command gave a value of 3.33 ns for the MIPS core, which is clearly erroneous. Whether the faulty value is caused by an error in the `report_timing` command or an error in the clock gated design, is unknown.

Entity	Data arrival time	Maximum clock frequency
avr_core	54.70 ns	18.281 MHz
sc_risc	20.78 ns	48.123 MHz

Table 8.9: Timing information from the Design Compiler

8.5 RTL implementation results

Correct functionality was verified in SystemC RTL simulation with several test programs. The designed platforms run both compiled programs and a TinyOS example. This has been verified with waveforms, debug output and by looking at the data written to memory entities.

All implemented features and entities have been verified in RTL simulation:

- Executing `wait` instruction in MIPS core.
- Entering and returning from halt-mode.
- Hamming encoding a block of text with both cores and decoding the text with an external program to verify correctness.

Benchmarking of the processors are shown in table 5.5.

8.6 FPGA Results

The MIPS core was synthesized successfully and tested with both the BLINK TinyOS application and the Hamming encoder program. The program counter was monitored on the display and the correct sequences were confirmed. The MIPS does, however, have the problem that the `wait` instruction doesn't stall the processor and waits for an interrupt to occur.

The AVR core caused problems with jumps from the first implementation. The initial source of the error was suspected to be the conversion to SystemC so the donor-core from OpenCores was tested and showed to have the same problems with jumps.
This problem could be caused by one (or both) of the following:

- The core was setup wrong
- The core has a bug somewhere
- The core is not reset correctly in every process.

The setup of the `avr_core` has been verified to the extreme. The memory has been set up with the address connected to a counter and the output connected to a display. This setup verified that the BlockRAM contents is correct. The design was tested with the `avr_core` as a back-annotated design and no problems were found. This leads to the conclusion that the donor probably has a bug somewhere but this bug has been very hard to detect.

An attempt has been made to synthesize a clock gated design with Xilinx ISE but the VHDL produced by the Power Compiler unfortunately does operations with values of the SIGNED type that are not compatible with the XST VHDL standard that Xilinx uses.

8.7 Discussion

The area and timing cost of clock gating is drastic, but the saved switching power shows that a good amount of power is saved. The saved power should be more than the increase in internal power for the process. Otherwise, the clock gating is not worth the effort. Since clock gating is an automated task, this should be made a standard feature when creating low-power platforms.

The power consumption for the processors with and without their interrupt handler is shown. The cell power increases almost 200 μW when introducing the coprocessor 0 in the MIPS processor. This is due to the fact that the `timer_counter` entity runs a counter at full speed, hence introducing many power consuming switches.

8.8 Evaluation

Two cores have been implemented and verified in RTL simulation. The designs look very solid in RTL simulation, but the AVR core does not behave correctly when running on the FPGA.
The MIPS, core however runs smoothly except for the bug in the `wait` instruction which should be a minor task to fix. The TinyOS example has been verified on FPGA by looking at the instruction address on the board display.

The clock gated designs seem to save some power at the cost of area but it is a problem that the designs can't be synthesized to an FPGA. Without such verification the measured values for the design might be wrong.

8.9 Summary

This chapter presented the results of the power measurements. The AVR and MIPS cores were analyzed with the Synopsys Power Compiler and the results were commented.
A status of the implementations was given for the SystemC code and the effort of running the processors on an FPGA.
The results and their reliability and usability was discussed.

Chapter 9

Conclusion

Two platforms have been implemented for this project. The platforms have been evaluated and with regards to power consumption. Clock-gated versions have created but errors in the code produced by the Design Compiler have made it impossible to use the new cores for anything other than power measurements.

Using existing cores was a challenge. The MIPS core was well-known but the AVR core took some effort to understand fully. The end-result is two cores with similar capabilities but with very different characteristics.

The TinyOS embedded operating system has been executed in simulation and on the SEAD board with the MIPS processor. The verified correct functionality of the blink example is promising for future work with the MIPS version of TinyOS.

Both cores have been verified with many applications to be working correctly.

9.1 SystemC

The experience of using SystemC is not flawless. The task of verifying the design on an FPGA involves converting files to VHDL. If an error is found, it must be corrected in the SystemC source files. This is necessary to avoid maintaining 2 versions of the same processor.
The SystemC cores can be programmed to Linux programs that can execute any program in MIPS or AVR assembly. This is very valuable and would not have been possible with VHDL. The cores can be configured to output almost any extend of debug output to console and/or VCD waveforms.

9.2 Future work

Here are some recommendations for future work on the elements of this project:

1. Find and correct the errors described in section **??**.

2. Compile more TinyOS applications for AVR and MIPS.
3. Find bug in AVR core and synthesize to FPGA
4. Finally, design and build a hardware prototype board with single FPGA, sensor and an antenna for deployment of nodes.

Lyngby, 30/11 2003, Nicolai Ascanius Jørgensen

9.3 Acknowledgments

I would like to thank Klaus S.Madsen for valuable comments in the final part of writing.
I would also like to thank Hans Holten-Lund of IMM for assistance with the FPGA boards during the project.
Finally, I would also like to thank the good people at Wittenborg for creating a coffee machine with a "strength +" button.

Bibliography

[1] David A. Patterson and John L. Hennessy: Computer Organization & Design. The hardware/software interface, second edition (1997). Morgan Kaufmann.

[2] Neil H.E.Weste and Kamran Eshraghian: Principles of CMOS VLSI design, second edition (1994). Addison Wesley

[3] Gerry Kane: MIPS R2000 RISC architecture, (1987). Prentice-Hall.

[4] Dominic Sweetman: See MIPS Run, (1999). Morgan Kaufmann.

[5] MIPS Technologies: MIPS32™Architecture For Programmers Volume I-III (2001). MIPS Technologies Inc.

[6] Krste Asanovic, Marc Hampton, Ronny Krashinsky and Emmett Witchel: Power aware computing, Chapter 1, Circuit level power management, (2002). Kluwer Academic/Plenum Publishers.

[7] Krste Asanovic, Marc Hampton, Ronny Krashinsky and Emmett Witchel: Power aware computing, Chapter 5, Energy-exposed instruction sets, (2002). Kluwer Academic/Plenum Publishers.

[8] Krste Asanovic, Marc Hampton, Ronny Krashinsky and Emmett Witchel: Power aware computing, Chapter 14, The case for power management in web servers, (2002). Kluwer Academic/Plenum Publishers.

[9] Xilinx®: Spartan-II 2.5V FPGA Family: Complete Data Sheet.
`http://www.xilinx.com/`

[10] Xilinx®: Virtex™ 2.5V Field Programmable Gate Arrays: Product Specification.
`http://www.xilinx.com/`

[11] Anantha P. Chandrakasan, Samuel Sheng and Robert W. Brodersen: Low-Power CMOS Digital Design, (1992). IEEE Journal of solid-state circuits. Vol.27, No. 4, April 1992.

[12] Sornavalli Ramanathan and Rituparna Mandal: Low Power Solution for Wireless Applications, (2002). Preceedings of the 15th International Conference on VLSI Design.

[13] Carla-Fabiana Chiasserini and Ramesh R. Rao: Improving Battery Performance by Using Traffic Shaping Techniques, (March 1, 2001). IEEE Journal on selected areas in communications. Vol.19, No. 7, July 2001.

[14] Yung-Hsiang, Luca Benini and Giovanni De Micheli: Dynamic Frequency With Buffer Insertion for Mixed Workloads (April 3, 2002). IEEE Transactions on computer-aided design of integrated circuits and systems. Vol.21, No. 11, November 2002.

[15] Ming-Dou Ker and Wen-Yu Lo: Design on the Low-Leakage Diode String for Using in the Power-Rail ESD Clamp Circuits in a 0.35 μm Silicide CMOS Process, (1999). IEEE Transactions on solid-state circuits, vol.35, No.4, April 2000.

[16] A. v. Schwerin, D. Schumann and J. Berthold: CMOS Process Design for Minimization of IC Power Consumption using TCAD, (1995). IEEE/Siemens.

[17] L. Daldoss, D. Sciuto and C. Silvano: State encoding for low power embedded controllers, (1998). IEEE.

[18] Paul E. Landman: Architectural power analysis: The dual bit type method, (1994). IEEE transactions on VLSI systems.

[19] Synopsys®Inc.: Power Compiler Reference Manual, (August 2001). Synopsys®Inc.

[20] Synopsys®Inc.: CoCentric SystemC Compiler RTL User and Modeling Guide, (June 2003). Synopsys®Inc.

[21] ATMEL®: AVR, Embedded RISC Microcontroller core, (1999). ATMEL®

[22] MIPS Technologies: MIPS32™Architecture For Programmers Volume I-III (2001). MIPS Technologies Inc.

[23] Digilent Inc.: Digilab 2 Reference Manual, (Revision: May 7, 2002). Digilent Inc.

[24] Digilent Inc.: Digilab DIO2 Reference Manual, (Revision: May 7, 2002). Digilent Inc.

[25] Agilent Technologies: Eight Character 5 mm Smart Alphanumeric Display, (October 29th, 2001). Agilent Technologies.

[26] Barry Pangrle, Ph.D. Sr. R& D Manager: Presentation slides for SAIF presentation, (October 15th, 2002). Synopsys®, Inc.

[27] Intel®: Intel 855GM Chipset GMCH Datasheet, (2003). Intel corp.

[28] Intel®: Intel Pentium M Processor Datasheet, (2003). Intel corp.

[29] About li-ion Polymer Batteries,
`http://www.ion-energy.com/ier/6`

[30] The Energy Crisis on Your Lap,
`http://www.msnbc.com/news/992103.asp?0dm=N11KT`

[31] OpenCores.org - project page for AVR core,
`http://www.opencores.org/projects/avr_core/`

[32] The European SystemC users group, http://www-ti.informatik.uni-tuebingen.de/~systemc/systemc.html

[33] SAVANT, VHDL analysis tool, http://www.ececs.uc.edu/~paw/savant/

[34] TinyOS main page, http://webs.cs.berkeley.edu/tos/

[35] Ascend EDA, http://www.ascend-eda.com/

[36] Andreas Vad Lorentzen and Nicolai Jørgensen: Design of a synthesizable processor core in SystemC, (February 2003). IMM, DTU.

[37] Morten Elo Petersen, Power Estimation for High Level Synthesis, (August 31, 1995). IMM, DTU

Appendix A

Converting VHDL to SystemC

This appendix describes the efforts made to do automated translation of VHDL code to SystemC code. The AVR Core from [31] is written in VHDL. Since the ambition was to develop the core using SystemC, the code was converted. Ideally, there would be a program that could do this, but the efforts to make such a program have been very limited.

A.1 VHDLParser

The most promising software is VHDLParser from the The European SystemC users group [32]. The tool translates most code to SystemC but there are annoying limitations. Only lines of code within a VHDL process is translated. Concurrent statements and WHEN-statements that aren't written inside a process are ignored. Concurrent statements are often used in VHDL since the sensitively list is implicit. In SystemC a process must have a sensitivity list.
Another problem is that the translated code requires a header file, vhdlparser.h because some constants from an IEEE package are needed. This file must either be included in the design or these constants must be replaced with similar constants included in SystemC.

A.2 Savant

Savant [33] is another project that claims to translate from VHDL to SystemC. This is not a direct false claim but the SystemC code is timed simulation code and hence no good for RTL simulation. The VHDL design is first translated to an internal format called IIR[1] from which the code again can be exported as behavioral VHDL code and/or timed simulation SystemC code which is useless for further work as all entity and signal names are lost in the translation.

A.3 Verilog2SC

A commercial tool is Verilog2SC from Ascend EDA [35] which, as implied in the name, converts Verilog code to SystemC. The Synopsys tool `compile_systemc` can convert from VHDL to

[1] Internal Intermediate Representation

Verilog so this tool would perhaps be interesting had there been funds available to try it out.

A.4 The chosen path

The AVR Core design was converted using the VHDLParser tool and a lot of manual translation. The primary work consists of locating concurrent statements in the VHDL code and translating it. After writing the code in c, a complete sensitivity list must be written to make statements combinatorial and to avoid registers when multiplexers are desired.

(a)
```
dbusout <= signal1 when x = "101"
           else signal2;
```

(b)
```
In header file:
...
SC_METHOD(execute);
sensitive << signal1 << signal2 << x;
...

In .cc file:
...
void module_name::execute()
{
  if (x.read() == "101")
    dbusout = signal1.read();
  else
    dbusout = signal2.read();
}
```

(c)

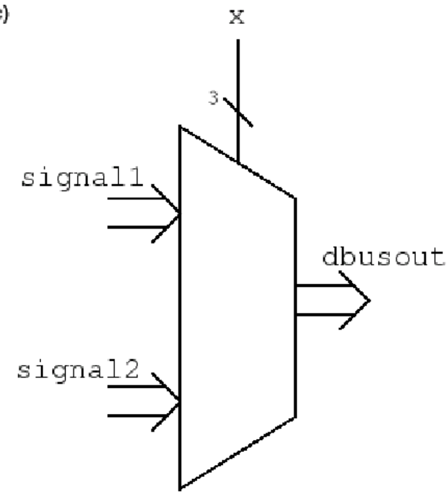

Figure A.1: (a) VHDL code. (b) SystemC code. (c) The modeled MUX.

Much translation can be done with search and replace, e.g. replacing "or" with "|", "and" with "&" and "not" with "~".

Appendix B

Converting SystemC to VHDL

After verifying a SystemC design, the entity must be translated to VHDL before synthesis. This is done with the `compile_systemc` program from Synopsys.
When doing this conversion, the SystemC types sc_int<> and sc_lv<> are converted to a type called, UNSIGNED. This data type needs some arithmetic function that are defined in two Synopsys-provided files, listed in table B.1. The paths should be the same on workstation installations. Otherwise substitute `/usr/synopsys/` with the local install path.

File	Location
ccsc.vhd	/usr/synopsys/syn/packages/ccsc/src
attributes.vhd	/usr/synopsys/syn/packages/synopsys/src

Table B.1: Files needed for synthesis of translated SystemC entities

B.1 dc_shell script example

The script in figure B.1 read `alu_avr.cc` and translates it to RTL level VHDL code.
A header file, `alu_avr.h` is included in `alu_avr.cc`, i.e. the entity is actually two files, since sensitivity is defined in the header file.
A complete script for converting the AVR core is shown in appendix H.9.2.

B.2 Quality of translation

The VHDL files are functionally equal to the SystemC files. There has been very few examples where the VHDL code had to be modified before synthesis with Xilinx ISE was possible. The errors encountered in this project have been corrected in newer versions of the Synopsys software and there has been no problems with the latest 2003.06 release.

```
company = "IMM, DTU";
designer = "NJ";

target_library    = {"tc6a_cbacore.db"};
synthetic_library = {"dw01.sldb","dw02.sldb"}
link_library      = {"*"} + target_library + synthetic_library

define_design_lib WORK -path ./WORK

read dw01.sldb
read dw02.sldb
read standard.sldb

compile_systemc -rtl alu_avr.cc -rtl_format vhdl

exit
```

Figure B.1: Example script for converting from SystemC to VHDL

Appendix C

FPGA setup

The implementations are have been synthesized to two Xilinx FPGA-based boards, one Spartan-II and one Virtex. The two FPGAs are present on a small and a large board, respectively. After a short while it was clear that the smaller board had insufficient logic to implement the designs. So a larger board was located.
The small board is manufactured by Digilent and is connected to the Digilent DIO II I/O board [24]. The board is based around the FPGA shown in table C.1 and the I/O board is very versatile with LEDS, 7-segment displays, buttons, switches and a LCD display.
The larger board is a MIPS SEAD prototype board, based primarily around the device in table C.2. It also has a MIPS processor, SRAM, DRAM, micro-switches, 7-segment display and a CPLD. For this project the board was configured to not use the MIPS processor at all.

For both boards a display driver was written in VHDL to use the available displays for debugging. The implementation of the display drivers are described below in section C.2.

Device	Logic cells	System Gates (Logic and RAM)	CLB Array (R x C)	Total CLBs	Maximum Available User I/O	Total Block RAM Bits	Total Distributed RAM Bits
XCS2S200	5292	200,000	28 x 42	1176	284	57,344	75,264

Table C.1: Excerpt from Table 1 in [9], XC2S200-PQ208

Device	Logic Cells	System Gates	CLB Array	Maximum Available I/O	Block RAM Bits	Maximum SelectRAM+ Bits
XCV1000	27,648	1,124,022	64x96	512	131,072	393,216

Table C.2: Excerpt from Table 1 in [10], XCV1000-FG680

C.1 On-chip memory

As shown in table C.1 and table C.2, both devices are equipped with BlockRAMs memory. The memory is organized in blocks of 4096 bit each. The FPGA allows the BlockRAMs to be config-

ured in 5 single-port and 15 dual-port configurations allowing for data widths from 1-16 bits.
For the AVR core, 4 16-bit memories were combined to one 16-bit memory (`ram16bit.vhd`) for the instructions. The 2 MSB in the address, are used to create internal chip-select signals for the 4 memories. An 8-bit memory was also created (`ram8bit.vhd`) for data/stack.
For the MIPS core, 4 8-bit memories were initially combined to one 32-bit memory (`ram32bit.vhd`) with byte-select. This version was too small for the applications at hand, so another version was made with 8 8-bit memories to create a 32-bit memory with double capacity (`ram32bit_large.vhd`). For data memory, a third version was created (`ram32bit_upper.vhd`) where the address area was changed to make room for a stack in the upper part of the memory.

For initializing the memories with code, the code must be written as 256-bit big-endian initialization-strings (as hex) in the VHDL files. Two programs were designed to convert binary code files to lines of this format for either 16-bit blocks (AVR) or 8 bit blocks (MIPS). The programs are called `mkram` and can be found in appendices H.7.2 (for MIPS) and H.7.1 (for AVR).

In table C.3, the memory configurations are listed along with their valid address area. Note that for the 32-bit memories, the input address is shifted twice to the right as the MIPS core operates with 4-byte alignment.

Platform	VHDL entity	Address area	Size
AVR	`ram8bit.vhd`	0x0000 - 0x01FF	512 bytes
AVR	`ram16bit.vhd`	0x0000 - 0x03FF	1024 16-bit words
MIPS	`ram32bit.vhd`	0x00000000 - 0x000007FC	1024 32-bit words
MIPS	`ram32bit_large.vhd`	0x00000000 - 0x00000FFC	2048 32-bit words
MIPS	`ram32bit_upper.vhd`	0x00000800 - 0x00001FFC	2048 32-bit words

Table C.3: Overview of the memory entities created for synthesis

C.2 Display drivers for FPGA boards

For this project, two display "drivers" were programmed in VHDL for this project. The drivers create an entity that receives one or two 32 bit signals and displays them as hex-numbers on the available display. The timing of each character and the initialization of the displays is all handled by the entity.

C.2.1 Digilent Board

The FPGA board is connected to an I/O board [24] with 16 LEDs, a 2x16 character display, 8 switches and 15 buttons. For debugging and testing, a module was written to display the instruction address (a.k.a. program counter) and the contents of the instruction memory. All I/O runs through a CPLD which routes predefined addresses to different elements on the board.
The module is a basic state machine (SM). First, the display is configured and cleared. After initialization, the SM runs a look refreshing the display continuously. The display requires a

separate clock input that must be kept below 50KHz. On the board, this clock can be created by taking the 10th bit in a counter-signal connected to the 50MHz input clock.

C.2.2 SEAD board

The MIPS SEAD board [25] features a 8-element ASCII-display from Agilent, 8 direct-accessible LEDs and 8 direct-accessible micro-switches. More LEDs and switches are available through an Altera CPLD.
The display driver for this board is also a simple SM that runs with an external clock. The display refreshes the characters with an internal 32KHz clock. The state machine must have a clock below 700Hz, which can be created by taking the 14th bit on a counter-signal connected to the 12MHz input clock.

C.3 Programming via JTAG

The boards are programmed via a PC parallel port with the JTAG protocol. The programming was done with IMPACT, a tool distributed with Xilinx ISE. In table C.4, info about the BIT-files and the programming time is listed. Unfortunately, an USB cable was not available or the programming could have been done nearly 20 times faster.

Device	.bit file size	Programming time	PC interface
XCS2S200	766,045 bytes	32 seconds	Parallel JTAG cable
XCV1000	167,053 bytes	7 seconds	Parallel JTAG cable

Table C.4: Programming times for the FPGAs in this project

Appendix D

SAIF - Switching Activity Interchange Format

The SAIF file format was designed internally by Synopsys as a 7-year long effort to create a common format for annotating switching activity. Synopsys has also been working on getting other companies to accept SAIF as *the* method for annotating designs but there are currently no simulation programs that produce SAIF files directly.

There are several formats that contain different amounts of information such as WIF, VCD, ISDB, EVCD and LXT. These formats all capture value changes in signals and ports in a design. The goal for SAIF isn't to capture cycle-to-cycle information but to sum up the information important timing information, such as glitching information. The gathered information is particularly useful when calculating power consumption.

To help the transition period, Synopsys has created the program `vcd2saif` that translates VCD files to SAIF files.

D.1 The data in SAIF files

SAIF files contain two types of information for every signal:

- Timing information
- Toggle information

The parameters in table D.1 are pretty much self-explanatory. The time for each value is recorded and the sum of all the times for a signal is equal to the simulation time.

These values are essential for calculating static power.
Toggle parameters are listed in table D.2. Toggle is a term often used by Synopsys and is a common term for rising and falling transitions.
A gate will see glitches on most inputs but only glitches of a certain length (> gate delay) will cause a power-consuming glitch on the output of the gate. This type of glitches are called transport glitches. When the length is too short to cause an output glitch, the glitch is called an inertial glitch.

Label	Description
T0	Total Time Design Object is in 0 State
T1	Total Time Design Object is in 1 State
TX	Total Time Design Object is in Unknown X State
TZ	Total Time Design Object is in Floating Z State
TB	Total Time Design Object is in Bus Contention[1]

Table D.1: SAIF Timing Attributes. Source: [26]

Inertial glitches can be ignored for simplicity.
The parameter, IK, is a factor describing the ratio between initial glitches compared to the total number of glitches.

These values are essential for the calculation of dynamic power.

Label	Description
TC	Toggle Count, Number of 0 to 1 and 1 to 0 Transitions
TG	Number of Transport Glitches
IG	Number of Inertial Glitches
IK	Inertial Glitch De-Rating Factor

Table D.2: SAIF Toggle Attributes. Source: [26]

D.2 Converting VCD to SAIF

Since no simulation programs are able to produce SAIF files directly, the information in VCD files must be used in stead. The files that are generated in a SystemC simulation, can't be converted directly. The VCD file must be modified as the following explains:

First, change VCD file from:

```
$scope module SystemC $end
$var wire    1  aaa  cp2         $end
$var wire    1  aab  ireset      $end
...
$var wire    8  aaj  reg_out     $end
$upscope $end
...
```

To:

```
$scope module top $end
$scope module avr_core $end
$var wire    1  aaa  cp2         $end
$var wire    1  aab  ireset      $end
...
$var wire    8  aaj  reg_out     $end
$upscope $end
```

```
$upscope $end
...
```

When the VCD file is changed, it can be converted with the Synopsys program `vcd2saif`:

```
vcd2saif -i trace.vcd -format VHDL -instance avr_core -o trace.saif
```

SAIF files are used with the Synopsys Power Compiler to annotate switching activity. This task is described in chapter 7.

Appendix E

Battery capacity

Batteries are divided into primary and secondary cells. Secondary have the extra feature of being rechargeable and also have higher capacity. This appendix will cover both types as well as two other technologies.

E.1 Capacity

Battery capacity is measured as $A \cdot h$ (Amps · hours) or Wh/kg (Watts · hours pr. kilogram). $A \cdot h$ values are measured for a battery by attaching the battery to a constant load. The time, h, is noted when the voltage V_{OC} drops to V_{CUT}.
Current Li-ion batteries have a capacity of 120-140Wh/kg [13, 29] and the annual increase for battery technology is only 5-10% a year [30]. Battery capacity is also written as e.g. 1600 mAh for a 45g 3.6 V li-ion battery.

> "Power, size, and longevity: choose any two" - Unknown

Since battery capacity is linear proportional to the size, a larger battery can always be chosen to meet demands. The disadvantage is that the time to recharge such a battery increases with the capacity. The cost also increase as the materials are rather expensive in high-capacity batteries.
An overview of current battery technologies is shown in table E.1.

Technology	**Rechargeable**	**Specific energy density**	**Voltage/Cell**
Pd-acid	No	38 Wh/kg	1.2 V
NiCd	Yes	56 Wh/kg	1.2 V
NiMH	Yes	76 Wh/kg	1.2 V
Li-ion	Yes	140 Wh/kg	3.6 V
Li-ion Polymer	Yes	168 Wh/kg	3.6 V

Table E.1: Current battery technologies and their capacities [29]

E.2 Other technologies

Methanol-powered fuel cells are looking very promising when looking at capacity but it will take a long time before extra methanol cells are available in every store. Another disadvantage is that fuel cells have to "warm up" so a device will either have to wait or use another source until the fuel cell is up to 100%.

If the power source is a solar cell, the device will be limited to daytime usage and solar cells are only usable in very low-power static-mounted devices.

Appendix F

Power statistics programs

Two small programs were created to analysis of input data and creating of Markov diagrams. The programs were designed for use with the AVR core. The `transitioncount` program only does bit statistics on 16-bit signals. The program could easily be modified to do the same for 32-bit signals.
The results of the programs are not used directly but they are presented as graphics in chapter 5.
Both programs work on specific lines of the AVR core debug output. These lines can be created by "piping" the debug output to the Linux tool, `grep`.
The source code for the programs are included in appendix H.8.1 and H.8.2.

F.1 Sequencegenerate

`sequencegenerate` uses the lines starting with "instruction: " for generating thousands of lines like these:

```
ldi    -> ret
call   -> ldi
ldi    -> call
```

These lines can then be sorted and grouped with equal lines to give a number of how many "ldi -> ret" sequences a program has.
This information is useful for creating Markov diagrams.

F.2 Transitioncount

`transitioncount` uses debug lines starting with "inst = ". These lines contain the instruction memory output as a 16-bit binary number. The program counts the number of bit shifts pr. bit in the signal and outputs the percentage of cycles where a shift takes place.
No distinction between rising and falling transitions are made, since every other transition clearly must be rising.

Appendix G

CD contents

With this report, a CD-ROM is included containing all the files created and used in the project. There are 4 folders in the root of the file system: `source`, `dist`, `bib` and `report`. The contents of the 4 folders is described below.

G.1 Source code

The source code for the AVR and MIPS cores are included in this folder. Some VHDL source is also included with all the different memory setups. The test suite of small programs for the two processors is also included.
In the source folders for each core, is a folder called `WORK`. These folder contains the core translated to VHDL. The files in these folders should not be modified for development. All changes in the design should be made with SystemC.

G.2 Distributions

This folder contains the current versions of the tools used. This includes programs like GTKWave, the SystemC 2.0.1 distribution, the ATMEL GCC for AVR and the TinyOS distribution. This folder should serve as an aid for anyone wishing to continue the work done in this project.

G.3 Bibliography

This folder contains articles, data sheets, reports and manuals. All references in the bibliography are included along with many other articles on low-power design.
Since the average page on the web has a lifespan of 100 days, the articles linked from the internet in the bibliography are included in a sub-folder named `www`.
Note that the naming of the files is not always precise enough to identify the document.

G.4 Report

The report is also included on the CD. There are two versions. The first version is the file `report_short.pdf`. This file does not contain the source code appendix. The file `report_full.pdf` does however contain everything.