Low-Power Processors For The Hogthrob Project

Andreas Vad Lorentzen

LYNGBY DECEMBER 2004
M. SC. THESIS
NR. 91

IMM

Printed by IMM, DTU

Preface

The work presented in this Masters thesis has been carried out by Andreas Vad Lorentzen.

This report is part of the results from the masters thesis project "Low-Power Processors For The Hogthrob Project" conducted at department of Informatics and Mathematical Modelling (IMM), Computer Science and Engineering division (CSE), Technical University of Denmark (DTU). The thesis was supervised by Jan Madsen and Jens Sparsø.

A special thank you to Martin Leopold for inspiring cooperation. The test programs which are described in sections 2.2.3 and 2.2.4 were implemented and used to measure the power consumption of an ATmega128L section 2.4.1 were part of this cooperation.

Further an appreciation is awarded to Hans Holten-Lund and Alberto Nannarelli for valuable insight information of using the FPGA and ASIC developing tools.

Finally thanks to Tina Chawes, Lars Bregnbæk and Nicolai Jørgensen for proof reading of this report.

The project is accompanied by a CD with all source code. These can also be found on the following homepage:
`http://vadlorenten.dk/andreas/dtu/thesis/`
A description of the CD contents can be found in appendix G.

DTU, 31st of December 2004

Andreas Vad Lorentzen

Abstract

The project was inspired by the Hogthrob project, which is an internal research project funded by the department of IT-research from May 2004 to 2007. Hogthrob is a sensor network project whose goal is to monitor the behaviour of sows in order to determine the exact time of their ovulation. The project involves three Danish universities and firms and among these participants is the department of Informatics and Mathematical Modelling (IMM) of Technical University of Denmark.

The contribution to the Hogthrob project was to explore the low-power AVR microprocessor Atmel ATmega128L on a special developed Hogthrob test board (A mote). TinyOS, an embedded operating system for sensor network motes, was running on the microprocessor.

The Nimbus microprocessor is a customised version of the AVR microprocessor from `opencores.org` for this project. The Nimbus microprocessor was synthesised for a Xilinx Spartan3e400, which is present on the board. It was then demonstrated that Nimbus microprocessor was able to run TinyOS. This is interesting because TinyOS is a very complex application.

The Nimbus microprocessor was synthesised for an ASIC based on two different cell libraries. This was done in order to compare the power consumption for the Nimbus microprocessor with the ATmega128L in the perspective of using the Nimbus microprocessor for sensor networks. The measurements of the power consumption showed that the ATmega128L consumed more power than the two Nimbus microprocessors. Furthermore the measurements showed that one had much lower dynamic power consumption than the other. The average execution of a microprocessor is only 1 % in a sensor network.

Therefore the microprocessor with the lowest leakage current was most appropriate for a sensor network. If it was possible to power down parts of the microprocessor, which were not in use, the other could be more efficient low-power microprocessor for sensor network.

The Disa microprocessor is an asynchronous version of the Nimbus microprocessor designed during this project. Disa was designed using the de-synchronisation technique. A design study of the de-synchronisation technique was successfully implemented. All the different components for the Disa microprocessor were implemented, but they werenot assembled.

Finally a programming flow was developed in order to help getting the Nimbus mi-

croprocessor to execute software programs. The programming flow defined a method to compile a software program, convert the binary program file in a hardware description and include hardware description in the hardware model for the Nimbus microprocessor.

Resume

Dette projekt er der hentet inspiration til fra Hogthrob projektet, som er finansieret af afdelingen for IT-forskning i perioden maj 2004 til 2007. Hogthrob er et sensornetværkprojekt, som har til formål at overvåge grises adfærd for at fastslå tidspunktet for deres ægløsning. Projektet involverer tre danske universiteter og firmaer og i blandt disse participanter er afdelingen for Informatik og Matematisk Modellering (IMM) ved Danmarks Tekniske Universitet.

Dette projekts bidrag til Hogthrob var at undersøge lav-effekt AVR mikroprocessoren Atmel ATmega128L på et speciel designet Hogthrob prøvebræt (en mote). TinyOs, et indlejret operativ system for sensornetværk motes, kørte på mikroprocessoren.

Til dette projekt blev der udviklet en specialkonstrueret version af AVR mikroprocessoren fra `opencores.org`, Nimbus mikroprocessoren. Nimbus mikroprocessoren blev syntetiseret til en Xilinx Spartan3e400, som er til stede på brættet. Det blev her vist, at det var muligt at køre TinyOs med Nimbus mikroprocessoren. Dette er interessant, da TinyOS er en meget kompliceret applikation.

Nimbus mikroprocessoren blev syntetiseret til en ASIC baseret på to forskellige cellebiblioteker. Dette blev gjort for at sammenligne effektforbruget for Nimbus mikroprocessoren med ATmega128L med henblik på at benytte Nimbus mikroprocessoren i et sensornetværk. Målingerne af effektforbruget viste, at ATmega128L har et større forbrug end begge cellebiblioteker. Ydermere viste målinger, at den ene Nimbus mikroprocessor havde et langt lavere dynamisk effektforbrug end den anden. Den gennemsnitlige køretid for en mikroprocessorer er blot 1 % for et sensornetværk.

Dette betyder, at mikroprocessoren med det laveste lækstrøm var den mest velegnede til sensornetværket. Hvis det var muligt at lukke dele af mikroprocessoren ned, når disse ikke var i brug, ville den anden Nimbus mikroprocessoren vil formentlig være den mest egnede lav-effekt mikroprocessor for sensornetværket.

Disa mikroprocessoren er en asynkron version af Nimbus mikroprocessoren, der blev designet i forbindelse med dette projekt. Disa blev designet ved hjælp af desynkroniserings teknikken. Et prøve eksempel af de-synkronisations teknik blev succesfuldt implementeret. Alle de forskellige komponenter for Disa mikroprocessoren blev implementeret, men disse var ikke sammensat.

Tilsidst blev der udviklet et programmeringsforløb for at hjælpe Nimbus mikroprocessoren med at køre software. Programmeringsløbet definerede en metode til at kompilere et stykke software, konvertere det binære program til en hardwarebeskrivelse og

inkludere hardwarebeskrivelsen i hardwaremodelen for Nimbus mikroprocessoren.

Word list

- **Atmel** Advanced Technology Memory and Logic.
- **AVR** People claims it stands for: Advanced Virtual Risc.
- **DIKU** Computer Science Department of University of Denmark
- **ELF** Executable and Linkable Format. A standard for binary assembly files in UNIX.
- **EMN** Electro-magnetic noise.
- **FPGA** Field Programmable Gate Array. A device that is programmed using VHDL.
- **GPL** Gnu Public License, a open source license.
- **IMM** Informatics and Mathematical Modelling of Technical University of Denmark.
- **KVL** The Royal Veterinary and Agricultural University of Denmark.
- **Motes** is the nodes is a sensor network.
- **Sensor Network** I a network of small embedded system, which can sense of transmit the information through a radio to base station.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Today, embedded systems are used for many purposes in the industry. An example is the co-operation between the University of California at Berkeley and Intel a small embedded device was developed which consisted of a microprocessor, some sensors and a radio transmitter. This small embedded device might be able to sense or monitor e.g. movements, temperature, humidity and/or acceleration. Using the radio, embedded systems are able to communicate with each other.

A sensor network is a network of many small embedded devices, which are able to sense and communicate. These small embedded systems are nodes called motes by Berkeley in the sensor network.

The aim of a sensor network is to use the motes to sense and transmit the information to a base station. The base station is where all the data from the network is collected and analysed.

The transmission can be done in two ways. The first method is a single hop where the information is sent directly to the base station. The second method is a multi hop where the information is transmitted through other motes to the base station. The motes might not be able to communicate with the base station and therefore the motes have to send the data through motes, so the data can reach the base station.

[1] describe different use of a sensor network. The most famous sensor network project, which is commonly applied as a literature example, is on Great Duck Island [2]. The purpose of the network is to observe the weather and nesting behaviours of seabirds on the island. The network consists of 150 motes. Figure 1.1 from [53] is an illustration of the sensor network on Great Duck Island. The motes are placed on the seabird nests in the ground (1) and just outside the nests exit (2). The monitored information is transmitted from the motes to a gateway (3). The gateway transmits the data to the base station (4) which has a satellite connection to the main lab (5).

There are many other examples of sensor network projects. For instance in Portland, Oregon and Las Vegas sensor networks are used to measure motion, pressure and infrared in elder care facilities in order to analyse activity of residents. The US military also uses sensor networks for instance in antitank mines. An other other project is about monitoring movements on Golden Gate Bridge.

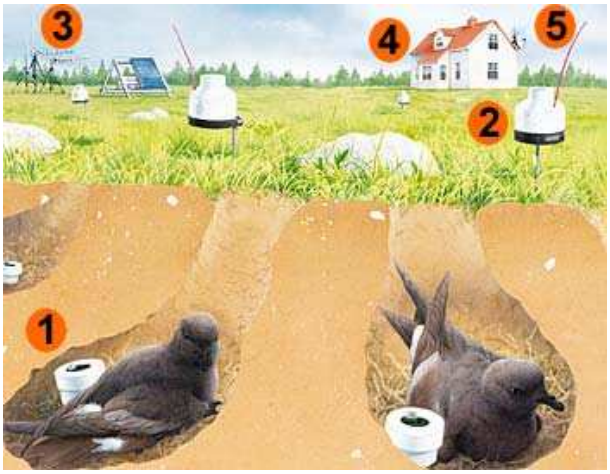


Figure 1.1 *Sensor Network on Great Duck Island*

1.1 Hogthrob - A Danish Sensor Network Project

Research has also been performed in the field of sensor networks in Denmark. Hogthrob is a national research project funded by the department of IT-research in may 2004-2007 [50]. This is a multi-disciplinary project between department of Informatics and Mathematical Modelling (IMM) of Technical University of Denmark, Computer Science Department of University of Copenhagen (DIKU), The Royal Veterinary and Agricultural University of Denmark (KVL), National Committee for Pig Production (NCPP) and IO Technologies. The goal of the project is to develop a sensor network that monitors the behaviour of sows. This is done by placing a sensor mote on the back of the sow.

The monitoring of sows is important in order to stat if the sow is ovulating and needs to be fertilised. A sow is only fertilised over a limited period of a few days. In this period the sow is very active and moves around a lot in the pen. The movements can be observed by an accelerometer which is connected to a sensor board. Information regarding the sows behaviour is then sent from the sensor to a stationary computer using a radio transmitter. Under normal circumstances it is up to the sow farmer to observe the sow and find the right moment of artificial insemination. This can be complicated if the farmer has 10.000 sows.

1.2 Issues of a Sensor Network

A mote consists, as mentioned, of a small microprocessor, some sensors and a radio transmitter. The problem with the mote is that it runs on batteries and the radio uses a lot of energy. If the microprocessor in the mote runs all the time and uses the radio, the batteries will only last a few days [1].

To handle this problem a range of measures have been taken. The motes have to be equipped with a low-power microprocessor. A low-power microprocessor needs to feature a sleep mode, which allows the microprocessor to turn off parts of or the whole chip, when the chip is idle. This includes the sensors and radio transmitter.

The speed requirement of a microprocessor in a sensor network is limited. The microprocessor only needs high speed operation when transmitting and receiving data through the radio or when sensing.

The mote has to be equipped with a low power radio. This entails that the mote has a limited transmission range of less than 30 meters. Transmission of longer ranges consumes too much energy.

The motes can be placed in hostile environments or indoors. These areas can have obstacles that influence the radio transmission. This can lead to a retransmission of the messages and will lead to unnecessary use of energy.

The size of the motes is an important factor to take into consideration. The motes must be really small so they can placed in the field e.g. in nests on Great Duck Island.

When the motes are placed in the field like the Great Duck Island it is very important that the motes a reliable. The birds on the island are preserved and the time is therefore limited to place the motes because the birds are not allowed to be disturbed. If the motes break down, they are lost.

1.3 Product Solution for Sensor Network

A microprocessor, which can handle the above mentioned issues, could be an AVR microprocessor from Atmel [46]. AVR is a small 8-bit RISC processor. The microprocessor is designed specifically for low-power consumption and is easy to integrate with sensors and radios. It has been used for many years and is therefore very reliable.

The AVR microprocessor can be used with the embedded operating system from Berkeley called TinyOS [47]. TinyOS is specially designed for sensor networks. It has the characteristic that when the AVR microprocessor is not executing, it is put to sleep. This is done to minimise the power consumption.

Normally in a sensor network the motes spend 99 % of the time dormant. Only 1 % of the time is used for transmission and sensing. Both AVR and TinyOS support these conditions.

1.4 Low-Power Techniques

There are many possibilities to optimise a microprocessor for low-power. Some are easier to implement than others. The following is a description of four such techniques:

1.4.1 Synchronous Circuits

A way to reduce power consumption for a synchronous digital circuits is to lower the voltage. This might have some timing consequence where data would not be ready in time.

Another option is to lower the dynamic power consumption by stopping the clock for the microprocessor. This can be done by clock gating or by stopping the oscillator. Nevertheless the leakage current in the microprocessor will still be a problem.

1.4.2 Leakage Current

Leakage current is becoming an increasing factor of low-power design. Figure 1.2 from [3] shows the power consumption in relation to the gate length.

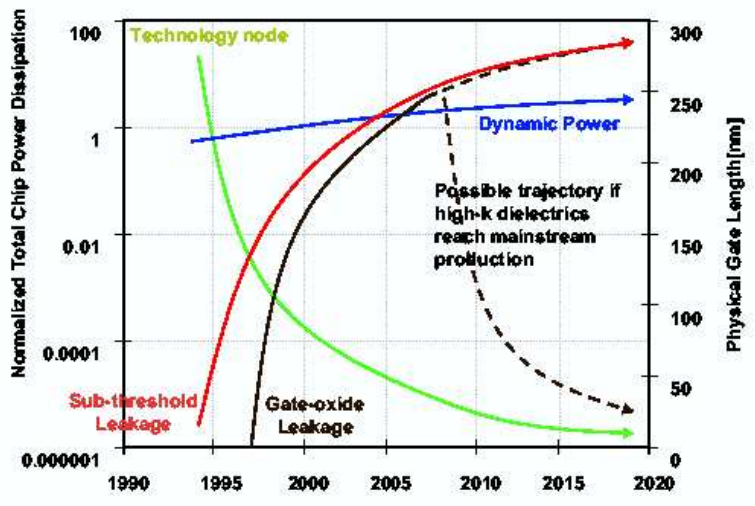


Figure 1.2 *Leakage power consumption versus technology from [3]*

There are three openly ways to reduce the leakage power. The leakage power may be reduced by lowering the voltage. This can involve timing problems in a synchronous circuit

A more dramatic way is to power down the parts of the microprocessor that are not in use. It costs a lot of power to power up the parts of microprocessor again, but in

a sensor network, this might be a good solution because it spends so much time in a dormant state.

Finally It is possible to reduce the leakage current by changing the gate length.

1.4.3 Synthesis Tools and Cell Library

The chosen cell library could also influence the power consumption of the microprocessor. Newer cell library technologies have showed that the dynamic power consumption for circuit have decreased in comparison with an old cell technology when they are running at the same frequency. But the new cell library also has increased leakage current and this may result in that is better to use an old cell library. A cell library can be designed for low-power, this can be done by using low-power transistors.

Modern synthesis tools have limited possibilities for optimising a circuit for low-power. Normally the synthesis tools can only optimise for speed or area, but a minimisation with respect to size will also lead to an reduction of leakage current.

1.4.4 Asynchronous Circuit for Sensor Networks

Asynchronous circuits are interesting because the design technique could lead to a reduction of the dynamic power consumption. [5] introduces low power implementations of asynchronous circuits for DCC error correction. The article explains different asynchronous techniques for an implementation of the circuit. The article shows that the asynchronous circuit obtains reduction of power consumption up to 5 times compared to the synchronous version. The cost to use an asynchronous design is an increase of area of 20 % due to the increase in control logic source.

The articles [6, 7, 8] explain the advantage of self-timed or delay-insensitive circuits for design of asynchronous microprocessors. They show the robustness of chips with different supply voltage and temperature. An asynchronous circuit can operate acceptably on a much lower supply voltage than a synchronous circuit. Asynchronous circuits also support the possibility of continuant changes in supply voltage due to the self-timed circuit. A synchronous circuit may suffer from clock skew because of the low voltage.

This results in an asynchronous circuit supporting the possibility to go into a standby mode with zero dynamic power consumption and that the circuit has limited wakeup time due to the delay-insensitive design. Clock gating for synchronous circuit might achieve similar benefits.

An other advantage of asynchronous circuits is the reduction of electro-magnetic interference (EMI). The articles [9, 10, 11] illustrate the difference in the level of electromagnetic noise between a synchronous circuit and an asynchronous circuit. The asynchronous does not have the EMI peak.

The problem with EMI is that it might influence the performance of an A/D converter. This is important for the radio receiver on a mote and it means result that some data having to be retransmitted.

This is why asynchronous circuits are particularly interesting for sensor networks. The technique may reduce the power consumption and supports the possibility to wakeup, execute an event and fall back to sleep.

This results in that an asynchronous design technique may reduce dynamic power consumption. An asynchronous circuit is more robust and has less EMI than a synchronous circuit. The disadvantage is an increase of circuit area which will increase the leakage current.

1.5 Project Goals

The project goal is split up into three parts.

- The first part is to explore the architecture of a AVR microprocessor and get TinyOS running on the AVR microprocessor. This includes measurements of power consumption using the different power saving modes in the microprocessor.
- The second part of the project is to download and explore an AVR microprocessor from the website `opencores.org`. A work flow has to be structured in order to program and test the microprocessor. The microprocessor should be able to run on a FPGA and to be synthesised for an ASIC library. Finally some power estimation for the microprocessor is to be calculated.
- The third part of the project covers how to convert the customised synchronous AVR to an asynchronous AVR microprocessor using the de-synchronisation method.

Finally all three microprocessor will evaluated.

1.6 How to Read This Report

The report contains 7 chapters and an appendix:

Chapter 1 is an introduction, which describes the background and aim of the project. The chapter discusses the problems with sensor networks, interesting issues of asynchronous design and how it is possible to reduce the power consumption of a microprocessor.

Chapter 2 describes the ATmega128L, an AVR microprocessor from Atmel and a special design test board for the Hogthrob project. The test programs are all described and these are used for measuring the power consumption of the ATmega128L. Finally TinyOs is explored.

Chapter 3 contains a description and evaluation of the AVR microprocessor from Opencores called Nimbus. Next a description of the design flow which is used to program and test the microprocessor. The Nimbus microprocessor is synthesised for the FPGA on the Hogthrob board and an ASIC library and then the power consumption for the Nimbus microprocessor is measured based on the ASIC library.

Chapter 4 explains the theory of asynchronous design and the technique for implementation of de-synchronous AVR microprocessor called Disa.

Chapter 5 compares the measurement of power consumption for the ATmega128 and Nimbus and looks at which microprocessors are best for use in a sensor network.

Chapter 6 is a discussion of the measurements and possibilities for the microprocessors. The experience of using the downloaded core and the programming tools are also discussed.

Chapter 7 sums up the present work and the achievements and at the end there is a recommendation for future work.

The appendix contains waveforms, pictures of the ATmega128, measurements and all the source code.

CHAPTER 2

ATmega128L

The object of this chapter is to introduce the ATmega128L microprocessor. The ATmega128L microprocessor is a low-power AVR compatible microprocessor from Atmel.

To begin with there will be a discussion of reasons to select the Atmel ATmega128L microprocessor for the Hogthrob project. The microprocessor will be compared with some other low-power microprocessor which are available on the market. The introduction of the ATmega128L will include a description of the architecture and features. The Hogthrob board has some hardware components which can be used by the microprocessor which are relevant from a sensor network perspective.

In order to measure the ATmega128L microprocessor there has been written some test programs. This chapter includes a description of the different test programs and the embedded operating system TinyOS . The test programs are used to measure the power consumption of the AVR ATmega128L.

The reason for exploring the ATmega128L is that later it is going to be compared with an other AVR microprocessor.

2.1 The Hogthrob Board

For the Hogthrob project, a special hardware test board was designed, which will be refereed to as the Hogthrob board. This board contains a microprocessor and some other sensor network related components that are described later. In the next section, reasons for selecting the AVR microprocessor for the board will be described.

2.1.1 Selection of the Microprocessor

There are many companies today which develop low-power microprocessors. Some of them are more useful than others in a sensor network .

Table 2.1 is a list of different low-power microprocessors. The microprocessors are from Atmel [12], Motorola [13], Intel [14], MIPS [15] and Texas Instruments [16]. The first column contains the product name of the microprocessors, the next column contains the data size and the instruction set size for the microprocessor. The last column

states whether the microprocessor is supported by the GNU programming tools [55].

Product	Inst	Data	Supported by GNU
Atmel Atmega128 (AVR)	16-bit	8-bit	yes
Motorola HCS08	8-bit	16bit	no
8051	8+	8-bit	no
Mips16 (TinyRisc)	16-bit	32-bit	yes
TI MSP430	8-bit	16-bit	yes

Table 2.1 *Table include names of different low-power microprocessors and whether the microprocessors are support by GNU.*

All the microprocessors can be used in a sensor network but some of them are more suited than others. An example of this is the HCS08. It has an onboard radio, but the controller is not supported by GNU programming tools. The GNU tools include many useful programs as a compiler and debugging tools. The problem is also that TinyOs is only supported by the GNU compuler, see section 2.3.

The reason for choosing the AVR microprocessor as the platform for the project was that the embedded TinyOS was originally written for AVR. This means that it is 100 % supported and there are many people which can assist if there is a hardware comparability problem with TinyOS.

The people from DIKU have also done some previous research using the ATmega128L microprocessor on another sensor network platform. The microprocessor has many I/O ports which are easy to use and the I/O ports can be connected to sensors.

The MSP430 would also have been a good choice. It is a new ultra low-power microprocessor. The microprocessor is interesting because it just recently the processor got supported by TinyOs [17]. The MSP430 has the advantages that it is not a Harvard architecture. It has a combined instruction and data memory instead of two separated memories as in the ATmega128L microprocessor. If the whole Hogthrob project was restarted today, it would probably included the MSP430 microprocessor on the board.

2.1.2 Description of ATmega128L

The ATmega128L is from Atmel (Advanced Technology Memory and Logic) and it is a high performance, low-power AVR 8-bit microprocessor. [19] is the documentation of the ATmega128L and contains a complete description of the architecture. [18] is a description of the AVR 8-bit instruction set. There is no official translation of the abbreviation from Atmel part, but people claim that AVR stand for Advanced Virtual RISC [20].

The ATmega128L has a two stage pipeline. The first stage is a instruction fetch stage and is used for instruction pre-fetch. The second of the is the execution stage, which handles the instruction decoding, instruction execution and memory access. The AVR instruction set is 16 bit and many of the instructions are multi-cycle.

The ATmega128L uses a Harvard memory architecture. It means that the microprocessor has a separated instruction memory and data memory.

The GNU tools are used to compile the programs for the AVR microprocessor. The AVR boot strap from GNU tools works as followed. When an AVR microprocessor is booted, it copies all variables, which are used in the system, from the instruction memory to the data memory. This is done by the instruction called load immediate. This may take some time if a program has many global variables.

The microprocessor has many interesting components as UARTs, timers, a Serial Peripheral Interface (SPI) and different sleep modes from a sensor network perspective.

The timers can use the microprocessor clock (the internal clock) or an external clock. Section 2.1.3 describes how to setup timer.

The timer invokes a interrupts when the specified time has elapsed. This action stops the current process in the microprocessor. The microprocessor jumps to interrupt vector, which is placed in the first part of the instruction segment. The interrupt vector specifies the address for the program segment, which should take care of the interrupt. After the interrupt has been handled, the microprocessor returns to the place it was interrupted from.

The ATmega128L has six different sleep modes which can be used when the microprocessor does not have to execute parts of a program. The clock is stopped in order to reduce the power consumption. The different sleep modes are described in section 2.1.4.

The ATmega128L has seven 8-bit ports, which is called PORTA, PORTB, ... , PORTG. The pins of the port are named in the following way: pin 6 of port B is called PB6. All the ports are connected to the main bus and use the same port components. Some of the pins are used for more than one thing. For instance PORTE pin 2 and PORTE pin 3 are used for the SPI and the UARTs are connected to other pins. The SPI can be used for radio transmission.

2.1.3 Timer Setup

The ATmega128L has two 8-bit timers and two 16-bit timers. The timers have individual prescalers and comparison registers. The prescaler is used to divide the clock with 1,2,4,8 and so on to 1024.

A timer can be used in two ways. It can signal an interrupt when the compare register is equal to the timer counter. The other possibility is when a timer wrap around it can signal an interrupt.

Table 2.2 shows the register names and the names of bit in a register which are used by TIMER0 that is one of the 8-bit timers. Not all the bits in the registers are described and this is because they are not used by TIMER0.

The timer also supports the use of an external clock. The external clock is much slower than the system (internal) clock. The timer can be set either to internal or external clock.

Register	Bit	Bit Name	Description
TCNT0	7-0		Timer/Counter0 (8 Bit) (Timer 0)
OCR0	7-0		Timer/Counter0 Output Compare Register
TCCR0			Timer/Counter Control Register
	2-0	CS02:0	Clock Select (prescale)
TIMSK			Timer/Counter Interrupt Mask Register
	1	OCIE0	Timer/Counter0 Output Compare Match Interrupt Enable
	0	TOIE0	Timer/Counter0 Overflow Interrupt Enable
ASSR			Asynchronous Status Register
	3	AS0	Asynchronous Timer/Counter0
	2	TCN0UB	Timer/Counter0 Update Busy
	1	OCR0UB	Output Compare Register0 Update Busy
	0	TCR0UB	Timer/Counter Control Register0 Update Busy

Table 2.2 *Appropriate register overview for the ATmega128L timer.*

2.1.4 Sleep Modes

The ATmega128L has six different sleep modes. Table 2.3 shows the sleep register and table 2.4 is a decoding of the sleep modes. The different sleep modes behaves as follows:

- `Idle:` The CPU is stopped but all the other components are still running i.e. UARTs, ports, timers and SPI.
- `ADC Noise Reduction:` The CPU and the ports are stopped to reduce the noise electromagnetic interference.
- `Power-Down:` The whole ATmega128L is turned off. Only an external interrupt can wakeup the microprocessor. The oscillator is also stopped.
- `Power-Save:` The same as Power-Down mode except, that the TIMER0 is still working if the external clock is enabled.
- `Standby:` The sleep mode is the same as Power-Down except that the Oscillator is not stopped.
- `Extend Standby:`The sleep mode is the same as Power-Save mode except that the oscillator is not stopped.

Register	Bit	Bit Name	Description
MCUCR			MCU Control Register
	5	SE	Sleep Enable
	4-2	SM1:SM0:SM2	Sleep mode select

Table 2.3 *The MCUCR register*

SM2	SM1	SM0	Sleep Mode
0	0	0	Idle
0	0	1	ADC Noise Reduction
0	1	0	Power-Down
0	1	1	Power-Save
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Standby
1	1	1	Extend Standby

Table 2.4 *Sleep Mode select*

2.1.5 Description of Hogthrob Board

For the Hogthrob a special test board was designed. The board was designed by IMM and DIKU. The precise description of the design can be found in the internal pages on the Hogthrog homepage. The paper [21] is a technical description of the Hogthrog board. This paper includes a figure, which shows a hardware overview of the different components and how they are connected. The figure is included in appendix B. The board was printed by IO Technologies [56]. They were also responsible for the acquisition of all the components for the board. Figure 2.1 shows a picture of the board.

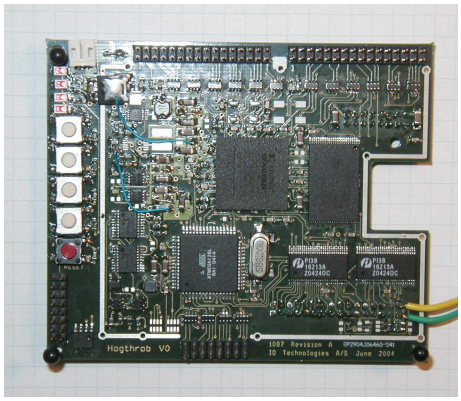


Figure 2.1 *Overview of the Hogthrob board*

On the Hogthrob board an ATmega128L, a Spartan3, a temperature sensor and memory is mounted. The ATmega128L is a microprocessor, which is described in section 2.1.2. The Spartan3 is a Spartan3 XC3S400 FPGA from Xilinx. The FPGA is described in section 3.4. The board also has LEDs and buttons, which can be programmed for different purposes.

The Hogthrob board has a interface to connect a radio board and sensor board. The radio board is designed so it can be placed on the back of the Hogthrob board.

2.2 Tests

In the order to test and evaluate the performance according to power consumption for ATmega128L some test program were developed.

The next section describes shortly how to program a AVR microprocessor. After this the different test programs and their purpose are described.

2.2.1 Programming of the Microprocessor

There are many good tools for programming a ATmega128. Atmel has an official homepage with programming tools for the Windows platform [60]. Another good place to look for tools is on AVR freaks homepage [61]. They have discussion forums and programing tools. The programming tools are based on the GNU compiler and is working on Windows and Linux. For this project the GNU tools were used.

The GNU programming tools can be used in the following way. `gcc` is used to compile and link a program. The binary program object is copied to a file in `srec` format. `srec` is a special format, which is used by `uisp` to program the ATmega128. `uisp` uses the serial port for programming the ATmega128. The `make` script can be found in appendix H.7.13. The programming can be done as followed:

```
avr-gcc -mmcu=atmega128 timer_blink.c -o timer_blink
avr-objcopy --output-target=srec timer_blink timer_blink.srec
uisp -v=3 -dprog=stk500 -dserial=/dev/ttyUSB0 -dpart=ATmega128 \
--erase --upload if=timer_blink.srec
```

Programming of a AVR microprocessor using GNU tools

The example is the compiling, object copy and programming of the an example called timer blink.

2.2.2 GNU AVR Library

The GNU library for AVR includes many useful functions. A C/C++ program must include the following lines to use the AVR library:

```
#define __AVR_ATmega128__ 1    // ATmega128L microprocessor
//#define __AVR_ATmega103__ 1 // or ATmega103 microprocessor
#include <avr/io.h>           // AVR headerfile
```

Header include of for AVR programmes

First the microprocessor is defined. This involves that when the AVR header files are included the correct variables and constant are available.

The AVR library includes some functions to define the behaviour of a port. There are some variables which refer to a port. The following listing is an example of how to

set port B bit 6 as an output signal and to set the values to 1. Function `_BV(6)` sets the sixth bit in a byte to 1 i.e. $0x20$ or $b00100000$ and together with the discrete mathematics it is possible to control the behaviour of the port.

```
DDRB  = _BV(6);  // Set port B bit 6 to a output port.
PORTB = _BV(6);  // Write a one to port B bit 6.
```

AVR library function to set 1-bit in a register.

The function `sbi` and `outp` are two other often used function. `sbi` is used to enable specific functionality in a AVR microprocessor. In the following example the *AS*0 bit in the *ASSR* register is set. The *ASSR* register is used to enable the external clock. The function `outp` stores a value in a specific register e.i. the $TCNT0$ is set to zero.

```
sbi(ASSR, AS0); // Enable async timer clock
outp(0, TCNT0); // Reset the timer0 counter
```

AVR library function to write a value in a register.

Further AVR microprocessor programming functionalities can be found in the documentation for the AVR library. All the register names and specific bit names can be found in the AVR microprocessor description. In the next section a small AVR program is described.

Timer Blink Example

The following example switches a LED on and off on the Hogthrob board. The example uses the external clock. The time between the led switches approximated one second if using an external clock of $32kHz$. The code can be found in appendix H.7.10. The program works as follows. First the led is turned on, next the external clock is enabled and the timer is reset and scaled by 2^7. Then the system waits for the external timer to be ready. When this happens the timer compares registers is reset and the microprocessor interrupt is enabled. The program then enters an infinite loop where the sleep mode is set and the microprocessor goes to sleep.

When the time is gone the microprocessor wakes up and the program counter jumps to the segment which handles the timer interrupt. The timer interrupt is disabled. The led is then turned on/off. The timer is reset and the interrupt for the timer is enabled. The program returns to the place where it went to sleep and waits for the external clock to start again. When this happens the microprocessor goes back to sleep.

```
#define __AVR_ATmega128__ 1
#include <avr/io.h>

int main(void) {
  DDRB = _BV(6);  // Enable the port for writing
  PORTB = _BV(6); // Turn the port on

  sbi(ASSR, AS0); // Enable async timer clock
```

```
  outp(0, TCNT0); // Reset the timer0 counter
  outp(7, TCCR0); // Scale the timer by 1024

  // Wait for the async clock to get going
  while (ASSR & 0x07);

  outp(128, OCR0); // Set the timer output compare to trigger at 128
                   // ticks.

  sbi(TIMSK, OCIE0);   // Enable the Output Compare interrupt
  asm volatile("sei"); // Enable interrupts

  while (1) {
    // Set the MCUCR so that we enter power-save mode (which will
    // leave the async clock going).
    MCUCR &= ~0x3C;
    MCUCR |= _BV(SE) | _BV(SM1) | _BV(SM0);
    asm volatile("sleep");
    asm volatile("nop");
    while (ASSR & 0x07); // Wait for the async clock to get going
                         // again.
  }

  return 0;
}

// Function to handle timer interrupt
void __attribute((signal)) SIG_OUTPUT_COMPARE0()
{

  cbi(TIMSK, OCIE0); // Disable the Output Compare Interrupt
  // Toggle the mb-led
  PORTB = PORTB & _BV(6) ? PORTB & ~_BV(6) : PORTB | _BV(6);
  outp(14, TCNT0);    // Reset the timer counter
  sbi(TIMSK, OCIE0); // Enable the Output Compare Interrupt
}
```

Timer example which turn on and off the led.

2.2.3 Memory and Arithmetic Programs

To test the power consumption for the ATmel128L, three programs were developed. All the programs are programmed in C and the source code can be found in appendix H.7. The programs were designed as followed:

- **add.c** tests how much energy the arithmetic function uses. The program uses embedded assembly to program the specific behaviour. The program uses minimal access to the data memory.
- **add-mem.c** is an arithmetic function and memory access. The idea was to make a program which uses the memory instruction as well as the arithmetic instruction.
- **encode_decode.c** is an implementation of the Hamming algorithm. This algorithm is an error correcting algorithm, which consists of encoding and decoding. It uses nearly all the data memory (4kByte) on the ATmega128L. The Hamming algorithm is interesting because it could be used in radio communication and it is a larger and more complex program.

2.2.4 Sleep Modes Programs

Another interesting thing to measure the efficiency of the different sleep modes for the microprocessor. To test the sleep modes on the ATmega128L six programs were written, one for each sleep mode except the reduce noise mode. A program with a infinity loop was also written. In table 2.5 is a list of which programs that test which sleep mode.

Name	Description
nop	A tight loop of no-operation instruction
idle	Idle mode of ATmega128L
power-save	Power-save mode of ATmega128L
power-down	Power-down mode of ATmega128L
standby	Standby mode of ATmega128L
ext-standby	ExtStandby mode of ATmega128L

Table 2.5 *Sleep mode test programs*

2.3 TinyOS

TinyOS is a embedded operating system, which is specially designed for sensor networks applications. The TinyOS project was started at the Computer Science Department at University of California, Berkeley [47] and is now a open source project and can be found on SourceForge.Net [48]. TinyOS is written in nescC [49], which has a C like syntax, but uses a special way of encapsulating the programming code. TinyOS provides the surroundings for the application and the program is given a strict interface for using hardware.

TinyOS is designed specifically for sensor networks in the way that it is a event driven operating system. The article [26] and slides [27] give a good introduction of the possibilities with TinyOS.

In a sensor network the motes spend the majority of time at sleep to reduce power. TinyOS is able to handle when motes wake up because of an event and the operating system is quick to start the processing. When the event activity is done, TinyOS makes the mote to return to sleep.

TinyOS was originally designed for the AVR microprocessor but now it also supports microprocessors from ARM and Texas Instruments. A big issue of sensor networks are radio communication and sensing and TinyOS has special support for radio communication. It has routing and broadcasting functionalities.

2.3.1 Hogthrob Platform

For the Hogthrob project has people from DIKU designed a special version of TinyOS, which supports the Hogthrob board. It can be found on the Hogthrob private home-

page.

2.3.2 Blink Example

The TinyOS blink example is the most simple example. Like the blink example from section 2.2.2 the blinking has a specific time interval. The example is designed in a way, so that it wakes up four times before the led is turned on or of. The source code for the blinking example can be found in appendix H.7.11 and in the TinyOS repository.

2.4 Measurements

The ATmega128L on the Hogthrob board has been tested by people from DIKU and everything seems to be working. As part of the experience with the Hogthrob board both timer blink and TinyOS has been executed on the microprocessor and they are working as expected.

2.4.1 Power Estimation

An important part of the investigation of the ATmega128L is to measure the power consumption of the microprocessor. The power consumption is very important from a sensor network perspective.

Unfortunately it was not possible to measure power consumption on the Hogthrob board because a resistance was blocking. Instead of using the measurements of the power consumption from the ATmega128L on Hogthrob board, it was done on a BTnode (version 2) [54], which has an ATmega128. A BTnode is a sensor network mote. The BTnode has a Bluetooth radio. Figure D.1 shows a picture of the BTnode. The red lines in the top of the picture are used for power consumption estimation and the red lines in the button is for programming the BTnode. More pictures of the setup are shown in appendix D.1.

Table 2.6 shows the results of measuring where current and power consumption of the test programs running 7 MHz and with a supply voltages of 3.273 V. The measurements show that the execution of a normal program has a current consumption of less than $10mA$.

The experiment illustrates that it is very important that the microprocessor is put to sleep if it is not executing a program. It can be seen that the `nop` program has the highest average consumption. This is a known problem that exists on all PCs. Powerdown is the most efficient sleep mode, but the drawback is that it takes several clock ticks to wakeup the microprocessor.

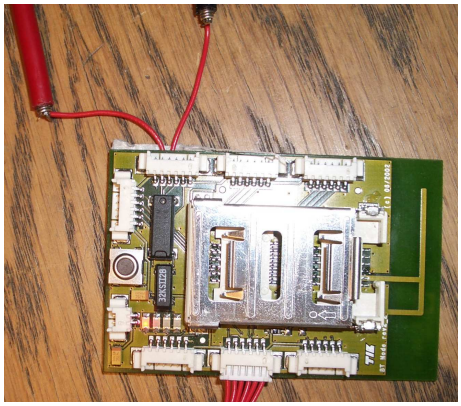


Figure 2.2 *Sensor network BTnode.*

Program	Current	Power
nop	14.5mA	47.5mW
idle	5.20mA	17.0μW
power-save	11.8μA	39.0μW
power-down	11.9μA	38.6mW
standby	0.71mA	2.32mW
ext-standby	0.71mA	2.32mW
add	9.18mA	30.0mW
add-mem	9.75mA	31.9mW
hamming	9.88mA	32.3mW

Table 2.6 *Current and power consumption for ATmega128L.*

2.5 Discussion

The ATmega128L is an easy microprocessor to understand. Both the architectural and instruction documentation from Atmel are excellent. They include small program examples which illustrates the utilisation of instructions or a special part of the microprocessor. It is understandable that computer system engineers like the AVR microprocessors. This is because the microprocessors has many ports and timers which can be used to control the behaviour of other hardware components.

2.6 Summary

This chapter has presented the ATmega128L and others microprocessors. It has been showed how the ATmega128L microprocessors can use the ports and timers. Some test programs have been written for the AVR microprocessors and the current consumption has been measured for the test programs. Finally TinyOS has been discussed.

CHAPTER 3

Customised Synchronous AVR

This section is about the open source AVR microprocessor named AVR_CORE. A customisation of the microprocessor will be referred to as Nimbus microprocessor. The customisation for the Nimbus microprocessor is implemnted so it can be synthesised for a FPGA and an ASIC.

This will lead to a new design of the sensor network platform, which can easily be customised for special purposes. The Nimbus microprocessor can be used for estimation of power consumption while executing different sensor network tasks like sensing or doing radio communication.

This chapter describes the AVR_CORE and the differences between the AVR_CORE and the ATmega128. There will also be a description of the customisation of the microprocessor together with the implementation of the AVR_CORE microprocessor.

A tool flow has been developed for programming and synthesising the Nimbus processor. The programming tool flow and the synthesis tools for FPGA and ASIC synthesis will be described.

At the end of the chapter the tests and measurements of the Nimbus microprocessor will be described and the microprocessor is discussed.

3.1 Open Cores

Open Cores [51] is a place like Source Forge, which has open source projects. Open Cores have many interesting processors and I/O interfaces.

Open Cores there is a project called AVR_CORE [52], which is a implementation of a look a like Atmel ATmega103. A description of the ATmega103 can be found here [22]. Both ATmega103 and ATmega128L use the same instruction set.

The AVR_CORE contains:

- Core
- Program memory
- Data memory
- UART
- Timer/Counter

- PORTA and PORTB

The AVR_CORE has limitations. The microprocessor does not support the instructions concerning the sleep modes and the watchdog control. This means that the clock cannot halt and different parts of the core cannot be disabled. Furthermore there is no implementation of a external clock. The watchdog is used to restart the microprocessor after a given time period.

The ports from C to F are not implemented. Port A and B are implemented as parallel ports and they do not support the advanced port functionalities.

3.1.1 Some Differences Between AVR_CORE, ATmega103 and ATmega128

The differences between an ATmega128L and ATmega103 can be found here [23]. The limitations for the AVR_CORE are described in the previous section and they are all supported in ATmega128L. The ATmega128L has another 2 timer counters where each of the them have their own prescaler as described in section 2.1.3.

The ATmega128L has two UARTs and one SPI, where the AVR_CORE only has one UART. The external memory interface for the ATmega128 has been improved compared to ATmega103. In the AVR_CORE the external and internal data memory are treated as one.

The AVR_CORE has some analog limitations compared to the two other microprocessors. Therefore the AVR_CORE does not have an implementation of an oscillator. A simple model could be implement in VHDL using and-gates and inverters.

3.2 Customisation

This section describes the customisation of the AVR_CORE microprocessor. The customisation includes changing the the RAM and ROM, implementation of the sleep modes and correction of bugs. To start with there will be a description of the Nimbus architecture.

3.2.1 Description of the Architecture

Microprocessor

The microprocessor will be described in a top down approach. Figure 3.1 shows the top level of the microprocessor. At the top-level it is possible to see the internal clock the external clock, and the reset signal. The external clock is connected to the timer and the internal clock is connected to the sleep control unit. The sleep control module regulates the clock for the other parts of the system. The sleep control module is described in detail in section 3.2.3.

The core is placed in the middle of the figure and is connected to the instruction memory (ROM), data memory (RAM) and the I/O hardware components. The core

is connected to the data memory and the I/O components through a common data bus. As described in the ATmega103 manual the I/O components memory is mapped. Each component listen to the address signal to find out if the data on the bus is the component. The data bus is made as input and output signal to the core.

The I/O ports of the Nimbus microprocessor is placed in left part of the figure. In the top left corner just beneath the timer are the I/O pins for external interrupts. Then there are the UART and the two ports.

The I/O and interrupt component take care of the sequence of interrupt and which IO hardware component or the data memory has the right to write the data bus. The interrupts are arranged into a vector, so the microprocessor can handle multiple interrupts.

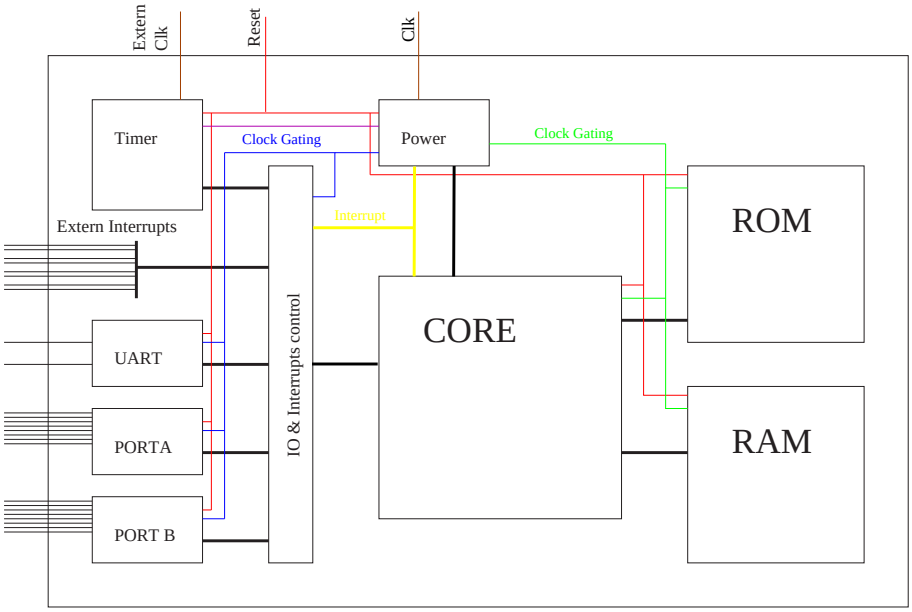


Figure 3.1 *Top level of the Nimbus design*

In the listing below are names of the components, the belonging file and a short description of the component purpose. The source for the file can be found in appendix H.2.

- **Top level (top_avr_core_rtl.vhd):** Top-level design of the Nimbus microprocessor.
- **Package (AVRuCPackage.vhd):** Constants and types.
- **IO & Interrupts (external_mux.vhd):** Data bus multiplexer and interrupt vector.
- **IO & Interrupts (Service_Module.vhd):** Special registers.
- **IO & Interrupts (RAMDataReg.vhd):** Data-bus register.
- **ROM (rom_binary/ram16bit_XXX.vhd):** This is the program memory where the XXX is the name of program.

- **RAM (data_ram_rtl.vhd):** Data RAM.
- **Port A (porta.vhd):** Parallel ports A.
- **Port B (portb.vhd):** Parallel ports B.
- **Timer (Timer_Counter.vhd):** Timer/Counter.
- **Extra Timer (simple_timer.vhd):** Simple timer.
- **UART (uart.vhd):** UART
- **Sleep Control (power_control.vhd):** Sleep mode control.

Core

The core is the main part of the Nimbus microprocessor. Figure 3.2 shows the connections between different components in the core.

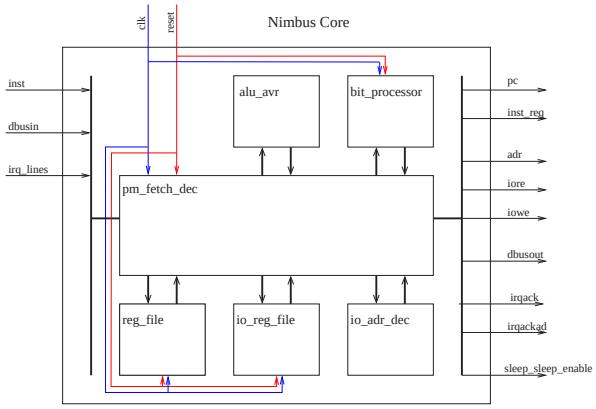


Figure 3.2 *The structure of the Nimbus core.*

- **CORE (avr_core.vhd):** top-level design of AVR core.
- **ALU (alu_avr.vhd):** ALU.
- **Bit processor (bit_processor.vhd):** The Bit processor is used for bitwise operation like XOR.
- **Register file (reg_file.vhd):** Register file.
- **Decode (pm_fetch_dec.vhd):** Includes program counter, instruction decoder, memory and I/O memory.
- **IO Register file (io_reg_file.vhd):** I/O registers.
- **IO Address Decode (io_adr_dec.vhd):** Address decoder and data bus multiplexer for the I/O registers.

3.2.2 ROM & RAM

The ROM and RAM specification has been chanced to a VHDL description that is supported by Xilinx XST standard 3.4.1. The size of the RAM has chanced from 128Byte to 4096Bytes and the size of ROM depends on program 3.4.1.

The ROM reads and RAM reads and writes are not done on the event when the clock signal becomes zero. Before the ROM and RAM were accessed asynchronously.

3.2.3 Sleep Mode

The sleep mode for the Nimbus microprocessor has been implemented as described in the ATmega103 documentation. The difference between the ATmega128 and ATmega103 is that ATmega103 does not support the standby functions as described in section 2.1.4.

The sleep mode is implemented in form of clock-gating. The sleep instruction is executed, the sleep mode state machine looks in the sleep mode register. If the `idle` mode is set, the clock for the Core, RAM and ROM is stopped. This means that interrupts from external interrupt, the timer, ports or UART can wake the microprocessor up.

The `Power-Save` mode stops all the components except the timer and the external interrupt and the `Power-Down` mode stops everything so only an external interrupt can wake up the system.

On figure 3.1 it is possible to see the top level of the Nimbus microprocessor. The green signal is the clock for the Core, RAM and ROM, the blue signal is the clock for the I/O and the purple is the clock signal for the timer.

The clock stops when the clock signal is low. The clock is first enable when the internal clock goes high afters an interrupt. Every component then functions normally again. The clock starts and stops in this way to avoid that the clock period is too small.

The way the sleep model is implemented in the Nimbus microprocessor is more like the standby sleep mode in the ATmega128. The power down and power save sleep modes for the ATmega128 turns off the oscillator and when it is turned on, it takes a long time for the clock to be stable again. The standby mode is not available in the ATmega103.

3.2.4 Identified Bugs

Only a few bugs were found, but these were very essential. Some of the bugs were described in [29] and they were wiped out. The bugs caused an incorrect handling of the following instructions:

- `push` and `pop` instruction: The data write was done and address were set too early. (push)
- `ld rD, Y` and `ld rD, Z` instruction: The instructions were not detected.
- `st rD, Y` and `st rD, Z` instruction: The instruction were not detected.
- ld rD, X+ and ld rD, -X instruction: The post increment (`X+`) and pre-decrement (`-X`) were not detected.
- `bst Rd,b` instruction: There was an internal error where the instruction was mistaken for the `bset` instruction.

In addition to these bug there were found the following bugs:

- `reti` instruction: Too early calculation of return address.

- `pop` instruction: The load address was wrong.

The interrupts were also handle wrong. The current program counter was calculated too early.

All these errors mean that the AVR_CORE could only have been tested using logical instructions. The problems with these bugs are, that there has never been tested complected programs on the AVR_CORE. This would have lead to detection of the bugs.

3.3 Programming Tool Flow

This section covers the tools that have been used for implementation and realisation of the Nimbus microprocessor. First there is a introduction to the program flow and how the AVR_CORE was explored in the first place.

In the end there is a description at how the synthesis tool from Xilinx ISE was used for programming the Spartan3 on the Hogthrob board and how Synopsys was used for synthesising the design for an ASIC.

3.3.1 Getting Started

It was difficult to get started using the AVR_CORE and get it to execute a program. The source code did not include a test bench. The project was very poorly described. Only the things which were not implemented was described.

The project also included Windows assembler which could assemble an AVR assembly program into a text file with the binary program code. This was not very helpful.

The problem was how to get a C program running on the VHDL Nimbus microprocessor model. To overcome this a flow was developed.

3.3.2 From Program to Chip

This section addresses how to write a program and get the program to execute on the Nimbus. Figure 3.3 shows the programming flow.

To begin with a program is compiled and linked using the GNU AVR compiler in a program file. The binary code from the program file is then dumped into a binary file which only included the raw program, instead of dumping the binary code into the srec-format as described in section 2.2.1.

A program was developed which converted the binary file into a VHDL file. The VHDL file includes a ROM description for the AVR microprocessor. The VHDL ROM description has the same size as the binary file. The program is called *vhdl2init-ext2* and the source code can be found in appendix H.7.12. Finally the VHDL file was moved into the other Nimbus VHDL files. These five steps were included in one script and the red box on figure 3.3 shows four out of five steps. If a new program was developed, it was very easy to get the VHDL model of the program.

The program could together with the rest of the description of the Nimbus microprocessor be simulated. The model could also be converted into a FPGA or ASIC netlist

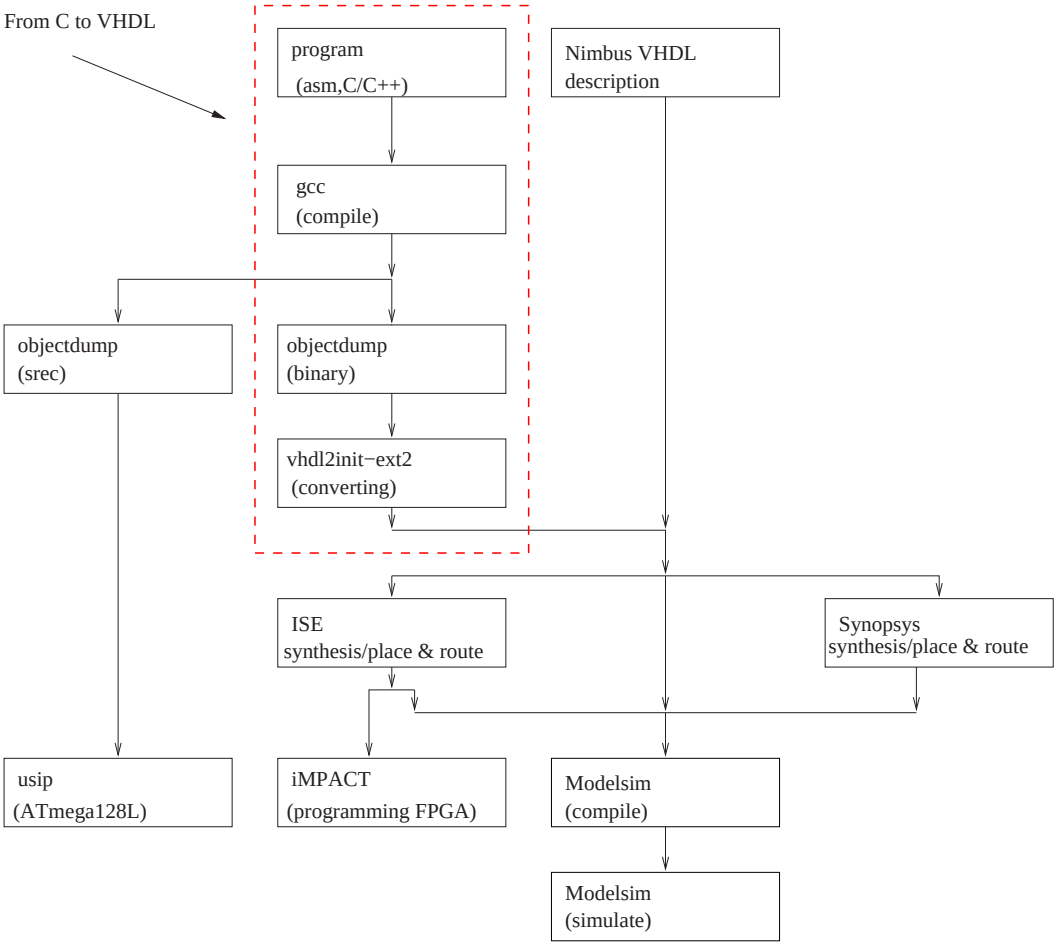


Figure 3.3 *The programming flow.*

and then make a back-annotated simulation of the model. The model could also be placed into a FPGA.

Boot Loader

The GNU AVR compiler includes a linker which intrun a boot strip that matches the interrupt vector as described the ATmega103 architecture description. The first instruction in a program is a jump to function, which loads every program variable into the data memory and then jumps to main.

Instead of using the GNU boot strip a specially developed one was used which runs a pair of `nop` instructions and via the `icall` instruction jumps to the main function. The advantage is that it is easier to monitor what is going on to start with inside the microprocessor and that it is started correctly. The GNU tool does many things to a program which is difficult to foresee if it is the first time working with it.

When the Nimbus was starting as expected, the GNU boot strip was used.

3.4 Hogthrob FPGA

On the Hogthrob board, a Xilinx Spartan3-400 is placed. The idea with the FPGA on the Hogthrob board is to be able to test different microprocessors and components, which could be interesting in a sensor network perspective. Spartan3 is from Xilinx [57] and the FPGA data-sheet can be found here [59]

For example it could be imagined that instead of using the microprocessor for encoding and decoding radio package, there could be developed a special hardware component to do the job. The FPGA could then be used to see if it is works in reality.

The idea with FPGA on the Hogthrob is to se that the Nimbus design works as expected. In order to "synthesise" and "place and route" the design into the Spartan3 Xilinx ISE 6.1a was used. The FPGA top level for the Nimbus design and the pin connection is specified in appendix H.2 in the files `top_spartan3.vhd` and `top_spartan3.ucf`. To download the design into the Spartan3 board Xilinx iMPACT was used. iMPACT is using the JTAG port on the Hogthrob board.

3.4.1 XST ROM and RAM

Xilinx Synthesis Technology (XST) [25] is a description of supported HDL languages Xilinx devices, and constraints for the ISE software. It is possible to map the ROM and RAM into memory blocks in the FPGA if the VHDL description of the RAM and ROM follows the XST specification. ISE will otherwise map the ROM and RAM to flip-flops and it is then not possible to have the microprocessor in the FPGA.

3.4.2 Visual Verification

One of the features in the Xilinx design tools is a program which is able to look inside the FPGA and see what is going on. The program is called ChipScope during execution.

ChipScope uses the JTAG like iMPACT. ChipScope is used to verify that Nimbus is running as expected in section 3.7.1

3.5 ASIC Synthesis

An important part of analysing the Nimbus microprocessor is to compare it with a real microprocessor. This can be done by using the Synopsys synthesis tools.

3.5.1 Synopsys Synthesis

Synopsys release 06-2004 was used to synthesise the Nimbus microprocessor. Synopsys is then able to report the timing, area and power consumption after the synthesis of the microprocessor. Figure 3.4 shows the flow.

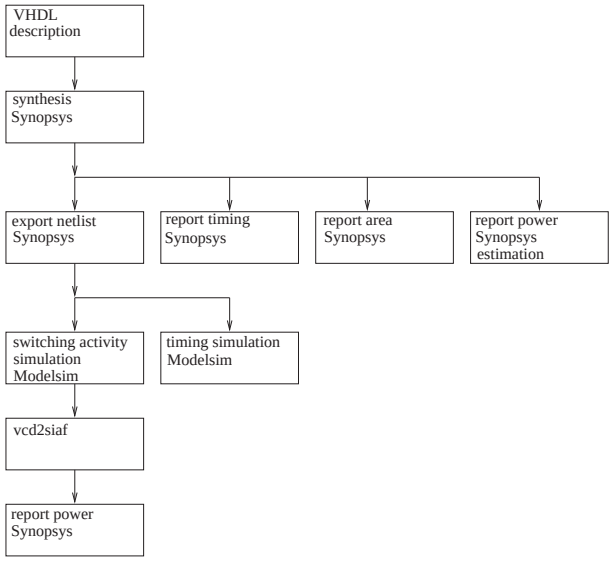


Figure 3.4 *Synopsys tool flow.*

Some scripts were developed which do all the work. To begin, with every VHDL file is synthesised separately and in the end they are put together. The files are analysed, elaborated and optimised. The optimisation is done with the highest effort and target minimisation of area. Each part of the design is synthesised separately because this makes it possible to keep the design hierarchy and reduces the optimisation timer by several minutes, because Synopsys does not try to optimise the design through the components.

The whole design is then synthesised with Synopsys Power Compiler to report an estimation of area without the wiring.

To get a more accurate model of the power consumption a switching activity model of a Nimbus microprocessor executing a specific program can be created. To do this a netlist and a switching delay of the netlist for the design have to be exported. The netlist was created in a Verilog file, because the VHDL part of Synopsys did not work correctly. The switching delay was saved in a file called `sdf` (Switching Delay File).

The netlist, sdf and test-bench are loaded into the ModelSim simulator. It is not possible to see if the circuit timing is maintained. This is called back-annotated simulation.

It is possible to enable the switching activity in ModelSim and activity is written into a VCD (Value Changed Dump) file. The VCD file is converted into a Synopsys switching activity file (SAIF). The netlist of the microprocessor and the SAIF are loaded into Synopsys and then Synopsys is able to estimate the power consumption based on switching activity. This gives a more realistic estimation.

3.5.2 Cell Library

Two different cell libraries were used for synthesising the Nimbus microprocessor for an ASIC. The libraries are from [58]. The first cell library is called CORELIB7 from November 1998 and is based on a 0.25μm cell technology and the other cell library is called CORE9GPLL from October 2001 and is based on a 0.12μm cell library. CORE9GPLL is a special low leakage current cell library. There is also a high speed cell library for the 0.12μm cell library, but this is not considered because this project only concerns the low-power microprocessors. There does not exist a low leakage version of the 0.12μm cell library.

Leakage current is becoming an increasing problem for the new cell library. Examination has been done using the cell libraries to find out how dominating the leakage current is in the cells. A full-adder have been used for the experiments. Figure 3.5 shows the full-adder run at different frequencies.

It can be seen that the leakage current has as influence. The full-adder has run at 70 MHz before the full-adder based on the two different technologies used the same amount of power. This may prove that the 0.12μm cell library is not the best cell library for the sensor network because the motes spend much more time asleep.

The Nimbus microprocessor which is synthesised using the 0.25μm cell library is called Nimbus 0.25 and if it is synthesised using the 0.12μm cell library the microprocessor is called Nimbus 0.12.

3.5.3 Memory

The cell libraries presented in the previous section do not have any memory support included. It has been possible to get information of some memories from STMicroelectronics. The memories are based on the 0.12μm cell library and [43, 44, 45] contain information on the different memory modules. The memories are from memory generator program from STMicroelectronics.

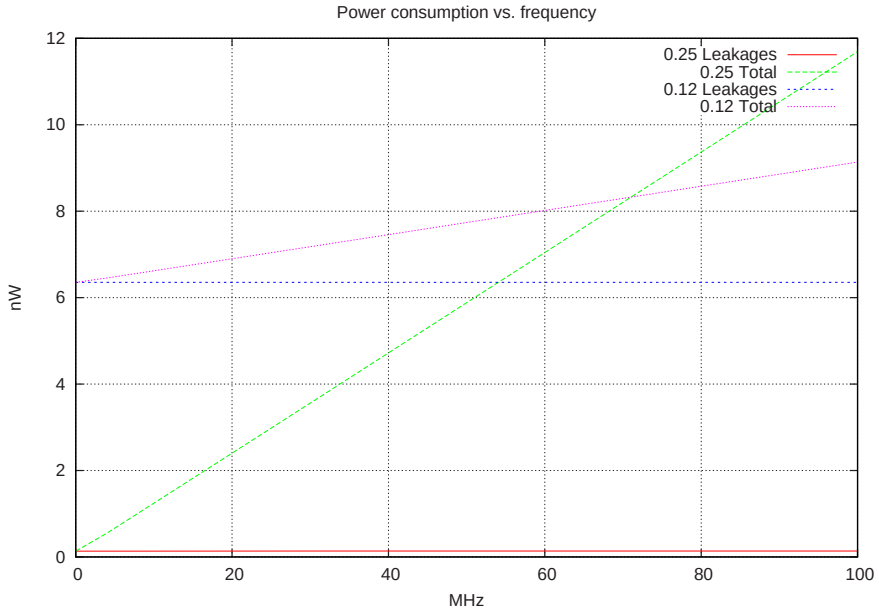


Figure 3.5 *Power consumption for a full-adder using a* 0.12μm *and a* 0.25 *cell library.*

Name	Unit	SPHS9gp	SPSMALL9gp
Technology		LL+ULL	LL
Dimension		8192x16	128x64
Cell lib.	μm	0.12	0.12
Cell area	μm^2	2.5064	3.584
Voltage	V	0.9	0.8
Size	KB	16	1
Dynamic consumption (read)	$\mu A/MHz$	85	36
Dynamic consumption (write)	$\mu A/MHz$	96	36

Table 3.1 *Dynamic power consumption for the memories.*

3.6 Test

The article [30] discusses the issue of the design CMOS VLSI using the variable design tools and the issue of testing and evaluating a digital circuit. They argue that the most dominant way to test a chip is random pattern generation. The technique is used to detect data hazards in different parts of the chip.

To generate random test patterns for this project is a lot of work, so instead some test programs have been developed. The programs have been verified first in behavioural simulation. Then everything was working perfectly and the design was converted in a netlist using the synthesis tool. The nestlist has been simulated with the same test program. The netlist includes timing information for the design library.

First in this section there is a description of the test programs, which were used in the beginning to check if the microprocessor worked. Then there is back-annotated simulation illustrating that the sleep modes and interrupts are working as expected and finally ChipScope is used to illustrate that the Nimbus microprocessor runs perfectly on the Hogthrob FPGA.

3.6.1 Nimbus Test Programs

Three kinds of test programs were developed. The test programs were used to see if the Nimbus microprocessor was working in the first place. Test programs for verification of the timing made it possible to find and correct the bugs described in section 3.2.4. Finally the ATmega128L test programs were run on the microprocessor as described in section 2.2. All the test program can be found in appendix H.7.

The First Programs

In the list below is a description of the first program, which was used to test if the Nimbus microprocessor was running. The test program used the boot loader, that was described in section 3.3.2. The timing test programs is also a boot loader.

- **simple_test1_s.s** loads some data and adds it together.
- **test1.c** tests if the boot-loader jumps to the main function. Between the main function and `icall` is an unlimited while-loop. So if the `icall` function does not succeed jumping to the `main` function, the program will be stuck at the while-loop.

Timing Test Programs

Some bugs were found and in order to test it the system was working perfectly these timing test programs were developed.

- **ret_test.c** tests that the function jumps correctly to a new function and returns again.

- **pin_test.c** three bit counts that blink the leds.
- **push_pop_test.c** push and pop and data to the data memory.
- **sub_test.c**

3.7 Visual Verification

This section shows confirmatory evidence that the Nimbus microprocessor runs as expected. All the test programs were visually verified. This includes all the test programs that were described in section 2.2 and 3.6. It was also verified that the TinyOS blink example was running as expected.

It is very important that the TinyOS program was running without errors because TinyOS is a more complex program in contrast to all the other test programs which are very simple and whose goals are to test a specific thing, where TinyOS includes parts which are unknown and therefor if TinyOS was performing wrong, it would be really complicated to find the error.

The verification of the Nimbus microprocessor was done using the FPGA and the back-annotated simulation based on the ASIC cell libraries.

3.7.1 FPGA

Visual verification using the FPGA was done in three ways.

Back-annotated Simulation

ModelSim was used for back-annotated simulation of the Nimbus microprocessor. This required that the FPGA cell library was compiled to ModelSim as explained in appendix E.

Visual

Furthermore the timer blink and TinyOS blink example were downloaded to the FPGA for visual verification. The examples were blinking with about the same interval as for the ATmega128L. It was not possible to have the same clock frequency as the ATmega128L. The FPGA has only a $40Mhz$ clock and therefor a clock divider was used and the internal clock was set to $5Mhz$ and the external clock was set to $39,0625Khz$.

ChipScopre

Finally ChipScope was used to monitor the workings inside the FPGA for the timer blink and TinyOS blink example. Figure 3.6 shows how ChipScope have been used to verify the timer blink example and it illustrates when the led is turned on. Figure 3.6(a) shows a screen dump of ChipScope which has collected data from the FPGA. ChipScope is setup to sample the program counter and the loaded instructions from

Clock ticks	PC	Description
1	*16E	The sts instruction is loaded from the instruction memory
2	*16E	The previous instruction is a multi cycle instruction and therefore the same instruction is loaded again.
3	*170	The sts instruction is decoded and it is determined that the sts is a 32 bit instruction multi cycle instruction. The second part of the sts instruction is loaded from the instruction memory.
4	172	The sts instruction is executing.
5	172	The sts instruction is continue executing the instruction
6	174	It is possible to see that the led is turned on and a new instruction is decoded.

Table 3.2 *Description of STS instruction*

the FPGA which is listed in figure 3.7.1. Figure 3.7.1 shows the matched disassembled code with c-source intermixed of the data.

In figure 3.7.1 the instruction with the "*" is a store direct to data space (sts) instruction, which turns the led on and off. sts is a 2 times 16 bit and is executed in 2 cycle (multi cycle instruction). Table 3.2 explains what is going on in the microprocessor.

In appendix F.2 more waveforms made by Chipscope can be found. The waveforms show the running of the timer blink example and TinyOS blink example.

3.7.2 ASIC

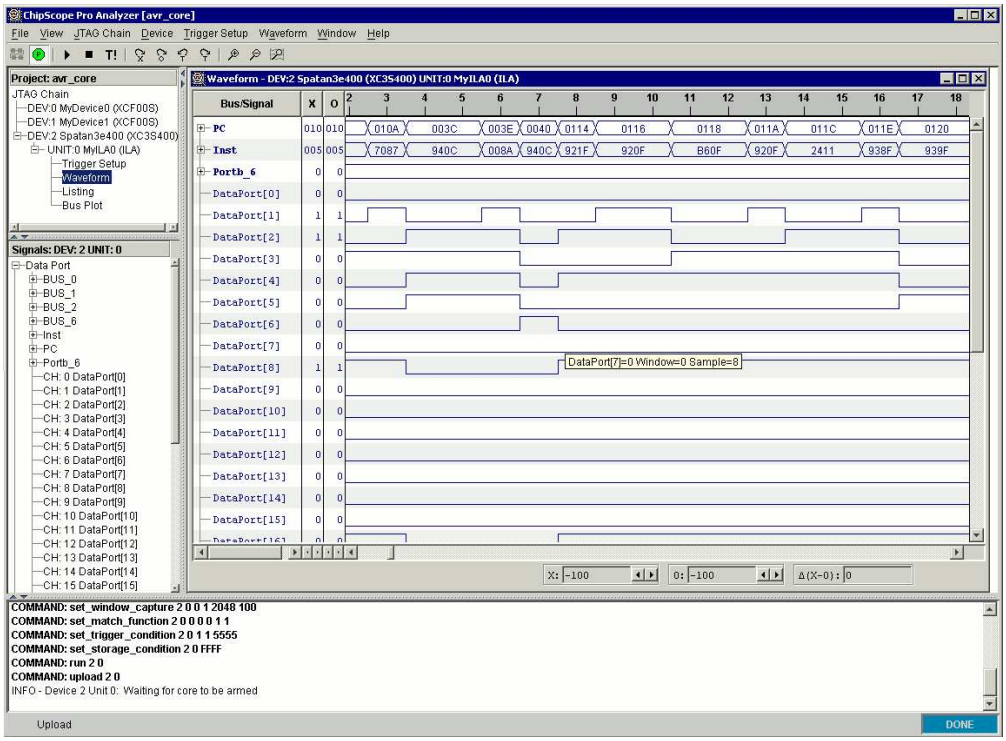
In order to show that the Nimbus microprocessor is working correctly the two next examples are included. The two examples are looking at the Nimbus microprocessor going to sleep and when it wakes up again. The examples are presented using wave forms, which are created by ModelSim running a back-annotated timing simulation. The simulation tool use the ASIC netlist from Synopsys based on the $0.12\mu m$ cell library.

In appendix F.1 there are more waveforms made by ModelSim from back-annotated simulations of Nimbus 0.12. There are waveforms of the Nimbus 0.12 running the timer blink and the TinyOS blink examples.

Sleep Mode

Figure 3.7(a) illustrates when the Nimbus 0.12 microprocessor is going to sleep. The internal clock, `clk_int` is running at 4*Mhz* and the external clock, `clk_ext` is running at 8*Mhz*. The external clock is set to this high frequency in order to reduce the simulation time to less than a minute.

When Nimbus 0.12 is put into sleep the signal `mode_idle`, `mode_power_save` and `mode_power_down` defines the sleep mode. It can be seen that the clock of the core and all the I/O components are stopped. The `clock_core_enable` and `clock_dev_enable`



(a) ChipScope

PC(14-0)	Inst	Portb_6
10a	7087	0
3c	940c	0
3c	940c	0
3e	008a	0
40	940c	0
114	921d	0
116	920f	0
118	b60f	0
11a	920f	0
11c	2411	0
11e	938f	0
120	939f	0
...		
16A	8389	0
16C	8189	0
16C	8189	0
*16E	9380	0
*16E	9380	0
*170	38	0
172	E08E	0
172	E08E	0
174	9380	1
176	52	1
178	9180	1
178	9180	1
17A	57	1
...		

(b) Data from Chipscope

```
00000000 <__vectors>:
...
  38:   0c 94 50 00     jmp     0xa0
  3c:   0c 94 8a 00     jmp     0x114
  40:   0c 94 50 00     jmp     0xa0
...
void __attribute((signal))
SIG_OUTPUT_COMPARE0(){
 114:   1f 92           push    r1
 116:   0f 92           push    r0
 118:   0f b6           in      r0, 0x3f
 11a:   0f 92           push    r0
 11c:   11 24           eor     r1, r1
 11e:   8f 93           push    r24
 120:   9f 93           push    r25
...
PORTB = PORTB & _BV(6) ?  PORTB & ~_BV(6) :
            PORTB | _BV(6);
...
 16a:   89 83           std     Y+1, r24
 16c:   89 81           ldd     r24, Y+1
*16e:   80 93 38 00     sts     0x0038, r24
outp(14, TCNT0);    // Reset timer counter
 172:   8e e0           ldi     r24, 0x0E
 174:   80 93 52 00     sts     0x0052, r24
sbi(TIMSK, OCIE0); // OutputCompareInterrupt
 178:   80 91 57 00     lds     r24, 0x0057
...
```

(c) Disassembled code with c-source intermixed

Figure 3.6 *Gathering instruction trace from AVR-Core*

signals indicate whether the clock of the core and I/O components are enabled or not. It can then be seen that the clock signal of core `clk_core` and to the I/O components `clk_dev` are stopped.

Wakeup

The wakeup of the Nimbus microprocessor is illustrated on figure 3.7. It can be seen that there are no activity signals except the external clock and the internal clock. When there is an interrupt from the timer the Nimbus microprocessor wakes up. The program counter is then set to the address 003*C*, this is the position in the instruction memory, which handles the timer interrupt. The instruction is a jump instruction which jump to the program segment that takes care of the interrupt.

3.8 Measurements - FPGA

There were only done measurements of the used slices in the FPGA. Table 3.3 summarises the used slices, RAM block and the other FPGA logic blocks, which are available in the FPGA. The summary is from after the place and route of the Nimbus microprocessor in the FPGA based on the TinyOS blink example.

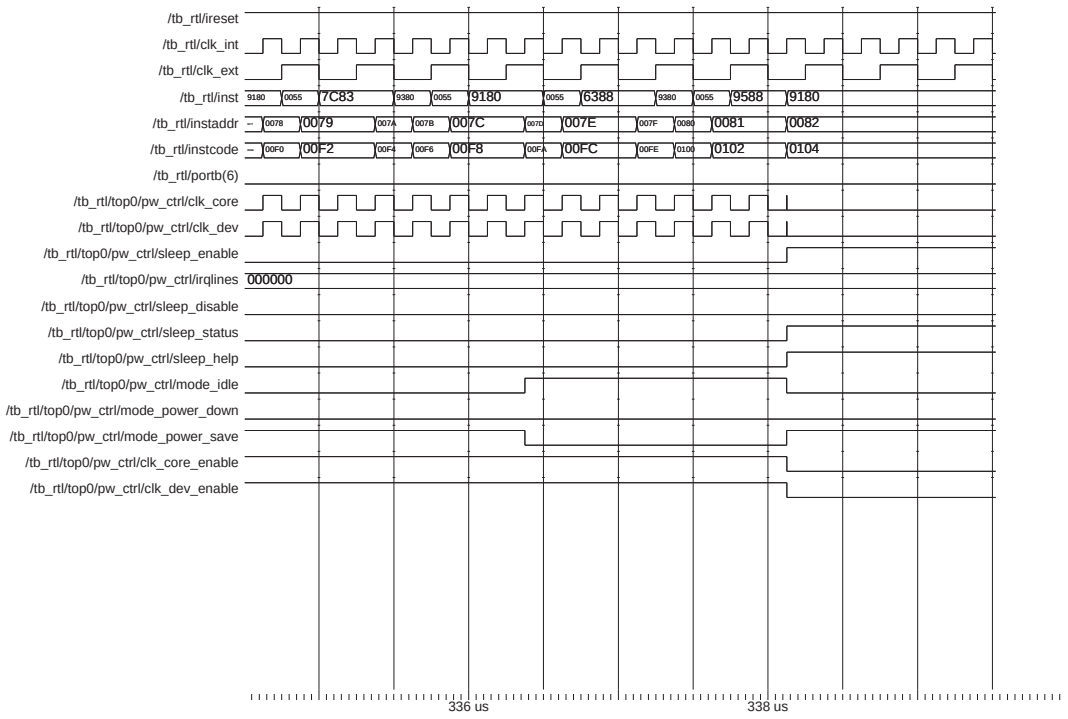
The synthesis of the microprocessor informs that the microprocessor is able to run at a maximum frequency of 26*MHz*.

Number of External IOBs	5	out of	173	2%
Number of LOCed External IOBs	5	out of	5	100%
Number of RAMB16s	1	out of	16	6%
Number of Slices	2916	out of	3584	81%
Number of SLICEMs	1152	out of	1792	64%
Number of BUFGMUXs	3	out of	8	37%

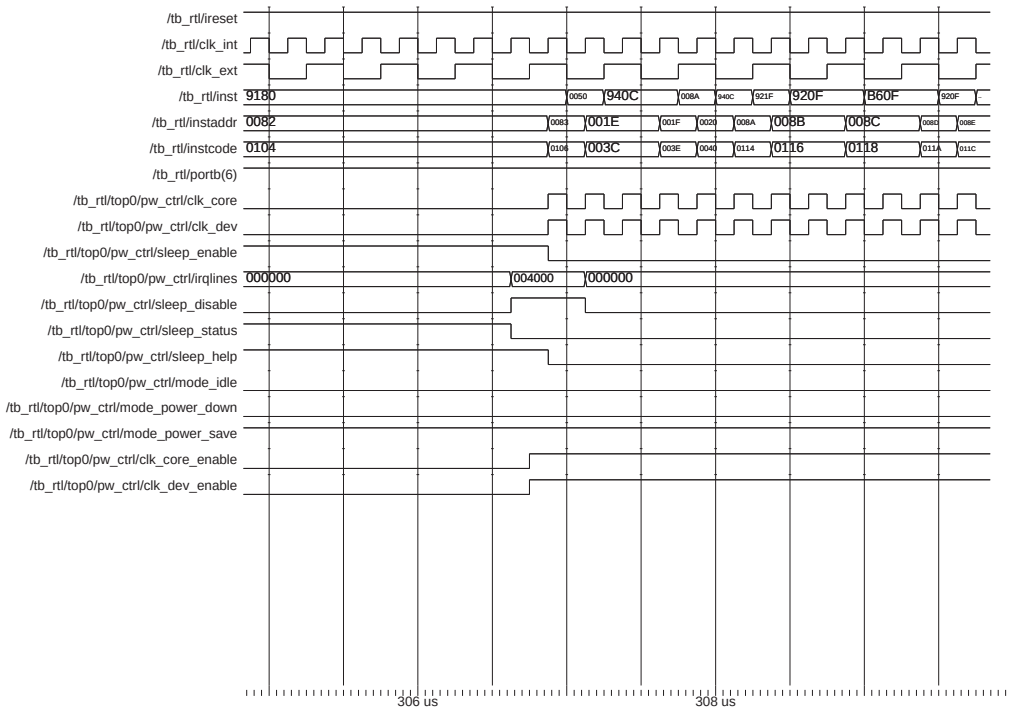
Table 3.3 *Spartan3 device utilisation summary*

3.9 Measurements - ASIC

This section is about the measurements of the Nimbus microprocessor synthesised with an ASIC cell library. The section is split up in two parts where first the area measurements will be evaluated and secondly the current and power consumption will be discussed. The measurements are done for both cell libraries i.e. Nimbus 0.12 and Nimbus 0.25.



(a) Nimbus microprocessor goes to sleep



(b) Nimbus microprocessor wakes up from sleep

Figure 3.7 *The figure is from a back-annotated simulation for the timer blink example.*

3.9.1 Area

The area calculations of the Nimbus microprocessor are based on synopsys and the memory data sheet.

Nimbus Microprocessor Without Memory

Table 3.6 shows the estimated area of the Nimbus microprocessor from Synopsys. The area of the microprocessor does not include memory, because the memory was not able to synthesised. It can be seen that the area of the Nimbus 0.12 is 4.5 times smaller than the Nimbus 0.25.

Microprocessor	Nimbus 0.12	Nimbus 0.25
Total	69008.41μm^2	307143.00μm^2

Table 3.4 *Area of the Nimbus microprocessor without the memory based on the ASIC cell library.*

Memory

Table 3.5 shows the calculations of the area for ROM and RAM. The calculation is based on the memory from section 3.1. The size of a cell area was only available for 1 bit RAM based on the 0.12 cell library. The area of the Nimbus 0.25 is 4.5 times larger than the Nimbus 0.12 and this constant was used to calculate the cell area of an 1 bit memory cell based on the 0.25 cell library. This gives the size on an 1 bit memory cell 11.16μm^2.

The size of a ROM is normally smaller than a corresponding size of a RAM. This is because it is only possible to read from the ROM and it is therefore estimated that the ROM is only 70 % of the size of the RAM. In the bottom row of the table it is possible to see the size of the RAM and ROM based on the two cell library.

	ROM	RAM	ROM	RAM
Technology	0.12μm	0.12μm	0.25μm	0.25μm
Cell area	2.51μm^2	2.51μm^2	11.16μm^2	11.16μm^2
Size	8 kB	8 kB	8 kB	8 kB
Scaled	0.7	1.0	0.7	1.0
Total area	57490μm^2	82129μm^2	255880μm^2	365543μm^2

Table 3.5 *Area of memory for the Nimbus microprocessor based on the ASIC cell library.*

Total Area

Table 3.6 shows the total size of the microprocessor when adding the size of the Nimbus microprocessor without the memory ROM and RAM. It can be seen that the size of the

Nimbus 0.25 is about $1mm^2$ whereas the Nimbus 0.12 is only about $0.25mm^2$.

Microprocessor	Nimbus 0.12	Nimbus 0.25
Core	$69008.41\mu m^2$	$307143.00\mu m^2$
ROM	$57490.80\mu m^2$	$255880.35\mu m^2$
RAM	$82129.72\mu m^2$	$365543.35\mu m^2$
Total	$208628.93\mu m^2$	$928566.70\mu m^2$

Table 3.6 *Total area of the Nimbus microprocessor based on the ASIC cell library.*

It should be noted that the calculation of the area of the Nimbus microprocessor does not include any wiring. The wiring would probably increase the size of the ROM and RAM by 1/3 because the memories are arranged into block. The memory blocks need to be connected and the memories have to be connected to the rest of the Nimbus microprocessor. The size of the part of the core is not increased much because it is possible to put many metal layers.

3.9.2 Current and Power Consumption

The evaluation of the current and the power consumption for the Nimbus microprocessor are discussed in three parts. First the measurements of the synthesised microprocessor is presented. This does not include the memory. Secondly it is explained how the power consumption of the ROM and RAM are calculated. Finally the two parts of measurements are added together.

The measurements are based on the test programs for the ATmega128L which are described in section 2.2.

Nimbus microprocessor without memory

Table 3.7 shows the current and the power consumption for the Nimbus 0.12 and Nimbus 0.25 without memory executing at 7 MHz. It is possible to see the leakage current and power and the total current and power consumption.

It can be seen that the leakage current has a dramatic influence on the power consumption for the Nimbus 0.12 microprocessor. The loop test program has a much higher power consumption than the add, add-mem and hamming test programs.

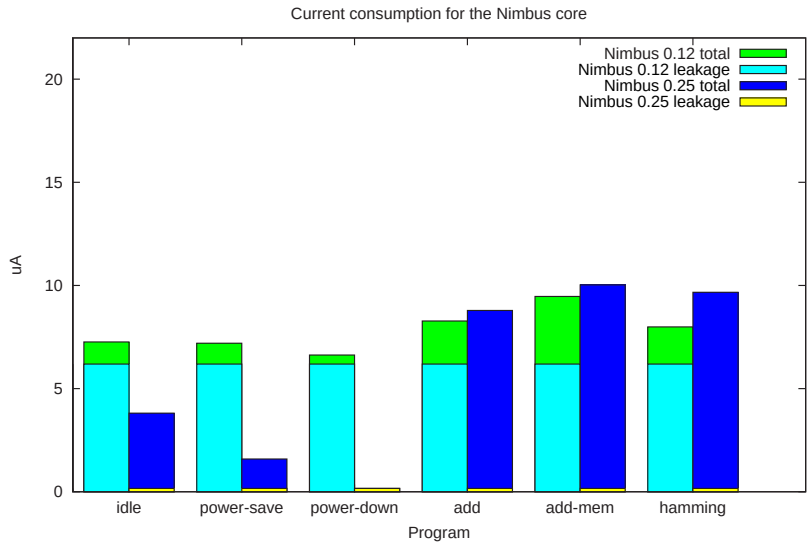
Figure 3.8 is a plot of the current and power consumption of table 3.7 except for the loop test program, which is not plotted because of the large current and power consumption.

The measurements from table 3.7 were originally made with a microprocessor running at $4MHz$, but the measurements of the dynamic power consumption have been linear scaled with the same slope as the one of the full-adder from 3.5.2. The dynamic power consumption for the microprocessor can be assumed to be linear because the microprocessor is running at a very low frequency. If the microprocessor was running much faster, the dynamic power consumption would increase exponentially. This is

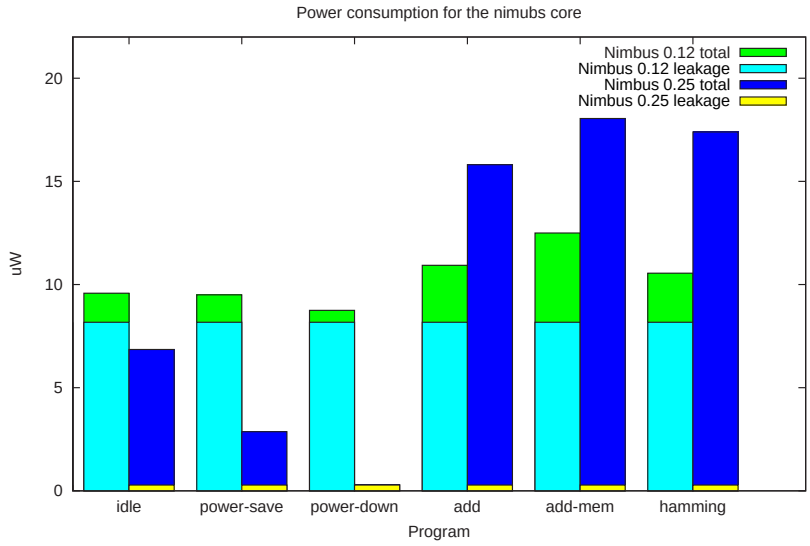
Current				
	Nimbus 0.12		Nimbus 0.25	
	Leakage	Total	Leakage	Total
loop	6.19μA	48.76μA	0.16μA	439.32μA
idle	6.19μA	7.26μA	0.16μA	3.81μA
power-save	6.19μA	7.20μA	0.16μA	1.59.μA
power-down	6.19μA	6.63μA	0.16μA	0.16μA
add	6.19μA	8.28μA	0.16μA	18.79μA
add-mem	6.19μA	9.47μA	0.16μA	10.03μA
hamming	6.19μA	7.99μA	0.16μA	9.67μA

Power				
	Nimbus 0.12		Nimbus 0.25	
	Leakage	Total	Leakage	Total
loop	8.17μW	64.36μW	0.28μW	790.15μW
idle	8.17μW	9.58μW	0.28μW	6.85μW
power-save	8.17μW	9.50μW	0.28μW	2.87μW
power-down	8.17μW	8.75μW	0.28μW	0.29μW
add	8.17μW	10.96μW	0.28μW	15.81μW
add-mem	8.17μW	12.50μW	0.28μW	18.05μW
hamming	8.17μW	10.55μW	0.28μW	17.40μW

Table 3.7 *Current and power consumption of Nimbus 0.12 and Nimbus 0.25 synthesised without memory.*



(a) Current



(b) Power

Figure 3.8 *Measurements of current and power consumption for the Nimbus microprocessor running at 7 MHz.*

also shown in [3] for a much larger microprocessor and it can be seen that the frequency of this microprocessor has to be running at least 75 MHz before the dynamic power consumption can be assumed to increase exponential.

Memory

The calculations of the dynamic current and power consumption and leakage current and power consumption are based on table 3.1 and the results are shown in 3.8.

It was not possible to find a memory of exactly 8KB the dynamic current consumption to read from and to write to the memory. Therefore the dynamic current consumption for the read and write of the 0.12 RAM are calculations based on the slope of the SPH and SPS the memory between the dynamic current consumption per read or write and the size of memory. It is assumed that the dynamic current consumption to read an instruction from the ROM is the same as for the RAM and the structure of the RAM is not taken into consideration.

The dynamic current consumption to read and write for the RAM and ROM based on the 0.25 cell are calculated according to the differences in the slope between full-adder synthesised with the 0.12 and 0.25 cell library.

The leakage current consumptions are calculated by the differences in size of the Nimbus microprocessor and the memory. The difference in the size is then multiplied by the leakage current consumptions of the Nimbus microprocessor to get the leakage for the whole memory.

Finally the power consumption of the different parts is calculated based on the voltage in table 3.8.

Technology	$0.12\mu m$	$0.25\mu m$
Voltage	$0.9V$	$1.8V$
Dynamic consumption (read)	$47.4\mu A/MHz$	$196.5\mu A/MHz$
	$42.7\mu W/MHz$	$353.7\mu W/MHz$
Dynamic consumption (write)	$49.6\mu A/MHz$	$205.6\mu A/MHz$
	$44.6\mu W/MHz$	$370.1\mu W/MHz$
Total leakage	$16.55\mu W$	$0.5651\mu W$
ROM & RAM	$18.39\mu A$	$0.3140\mu A$

Table 3.8 *Calculation of dynamic current/power consumption and leakage current and power consumption for the ASIC library.*

In order to calculate the dynamic power consumption of the Nimbus microprocessor the number of instructions read and data read and write were counted for the same amount of times the test program was run. This was done by making a counter in the ROM and RAM and the results are shown in table 3.9.

The dynamic current and power consumption was then calculated based on measurements showed in table 3.9 and the dynamic current and power consumption to

Program	ROM read	RAM read	RAM write
loop	4000	763	380
idle	0	0	0
power-save	0	0	0
power-down	0	0	0
add	4000	0	0
add-mem	4000	752	753
hamming	4000	985	123

Table 3.9 *Instruction read and data write and read for the test program in the period of 4000 clock ticks.*

read and write to the memory from table 3.8. The results are shown in table 3.10 and 3.11.

Technology	Nimbus 0.12 (μA)			Nimbus 0.25 (μA)		
Program	ROM R	RAM R	RAM W	ROM R	RAM R	RAM W
loop	368.7	70.3	35.0	764.1	145.8	72.6
idle	0.0	0.0	0.0	0.0	0.0	0.0
power-save	0.0	0.0	0.0	0.0	0.0	0.0
power-down	0.0	0.0	0.0	0.0	0.0	0.0
add	368.7	0.0	0.0	764.1	0.0	0.0
add-mem	368.7	69.9	70.0	764.1	144.8	145.0
hamming	368.7	90.8	11.3	764.1	188.2	23.5

Table 3.10 *Dynamic current consumption for instruction read and data write and read of the test program.*

Technology	Nimbus 0.12 (μW)			Nimbus 0.25 (μW)		
Program	ROM R	RAM R	RAM W	ROM R	RAM R	RAM W
loop	331.8	63.29	31.5	1375.3	262.3	130.7
idle	0.0	0.0	0.0	0.0	0.0	0.0
power-save	0.0	0.0	0.0	0.0	0.0	0.0
power-down	0.0	0.0	0.0	0.0	0.0	0.0
add	331.8	0.0	0.0	1375.3	0.0	0.0
add-mem	331.8	62.9	63.0	1375.3	260.6	261.0
hamming	331.8	81.7	10.2	1375.3	338.7	42.3

Table 3.11 *Dynamic power consumption for instruction read and data write and read of the test program.*

3.9.3 Nimbus with Memory

The total current and power consumption for the test programs are showed in table 3.12. The current and power consumptions are calculated by adding the results from the Nimbus microprocessor without memory (table 3.7) the dynamic and leakage current and power consumption for the ROM and RAM from table 3.8, 3.10 and 3.11.

Program	Nimbus 0.12	Nimbus 0.25	Nimbus 0.12	Nimbus 0.25
loop	543.20μA	1421.90μA	509.36μW	2792.33μW
idle	27.69μA	3.98μA	27.98μW	13.00μW
power-save	27.63μA	1.77μA	27.89μW	8.69μW
power-down	26.08μA	0.34μA	27.16μW	3.02μW
add	397.39μA	773.03μA	361.14μW	1402.88μW
add-mem	538.39μA	1064.06μA	488.53μW	1933.22μW
hamming	499.23μA	985.56μA	452.67μW	1783.88μW

Table 3.12 *Total current and power consumption for the Nimbus microprocessor*

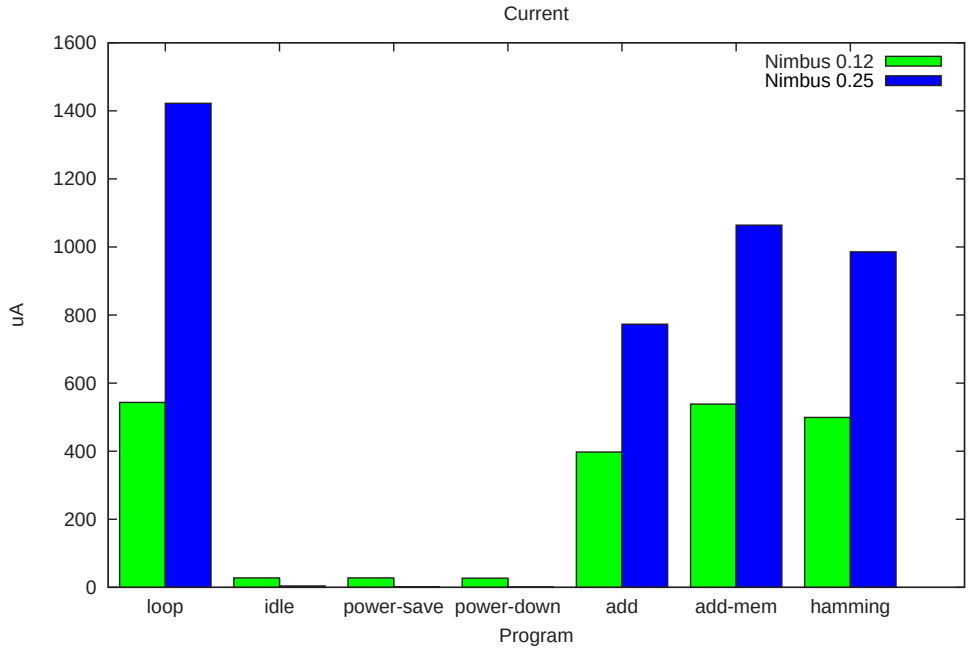
The results from table 3.12 are plotted in figure 3.9 and 3.10. Figure 3.9 shows the graph of the current consumption for the Nimbus microprocessor and figure 3.10 shows the power consumption. Figure 3.9(a) and 3.10(a) shows the consumption for all the test programs where figure 3.9(b) and 3.10(b) shows the consumption of the microprocessor in the sleep modes.

It can be seen on figure 3.10(a) that there is a big difference in of the power consumption for Nimbus 0.12 and Nimbus 0.25. The power consumption for the Nimbus 0.25 is much higher than the Nimbus 0.12 when executing the `add`, `add-mem` or `hamming` test program. The solution for this is as expected that the 0.12 cell library has a lower dynamic power consumption than the Nimbus 0.12.

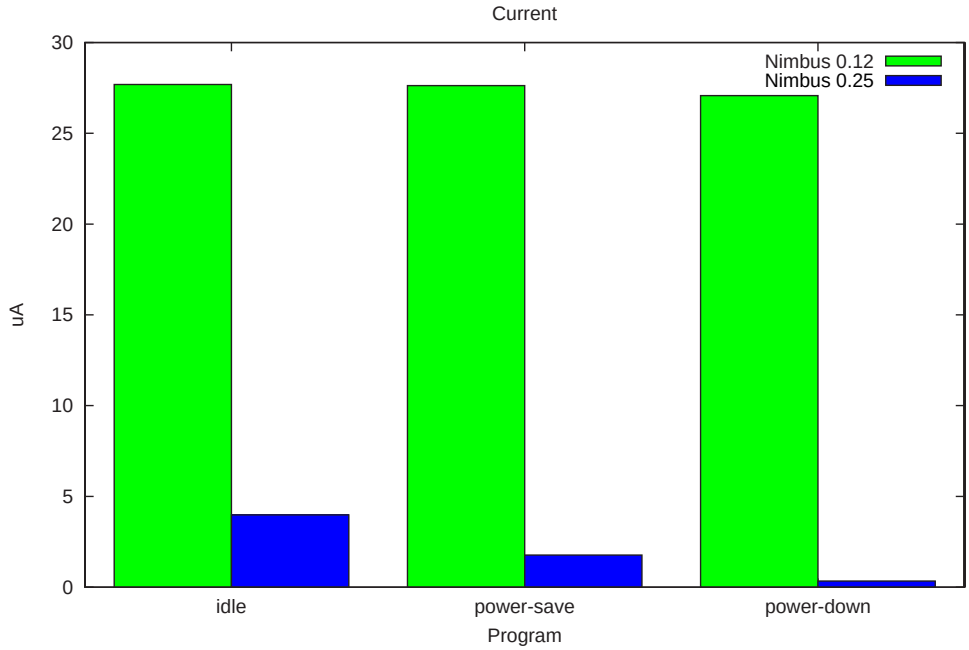
When the Nimbus microprocessor goes to sleep the picture chances. Figure 3.10(b) shows a zoom on the sleep mode test programs and it can be seen that the power consumption for the Nimbus 0.25 is much less than the Nimbus 0.12. This is because of the high leakage for the Nimbus 0.12. The same characteristics can be seen figure 3.9(a) and 3.9(b) where the current consumption of the Nimbus microprocessors is showed.

3.10 Discussion

The measurements had some reservation because not all the desired components and functionalities were available in the cell libraries.

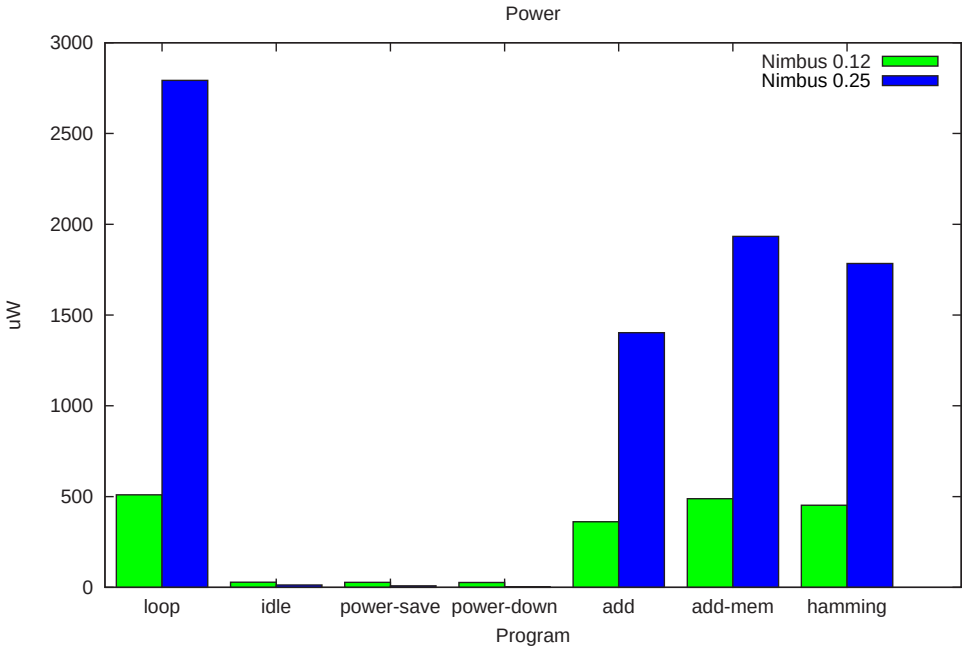


(a)

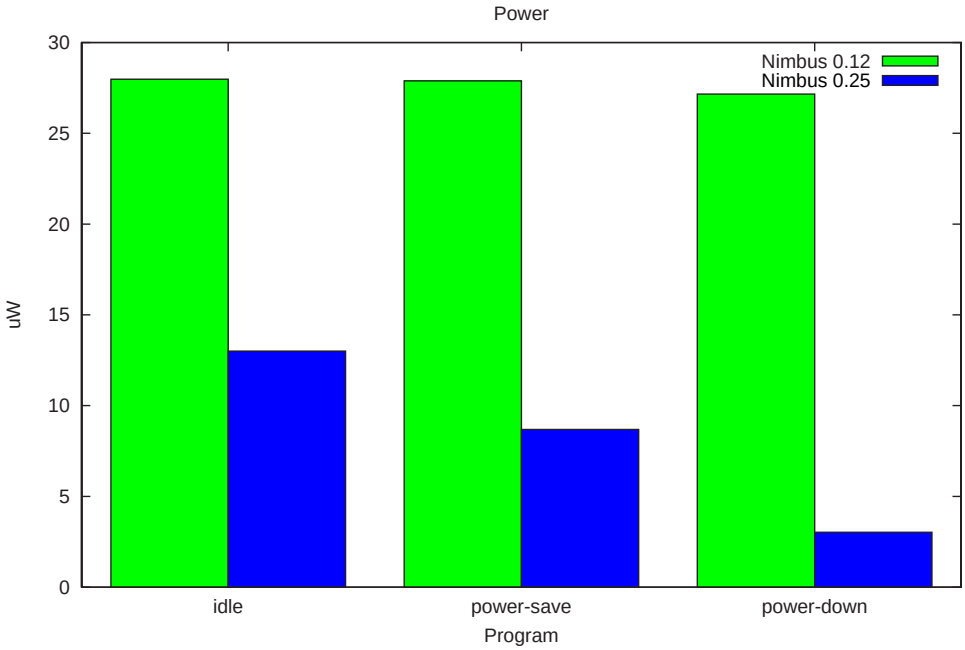


(b)

Figure 3.9 *Current consumption for the Nimbus microprocessor*



(a)



(b)

Figure 3.10 *Power consumption for the Nimbus microprocessor*

Power and Current Consumption

The power measurements give a good indication of the power consumption for the Nimbus microprocessor but the power estimations are not satisfactory. If the power estimations should be more precise it requires that memory can be synthesised.

The power estimation nor includes the power consumption for the external and internal clock. A clock generator can be implemented using and-gates and one inverter. But there was not enough time to implement a clock.

If it was possible to includes these things in the design the power estimation would have been more realistic.

Area

The above-mentioned problems may not change the size of the microprocessor much because the calculation of the area included the size of the memory.

3.11 Summary

The chapter is about the Nimbus microprocessor. It has been explained how it is possible to write a C program and convert the program into a hardware description.

It has been illustrated that the Nimbus microprocessor can be synthesised and placed & routed for a Spartan3 FGPA. The microprocessor in the FPGA was successfully able to execute a TinyOS program and other applications.

The Nimbus microprocessor was also synthesised with two different cell libraries. The power consumption for the microprocessor was measured based on the two cell libraries. The measurements showed that the Nimbus microprocessor performed better than the ATmega128L. The measurements also showed that the Nimbus microprocessor, which was synthesised with the newest cell library, has the lowest dynamic power consumption but it has a higher leakage current than the one synthesised with the other cell library.

CHAPTER 4

Asynchronous AVR Microprocessor

This chapter concerns the implementation of an asynchronous microprocessor called Disa. Disa takes starting point in the customised Nimbus microprocessor. The idea is to de-synchronise the Nimbus microprocessor based on common asynchronous techniques.

The chapter contains a basic introduction to asynchronous circuit design and then the de-synchronisation technique is presented. Some research has been done in order to find out how other asynchronous microprocessors have been implemented. Based on this research the asynchronous protocol is chosen.

Only limited work has been published about de-synchronised. The de-synchronisation technique is trailed in a design study and the Nimbus microprocessors is de-synchronisation based on these experiences. It is explained how Disa is structured and there is a description of the different components in the Disa microprocessor.

Finally the de-synchronisation design technique is discussed and evaluated. It is explained why the implementation did not prove successful.

4.1 Approach

Section 1.4.4 describes why asynchronous design techniques are well suited for low-power microprocessors. The idea is to implement an asynchronous AVR microprocessor for a sensor network and compare it with the microprocessors introduced in the previous two chapters.

There are many techniques to design asynchronous circuits but which one should be used. Implementation of an asynchronous microprocessor is often done from scratch and this is time consuming. Using the de-synchronisation technique all flip-flops in a microprocessors are replaced by latches and asynchronous latch controllers. In this way the structure on the synchronous microprocessor can be kept and this would hopefully lead to a faster implementation of the asynchronous microprocessor.

To understand exactly how this is done the next chapter will introduce the basic asynchronous technique and then the de-synchronisation design technique will be explained in detail.

4.1.1 General Theory

A synchronous system is characterised in that there is a central clock which controls data transfer as illustrated on figure 4.1(a).

An asynchronous system is characterised by the absence of a central clock. The data transfer is now controlled by two neighbouring circuits. The neighbouring circuits communicate using handshake signals as shown in figure 4.1(b). This means when a circuit wants to transfer data to an other circuit, it sends a request signal and when the other circuit is ready and has read the data, it sends an acknowledge signals. Since there is no global clock, all circuits have to have a built-in control, that can handle the transfer of data.

Both figures 4.1 are from [32]. This book explains everything necessary for asynchronous circuits development.

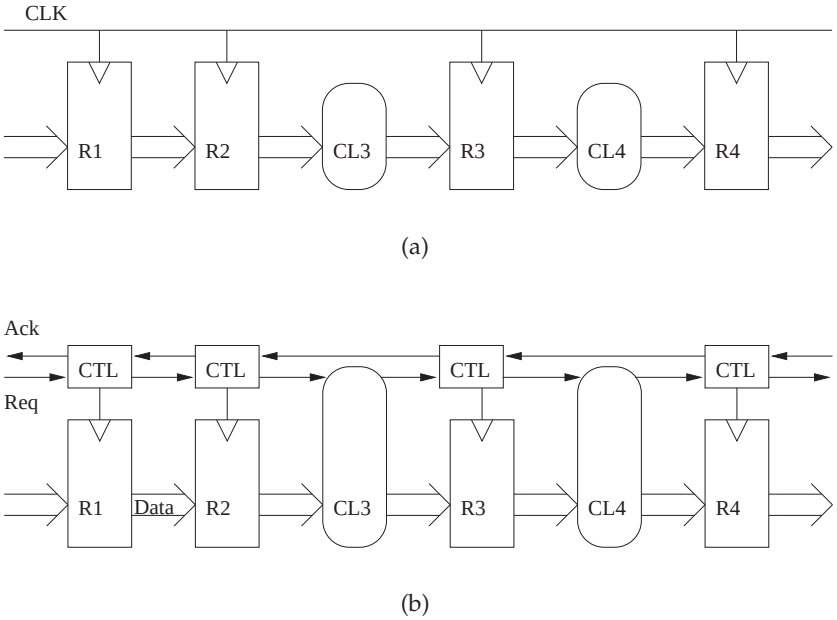


Figure 4.1 *(a) Synchronous circuit and (b) Asynchronous circuit*

Handshake Protocols

The four most know handshake protocols are 2-phase bundled-data protocol, 4-phase bundled-data protocol, 2-phase dual-rail protocol and 4-phase dual-rail protocol.

Bundled-Data Protocols

The bundled-data protocols utilise normal boolean encoding for the data signal. The request and acknowledge signals are bundled to the data and hereby is the name of the

protocol: bundled-data protocol. This is shown on figure 4.2(a).

Figure 4.2(c) shows the 4-phase bundled-data protocol. This protocol is a little more complicated than 2-phase bundled-data protocol. Then the sender issues data and sets request high. When the receiver has absorbed the data, it set the acknowledge signal high. Seeing this, the sender set the request signal low to indicate that the data is no longer valid. The receiver acknowledges this by setting the acknowledge signal low.

Figure 4.2(b) illustrates the 2-phase bundled-data protocol and this is the most simple protocol. However research has shown that the protocol uses more space than the 4-phase bundled-data protocol. The 2-phase bundled-data protocol requires 2 registers for every one register because the register has to store data when the request signal goes high and low. This is described in detail on page 177 in [34] from [33]. Since the protocol is going to be used for a low-power microprocessor where size of the microprocessor has a big influence on leakage current, the 2-phase bundled-data protocol is not going to be used.

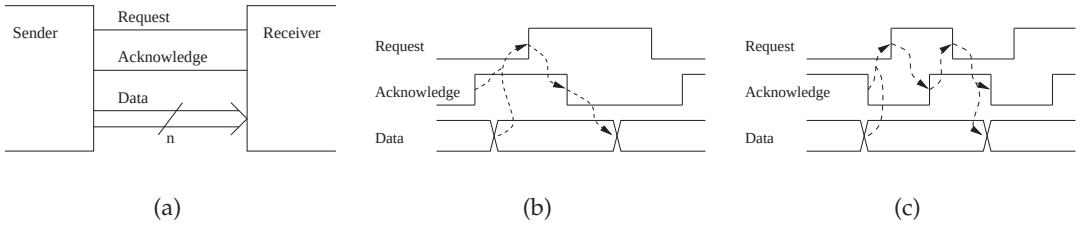


Figure 4.2 *(a) A bundled-data channel, (b) 4-phase bundled-data protocol and (c) 2-phase bundled-data protocol*

Dual-Rail Protocols

Dual-rail protocols encode the data in that it uses two wires per bit. The request and acknowledge work in nearly the same way as the other protocol.

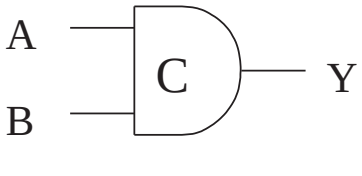
Since the dual-rail protocols use two wires to encode one bit, it means that it uses about twice the mount of space to implement a circuit than the bundled-data protocols which is explored on page 178 in [34]. Therefore the dual-rail protocol is not going to be used.

This means that the 4-phase bundled-data protocol is going to be used for implementation of the Disa microprocessors.

C-Element

In order to implement the behavior of the protocols there has been design a special asynchronous component. The components is called a c-element and in figure 4.3(a) the diagram of the c-element is shown and in figure 4.3(b) the truth table for the c-element is shown.

The c-element functions in that way that every input has to be the same in order to change the output. This means that when all the inputs are set to 1 the output becomes 1, otherwise if all the inputs are set to 0 the output becomes 0 and else does the c-element keep the last value. The c-element can have more than 2 inputs. It is described on page 21 in [32] how the c-elements are designed.



(a)

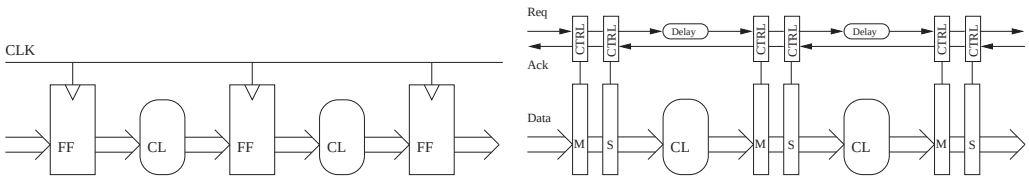
A	B	Y
0	0	0
0	1	Y
1	0	Y
1	1	1

(b)

Figure 4.3 *(a) the symbol for the c-element and (b) the c-element functionality*

4.1.2 De-synchronisation

The idea is to implement an asynchronous microprocessor based on the de-synchronisation technique described in [36]. It is starting from a synchronous microprocessor and replacing the global clock network with a set of local handshaking circuits. Figure 4.4(a) shows a synchronous circuit and figure 4.4(b) shows the synchronous circuit, which has been de-synchronized.



(a) (b)

Figure 4.4 *(a) Synchronous circuit, (b) De-synchronous circuit*

Handshake Protocols

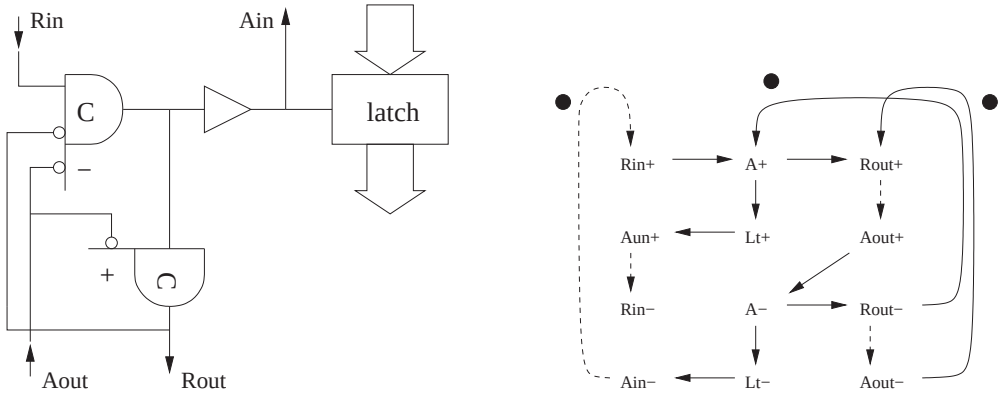
There are some criteria for the de-synchronisation technique which should be maintained for it to work. This concerns the selected the handshake protocols. The handshake protocols should achieve the liveness criteria. This implies that the handshake

protocols have a static speed of 1 i.e. the number of empty tokens between two valid tokens. The simple 4-phase bundled-data latch controller can therefore not be used.

The handshake should also achieve flow-equivalence and this requires that a handshake controller model has less than 8 states. The semi-decouple and fully-decouple handshake controllers both fore fill these two criterias. The semi-decouple handshake controller is selected because it requires two less c-elements than the fully-decouple handshake controller.

Semi-decoupled Latch Control Circuit

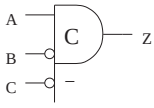
The semi-decoupled latch controller is designed by Furber and Day and it is presented in [35]. Figure 4.5 shows the semi-decoupled control circuit and the ancillary STG.



(a) Circuit (b) STG

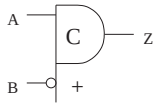
Figure 4.5 *(a) Semi-decoupled control circuit, (b) Semi-decoupled 4-phase STG*

Figure 4.6 shows the two c-elements used in a semi-decoupled control circuit and the corresponding logic function. The c-elements are implemented using state-holding gates, where the result is defined as follows: $z = z_{set} + z \cdot \overline{z_{resest}}$.



$$\begin{aligned} z_{set} &= a \cdot \bar{b} \\ z_{reset} &= \bar{a} \cdot b \cdot c \\ z &= a\bar{b} + za + z\bar{b} + z\bar{c} \end{aligned}$$

(a)



$$\begin{aligned} z_{set} &= a \cdot \bar{b} \\ z_{reset} &= \bar{a} \\ z &= a\bar{b} + za \end{aligned}$$

(b)

Figure 4.6 *C-element used by the semi-decoupled latch controller*

Putting It All Together

The reasons for using the semi-decoupled 4-phase bundled-data latch controller for desynchronisation has been explained. Figure 4.7 shows what the implementation would look like. The c-elements have been replaced with logic blocks. A reset signal has been inserted to ensure that semi-decoupled latch controller start correctly.

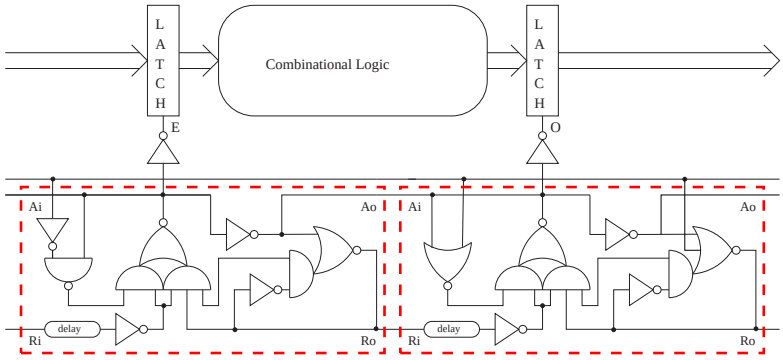


Figure 4.7 *Implementation of semi-decoupled controllers for even (E) and odd (O) latch*

4.1.3 An Other Asynchronous Microprocessor

In the process of the design of the Disa microprocessor, research was done to find out other implementations of asynchronous microprocessors and to find out how they were implemented. Two asynchronous microprocessors are briefly in introduced in this section.

AMULET

The AMULET microprocessor is probably one of the most known asynchronous microprocessors and is developed by the University of Manchester. AMULET are series of asynchronous microprocessors where the newest one is called AMULET3i [38].

The AMULET3i is an ARM microprocessor which is compatible with the 16-bit Thumb ARM instruction set that is used in the ARM9 microprocessors. It is implemented using semi-decoupled 4-phase bundled-data latch controller.

The first AMULET microprocessor (AMULET1) was originally implemented using a 2-phase latch control, but it was found out it was discovered that it is using too much space as explained in the previous section.

ARISC

ARISC[39] is asynchronous microprocessors developed at IMM. The ARISC is a reimplementation of a TinyRISC TR4101 from MIPS.

ARISC is implemented with Normally Opaque latch controller which is a special 4-phase low-power bundled-data latch controller. If time was available it would have interesting to use this latch controller for the Disa microprocessors.

4.1.4 Components

The implementation of the de-synchronous microprocessor requires implementation of extra component to ensure a safe communication between the different part of the microprocessor.

A fork component was implemented with two outputs and a join component with two inputs as described on page 59 in [32]. The join and fork component were expanded so they have multiple inputs or outputs respectively as described on page 21 in [32].

A demux and mux were also implemented as described on page 76 in [32]. These components were not used in the Disa microprocessor but they were used in design studies, which are explained in the following section.

4.1.5 Design Studies Using De-synchronisation Technique

In order to get familiar with the de-synchronisation design technique some design studies were performed. This section will explain the different design studies. The design studies were implemented at RTL hardware level and were simulated to verify they were behaving as expected.

Design Study 1

The goal of the first design study was to see if it was possible to implement the technique and get it to work as desired. The most simple test example is to make a loop consisting of a master latch, a slave latch and a combinatoric circuit as shown in figure 4.8. The master latch, semi-decoupled master control, slave latch and semi-decoupled slave control are implemented as shown figure 4.7. These four components are called a desyn-element in the project.

It is possible to see in the figure that the request and acknowledge signals are going around and the request signals is delayed in the function block.

Design Study 2

It was then the idea to start implementing a small microprocessor. The first design study includes a program counter that is stored in a desyn-element. The output request signal and data were forked where one part went into a function block which incremented the program counter. The other part was used to load an instruction for a memory and then the instruction was stored in a desyn-element. The instruction from the desyn-element was the then caught by a monitor. This was sending back the correct acknowledgements.

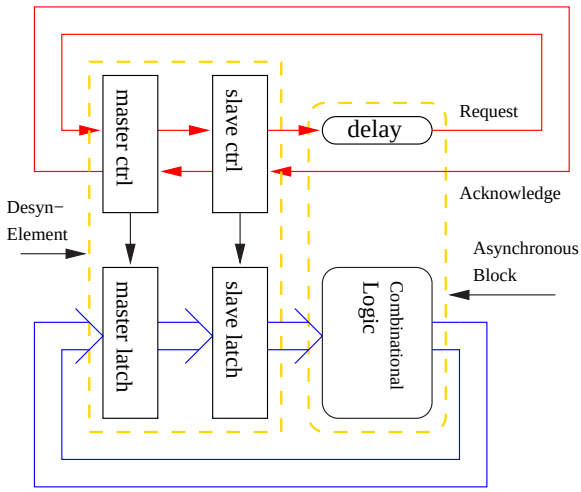


Figure 4.8 *A simple de-synchronised circuit*

Design Study 3

In setup 3 an instruction desyn-element get instruction from instruction test-bench struction. The instruction from the desyn-element was then decoded.

A register file was designed based on desyn-elements, an ALU and a bit-processor. The alu and bit-processor have only limited functionalities e.g. and, or, addition and subtraction. The output from the decoder was then sent to the register file, ALU and bit-processor to tell them what to do. The communications was ensured by fork and joins.

A demux and a mux were used to select whether the bit-processor or the ALU should be used.

Design Study 4

Finally, design studies 2 and 3 were combined. The result is shown on figure 4.9. The request signals and acknowledge signals are illustrated as one signal. The data signals and decode signals are also illustrated in the figure.

Summary

The design studies explored the de-synchronisation technique and it was easy to use. It was determined how-to reset the latch controllers, so they started correctly. The studies showed that the implementation of the fork, join, demux and mux were working as expected and the data was flowing correctly .

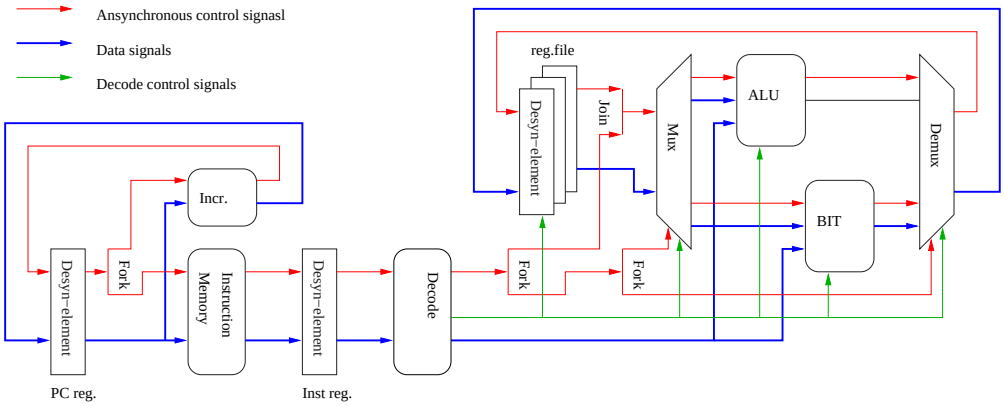


Figure 4.9 *Mini de-synchronised microprocessor*

4.2 Asynchronous AVR Architecture

This section covers the restructuring and implementation of the Disa microprocessor.

4.2.1 Restructuring the AVR Microprocessor

In order to de-synchronise the Nimbus microprocessor, the microprocessor needs to be restructured. The de-synchronisation can easily be done if the structure of the microprocessor is pipelined and it is clear which signals are dependant of each other. The Nimbus microprocessor did not satisfy these criteria.

The problem with the Nimbus processor was that all the main parts of the Nimbus microprocessor were written in one file as described in section 3.2.1. The Nimbus microprocessor was therefore restructured.

The restructuring included splitting up the core in many components each is responsible for controlling different parts of the microprocessor. The overall structure is described in the next section.

4.2.2 Overview

Figure 4.10 shows an overview of the Disa microprocessor. The different components on the figure will be described below.

- **PC ALU:** This component is responsible for calculation or selection of the next value for program counter.
- **Inst. Mem.:** This is the instruction memory. The AVR microprocessor pre-fetches the instruction.
- **Decode:** The instruction is decoded.

- **Status reg.:** The component selects and stores the status of the microprocessor. Many of the instructions are multi-cycled instructions and therefore require that the microprocessor knows how far it has come in executing the instruction.
- **Reg read:** Read data from the register file based on the decode instruction.
- **ALU:** The ALU.
- **Write Reg:** Data is written to the register file from the ALU or the data-bus.
- **IO Ctrl:** A read or a write to I/O components or the timer are detected and data is written to the out data-bus . The data-bus is split into an outgoing and an incoming data-bus.
- **IO:** The I/O components are accessed.
- **Ram Addr. Calc.:** The address is calculated.
- **RAM:** The memory reads/writes data from/to the data-bus.
- **Interrupt:** The interrupt component for detecting interrupts from the I/O components and the timer.

4.2.3 Timing

The instruction timing for the Disa microprocessor is the same as the Nimbus microprocessor. The timing of the different components are controlled by a number of forks and joins which are put in the right places.

The general kind of instructions will shortly be itemised below in order to understand the timing. All the instructions are pre-fetched.

- **Arithmetic and Logic Instructions:** This is the most simple instruction and are executed in one cycle. When the instruction is decoded the ALU executes the instruction and stores the result in the register file.
- **Branch Instructions** A branch instruction is executed in 1 or 2 stages. The ALU is first used to find out whether the instruction should branch. If not the instruction is executed in on cycle. If the instruction branches the status register is informed and so the next instruction is not used. It takes a cycle to load a new instruction from a different address.
- **Data Transfer Instructions** There are two different types of data transfer instructions:
  - **Load instruction:** The normal load instruction takes two cycles. The first cycle is the address calculated and the next cycle is data loaded from the memory and stored in the register.
  - **Store instruction:** The store instruction also takes two cycles like the load instruction. The address is calculated in the first cycle and data is handled by the `IO Ctrl`. In the next cycle is the data put on the data-bus and it is stored in the memory.
- **I/O access:** Access to components are done in the same way as the load and store instructions. The difference is that `IO Ctrl` detects that the I/O components and the timer are accessed.
- **Call instruction:** This is one of the most complicated instructions because it takes four cycles to execute. In the first cycle the instruction is decoded as a 32-bit

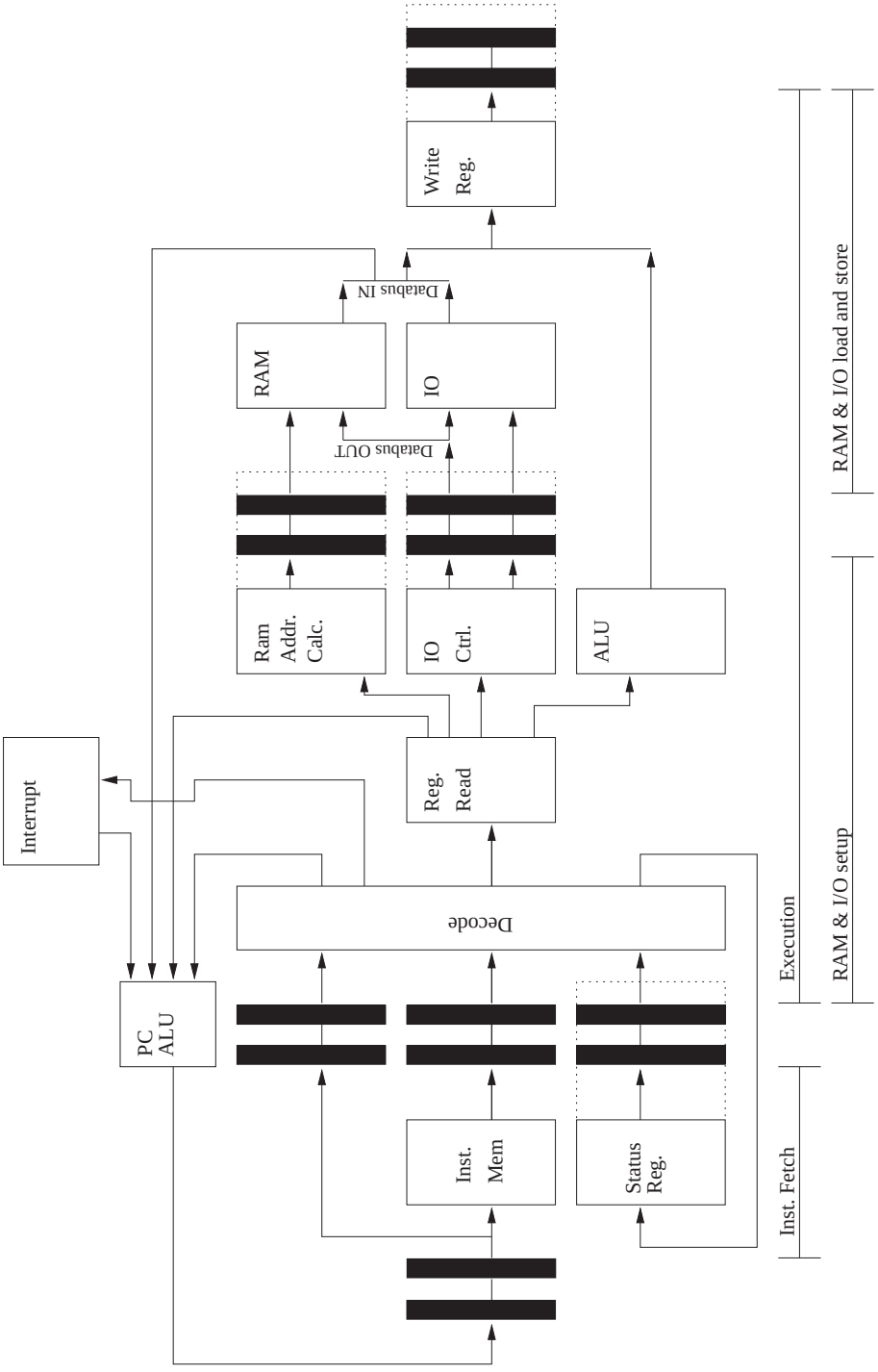


Figure 4.10 *Disa architectural overview*

instruction. In same cycle the current program counter is incremented by 2. In the second cycles, the program counter is stored in the data memory and the stack pointer is decremented by 2. In the third cycle the new program counter is set and in the fourth cycle the next instruction is pre-fetched.

- **Interrupts:** When an interrupt occurs the microprocessor is behaving like the call instruction except that the program counter is defined by the interrupt vector.

4.3 Implementation

This section covers the implementation of the Disa microprocessor. The basic asynchronous components and the Design studies components are described shortly to begin.

4.3.1 Asynchronous Components

All the asynchronous components have been implemented at gate-level. All the components have been compiled with ModelSim and all the components have be simulated in the design studies except the fork component with more than 4 outputs and joins with more than 4 inputs. All the components can be found in appendix H.3.

C-element

The C-elements have been compiled and simulated, but they are not possible to synthesis because the synthesis tool removes the logic functionality. The problem can be solved by using standard cells from the cell library.

The AO5LL is standard cell which has the boolean expression: $Z = !((A \cdot B + C) \cdot (A + B))$. If the Z signal is fed back into the A signal the c-element functionality is obtained. The component should be analysed and elaborated only. The component will be removed if the design is optimised with Synopsys.

Design Studies

The source code for the design studies can be found in appendix H.4. All the components were implemented at gate-level and they were simulating as expected.

4.3.2 Disa microprocessor

It was explained that the Nimbus microprocessor was re-structured in order to be desynchronisation. This sections describes the implementation of the Disa microprocessor. The implementation is described in a top-down approach. The source code for the microprocessor can be found in appendix H.5. The implementation of the different data blocks are the same in the Nimbus microprocessor and the Disa microprocessor. The data blocks have been split into more files.

The implementation of the Disa microprocessor was not finished. All the different blocks for the Nimbus microprocessor were design, but not put together. The different block will now be explained shortly according to figure 4.10. It is mentioned in the description of the component if the implementation is not completed. The components have been compiled, but they have not simulated.

Top

The top level includes the core, the timer and I/O components. It is the meaning that the I/O components and timer should still be synchronous. Therefore a synchronous module was needed to be implemented in order to ensure that the communication from the asynchronous core to the components were not corrupted. The top level is described in `top_asyn_avr_core_beh.vhd`

Core

The Core component is described in `asyn_avr_core_beh.vhd` and it includes connect all the components. This file has not been implemented yet.

PC Select

The PC Select component consists of two parts component. The first component,
`pc_selection_beh.vhd` controls the selection of the value program counter and the other,
`pc_regfile_beh.vhd` is the program counter register.

ROM

The ROM component `rom_interface_beh.vhd` encapsulates the synchronous ROM from the Nimbus microprocessor. It also includes the instruction register.

Decoder

The Decode component `decode_inst_beh.vhd` decodes the instruction.

Status Register

The Status register components `decode_status_reg_beh.vhd` controls the decoder and tells the decoder whether to load a new instruction or keep executing the current instruction.

Register File

The Register component is made by two modules. The first one `regfile_beh.vhd` controls the register file. The other component `regstore_beh.vhd` is responsible for selecting the data which is stored in the register. This can either be data from the databus or the ALU.

ALU

The ALU also consists two components the ALU `alu_beh.vhd` and the bit processor `bit_processor_beh.vhd`.

I/O components and Timer

This component `io_regfile_beh.vhd` is an interface to the I/O components and Timer. It is responsible for the communication between the asynchronous microprocessor and the synchronous microprocessor.

RAM

The RAM component `decode_ram_reg_beh.vhd` is like the ROM an encapsulation of the synchronous RAM module.

Interrupts

This component `interrupt_encoder_beh.vhd` is responsible of detection interrupt from the synchronous components.

4.4 Discussion

This section discusses the experience using the de-synchronisation technique. The technique is easy to understand because the asynchronous circuit is pipelined in the same way as a synchronous circuit. The only big different in the pipelining is the asynchronous request and acknowledge signals which ensure that the data is available.

4.4.1 Problems with De-synchronous

The problem with the Nimbus microprocessor is that the hardware description is written tortuous. It would have been good to re-implement the chip before starting the de-synchronization of the chip. It was difficult to find out how the routing of the request and acknowledge signals should be.

The status register in the microprocessor is constructed by many 1 bit registers that control the behaviour of a multi-cycle instruction. The re-implementation would have made it easier to de-synchronous this register and it would probably have reduced the area of the microprocessor.

4.4.2 The Optimal use of De-synchronous

The de-synchronisation of a microprocessor is most optimal if the microprocessor has a straightforward pipeline structure. An optimal microprocessor to de-synchronous would be the Mips processor described in [40]. The pipeline structure of the Mips

processor simple and it is very easy to find out where to place the fork and joins to ensure that the data is valid.

4.5 Summary

The chapter covers how to de-synchronise a microprocessor. The de-synchronisation technique has been demonstrated with a number of design studies. The design for the Disa microprocessor has been explained. All the components for the Disa microprocessor have been implemented, but they have not been combined because of lack of time.

CHAPTER 5

Comparison of Results

This chapter contains a comparison of the ATmega128L and the Nimbus microprocessor synthesised with the $0.12\mu m$ and $0.25\mu m$ cell library. First the current and power consumption will be compared for the microprocessors and then the measurements are discussed with sensor networks in mind.

5.1 Power Consumption Estimation

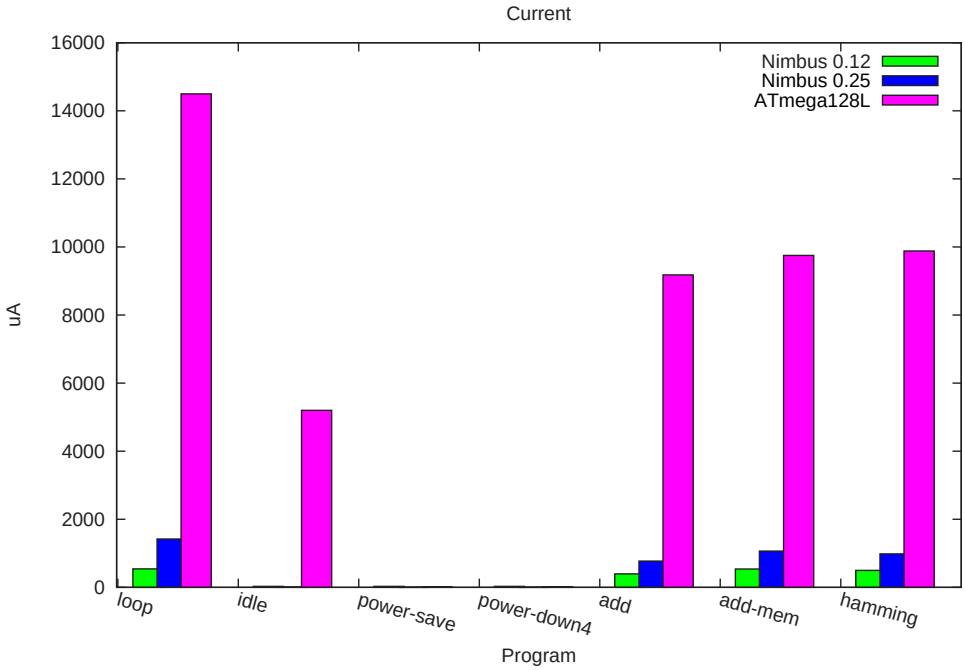
Figure 5.1 and 5.2 shows the power and current consumption for the ATmega128L, Nimbus 0.25 and Nimbus 0.12 microprocessor. These measurements are found in table 2.6 and table 3.12.

The figures show that the ATmega128L is completely outperformed by the Nimbus microprocessor. This can be explained by the fact that ATmega128L uses an old cell library. The chip is properly made by a $0.35\mu m$ library technology, but it has not been possible to find the precise technology information about the fabrication of the chip. For older cell libraries the dynamic current consumption are much higher than for newer ones.

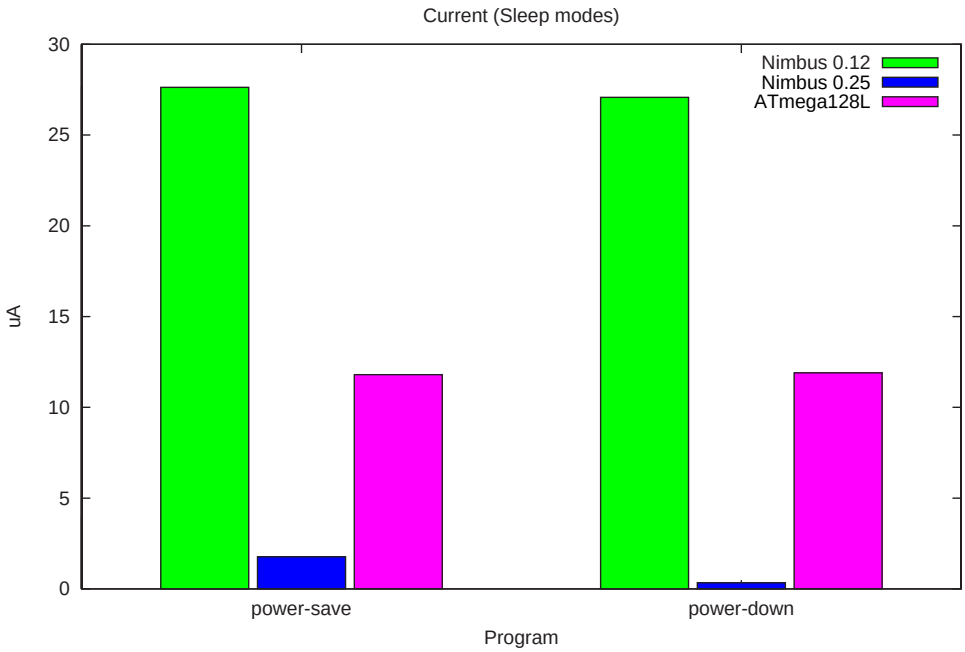
The ATmega128L uses much more current than the other microprocessor except during the sleep power modes that are shown in figure 5.1(a). Figure 5.1(b) only shows the current consumption for the sleep modes. The ATmega128L has a lower current consumption than the Nimbus 0.12. This is because of a higher leakage current for the Nimbus 0.12 than the ATmega128L. Old cell technology has normally a lower leakage current than the new ones.

Figure 5.1(b) shows that ATmega128L has a higher current consumption than Nimbus 0.25. This was not as expected, since the ATmega128L is based on an older technology which has a lower leakage current. The reason could be that the ATmega128L has five extra 8-bit I/O ports, two extra timers, some extra instructions, one extra UART and a SPI more than the Nimbus microprocessor. The extra components could be the reasons to the increase leakage current.

The power consumption for the microprocessor are seen in figure 5.2(a) and 5.2(b). These figures show a poorer performance by the ATmega128L. This can be explained

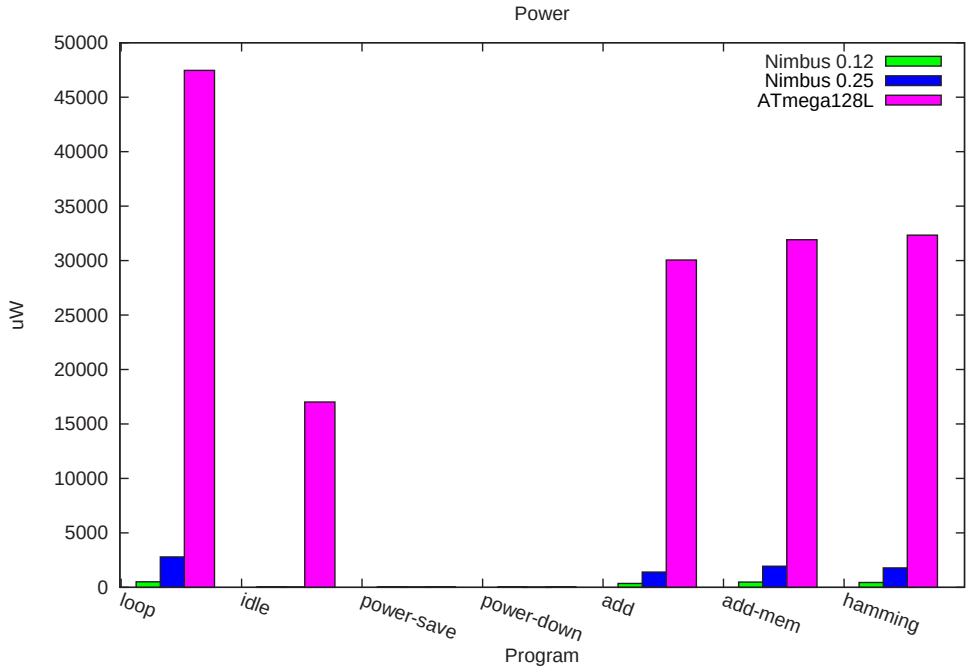


(a) Current consumption

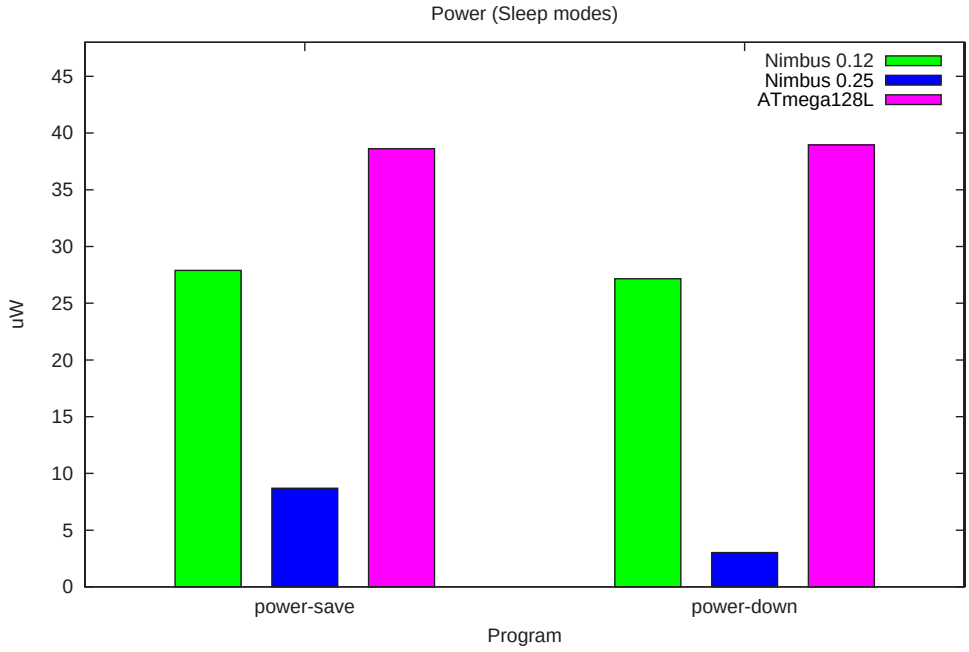


(b) Current consumption for the power-save and power-down sleep mode

Figure 5.1 *Current consumption for the ATmega128L, Nimbus 0.12 and Nimbus 0.25 for running the test programmes described in section 2.2.*



(a) Power consumption



(b) Power consumption for the power-save and power-down sleep mode

Figure 5.2 *Power consumption for the ATmega128L, Nimbus 0.12 and Nimbus 0.25 for running the test programmes described in section 2.2.*

by the fact that the ATmega128L, Nimbus 0.25 and Nimbus 0.12 use a supply voltage of 3.3V, 1.8V and 1.2V respectively (The Nimbus 0.12 memory uses 0.9V).

5.2 Usage of the Microprocessor for a Sensor Network

In a sensor network the motes often wake up, senses or transmit data and then goes back to sleep. The article [28] describes how PowerTOSSIM have been used for measuring power consumption for motes in a sensor network. PowerTOSSIM is an extension of TOSSIM, which is a sensor network simulator that can estimate power consumption. The article calculates the execution time for all the programming examples that are included with TinyOS operating system. In the article the tests are based on a Mica2 mote, which as the Hogthrob, contains an ATmega128L. Mica2's microprocessor is running at the same frequency as the microprocessor in this project. The article shows that the average the execution time of a mote in a sensor network is 1 % for all the test programs. This is very interesting in the perspective of using the Nimbus microprocessor for a sensor network because different cell libraries can be used for synthesising the microprocessor.

The measurements in section 5.1 showed that the Nimbus 0.12 has the lowest power consumption when executing a program whereas the Nimbus 0.25 has the lowest power consumption when the microprocessor is asleep. This rises the question about which Nimbus microprocessors is most suitable for a sensor network.

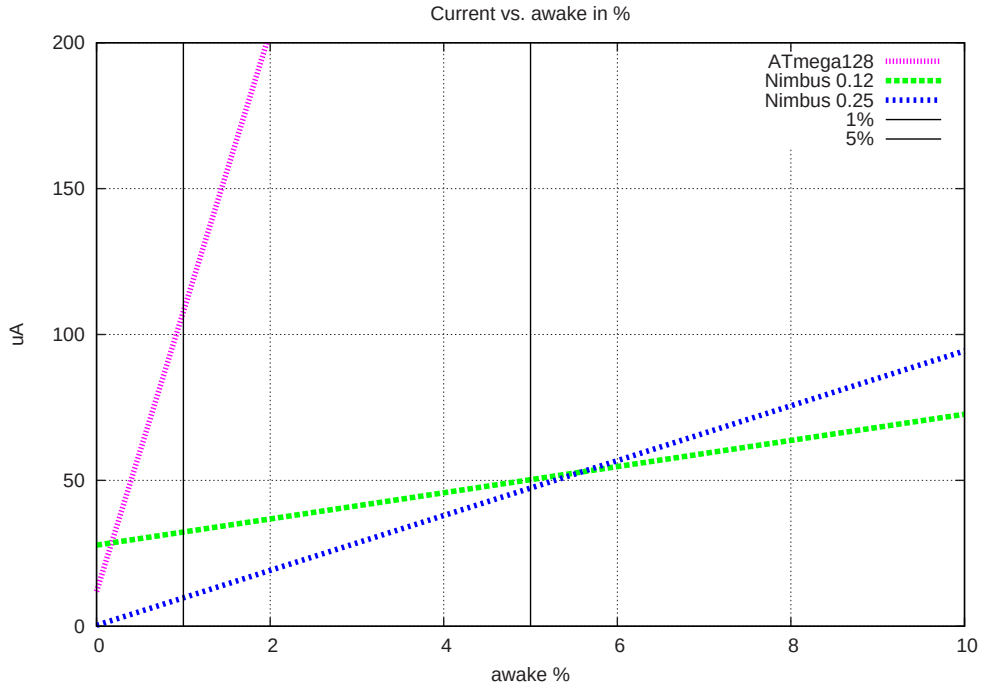
Figure 5.3 shows the average of current and power consumption in relation to the execution time for the microprocessors in %. The test programs `add`, `add-mem` and `hamming` have been used to calculate the average power consumption, when the microprocessors is executing. The test program `power-down` is used to determine the power consumption when the microprocessors is asleep. This program is chosen because of its low current and power consumption.

The calculation of the execution time in % does not includes startup time or power down power consumption of the microprocessors. The microprocessors are either asleep or awake.

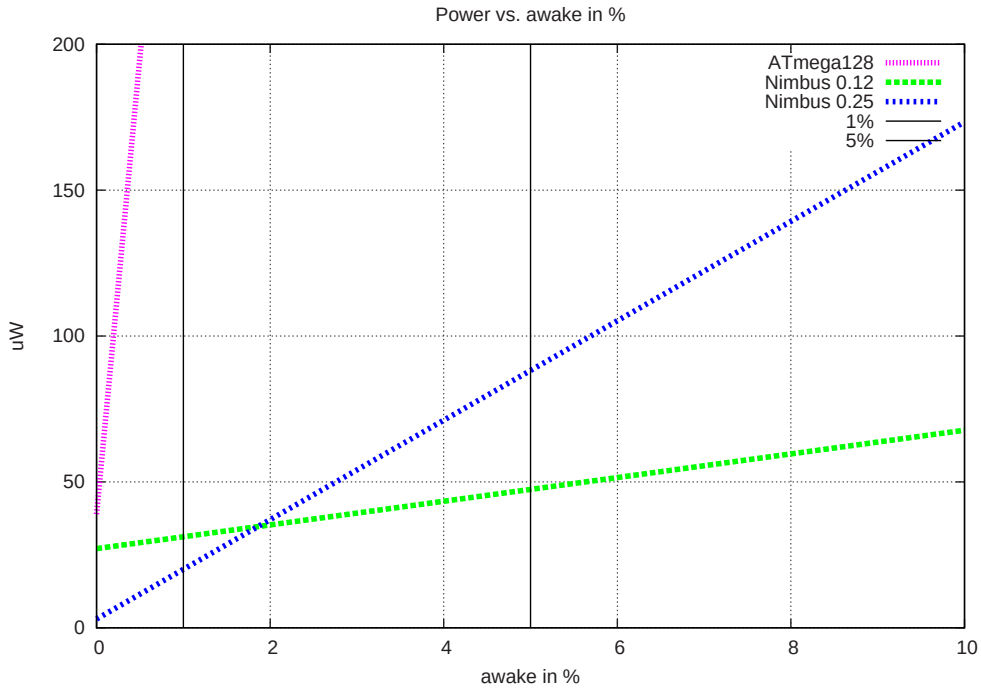
The figures shows that the ATmega128L is completely outperformed as discussed in the previous section, but what about the two other microprocessors? The voltage have a significant influence on the results for the Nimbus microprocessors. Figure 5.3(a) shows that the current consumption of the Nimbus 0.25 is much lower than the Nimbus 0.12 if the execution time is less than 5 %.

Since the Nimbus 0.25 has twice the amount of supply voltage as the Nimbus 0.12 the situation changes. The graph on figure 5.3(b) shows that only if the execution is less than 2 % the Nimbus 0.25 is more efficient than the Nimbus 0.12.

In a sensor network the execution time is about 1 %. The figures show that the Nimbus 0.25 is a better choice of microprocessor for a sensor network. However this is with subject to certain reservations. If the execution time increases to 2 % the Nimbus 0.12 may be a better solution.



(a) Current consumption versus awake in %



(b) Power consumption versus awake in %

Figure 5.3 *Estimation of average power and current consumption for the three AVR microprocessors when they are awake and running in %.*

It should also be considered that the Nimbus microprocessor does not contain information about the oscillators for both the internal clock and the external clock. A clock could be constructed by and-gates and an inverter. The oscillators will then lead to an increase in dynamic power consumption and leakage power. If the dynamic power consumption is comparatively greater than the leakage, the Nimbus 0.12 could be the best solution.

5.3 Summery

The ATmega128L and the Nimbus microprocessors have been compared which demonstrated a higher overall power consumption by the ATmega128L. The Nimbus 0.12 has the lowest power consumption when executing whereas the Nimbus 0.25 has the lowest power consumption when the microprocessors is asleep. In a sensor network perspective it was demonstrate that the Nimbus 0.25 has a lower power consumption compared to the Nimbus 0.12.

CHAPTER 6

Discussion

This chapter discusses the experiences and results obtained through the work of the project. This includes the experience of using open cores. Is it a good idea to use open cores in commercial products? There have been used and implemented some microprocessors for this project. Here the different modelling possibilities using the microprocessors are considered. The costs for production of a sensor network mote using different technologies is also discussed. The results show that leakage current is increasing in the new cell libraries. Is it possible to reduce leakage current? The asynchronous microprocessor was not successfully implemented. What is the advantage of the power consumption in an asynchronous microprocessor? At last the different design tools are evaluated.

6.1 Open Cores

There are many reasons for using open cores instead of either developing yourself or purchasing off-the-rack products. In this section the motivation for and against is discussed.

`opencores.org` is the most known homepage where hardware description of components are available for free. These are known as open cores and most of these cores are subject to the GNU Public license (GPL). Many of the cores are implementations of open standards or reimplementation of commercial products.

The problem with many of the cores are that they are very poorly certified and may include many bugs. This is due the fact that the community for developing open hardware is not as organised as the community for open software. This means that bugs in cores may never be notified.

The AVR microprocessor from `opencores.org` was very poorly documented and many bugs were found. The microprocessor is not logically structured. Looking back at this project it would probably have been useful to re-implement the microprocessor in order to ensure a more suitable microprocessor. In the re-implementation the microprocessor could have been prepared for de-synchronisation.

Not all the cores on opencores.org are poorly documented. Many of the communication cores are not complicated to integrate in a hardware model.

If an open core is going to be integrated in a design it should be carefully examined. Some of the main criteria for using a core is an easily understandable structure and the level of documentation. It is a problem that no certified information for testing of the core is available. However, if the core is put in a commercial product, it has to be tested.

The advantage of an open core is that the core is simple to start up, since many of the cores include a test bench. If a product needs i.e. a UART it is easy to get the core from opencores.org. The UART can be simulated and downloaded to a FPGA. Later in the design process it can be discussed whether it is more affordable to buy the core from an IC producer or make a full-custom chip.

6.2 Modelling Possibilities

This project concerns the use and development of different AVR microprocessors that utilise different cell libraries. The idea was to investigate all the different platforms which should estimate the efficiency for a sensor network.

Figure 6.1 shows how all the AVR microprocessors can be implemented. The first possibility is to buy an ATmeage128L, which is on the Hogthrob board. Another possibility is the Nimbus microprocessor that is a customised AVR microprocessors from opencores.org. The Nimbus microprocessor can then be synthesised and placed & routed for the FPGA on the Hogthrob board. Hereby it is possible to observe in real time that the Nimbus microprocessor behaves like the ATmega128L.

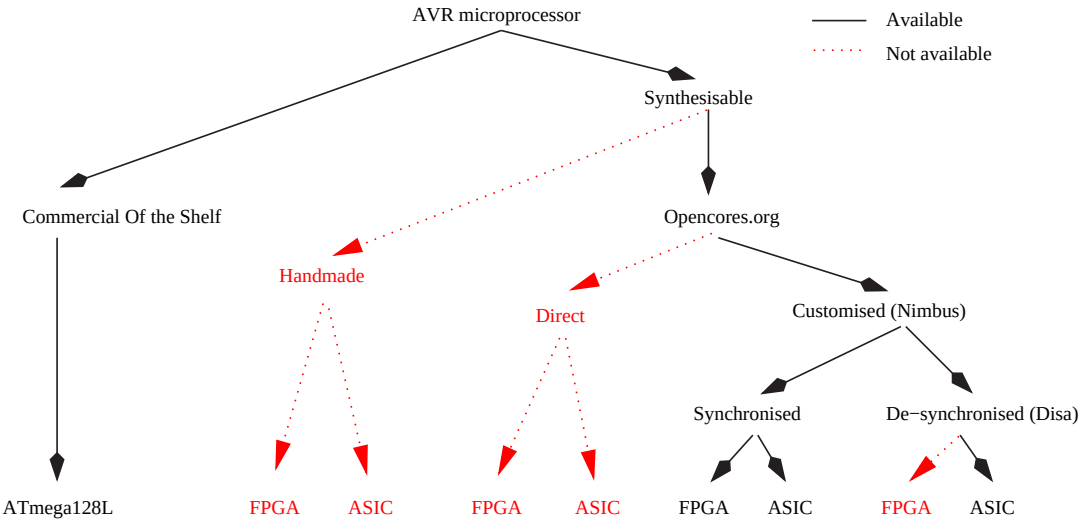


Figure 6.1 *A diagram of the different possibilities for the use of the AVR microprocessors*

The Nimbus microprocessors can be synthesised for an ASIC using different cell

libraries. The performance of the ASIC microprocessors can then be compared with the ATmega128L which was done in this project.

It was the intention to implement an asynchronous version of the Nimbus called Disa. This should have been compared with the other microprocessors.

The microprocessors make it possible to experiment with customisations of the Nimbus processor. In a sensor network perspective, the Nimbus microprocessors could integrate hardware components which are able to manipulate sensed data.

The Nimbus microprocessor could also integrate a hardware component for error correction that can be used for radio transmission. This is today done either in the microprocessor or in the radio. If a mote in a sensor network sends large amount of data, it would be useful to have a module designed for data compression. This may reduce the radio transmission time and would lead to a reduction in power consumption.

Finally the microprocessors can be used in a more common perspective i.e. in a system on a chip (SoC) or some kind of multimedia device with MP3, radio and camera. The microprocessors could then control the behaviour of the other components.

6.3 Technology Versus Mote Costs

Motes are sometimes placed in the field where their sensitivity towards weather conditions can limit their lifetime and it may not be possible to find and/or use them again. A sensor network may consists of thousands of motes and it is therefore important that the cost per mote is low or it will be too expensive to have a sensor network.

All the components, like sensors for the motes, have to be made in a mass production in order to reduce the costs of the mote. The components are usually not the latest technology, because the components based on the newest technology are extremely expensive. The motes are therefor made by components that are based on rather old technologies. The ATmega128L consists of old technology. Nevertheless the microprocessor is still popular because of its low cost.

It has been illustrated that the Nimbus microprocessor has a much lower power consumption than the commercial ATmega128L. Production of new chips is a very expensive undertaking. The question is, does it payoff to produce full-custom microprocessors like the Nimbus in order to save power or should an ATmega128L be purchased.

The price of a full-custom microprocessor may equal an IC. It is only a matter of the number of chips that are needed and which technology to apply. It was shown in section 5.2 that the newest technology may not be the best technology for a mote in a sensor network which could reduce the production costs. It was not possible to find a price to manufacture the microprocessor.

6.4 Power Consumption

The advantage of full-custom microprocessors is the possibility of deciding which functionalities the microprocessors should include. The ability to customise the microprocessors affects performance. The customisation can include removal of all the functionalities that are not being used or situations where other features are added. The customisation issue can reduce the power consumption and this may lead to a longer lifetime of the microprocessor in a sensor network.

As mentioned in section 5.1 the Nimbus microprocessor does not include all the I/O components and timers, which are available for the ATmega128L. The Nimbus microprocessor does nor have an oscillator. This helps reducing the leakage current of the Nimbus microprocessor.

The problem with the new cell library is an increase in leakage current. The microprocessor for the project has been synthesised with low leakage current cells and even, then the leakage current is about ten times larger than the older cell library. The old cell library does not have a low leakage current cell.

Another important factor is to reduce the leakage current. This can be done by using the low leakage cell library for synthesis as it is done in this project. The low leakage cell library uses special high-V_{th} transistors.

There are many solutions to reducing leakage current and some of them are easier to implement than others. An option is to power down parts of a circuit, which are not used. However it is difficult to estimate the amount of current that the component consumes when being switched on and off. This is a very essential aspect in a sensor network because a mote spends a high percentage of the time in a dormant state.

A master thesis [42] looked at the possibility of reducing leakage current. The thesis presented some new ideas. One of them is to make larger logic blocks on-the-fly in the synthesis process. The idea was to connect transistors in series which showed a reduction in leakage current. The problem with this technique is that it requires a company like to Synopsys change the way a hardware description is synthesised.

The would have been useful if it was possible to change the supply voltage. This was not possible when using Synopsys for synthesis. Many low-power microprocessors have voltage scaling and it could have interesting to see the effect on the power consumption and the timing in the circuit. The Nimbus microprocessors can run at a speed around 100 MHz.

If the Disa microprocessors was successfully implemented it would have been interesting to see how the Disa microprocessor performed compared to the Nimbus microprocessor. An asynchronous microprocessor is able to execute with a much lower voltage because it is speed independent.

6.5 Hardware Development Tools

Many tools were explored when implementing, simulating and synthesising the microprocessors.

HDL Designer from Mentor Graphic, ModelSim editor, ISE editor from Xilinx and Emacs were used for writing VHDL. All the editors have syntax highlight. All the editors are available for both Windows and Linux.

The emacs is the most advanced editor and it supports many programming languages aside from VHDL and C/C++. It was mainly used for writing VHDL because it has many auto complete functionalities and it gives the programmer the advantage to develop both VHDL and C/C++ in the same environment. This makes the developing of software and hardware much faster

HDL Design is the most advanced tool since it integrates a VHDL simulator and FPGA tools in the same product. The editor has a simple VHDL error detector and there is a graphical tool for connection components. HDL Designer is the best editor for developing big circuit with many components because it has all the helping tools.

Compilation and simulation was mainly done by ModelSim, but the open source simulation GHDL was also tried. The problem is that GHDL is not fully developed and it does not support all the VHDL packages. It is there not recommended to use GHDL.

Xilinx ISE, Precision Synthesis and Leonardo Spectrum were used for synthesising and placing & routing of the Nimbus microprocessor in the FPGA on the Hogthrob board. ISE was most commonly used, but they are all simple to use.

Finally the Synopsys Compiler was used to synthesise the Nimbus microprocessor for an ASIC library. Even though the Synopsys is the most advanced program the user interface is very primitive.

Based on the experience working with all these different tools the fastest way to develop hardware was to use FPGA's. These design tools are much better integrated.

The problem involving the use of Synopsys is that it takes long time to setup and synthesise a design compared to the FPGA tools.

6.6 Summary

In this chapter, many important issues related to sensor networks have been discussed. It was argued that the use of open cores may save development time. The influence of the different technologies have been evaluated and it showed how power consumption can be reduced in a microprocessor. The potential power consumption for a asynchronous microprocessor has been discussed. Finally the different tools for realisation of the microprocessor have been evaluated which showed that a FPGA environment is the most suitable.

CHAPTER 7

Conclusion

The conclusion consists of three different sections. The first section is about the contribution and the results achieved and realised.

The project encountered some problems because of the structure of the AVR microprocessor from OpenCores and because of unforeseeable error from Synopsys Synthesis tools. These problems are discussed in the second section.

Finally the last section will present some ideas for future work.

7.1 Contribution

The main object of the work was to explore the Atmel ATmega128L and compare it with a customised and a de-synchronised implementation of an AVR microprocessor, which is based on a core from `opencores` in the perspective of using the microprocessor for sensor networks.

The work has illustrated that open cores can be integrated in embedded systems, but all the cores are not ready to use. The core can be bug infested and may require customisation.

A tool has been developed, which integrates a software program in a hardware model. The tools make it possible, in one step, to compile a software program, convert the binary program file to a hardware description and include it in the other parts of the hardware model.

The customised AVR microprocessor was successfully synthesised and placed & routed for FPGA. The AVR microprocessor in the FPGA could successfully execute the same programs as ATmega128L. The customised AVR microprocessor was also successfully synthesis for an ASIC based on two different cell libraries and then compared with the ATmega128L in order to measure the power consumption.

It was shown that a microprocessor synthesised with old cell library technology could be more appropriate for sensor networks. This is because the leakage current is becoming an increasing factor for low-power design.

Finally the contribution demonstrated the principle for de-synchronising a synchronous microprocessor. A design study was successfully implemented which illus-

trated that the de-synchronising technique works. A microprocessor was successfully designed and all the different components for the microprocessor were implemented, but the they were not assembled because of limited time.

7.2 Discussion

The project encountered two types of problems during the implementation phase of the Nimbus and Disa microprocessor. The first problem concerns using the Synopsys for synthesis and the other problem is about the structure of the AVR microprocessor.

7.2.1 Synopsys Synthesis

Some problems were encountered during synthesis of the AVR microprocessor. They resulted in that it was impossible to do back-annotated simulation of the circuit. The problem is described in the next subsection.

It would have been a good idea in the beginning of the project to have been familiar with synthesis flow and done some back-annotated simulation before. Then the problems would have been expected when synthesising the whole Nimbus microprocessor. It was very difficult to find what was going wrong because ModelSim was reporting thousands of errors.

Back-annotated Simulation Problems

A problem occurred during synthesising process of the Nimbus microprocessor because Synopsys Synthesis was using incompatible wires names, which are not supported by ModelSim. This is a known bug could be avoided by writing the command to `dc_shell`. The solution was found on Synopsys SolvNet, which is a solution database that contains answers to problems and bugs.

Another problem occurred when reading the top-level netlist and the static timing information from Synopsys into ModelSim. The netlist variable names did not match the static timing information names. A solution could NOT be found on SolvNet. The problem was solved by using a Verilog netlist instead of a VHDL netlist. It was then possible to back-annotated simulation using ModelSim.

Memory Problems

A lot of time was also spent on synthesising the memories using Synopsys Synthesis, which is not possible using the cell library. Therefore the memory from STMicroelectronics was used instead. It may have been possible to make some memory modules, which could be synthesised. This could have be done by breaking the memory into small pieces and put them together.

7.2.2 Structure of the AVR Microprocessor

The other problem results from the structure of the AVR microprocessor from Open Cores. A lot of time was used to understand how the AVR microprocessor was implemented. The structure proved to be much more complicated when the process of de-synchronising the Nimbus microprocessor was begun.

This is because there are many signals that are used in different components of the microprocessor. The placement of the signals in the components is not organised logically and they may be used in components where they look like a small hack to solve a bug. This can be confirmed by the fact that many bugs were found. If the microprocessor was better structured it could have led to a successful implementation of Disa.

7.3 Future Work

This section presents some ideas for future work. A re-implementation of the Nimbus microprocessor and a successfully implementation of the de-synchronised microprocessor is an obvious idea. An other solution could be to implement an asynchronous version of the AVR microprocessor, which should be developed from scratch and compare it with the other microprocessors.

The Nimbus microprocessor is curently statically preprogrammed. This means that every time a new program has to run on the microprocessor the microprocessor has to be synthesised and this takes several minutes. It could be interesting to implement components, which make it possible to download a new program to the core. This could be done using a serial interface which is connected to the PC.

Finally it could be useful to interface the Nimbus microprocessor with the radio and temperature sensor from the Hogthrob board so that it is possible to construct real a sensor network.

Bibliography

[1] David E. Culler and Hans Mulder, Smart Sensors to Network the World, Scientific American, 2004

[2] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, John Anderson, Wireless Sensor Networks for Habitat Monitoring, 2002 ACM International Workshop on Wireless Sensor Networks and Applications September 28, 2002, .Atlanta, GA.. (also Intel Research, IRB-TR-02-006, June 2002.)

[3] Trevor Mudge, Low Power Systems on a Chip - Today's Challenge, Bredt Professor of Engineering, The University of Michigan, Ann Arbor, Slides, July 2004

[4] Kees Van Berkel, Ronan Burgess, Joep Kessels, Marly Roncken and Firts Schalij, Asynchronous Circuits for Low Power: A DCC Error Corrector, IEEE Design & Test of Computers, pages 22-32, 1994.

[5] Kees Van Berkel, Ronan Burgess, Joep Kessels, Marly Roncken and Firts Schalij, A Single-Rail Re-implementation of a DCC Error Detector Using a Generic Standard-Cell Library, In 2nd Working Conference on Asynchronous Design Methodologies, London, May 30-31 1995, pages 72-79, 1995.

[6] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic and Pieter J. Hazewindues, The First Asynchronous Microprocessor: The Test Results, Department of Computer Science California Institute of Technology Pasadena CA 91125, USA, pages 95-109, April 1989.

[7] C.D. Nielsen, J. Staunstrup and S. R. Jones, Potential Performance Advantages of Delay Insensitivity, Department of Computer Science, Technical University of Denmark and Department of Electrical and Electronic Engineering, The University of Nottingham, 1991.

[8] Lars S. Nielsen, Cees Niessen, Jens Sparsø and Kees Van Berkel, Low-Power Operation Using Self-Timed Circuits and Adaptive Scaling of the Supply Voltage, IEEE Transactions on very large scale integration (VSLI), System, Vol. 2, No. 4 December 1994, Pages 391-397, 1991

[9] Kees Van Berkel, Mark B. Josephs and Steven M. Nowick, Scanning the Technology, Application of Asynchronous Circuits, Proceedings of The IEEE Vol. 87, No 2, February 1999, pages 223-231, 1999.

[10] N.C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien and J. Liu, A Low-Power, Low Noise, Configurable Self-Timed DSP, Cogency Technology Inc., 120 Eglinton Ave. E, Suite 500, Toronto, Ontario, M4P 12 Canada, IEEE, 1998

[11] S. Furber, Industrial take-up of asynchronous design, slides 2, Second ACiD-WG Workshop of The European Commission FiFth Framework Program, Munich, Germany, 28-29 January, 2002.
[12] AVR Homepage. http://atmel.com/products/avr/
[13] Motorola http://www.microcontroller.com/news/motorola_hcs08.asp and http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MC9S08GT60&nodeId=01624684498634&tid=EMK200409EMK5246
[14] 8051 from Intell, http://support.intel.com/design/embcontrol/
[15] TinyRISC from MIPS http://www.mips.com/content/PressRoom/TechLibrary/Backgrounders/mips_processors
[16] MSP430 from TI. http://www.ti.com/msp430
[17] TinyOs supports TI MPS430 now. http://mail.millennium.berkeley.edu/pipermail/tinyos/2004-May/000253.html
[18] Instruction set of 8-bit AVR http://atmel.com/dyn/resources/prod_documents/doc0856.pdf
[19] Description of Atmel AVR ATmega103 http://atmel.com/dyn/resources/prod_documents/doc2467.pdf
[20] AVR FAQ http://partsandkits.com/avr-faq.htm
[21] Kashif Virk, Martin Leopold, Martin Hansen, Phillipe Bonnet and Jan Madsen, Design of A Wireless Sensor Node Development Platform for Sow Monitoring, expected released in 2005 and can be found on Hogthrob internal homepage.
[22] Description of Atmel AVR ATmega103 http://atmel.com/dyn/resources/prod_documents/Doc0945.pdf
[23] The differences between the ATmega103 and ATmega128. http://atmel.com/dyn/resources/ prod_documents/doc2501.pdf
[24] Martin Leopold, Power Estimation using the Hogtrob Prototype Platform, M. Sc. Thesis, Computer Science Department of University of Denmark, 2004
[25] XST User Guide www.xilinx.com 1-800-255-7778
[26] Philip Levis, Sam Madden, David Gay, Joe Polastre, Robert Szewczyk, Alec Woo, Eric Brewer and David Culler, The Emergence of Networking Abstractions and Techniques in TinyOS, Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004).
[27] Joseph Polastre, Robert Szewczyk, Cory Sharp, David Culler, The Mote Revolution: Low Power Wireless Sensor Network Devices, in Proceedings of Hot Chips 16: A Symposium on High Performance Chips. August 22-24, 2004.
[28] Victor Shnayder, Mark Hempstead, Borrong Chen, Geoff Werner Allen, and Matt Welsh Division of Engineering and Applied Sciences, Simulating the Power Consumption of LargeScale Sensor Network Applications, Sensys2004, 2004
[29] Nicolai Jørgensen, Design of low-power platform running an embedded operating system. master thesis, department of Informatics and Mathematical Modelling of Technical University of Denmark, november 2003.
[30] Randal E. Bryant, Kwang-Ting Cheng, Anrew B. Kahng, Kurt Keutzer, Wojciech Maly, Richard Newton, Lawrence Pileggi, Jan M. Rabaey and Alberto

Sangiovanni-Vincentelli, Limitations and Challenges of Computer-Aided Design Technology for CMOS VLSI, Proceedings of The IEEE, Vol. 89, No. 3, March 2001, IEEE, 2001

[31] Ran Ginosar, Fourteen Ways to Fool Your Synchronizer, Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03), 2003 IEEE

[32] Jens Sparsø and Steven Furber, Principles of Asynchronous Circuit Design, A System Perspective, 2001.

[33] S. Furber and M. Edwards. Asynchronous Design Methodologies, Mancheter, UK, 31 March - 2 April, 1993

[34] Jens Sparsø, Christian D. Nielsen, Lars S. Nielsen and Jørgen Staunstrup. Design of Self-timed Multipliers: A Comparison. Department of Computer Science, Technical University of Denmark, 1993.

[35] Stephen B. Furber and Paul Day, Four-Phase Micropipeline Latch Control Circuits, IEEE Transaction on very large scale integration (VLSI) system, Vol 4, No 2, June 1996, Pages 247-253, 1996.

[36] I. Blunno, J. Cortadella, A. Kondratyev, K. Lwin and C. Sotiriou, Handshake protocols for de-synchronization, Proceedings of the 10th Inernational Symposium on Asynchronous Circuits and System (ASYNC'04), 2004.

[37] Rakefet Kol and Ran Ginosar, A Doubly-Latch Asynchronous Pipeline, VLSI System Research Center, Electrical Engineering Department Technion, Isreal Institute of Technology, IEEE, Pages 706-711, 1997

[38] J.D. Garside, WJ Bainbridge, A Bardsley, D.M. Clark, D.A. Edwards, S.B. Furber, J. Liu, D.W. Lloyd, S. Mohammadi, K.S. Pepper. O. Petlin, S. Temple and J. V. Woods, AMULET3i - an Asynchronous System-on-Chip, Dept. of Computer Science, The University of Manchester, p. 162 -175. In: Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000)

[39] Kåre T. Christensen, Peter Jensen, Peter Korger and Jesper Sparsø. The Design of an Asynchronous TinyRISC TR4101 Microprocessor Core, Department of Information Technologiy at Technical University of Denmark, 1998, p. 108 -119. In: Advanced Research in Asynchronous Circuits and Systems, 1998

[40] David A Patterson and John L. Hennessy, Computer Organization & Design, The Hardware/Software Interfase, Second Edition.

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

[42] Jacob Gregers Hansen, Design of CMOS cell library for minimal leakage current, Master's Thesis, Project Number 55, department of Informatics and Mathematical Modelling of Technical University of Denmark, 2004

[43] SPHS9, Single Port High Speed SRAM Generator in HCMOS9gp, Product Specification, Version - 2.1, ST Microelectronics Central R&D, 27 November 2002.

[44] SPSMALL9gp, Small Size Memory Generator in HCMOS9gp, Project Specifications, Version - 1.1, November 2002, ST Microelectronics Central R&D.

[45] SPLarge, Single Port, Low Power High Speed memory generator in HCMOS9,

specification, version 1.2, September 2001, ST Microelectronics Central R&D.
[46] Atmel main page, http://atmel.com
[47] TinyOS main page, http://webs.cs.berkeley.edu/tos/
[48] TinyOS project Homepage. http://sourceforge.net/projects/tinyos/
[49] nescC project Homepage. http://sourceforge.net/projects/nescc/
[50] Hogthrob.dk - Hogthrob, Networked on-a-chip nodes for snow monitoring, Official Home. http://hogthrob.dk
[51] OpenCores.org, http://www.opencores.org/
[52] OpenCores.org - project page for AVR core, http://www.opencores.org/projects/avr_core/
[53] Pictur af Sensor Network on Great Duck Island, The Ultimate on-the-fly Network, Wired Magazine, Issue 11.12, December 2003, http://www.wired.com/wired/archive/11.12/network.html
[54] BTnode Homepage http://www.btnode.ethz.ch/
[55] GNU Operating System - Free Software Foundation http://gnu.org
[56] IO Technologies homepage. http://www.iotech.dk/
[57] Xilinx Home http://www.xilinx.com
[58] Curcuis Multi-Projects, Cell library vender. http://cmp.imag.fr
[59] Spartan3 Documentation homepage. http://www.xilinx.com/spartan3/
[60] Atmel official programming tools. http://atmel.com/dyn/products/tools.asp?family_id=607
[61] Forum of Tools and Tips for programming AVR microprocessor. http://www.avrfreaks.net/

APPENDIX A

Working description of M. Sc. Thesis

Design of a synthesizable asynchronous microcontroller

NR.:	1029
Master's Thesis Project:	
Title:	Design of a synthesizable asynchronous microcontroller
Student:	Andreas Vad Lorentzen
Period:	15.02.2004 - 31.12.2004
Project description:	Målet med projektet er at opbygge en asynkron mikrokontroller til afvikling af et indlejret operativsystem. Mikrokontrolleren tænkes brugt i forbindelse med forskningsprojektet Hogthrob, hvor brugen af en asynkron processor forventes at give en reducering i strømforbrug. Der skal implementeres en asynkron version af ATMEL's AVR mikrokontroller, som skal afvikle simple applikation under TinyOS operativsystemet. Eksamensprojektet er struktureret på følgende måde. Først skal der opsættes et test miljø for en eksisterende synkron AVR mikrokontroller. Den synkrone processor skal syntetiseres til en FPGA. For at teste den synkrone processor skal der implementeres en seriel port, som skal benyttes af GDB til at debugge systemet. Det er meningen at test miljø senere skal bruges til test af asynkrone AVR mikrokontrollerer. Herefter skal en simpel version af den asynkrone AVR processoren implementeres for at undersøge om instruktionssættet virker korrekt. Der lægges særlig vægt på design forløbet for implementering af en asynkron mikrokontroller. Der ønskes benyttet et asynkront design værktøj til design af mikrokontrolleren. Der skal undersøges, hvordan det mest hensigtsmæssigt kan lade sig gøre at få et asynkront design til at kører på en FPGA. Den simple version skal herefter syntetiseres til en FPGA og det kontrolleres at den kører korrekt. Den simple version skal herefter optimeres mht. strømforbrug ved passende pipelining og anden form for teknik. Den optimerede AVR processor skal syntetiseres og sammenlignes med den simple version. Endelig ønskes det at afvikle det indlejrede operativsystem, TinyOS på FPGA'en og køre nogle forskellige programmer. Her skal de forskellige versioner af den asynkrone AVR sammenlignes med en synkron AVR. Supervisor: Jan Madsen og Jens Sparsø

APPENDIX B

Hogthrob board - Hardware overview

Figure B.1 show a overview of the Hogthrob board. The figure is from [21].

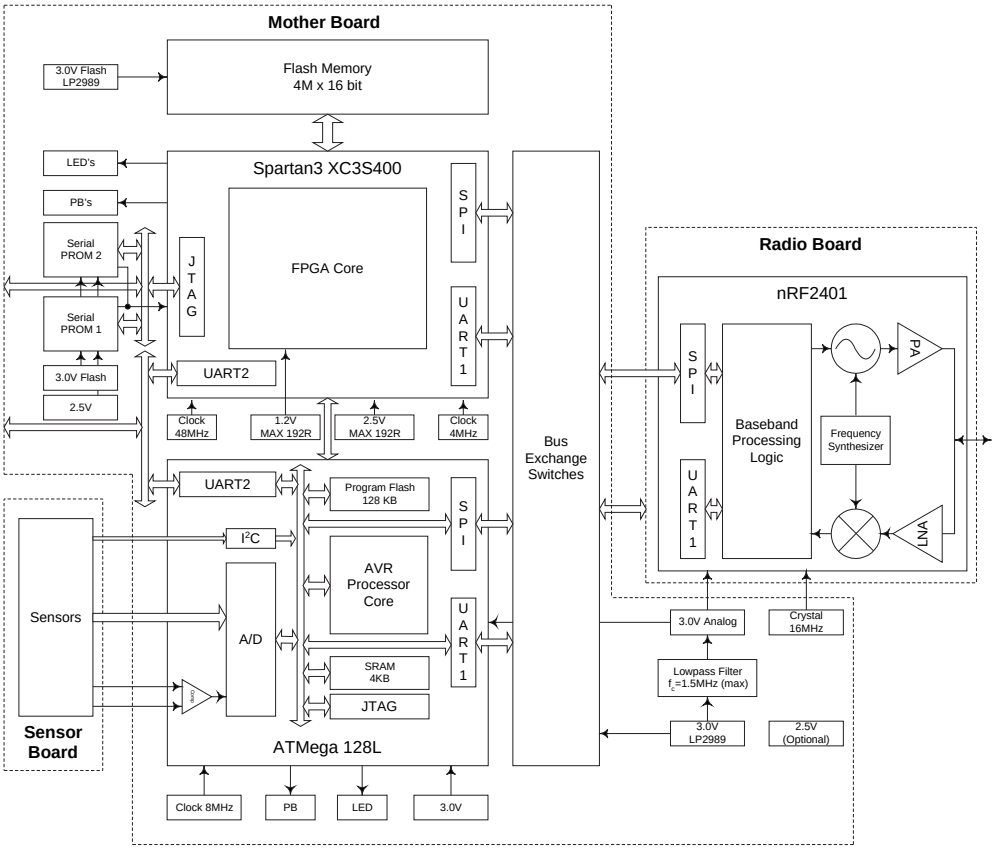


Figure B.1 *Hardware overview of the Hogthrob board*

APPENDIX C

Measurements

This section includes all the measurements from Synopsys Power Report, activity count for the memory entry and current estimation for the ATmega128L.

C.1 Full-adder measurements

Table C.1 and C.2 includes a Synopsys Power Report for a full-adder using the $0.25\mu m$ and $0.12\mu m$ technology libraries. The power estimation have be done using different frequencies. The test bench includes all eight combination for the full-adder.

Frenquency	Cell Internal Power	Net Switching Power	Total Dynamic Power	Cell Leakage Power
$4Mhz$	$0.29779nW$	$0.11662nW$	$0.41441nW$	$0.13559nW$
$8Mhz$	$0.63176nW$	$0.24563nW$	$0.87738nW$	$0.13560nW$
$32Mhz$	$2.63500nW$	$1.01970nW$	$3.65470nW$	$0.13598nW$
$128Mhz$	$10.67670nW$	$4.13150nW$	$14.80820nW$	$0.13562nW$
$512Mhz$	$42.61870nW$	$16.51090nW$	$59.12960nW$	$0.13565nW$
$2048Mhz$	$125.44910nW$	$62.84030nW$	$188.28940nW$	$0.13581nW$

Table C.1 *Power consumption for a full-adder using 0.25 library running different speeds.*

C.2 Nimbus Measurements

This section includes the measurement for the Nimbus test using the two technology library. Table C.3 and C.4 show the power consumption for the VHDL code of the Nimbus microprocessor running the test program at $4MHz$. Synopsys and ModelSim have be used to generate the result.

Frenquency	Cell Internal Power	Net Switching Power	Total Dynamic Power	Cell Leakage Power
4*Mhz*	0.07236*nW*	0.02679*nW*	0.09916*nW*	6.3549*nW*
8*Mhz*	0.15332*nW*	0.57230*nW*	0.21055*nW*	6.3549*nW*
32*Mhz*	0.63914*nW*	0.23986*nW*	0.87900*nW*	6.3549*nW*
128*Mhz*	2.59290*nW*	0.97409*nW*	3.56700*nW*	6.3564*nW*
512*Mhz*	10.36620*nW*	3.89620*nW*	14.26240*nW*	6.3548*nW*
2048*Mhz*	41.50380*nW*	15.59050*nW*	57.09430*nW*	6.3544*nW*

Table C.2 *Power consumption for a full-adder using 0.25 library running different speeds.*

Program	Cell Internal Power	Net Switching Power	Total Dynamic Power	Cell Leakage Power
nop	294.0621μW	157.6521μW	451.7142μW	0.2803μW
idle	2.553μW	1.203μW	3.756μW	0.2798μW
power-save	0.999μW	0.4798μW	1.479μW	0.2794μW
power-down	0.001μW	0.005μW	0.006μW	0.2796μW
add	5.819μW	3.057μW	9.645μW	0.2785μW
add-men	6.198μW	3.958μW	10.156μW	0.2777μW
ham-ming	6.090μW	3.693μW	9.782μW	0.2788μW

Table C.3 *Power estimation of the Nimbus microprocessor using* 0.25μm *library run* 4*MHz.*

Program	Cell Internal Power	Net Switching Power	Total Dynamic Power	Cell Leakage Power
nop	23.8702μW	8.2384μW	32.1086μW	8.1698μW
idle	0.6038μW	0.2010μW	0.8048μW	8.1779μW
power-save	0.5721μW	0.1873μW	0.7594μW	8.1745μW
power-down	0.2511μW	0.0826μW	0.3337μW	8.1867μW
add	1.1103μW	0.4644μW	1.5747μW	8.1905μW
add-men	1.6254μW	0.8478μW	2.4742μW	8.1772μW
ham-ming	1.0211μW	0.3400μW	1.3611μW	8.1860μW

Table C.4 *Power estimation of the Nimbus microprocessor using* 0.12μm *library run* 4*MHz.*

C.3 Nimbus memory access

In order to calculate the power consumption for the Nimbus microprocessor running the test programmes. Table C.5 and C.6 shows the number of ROM reads and RAM writes and reads. The microprocessor is running 4*MHz* when running the test.

P2rogram	nop	idle	power-save	power-down
Time (ns)	1000000	1000000	1000000	1000000
Start instruction reads	160	0	0	0
Start data reads	20	0	0	0
Start data writes	14	0	0	0
End instruction reads	4160	0	0	0
End data reads	783	0	0	0
End data writes	394	0	0	0

Table C.5 *Count of memory access for the Nimbus microprocessor running the test program at 4MHz*

Program	add	add-mem	hamming
Time (ns)	1000000	1000000	1000000
Start instruction reads	160	160	2000
Start data reads	0	22	303
Start data writes	0	22	180
End instruction reads	4160	4160	6000
End data reads	0	780	1288
End data writes	0	781	303

Table C.6 *Count of memory access for the Nimbus microprocessor running the test program at 4MHz (continued)*

C.4 ATmega128L

In the following listings is the test report of current consumption for estimation of the ATmega128L on the BTnode.

```
* Instruktions - benchmarks
add/sub
bit manipulation

--

call/jump

--

* Memory vs. registre
 register adition vs. memmory

* CRC på noget data

* Power modes
 - loop m. nop
 - IDLE
 - POWE Save

* Blink
Viser TOS program, men svært at lave præcise målinger på.


========================
=       Journal        =
========================
Vi sætter BTNoden til AVR-devkittet og måler spændingen 4.89 V -
btnode og programmer bliver _meget varme_.

======== add-mem
12. nov 12:16
add-mem (med 3xFF loops), tænder LED kører i et stykke tid og tænder LED
 * 27.25 mA, 27.35 mA, 27.41 mA, 27.51 mA
 * Ligger stabil omkring 27.3 mA
 * Går i uendelig lykke med 59.9 mA
```

```
======== add
12. nov 12:30
add (med 3xFF loops).. Tænder _ikke_ LED (output pins forstyrrer målinger)
 * 18.65 mA (meget stabilt og varierer til 18.64 mA med 5 s mellemrum)
 * Kører i ca. 3 min og tænder LED

======== add-mem
12. nov 12:35
add-mem (med 3xFF loops)
 * 20.20, 20.17, 20.11, 20.23, 20.28
 * Ligger omkring 20.18 mA
 * Max: 19.57, 20.57

 * Kører igen.
 * Ligger omkring 19.6 mA (dvs. 95.8 mW)
 * Max/min: 19.51, 19.67

 * Flere exp. liggger omkring de. 19.6 mA
 * Køler lidt af og nu ligger vi på 18.38 mA
 * Kølet helt af under frokost 18.89 mA, 10.10 mA


Prøver at bytte programming board (til Martins). Måler spænding til
3.273 V (er sikkert stillet med AVRStudio). Om igen....

======== add-mem
add-mem (med 3xFF loops), kører loop og tænder LED
12. nov 13:42
 * Starter med højt strømforbrug og falder derefter.
 * Ligger omkring 9.75 mA
 * Min/max: 9.65, 9.74
 * Slutter med 14.5

 * Gentager exp.
 * Ligger omkring 9.65 mA (dvs. 31.6 mW)
 * min/max: 9.69
 * Flere gentagelser giver samme resultat

======== add (med 3xFF loops)
 * Starter med højt forbrug og falder derefter til stabilt:
 * Ligger stabilt på 9.18 (og skifter til 9.19 med 5-10 s mellemrum)
 * Min/max. 9.17/9.19
 * Gentagne exp. giver samme res.
```

```
======== power-mode-idle
 * Starter højt og falder øjeblikkeligt
 * 5.20 mA (gentagelser giver samme res.)

======== power-mode-powerDown
 * starter højt og falder øjeblikkeligt
 * 11.9 uA (gentagelser giver samme res.)

======== power-mode-powerSave
 * starter højt og falder øjeblikkeligt
 * 11.8 uA (gentagelser giver samme res.)

======== power-mode-Standby
 * 9.53 mA (gentagelser giver samme res.)

======== power-mode-ExtStandby
 * 0.71 mA (gentagelser giver samme res.)

======== timer_blink
Scale 7 == 1024 == TCCR0, Ticks == 128 == OCR0
Blinker med ca. 1 sekunds interval og driver en pin (driver den lav,
når den er slukket). Går "power-save" mellem interrupts
 * 4.12 mA med LED tændt (stabil)
 * 19.5 mA med LED slukket (stabil)

======== power-mode-Standby
Sun Nov 14. Fejl i bit-mønsteret for valg af standby power-mode.
 * 0.71 mA (gent. giver samme res).

======== encoder_decoder
Sun Nov 14 15:53 Efter fedteri med den spændende "-c" option kører encoder_decod
 * 9.88 mA (omkring) varrierer 9.87 mA, 9.89 mA
```

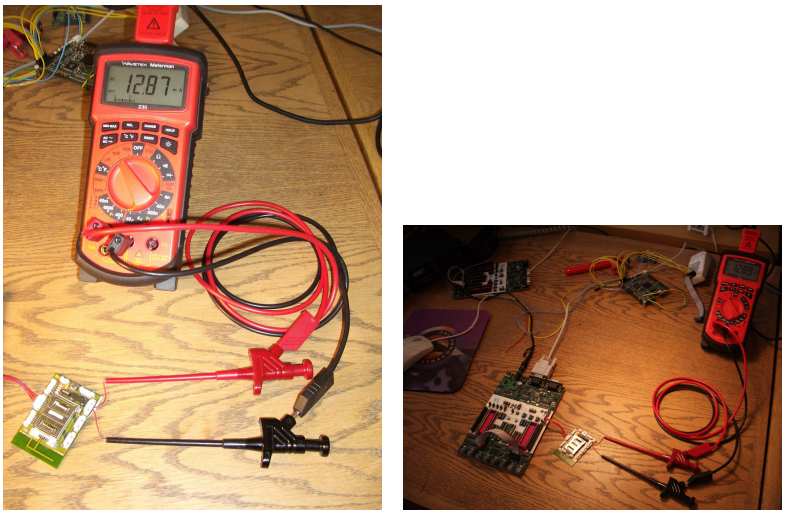
APPENDIX D

Pictures

This appendix includes pictures of the different hardware components used in the project.

D.1 BTnode mote

Figure D.1(a) and D.1(b) show experimental set-up for estimation of power consumption for the BTnode not used in this project. On figure D.1(b) is it also possible to see the programming board, which can be used to program a the ATmega128L on the BTnode or the Hogthrob board.



(a) (b)

Figure D.1 *Power estimation of a BTnode.*

APPENDIX E

Modelsim: Compilation of Xilinx library

- Make directory i.e. `c:\temp_modelsim` and go to the directory.
- Compile the modelsim liberies:
  `compxlib -s mti_se -f all -l all -o c:\Modeltech_5.8b\xilinx_lib`
- Go to the Modelsim installation directory: `c:\Modeltech_5.8b` Open the file: `modelsim.ini`
- Open the file `c:\temp_modelsim\modelsim.ini` and copy the the Libary path from the file to the file in the Modelteck directory.

```
UNISIMS_VER = c:\Modeltech_5.8b\xilinx_lib\unisims_ver
SIMPRIMS_VER = c:\Modeltech_5.8b\xilinx_lib\simprims_ver
XILINXCORELIB_VER = c:\Modeltech_5.8b\xilinx_lib\XilinxCoreLib_ver
AIM_VER = c:\Modeltech_5.8b\xilinx_lib\abel_ver\aim_ver
CPLD_VER = c:\Modeltech_5.8b\xilinx_lib\cpld_ver
UNI9000_VER = c:\Modeltech_5.8b\xilinx_lib\uni9000_ver
UNISIM = c:\Modeltech_5.8b\xilinx_lib\unisim
SIMPRIM = c:\Modeltech_5.8b\xilinx_lib\simprim
XILINXCORELIB = c:\Modeltech_5.8b\xilinx_lib\XilinxCoreLib
AIM = c:\Modeltech_5.8b\xilinx_lib\abel\aim
PLS = c:\Modeltech_5.8b\xilinx_lib\abel\pls
CPLD = c:\Modeltech_5.8b\xilinx_lib\cpld
```

APPENDIX F

Waveforms of Nimbus

This appendix includes waveforms of Nimbus from simulations. There are waveforms from back-annotated simulations using Modelsim and from the Xillinx tool, called Chip-scope.

The reasons for showing many waveforms for the timer blink examples are because they have a clear behaviour. Then the program is TinyOs Blink example and Timer Blink example is turning the led on and off, it is to see it is working.

Normally en a specific program is tested the behaviour is closely monitor. Each instruction has a one shot signal and with these signals is it easy to see if a specific program is behaving correctly.

F.1 Back-annotated simulation

This section contains waveform from Modelsim. All the waveforms are med from back-annotated simulation of the Nimbus, which have be synthesised for ASIC (see 3.5.

F.1.1 Timer blink example

Figure F.1 show a back-annotated simulation of the timer blink example. The interested thing is that it is possible to see the internal clock and external clock is clocking but the core clock and device clock is stop. It is also possible to see which sleep mode the system is in.

F.1.2 TinyOs - Blink

Figure F.2 and F.3 shows the waveforms of back-annotated simulation of the blink example. As shown on figure F.1 is it possible to how the clock in internal in the microprocessor is stopped because the same instruction is there for long period. Figure F.2 shows only the a few signal, the instruction, the program counter, reset and the port, which turns the led of and on. Figure F.3 show more signal like the sleep mode, the core clock, the device clock and sleep modes.

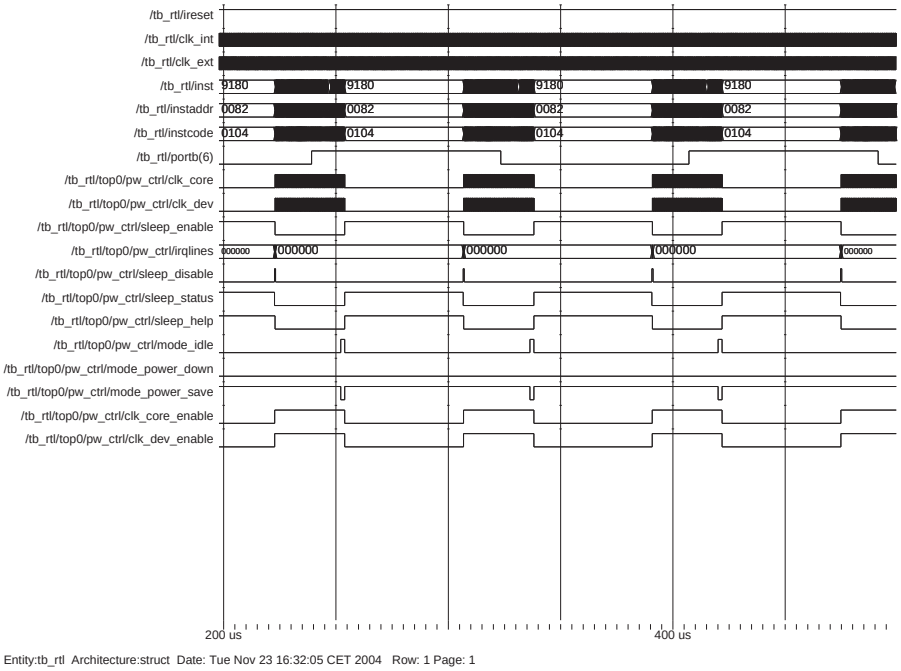


Figure F.1 *Back-annotated simulation of Timer blink example.*

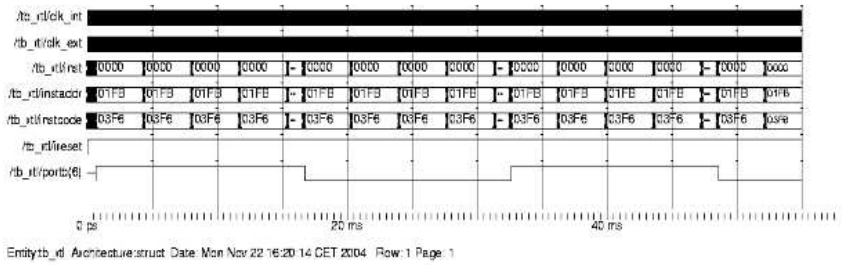


Figure F.2 *Back-annotated simulation of TinyOs blink example.*

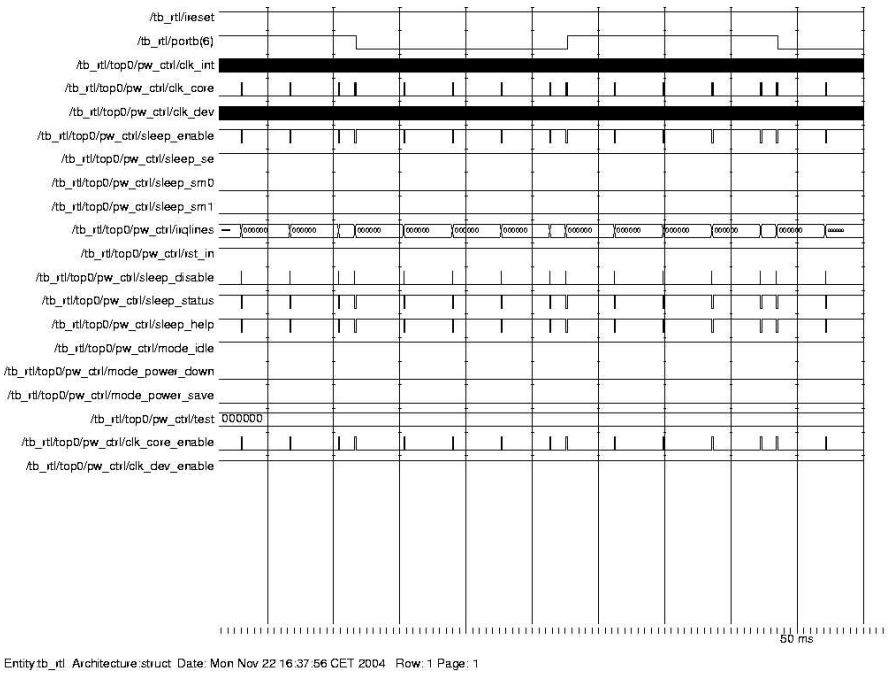


Figure F.3 *Back-annotated simulation of TinyOs blink example.*

F.2 Chip-scope waveforms

This section includes waveform and screen shots of wave from in Chipscope.

F.2.1 Timer blink example

Figure F.4 and F.5 are waveform using chipscope. The figures show the timer blink example running on the Hogthrob FPGA. `pc` is the program counter, `inst` is the instruction and `port` is the led turn on or off. In figure F.4 is the led turned on and figure F.5 is the led turn of.

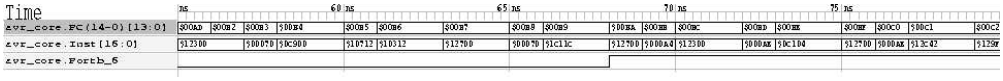


Figure F.4 *Waveform of Timer blink expample.*

F.2.2 TinyOs - Blink

On figure from F.6 to F.11 are waveforms of TinyOs running the Blink in the FPGA on the Hogthrob board. In the waveforms is it possible to see the current program

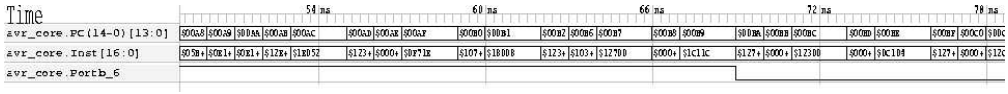


Figure F.5 *Waveform of Timer blink expample.*

counter, the loaded instruction and whether the led is on or off, using port B. Chipscope starts the sampling 100 ns before the interrupt from the timer. The tricker signal is the interrupt goes high.

Figures F.6 and F.9 shows waveforms from then TinyOs turns on and off the led. Figure F.7 and F.10 show a close up of the waveforms, when the interrupt occur and figures F.8 and F.11 show then the led is turn on or off.

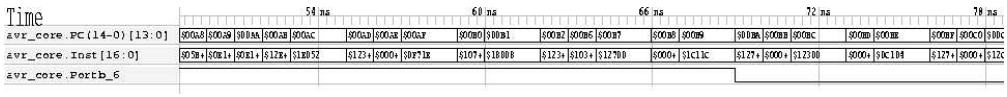


Figure F.6 *Waveform of TinyOs blink example.*

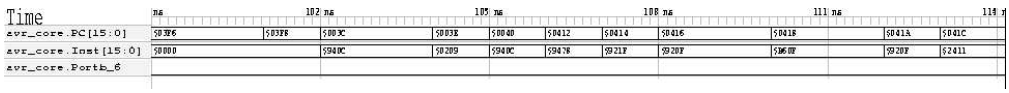


Figure F.7 *Waveform of TinyOs blink example.*

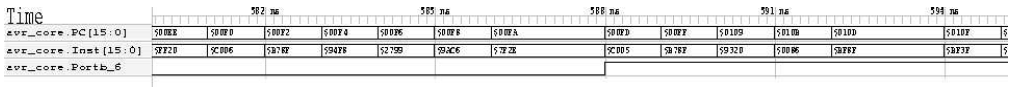


Figure F.8 *Waveform of TinyOs blink example.*

Figure F.9 *Waveform of TinyOs blink example.*

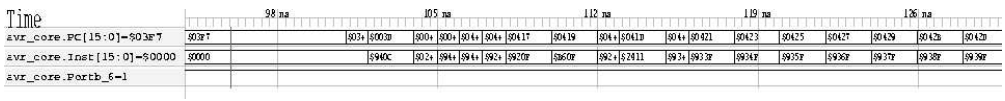


Figure F.10 *Waveform of TinyOs blink example.*

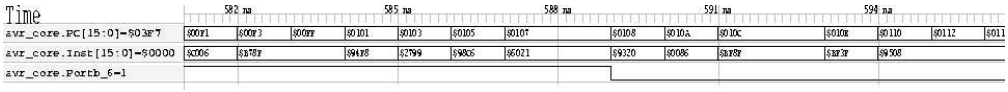


Figure F.11 *Waveform of TinyOs blink example.*

F.2.3 Hamming screen dump

In order to show that the hamming encode and decode was working on the Nimbus microprocessor, chipscope were used. Figure F.12 show a screen dump of the hamming algorithm running on the FPGA using chipscope. In order to get chipscope to sample the activity in the FPGA, the program where set to write a port.

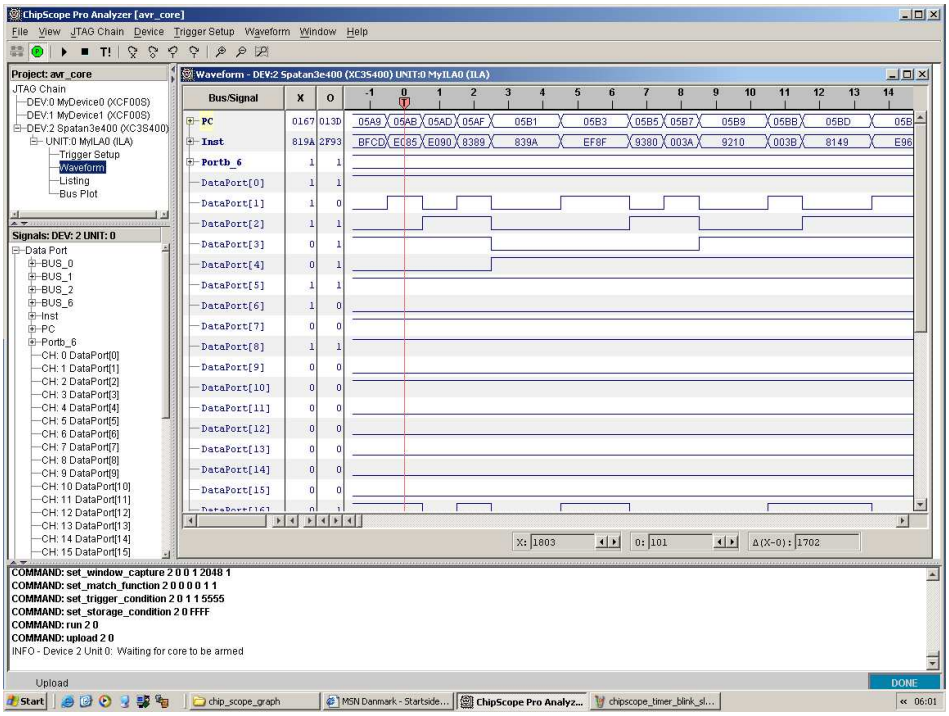


Figure F.12 *Hamming encode and decode rounding on the Hogthrob FPGA*

APPENDIX G

CD-rom

Overview of the CD-ROM:

G.1 General

- *doc:* Articles and Hardware documentation.
- *rapport:* The report.
- *src:* Source code.
- *xilinx:* Xilinx synthesis information

G.2 Documentation

- *doc/ATmega103:* Documentation of the ATmega103
- *doc/ATmega128:* Documentation of the ATmega103
- *doc/articles:* All the articles have been used which are available in a digital format.
- *doc/ram_specification:* Ram module specification.
- *doc/spartan3:* Documentation of the Spartan3.

G.3 Source Code

- *src/asyn_avr_core:* The Disa microprocessor.
- *src/add_syn:* The full-adder.
- *src/asyn_comp:* Asynchronous basic blocks used for de-synchronisation
- *src/avr_core:* The Nimbus microprocessor.
  - *src/avr_core/rom_binary:* The VHDL test programs.
  - *src/avr_core/scr:* Synopsys synthesis script.
- *src/c-asm:* The test programs.

- *src/mini_aram:* De-synchronous design study 1 and 2.
- *src/mini_avr:* De-synchronous design study 3 and 4.
- *src/opencores:* The original source code for microprocessor from opencores.

APPENDIX H

Source

The most of source code can only be found on the CD-ROM or this homepage: `http://vadlorenten`

H.1 Full-adder

The source code can be found on the homepage or on the CD-ROM in the following directory: *src/add_syn* .

H.2 Nimbus Microprocessor

The source code can be found on the homepage or on the CD-ROM in the following directories: *src/avr_core* and *src/avr_core/rom_binary*.

H.3 Asynchronous components

The source code can be found on the homepage or on the CD-ROM in the following directory: *src/asyn_comp*.

H.4 De-synchronous design study

The source code can be found on the homepage or on the CD-ROM in the following directories: *src/mini_ram* and *src/mini_avr*.

H.5 Disa microprocessor

The source code can be found on the homepage or on the CD-ROM in the following directory: *src/asyn_avr_core*.

H.6 Scripts

The source code can be found on the homepage or on the CD-ROM in the following directories: *src/avr_core/scr* and *src/c-asm*.

H.7 C and Assembly programs

The source code can be found on the homepage or on the CD-ROM in the following directory: *src/c-asm*. The source code of some of the programs are avaible here.

H.7.1 power-extstandby

c-asm/power-mode-ExtStandby.c

```
//
// A simple blink application.
//
// avr-gcc -mmcu=atmega128 blink.c -o blink
// avr-objcopy --output-target=srec blink blink.srec
// uisp -v=3 -dprog=stk500 -dserial=/dev/ttyUSB0 -dpart=ATmega128 --erase --upload if=blink.srec

//#define __AVR_ATmega103__ 1
#define __AVR_ATmega128__ 1
#include <avr/io.h>

int main(void) {
  MCUCR |= _BV(SE) // Sleep enable
    | _BV(SM2) | _BV(SM1) | _BV(SM0);

  /*
   SM2   SM1   SM0
   0     0     0   Idle
   0     0     1   ADC Noise reduction
   0     1     0   Power-down
   0     1     1   Power-save
   1     0     0
   1     0     1
   1     0     0   Standby
   1     1     1   Ext. standby

   */
```

```
  asm volatile("sleep");
  asm volatile("nop");
  asm volatile("nop");
  return 0;
}
```

H.7.2 idle

c-asm/power-mode-Idle.c

```
//
// A simple blink application.
//
// avr-gcc -mmcu=atmega128 blink.c -o blink
// avr-objcopy --output-target=srec blink blink.srec
// uisp -v=3 -dprog=stk500 -dserial=/dev/ttyUSB0 -dpart=ATmega128 --erase --upload if=blink.srec

#define __AVR_ATmega103__ 1
//#define __AVR_ATmega128__ 1
#include <avr/io.h>

int main(void) {
  MCUCR |= _BV(SE); // Sleep enable
  //MCUCR &= ~_BV(SM2) & ~_BV(SM1) & ~_BV(SM0); // 128
  MCUCR &= ~_BV(SM1) & ~_BV(SM0); // 103

  /*
   SM2   SM1   SM0
   0     0     0   Idle
   0     0     1   ADC Noise reduction
   0     1     0   Power-down
   0     1     1   Power-save
   1     0     0
   1     0     1
   1     1     0   Standby
   1     1     1   Ext. standby
   */

  asm volatile("sleep");
  asm volatile("nop");
  asm volatile("nop");
  return 0;
}
```

H.7.3 loop

c-asm/power-mode-NOPLoop.c

```
//
// A simple blink application.
//
// avr-gcc -mmcu=atmega128 blink.c -o blink
// avr-objcopy --output-target=srec blink blink.srec
// uisp -v=3 -dprog=stk500 -dserial=/dev/ttyUSB0 -dpart=ATmega128 --erase --upload if=blink.srec

#define __AVR_ATmega103__ 1
```

```
//#define __AVR_ATmega128__ 1
#include <avr/io.h>

int main(void) {
  uint16_t i,j,k,l,m;

  DDRB = _BV(6);
  PORTB = _BV(6);
  for(k=0 ; k<=0xFF ; k++){
    for(j=0 ; j<=0xFF ; j++){
      for(i=0 ; i<=0xFF ; i++){
      asm volatile("nop");
      }}
  }
  DDRB = _BV(7);
  PORTB = _BV(7);

  while(1) {
    asm volatile("nop");
  }
  return 0;
}
```

H.7.4 power-down

c-asm/power-mode-PowerDown.c

```
//
// A simple blink application.
//
// avr-gcc -mmcu=atmega128 blink.c -o blink
// avr-objcopy --output-target=srec blink blink.srec
// uisp -v=3 -dprog=stk500 -dserial=/dev/ttyUSB0 -dpart=ATmega128 --erase --upload if=blink.srec

#define __AVR_ATmega103__ 1
//#define __AVR_ATmega128__ 1
#include <avr/io.h>

int main(void) {
  MCUCR |= _BV(SE) // Sleep enable
    | _BV(SM1);
  //MCUCR &= ~_BV(SM2) & ~_BV(SM0); // 128
  MCUCR &= ~_BV(SM0); // 103

  /*
   SM2   SM1   SM0
   0     0     0   Idle
   0     0     1   ADC Noise reduction
   0     1     0   Power-down
   0     1     1   Power-save
   1     0     0
   1     0     1
   1     0     0   Standby
   1     1     1   Ext. standby

   */

  asm volatile("sleep");
  asm volatile("nop");
```

```
  asm volatile("nop");
  return 0;
}
```

H.7.5 power-save

c-asm/power-mode-PowerSave.c

```
//
// A simple blink application.
//
// avr-gcc -mmcu=atmega128 blink.c -o blink
// avr-objcopy --output-target=srec blink blink.srec
// uisp -v=3 -dprog=stk500 -dserial=/dev/ttyUSB0 -dpart=ATmega128 --erase --upload if=blink.srec

#define __AVR_ATmega103__ 1
//#define __AVR_ATmega128__ 1
#include <avr/io.h>

int main(void) {
  MCUCR |= _BV(SE) // Sleep enable
    | _BV(SM1) | _BV(SM0);

  /*
   SM2   SM1   SM0
   0     0     0   Idle
   0     0     1   ADC Noise reduction
   0     1     0   Power-down
   0     1     1   Power-save
   1     0     0
   1     0     1
   1     0     0   Standby
   1     1     1   Ext. standby

   */

  asm volatile("sleep");
  asm volatile("nop");
  asm volatile("nop");
  return 0;
}
```

H.7.6 power-standby

c-asm/power-mode-Standby.c

```
//
// A simple blink application.
//
// avr-gcc -mmcu=atmega128 blink.c -o blink
// avr-objcopy --output-target=srec blink blink.srec
// uisp -v=3 -dprog=stk500 -dserial=/dev/ttyUSB0 -dpart=ATmega128 --erase --upload if=blink.srec

//#define __AVR_ATmega103__ 1
#define __AVR_ATmega128__ 1
#include <avr/io.h>
```

```
int main(void) {
  MCUCR |= _BV(SE) | _BV(SM2) | _BV(SM1); // Sleep enable
  MCUCR &= ~_BV(SM0);

  /*
   SM2   SM1   SM0
   0     0     0   Idle
   0     0     1   ADC Noise reduction
   0     1     0   Power-down
   0     1     1   Power-save
   1     0     0
   1     0     1
   1     1     0   Standby
   1     1     1   Ext. standby

   */

  asm volatile("sleep");
  asm volatile("nop");
  asm volatile("nop");
  return 0;
}
```

H.7.7 add-mem

c-asm/add-mem.c

```
// avr-gcc add-mem.c -o
// avr-objcopy --output-target=srec tester2 tester2.srec
// uisp -v=3 -dprog=stk500 -dserial=/dev/ttyUSB0 -dpart=ATmega128 --erase --upload if=tester2.srec

//#define __AVR_ATmega128__ 1
#define __AVR_ATmega103__ 1

#include <inttypes.h>

#include <avr/io.h>

int main(void) {
  uint8_t register aa, bb, cc, dd;
  uint8_t ee;
  uint8_t *ptrr;

  // Tænder LED, men driver et output ben, som bruger strøm
  //DDRB |= _BV(6);
  //PORTB |= _BV(6);

  //register i, j;
  uint8_t ii, jj, kk;
  ee = 0x1334;
  ptrr = &ee;
  asm volatile ("       ldi  %[one], 0x01   \n\t"    /* load constant */
                "       mov  r26, %[ptr]    \n\t"
                "       ldi  %[k], 0xFF     \n\t"    /* outer loop counter */
                "loop3: ldi  %[j], 0xFF     \n\t"    /* outer loop counter */
                "loop2: ldi  %[i], 0xFF     \n\t"     /* inner loop counter */
                "loop:  ld   %[res], X \n\t"     /* fetch stored value */
                "       add  %[res], %[i] \n\t"     /* add something */
                "       st   X, %[res] \n\t"     /* store value */
```

```
                "       ld    %[res], X \n\t"      /* fetch stored value */
                "       add   %[res], %[i] \n\t"     /* add something */
                "       st    X, %[res] \n\t"      /* store value */
                "       ld    %[res], X \n\t"      /* fetch stored value */
                "       add   %[res], %[i] \n\t"     /* add something */
                "       st    X, %[res] \n\t"      /* store value */
                "       ld    %[res], X \n\t"      /* fetch stored value */
                "       add   %[res], %[i] \n\t"     /* add something */
                "       st    X, %[res] \n\t"      /* store value */
                "       ld    %[res], X \n\t"      /* fetch stored value */
                "       add   %[res], %[i] \n\t"     /* add something */
                "       st    X, %[res] \n\t"      /* store value */
                "       ld    %[res], X \n\t"      /* fetch stored value */
                "       add   %[res], %[i] \n\t"     /* add something */
                "       st    X, %[res] \n\t"      /* store value */
                "       ld    %[res], X \n\t"      /* fetch stored value */
                "       add   %[res], %[i] \n\t"     /* add something */
                "       st    X, %[res] \n\t"      /* store value */
                "       ld    %[res], X \n\t"      /* fetch stored value */
                "       add   %[res], %[i] \n\t"     /* add something */
                "       st    X, %[res] \n\t"      /* store value */
                "       ld    %[res], X \n\t"      /* fetch stored value */
                "       add   %[res], %[i] \n\t"     /* add something */
                "       st    X, %[res] \n\t"      /* store value */
                "       ld    %[res], X \n\t"      /* fetch stored value */
                "       add   %[res], %[i] \n\t"     /* add something */
                "       st    X, %[res] \n\t"      /* store value */
                "       ld    %[res], X \n\t"      /* fetch stored value */
                "       add   %[res], %[i] \n\t"     /* add something */
                "       st    X, %[res] \n\t"      /* store value */

                "       sub   %[i],   %[one] \n\t"      /* decriment loop counter */
                "       brne  loop           \n\t"
                "       sub   %[j],   %[one] \n\t"      /* decriment loop counter */
                "       brne  loop2           \n\t"
                "       sub   %[k],   %[one] \n\t"      /* decriment loop counter */
                "       brne  loop3          \n\t"
                : [res] "=r"(ii),
                  [one] "=r"(aa),
                  [i]   "=r"(cc),
                  [j]   "=r"(dd),
                  [k]   "=r"(kk)
                : [ptr] "r"(ptrr)
                );

  DDRA = 0xFF;
  PORTA= 0x2; // Marker for simulation

  DDRB |= _BV(7);
  PORTB |= _BV(7);

  while(1) {
    asm volatile ("nop" ::);
  }

  return 0;
}
```

H.7.8 add

c-asm/add.c

```
// avr-gcc tester2.c -o tester2
// avr-objcopy --output-target=srec tester2 tester2.srec
// uisp -v=3 -dprog=stk500 -dserial=/dev/ttyUSB0 -dpart=ATmega128 --erase --upload if=tester2.srec
//#define __AVR_ATmega128__ 1
#define __AVR_ATmega103__ 1

#include <inttypes.h>

#include <avr/io.h>

int main(void) {
  uint8_t a, b, c, d;
  register i, j, k;

  // DDRA = 0x00; // Data direction defaults to input

  //asm volatile ("clr r15" : /*Inputs*/: /*output*/: "r15");
  //asm volatile ("ldi r16, 1" ::);
  //for(i=0 ; i<1000 ; i++) {
  //  asm volatile ("add r15, r16" ::);
  //}

  //asm volatile ("clr %0" : "=r" (a):);
  //asm volatile ("ldi %0, 1" : "=r" (b) :);
  //for(i=0 ; i<=10 ; i++) {
  //  asm volatile ("add %0, %1" : "=r"(a): "r"(b) );
  //}

  asm volatile ("       ldi %[one], 1\n\t"         /* load constant */
                "       ldi %[k], 0xFF\n\t"         /* outer loop counter */
                "loop3: ldi %[j], 0xFF\n\t"         /* outer loop counter */
                "loop2: ldi %[i], 0xFF\n\t"   /* inner loop counter */
                "loop:  add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
                "       add  %[res], %[one]\n\t"   /* add something */
```

```
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       add   %[res], %[one]\n\t"   /* add something */
                "       sub   %[i], %[one]\n\t"     /* decriment loop counter */
                "       brne  loop\n\t"
                "       sub   %[j], %[one]\n\t"     /* decriment loop counter */
                "       brne  loop2\n\t"
                "       sub   %[k],   %[one] \n\t"     /* decriment loop counter */
                "       brne  loop3          \n\t"

                : [res] "=r" (i),
                  [one] "=r" (a),
                  [i]   "=r" (c),
                  [j]   "=r" (d),
                  [k]   "=r" (k)
                : );

  DDRB |= _BV(7);
  PORTB |= _BV(7);

  DDRA = 0xFF;
  PORTA= 0x2; // Marker for simulation

  return 0;
}
```

H.7.9 hamming

c-asm/encoder_decoder.c

```
#include <string.h>
#include <elf.h>


#define PC

// PC Version
```

```
#ifdef PC
#include <stdio.h>
#endif

//codeblock*sizeof(uint16_t) =  126
#define CODESIZE 126

//datablock*sizeof(uint16_t) =  114
#define DATASIZE 114

#define SIZE 456
char data[SIZE] = "\nFrom \"The Cathedral and the Bazaar\"\n\n" \
        "Linux is subversive. Who would have thought even five years ago (1991)\n" \
        "that a world-class operating system could coalesce as if by magic out\n" \
        "of part-time hacking by several thousand developers scattered all over\n" \
        "the planet, connected only by the tenuous strands of the Internet?\n\n"\
        "\t\t  -- ESR\n";


char edata[480]; // Temp data
char rdata[SIZE]; // result


/* The encodeH takes an in-block of words (a row of bits for a bitstream
 * in row major order) and produces a corresponding out-block with N
 * Hamming control words inserted for correction of single bit errors.
 * A total of N control words are inserted, so the input block must be
 * 2**N -1 - N words long.
 */
void encodeH(uint16_t in [], uint16_t out[],uint8_t N){
  uint16_t
    j=0,    /* The C index of the next code word to be inserted    */
    i,      /* The C index of the output word to be filled         */
    k,      /* The value of the last data word inserted in out     */
    q,      /* The C index of a code word                          */
    p=0;    /* Count of code words used                            */

  for (i=0; p<=N; i++)
    if (i==j) { out[i] = 0; j += j+1 ; p++; } /* Add new code word */
    else {  /* Find contributions to previous code words          */
      q = 0;
      k = out[i] = in[i-p];                  /* A data word        */
      while (q<=j) {
        if ((q+1) & (i+1)) out[q] ^= k;      /* Renew control word */
        q += q+1;
      }
    }
}


void encodeAndCopy(uint8_t codeblock, uint8_t datablock, uint8_t N){

  uint16_t inbuffer[datablock],      // data to be read and encoded
    temp[datablock],
    outbuffer[codeblock];     // encoded data to be written

  char *dataptr = &data[0];
  char *edataptr = &edata[0];     // pointer to save output
  uint16_t i,size = 4;

  for( i=0; i<size; i++){
    memcpy((char *) inbuffer,dataptr,DATASIZE);
```

```
    dataptr += DATASIZE;
    encodeH(inbuffer,outbuffer,N);
    memcpy(edataptr, (char *) outbuffer,CODESIZE);
    edataptr += CODESIZE;
#ifdef PC
    printf("%s",outbuffer);
#endif

 }
#ifdef PC
    printf("\nSizedata : %i\n",dataptr-data);
#endif
}


void decodeH(uint16_t in[], uint16_t out[],int N){
  uint16_t
    j=0,    /* The C index of the next code word                  */
    i,      /* Index into the block of code words                 */
    c[N+1], /* The table of control words                         */
    p=0,    /* The C index of first free control word, j=2^p-1    */
    q,      /* Index into the table of computed control words     */
    bit,    /* Bit position in a code word of a possible error    */
    marks=0 /* Bit positions to indicate presense of errors       */
    ;

  for (i=0; p<=N; i++)                 /* Find control words 0..p  */
    if (i == j) { c[p] = in[i]; p++; j += j+1; } else
      for (bit=1,q=0;q<p; bit<<=1,q++)
        if (bit & (i+1)) c[q]^=in[i];

  /* Check for presense of errors and find their bit positions     */
  for (q=0; q<N; q++) marks |= c[q];

  if (marks) { /* Correct errors in bit positions of marks        */
    for (bit=1; marks; marks>>=1,bit<<=1)
      if (marks & 1) { /* On error: find its index position       */
        i=0;
        for (q=N; q; q--) if (c[q-1] & bit) i += i+1; else i += i;
        in[i-1] ^= bit;  /* Correct the bit in the received word  */
      }
  }

  /* Move data from in to out                                      */
  for (j=p=i=0; p<=N; i++)
    if (i==j) {j += j+1; p++;} else out[i-p] = in[i];
}


void readAndDecode(uint16_t codeblock, uint16_t datablock, uint16_t N){

  uint16_t inbuffer[codeblock];      // code to be decode
  uint16_t outbuffer[datablock];     // data

  char *edataptr = &edata[0];      // pointer to save output
  char *rdataptr = &rdata[0];
  uint16_t i,size = 4;

  for(i=0; i<size; i++){
    memcpy((char *) inbuffer, edataptr, CODESIZE);
    edataptr += CODESIZE;
    decodeH(inbuffer,outbuffer,N);
```

```
    strncpy(rdataptr, (char *) outbuffer, DATASIZE);
    rdataptr += DATASIZE;
#ifdef PC
    printf("%s",outbuffer);
#endif
  }
}


int main(void) {

  uint16_t
    N,                  // The number of auxiliary bit in a Hamming code.
    codeblock,          // The length of an encoded block
    datablock;          // The length of a block of data

  N = 6;

  codeblock = (1<<N) - 1;
  datablock = codeblock - N;
  encodeAndCopy(codeblock,datablock, N);
  readAndDecode(codeblock,datablock,N);

  if(0==strncmp(data,rdata,SIZE)){
#ifdef PC
    printf("Succes\n");
#else
    while(1);
#endif
  } else{
#ifdef Pc
    printf("Error\n");
#else
    while(1);
#endif
  }
  // make compare

  return 0;
}
```

H.7.10 Timer Blink

c-asm/timer_blink.c

```
//
// A simple blink application.
//
// avr-gcc -mmcu=atmega128 blink.c -o blink
// avr-objcopy --output-target=srec blink blink.srec
// uisp -v=3 -dprog=stk500 -dserial=/dev/ttyUSB0 -dpart=ATmega128 --erase --upload if=blink.srec

#define __AVR_ATmega103__ 1
#include <avr/io.h>

int main(void) {
  // Turn on MB LED
  DDRB = _BV(6);
  PORTB = _BV(6);
```

```
  sbi(ASSR, AS0); // Enable async timer clock
  outp(0, TCNT0); // Reset the timer0 counter
  outp(7, TCCR0); // Scale the timer by 1024
  //outp(2, TCCR0); // Scale the timer by 32

  // Wait for the async clock to get going
  while (ASSR & 0x07);

  outp(128, OCR0); // Set the timer output compare to trigger at 128
                   // ticks.
  sbi(TIMSK, OCIE0); // Enable the Output Compare interrupt
  asm volatile("sei"); // Enable interrupts

  while (1) {
    // Set the MCUCR so that we enter power-save mode (which will
    // leave the async clock going).
    MCUCR &= ~0x3C;
    MCUCR |= _BV(SE) | _BV(SM1) | _BV(SM0);
    asm volatile("sleep");
    asm volatile("nop");
    while (ASSR & 0x07); // Wait for the async clock to get going
                         // again.
  }

  return 0;
}

//int valueB = 0;
void __attribute((signal)) SIG_OUTPUT_COMPARE0()
{

  cbi(TIMSK, OCIE0); // Disable the Output Compare Interrupt
  //PORTB = ~ _BV(6);

/*    if (valueB = 1){ */
/*      PORTB = ~ _BV(6); */
/*      valueB = 0; */
/*    }else{ */
/*      PORTB = _BV(6); */
/*      valueB = 1; */
/*    } */
  // Toggle the mb-led
  PORTB = PORTB & _BV(6) ? PORTB & ~_BV(6) : PORTB | _BV(6);
  outp(14, TCNT0);     // Reset the timer counter
  sbi(TIMSK, OCIE0); // Enable the Output Compare Interrupt
}
```

H.7.11 TinyOS Blink

c-asm/tinyos/blinkM.nc

```
// $Id: BlinkM.nc,v 1.1 2004/11/11 13:44:08 avl Exp $

/*                                                                  tab:4
 * "Copyright (c) 2000-2003 The Regents of the University  of California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software and its
```

```
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice, the following
 * two paragraphs and the author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT
 * OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF
 * CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
 * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Copyright (c) 2002-2003 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704.  Attention:  Intel License Inquiry.
 */

/**
 * Implementation for Blink application.  Toggle the red LED when a
 * Timer fires.
 **/
module BlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
  }
}
implementation {

  /**
   * Initialize the component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }


  /**
   * Start things up.  This just sets the rate for the clock component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    // Start a repeating timer that fires every 1000ms
    //return call Timer.start(TIMER_REPEAT, 1000);
    return call Timer.start(TIMER_REPEAT, 20);
  }

  /**
```

```
    * Halt execution of the application.
    * This just disables the clock component.
    *
    * @return Always returns <code>SUCCESS</code>
    **/
  command result_t StdControl.stop() {
    return call Timer.stop();
  }


  /**
    * Toggle the red LED in response to the <code>Timer.fired</code> event.
    *
    * @return Always returns <code>SUCCESS</code>
    **/
  event result_t Timer.fired()
  {
    call Leds.redToggle();
    return SUCCESS;
  }
}
```

c-asm/tinyos/blink.nc

```
// $Id: Blink.nc,v 1.1 2004/11/11 13:44:08 avl Exp $

/*                                                                      tab:4
 * "Copyright (c) 2000-2003 The Regents of the University  of California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice, the following
 * two paragraphs and the author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT
 * OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF
 * CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
 * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Copyright (c) 2002-2003 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704.  Attention:  Intel License Inquiry.
 */
/**
 * Blink is a basic application that toggles the leds on the mote
 * on every clock interrupt.  The clock interrupt is scheduled to
 * occur every second.  The initialization of the clock can be seen
 * in the Blink initialization function, StdControl.start().<p>
 *
 * @author tinyos-help@millennium.berkeley.edu
```

```
 **/
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

H.7.12 vhdl2init-ext2

c-asm/vhdl2init-ext2.cpp

```
//
// $Id: vhdl2init-ext2.cpp,v 1.9 2004/12/01 23:39:12 avl Exp $
//
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <fcntl.h>
using namespace std;

FILE *startwrite;   // output file
FILE *fdwrite;
char *memoryname; // input filename

int size; //codesize
unsigned char *code;


int  binarycopy();
int createfile();
int addmemory();
int closefile();

#define RS_BIT 100000
#define RS_BYTE RS_BIT/8
#define RS_LINE RS_BIT/16


int main(int argc, char* argv[])
{
  memoryname = argv[1];

  binarycopy();
  createfile();
  addmemory();
  closefile();

  return 0;
}

int binarycopy(){

  //unsigned int codepos = 0;
  unsigned char *buf;
```

```
  // Set op array temp array    (stort nok)
  buf = (unsigned char*)malloc(100000);

  // op file
  FILE *fid = fopen(memoryname, "r");
  size = fread(buf, 1, RS_BYTE, fid);
  //codepos = size;


  // create instruction array
  code = (unsigned char*)malloc(size);

  printf("Mkram v. 1.2\n");
  printf("Code size: %d\n",size);

  // Set all 0's...
  for (int i = 0; i<size; i++)
    code[i] = 0;

  printf("Moving Data\n");

  // Fill'er up

  for(int i=0; i<size;){ // 2000 lines * 16 bit = 32 kb = 4 kByte
                code[i] = buf[i+1];
                code[i+1] = buf[i];
                //code[i] = buf[i];
                //code[i+1] = buf[i+1];
                i=i+2;
  }

  free(buf);

}

int addmemory(){

  // Write all the bytes as init-strings
  //for (int i = 0; i<RS_BYTE;){
  for (int i = 0; i<size;){
    fprintf(fdwrite,"     ");
    for(int l=0; l<8 and i<size; l++){ // 8 instructions per linie
      fprintf(fdwrite,"X\"%02X", code[i++]);
      fprintf(fdwrite,"%02X\"",  code[i++]);
      if(i<size-1)
        fprintf(fdwrite,", ");
    }
    fprintf(fdwrite,"\n");
  }
  fprintf(fdwrite,"  );\n");

  free(code);

  return 0;
}

#define NAMESIZE 120
int createfile(){

  printf("Create file\n");
  char writename[NAMESIZE];
```

```
  char *wn = &writename[0];
  int length = strlen(memoryname);
  if(length >= NAMESIZE-4){
    printf("Buffer to small\n");
    exit(0);
  }

        sprintf(wn,"%s",memoryname);
        wn = &wn[length-3];
        sprintf(wn,"vhd");

  printf("Name: %s\n",writename);

  fdwrite = startwrite = fopen(writename,"w");

  fprintf(fdwrite,"LIBRARY IEEE;\n");
  fprintf(fdwrite,"USE IEEE.STD_LOGIC_1164.ALL;\n");
  fprintf(fdwrite,"USE IEEE.STD_LOGIC_ARITH.ALL;\n");
  fprintf(fdwrite,"USE IEEE.STD_LOGIC_MISC.ALL;\n");
        fprintf(fdwrite,"USE IEEE.STD_LOGIC_UNSIGNED.ALL;\n");
  fprintf(fdwrite,"\n");
//    fprintf(fdwrite,"LIBRARY UNISIM;\n");
//    fprintf(fdwrite,"USE UNISIM.All;\n");
//    fprintf(fdwrite,"USE UNISIM.VPKG.all;\n");
  fprintf(fdwrite,"\n");
  fprintf(fdwrite,"entity ram16bit is\n");
  fprintf(fdwrite,"port(\n");
  fprintf(fdwrite,"     clk:             in STD_LOGIC;\n");
  fprintf(fdwrite,"     rst:           in STD_LOGIC;\n");
//    fprintf(fdwrite,"     memrw:           in STD_LOGIC;\n");
  fprintf(fdwrite,"     memaddr:         in std_logic_vector (15 DOWNTO 0);\n");
  fprintf(fdwrite,"     memreq:          in STD_LOGIC;\n");
//    fprintf(fdwrite,"     memdatawrite:    in std_logic_vector (15 DOWNTO 0);\n");
  fprintf(fdwrite,"     memdataread:     out std_logic_vector (15 DOWNTO 0));\n");
  fprintf(fdwrite,"end ram16bit;\n");
  fprintf(fdwrite,"\n");
  fprintf(fdwrite,"architecture rtl of ram16bit is\n");
  //  fprintf(fdwrite,"  type rom_type is array (0 to %i) of std_logic_vector (15 downto 0);\n",RS_LINE-1);
  fprintf(fdwrite,"  type rom_type is array (0 to %i) of std_logic_vector (15 downto 0);\n",(size/2)-1);
  fprintf(fdwrite,"  constant ROM : rom_type := (\n");



}


int closefile(){

  fprintf(fdwrite,"  signal clk_i : std_logic;\n");
  fprintf(fdwrite,"  signal countread   : std_logic_vector(15 downto 0) := (others => '0');\n");
  fprintf(fdwrite,"begin\n");
  fprintf(fdwrite,"\n");
  fprintf(fdwrite,"  clk_i <= not clk; -- reading un down clk signal\n");
  fprintf(fdwrite,"\n");
  fprintf(fdwrite,"  process (clk_i)\n");
  fprintf(fdwrite,"  begin\n");
  fprintf(fdwrite,"    if (clk_i'event and clk_i = '1') then\n");
  fprintf(fdwrite,"      if (memreq = '1') then\n");
  fprintf(fdwrite,"        countread <= unsigned(countread)+1;\n");
  fprintf(fdwrite,"        memdataread <= ROM(conv_integer(memaddr));\n");
  fprintf(fdwrite,"      end if;\n");
  fprintf(fdwrite,"    end if;\n");
```

```
  fprintf(fdwrite,"  end process;\n");
  fprintf(fdwrite,"\n");
  fprintf(fdwrite,"end rtl;\n");

  fclose(startwrite);

}
```

H.7.13 Makefile

c-asm/Makefile

```
#
# Original script by Lars Munch Christensen
# Modified to be used with AVR platform
# $Id: Makefile,v 1.13 2004/12/01 23:39:12 avl Exp $
#

# ******************************************
# Programs to build
# ******************************************

PROGS = power-mode-NOPLoop \
        power-mode-Idle \
        power-mode-PowerSave \
        power-mode-PowerDown \
        add \
        add-mem \
        timer_blink_sleep_all \
        timer_blink_sleep_core \
        timer_blink_nop \
        ret_test \
        sub_test \
        push_pop_test \
        timer_blink \
        tinyos_blink \
        simple_test1_s \
        pin_test \
        test1

# ******************************************
# Bootstrap object file
# ******************************************

BOOTSTRAP = crt0.o
BS_TIMER = crt1.o

# ******************************************
# Compiler toolchain
# ******************************************

#CC         = avr-gcc
CC         = avr-gcc -mmcu=atmega103
#CC         = avr-gcc -g -mmcu=atmega128
LD         = avr-ld
OBJCOPY    = avr-objcopy
OBJDUMP    = avr-objdump
CONVERT    = vhdl2init-ext2
```

```
# ********************************************
# Compiler and linker options
# ********************************************

LD_SCRIPT = link.xn
LD_OPTS   = -G 0 -static -T $(LD_SCRIPT)
CC_OPTS = -c

# ********************************************
# Rules
# ********************************************

%.o : %.c
	$(CC) $(W_OPTS) $(CC_OPTS) -o $@ $<

%.o : %.S
	$(CC) $(W_OPTS_A) $(CC_OPTS) -o $@ $<

%.o : %.s
	$(CC) $(W_OPTS_A) $(CC_OPTS_A) -o $@ $<

% : %.c
	$(CC) $(W_OPTS) -o $@ $<

%.srec   : %
	avr-objcopy --output-target=srec $< $@

%.install : %.srec
	uisp -v=3 -dprog=stk500 -dserial=/dev/ttyUSB0 -dpart=ATmega128 --erase --upload if=$<

all: $(PROGS)

$(CONVERT): $(CONVERT).cpp
	c++ -O3 -o $(CONVERT) $(CONVERT).cpp

#encoder_decoder: $(CONVERT) encoder_decoder.c
#	$(CC) $@.c -o $@
#	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
#	./$(CONVERT) ram16bit_$@.bin
#	mv ram16bit_$@.bin binary
#	mv ram16bit_$@.vhd ../avr_core/rom_binary/


power-mode-NOPLoop: $(CONVERT) power-mode-NOPLoop.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/

power-mode-Idle: $(CONVERT) power-mode-Idle.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/

power-mode-PowerSave: $(CONVERT) power-mode-PowerSave.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
```

```
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/

power-mode-PowerDown: $(CONVERT) power-mode-PowerDown.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/



encoder_decoder1: $(CONVERT) encoder_decoder1.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/

encoder_decoder2: $(CONVERT) encoder_decoder2.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/


add-mem: $(CONVERT) add-mem.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/

add: $(CONVERT) add.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/

timer_blink_sleep_all: $(CONVERT) timer_blink_sleep_all.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/

timer_blink_sleep_core: $(CONVERT) timer_blink_sleep_core.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/

timer_blink_nop: $(CONVERT) timer_blink_nop.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/
```

```
ret_test: $(CONVERT) ret_test.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/

sub_test: $(CONVERT) sub_test.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/

push_pop_test: $(CONVERT) push_pop_test.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/

timer_blink: $(CONVERT) timer_blink.c
	$(CC) $@.c -o $@
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/


timer_test: $(CONVERT) $(BS_TIMER) timer_test.o
	$(LD) $(LD_OPTS) -o $@ $@.o $(BS_TIMER)
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/

tinyos_blink: $(CONVERT)
	./run_tinyos.sh
	$(OBJCOPY) -O binary  ram16bit_$@.exe ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/


simple_test1_s : $(CONVERT) $(BOOTSTRAP) simple_test1_s.o
	$(LD) $(LD_OPTS) -o $@ $@.o $(BOOTSTRAP)
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/


pin_test : $(CONVERT) $(BOOTSTRAP) pin_test.o
	$(LD) $(LD_OPTS) -o $@ $@.o $(BOOTSTRAP)
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/

test1 : $(CONVERT) $(BOOTSTRAP) test1.o
	$(LD) $(LD_OPTS) -o $@ $@.o $(BOOTSTRAP)
	$(OBJCOPY) -O binary $@ ram16bit_$@.bin
```

```
	./$(CONVERT) ram16bit_$@.bin
	mv ram16bit_$@.bin binary
	mv ram16bit_$@.vhd ../avr_core/rom_binary/


clean :
	rm -f $(PROGS) *.bin *.o
	rm -f *~
```