## Abstract

This report gives a short introduction of the Norwegian wireless electronics company Chipcon AS, and goes on to account for the state of the art of small IP processor cores. It then describes the NanoRisc, a powerful processor developed in this project to replace hardware logic modules in future Chipcon designs. The architecture and a VHDL implementation of the NanoRisc is described and discussed, as well as an assembler and instruction set simulator developed for the NanoRisc. The results of this development work are promising; synthesis shows that the NanoRisc is capable of powerful 16-bit data moving and processing at 50 MHz in an 18nm process while requiring less than 4500 gates. The report concludes that the NanoRisc, and none of the existing IP cores studied, satisfies the requirements for hardware logic replacement in Chipcon transceivers.

# Preface

This report and the NanoRisc development work was made as part of a master's thesis for the Norwegian University of Science and Technology (NTNU). The project proposal was given by Chipcon AS, and the work has been performed in their offices.

## *Acknowledgements*

I would like to thank my mentors Jørgen Langfeldt, Robin Osa Hoel and Dag-Sverre Skjelbreid at Chipcon for their many hours of help and great inspiration.

I would like to thank Chipcon AS for supporting me and providing me with a desk, computer and all necessary tools.

I would like to thank Lasse Natvig and Morten Hartmann of NTNU for helpful guidance in writing this report.

## *Outline*

This report consists of three parts. Part I is an introduction to the project and accounts for the state of the art. An introduction to Chipcon and the motivation behind this project is given in chapter 2 followed by the requirements specification in chapter 3. An overview and discussion of available IP cores is given in chapter 4. Part I describes the NanoRisc and its tools. An overview of the NanoRisc architecture is given in chapter 5 followed by a description of its implementation in chapter 6. Chapter 7 describes the tools developed for NanoRisc. Part III contains the results and discussion. Chapter 8 describes the testing performed on the NanoRisc and chapter 9 describes the results achieved in synthesis. Chapter 10 is a discussion on some of the more interesting trade-offs made during the development of the NanoRisc.

## Table of Contents

## Table of Figures

## Table of Tables

# Part I

# 1 Introduction

This chapter gives a short introduction to the project. Chipcon is a Norwegian company who designs low-cost, low-power wireless transceivers. The increasing complexity of the chips designed by Chipcon reveals a need for new design methods on their systems on chip (SoCs). One way to achieve this is through the use of an on-chip firmware processor. Contrary to the processors integrated in the Chipcon designs of today, an on-chip firmware processor would not be user programmable, but handle only internal control and data processing tasks.

The NanoRisc processor was developed for Chipcon as a part of this thesis. It is a compact and effective microcontroller core which can control complex processes and move and process data. It features 13 general 16-bit registers, a full 16-bit ALU, an 8x8 multiplier, a 16-bit barrel-shifter, and a load/store unit with auto-increment/decrement. Its up to 32 addressable I/O ports and interrupt handling contribute to its easy integration into any design. It is controlled by a compact and comprehensive set of 16-bit instructions, but is still capable of immediate 16-bit memory addressing without the use of paging. A complete implementation of the core requires less than 5K gates.

This report will show that the NanoRisc is more suited to Chipcon's firmware processor needs than the available intellectual property (IP) processor cores.

## 1.1 Project Description

The task as given by Chipcon is cited below:

*Develop an ultra-low-complexity CPU core aimed at replacing hardwired finite state machines (FSMs) and custom logic in ASIC designs. A typical application is packet processing and protocol handling. A suggestion for CPU core features is:*

- *Ultra-low complexity (2-5 kgates maximum)*
- *8/16-bit ALU/register width (e.g. sixteen 8-bit registers which can also be addressed as eight 16-bit registers)*
- *Simple, orthogonal, high-density instruction set. (preferably fixed size instructions)*
- *Common address space for code and data (12-16 bit). Separate memory busses for code and data allow the two to map to disjoint parts of the address space and be accessed simultaneously. All memory busses are compatible with synchronous SRAMs (synchronous read and write).*
- *Load/store architecture.*
- *Simple integrated interrupt controller.*
- *Parameterizable number of input/output ports.*
- *Low power consumption (< 25 uW/MHz in a 0.18 um process)*

*The goals of the thesis are:*

- *Detailed specification of CPU architecture / instruction set*

- *Develop cycle-accurate instruction set simulator (ISS) in C/C++ (preferably GUI-based).*
- *Develop assembler for the instruction set (Bison/Flex & C/C++).*
- *Implement core in synthesizable VHDL.*
- *Perform exhaustive testing of VHDL core against ISS.*
- *Perform test synthesis for a TSMC 0.18 um process to determine accurate gate count and power consumption.*

# 2 Chipcon and NanoRisc

This chapter gives a brief overview of Chipcon, and accounts for the motivation behind the NanoRisc project. Chipcon is a Norwegian microelectronics company which specializes in the design of wireless transceivers. They operate in an international market, selling chips for applications ranging from wireless game pads and security systems to industrial systems.

## *2.1 Corporate and Organization*

Chipcon started out as a specialized Application Specific Integrated Circuits (ASICs) design center in 1996, and has since changed strategy to become a leading international 'fabless' semiconductor company. It has a total of 103 employees where 17 are located in San Diego and 7 in sales offices around the world. The remaining 79 employees make up the headquarter and design center in Oslo where this project was undertaken [Chipcon].

The design center in Oslo handles

- Design of new chips
- Testing
- Support and applications development
- Development tools (HW/SW)

This thesis was written in close cooperation with the digital signal processing (DSP) and system-on-chip (SoC) groups.

Chipcon's largest owners today are Four Seasons Venture and the founders Geir Førre, Sverre Dale Moen and Svein Anders Tunheim.

## *2.2 Products*

Today, most of Chipcon's revenue comes from Application Specific Standard Products (ASSPs) for short-range wireless communication. The application which represents the largest portion of the orders is game pads for systems such as Sony PlayStation [Sony] and Microsoft Xbox [Microsoft]. One of it's main competitors in the wireless game controllers and other wireless applications is Nordic Semiconductor [Nordic] located in Trondheim.

Currently, Chipcon works to promote the ZigBee standard which is a low power consumption and low bandwidth protocol for wireless communication between potentially a large number of devices. It is based on the IEEE802.15.4, and requires a software Media Access Control (MAC) module. See [ZigBee] for details.

## *2.3 Motivation behind the NanoRisc Project*

While Chipcon maintains a range of cheap and simple wireless transceivers, they are also developing complex SoCs for wireless applications. Future Chipcon chips will require rather complex protocol handling, packet processing and buffer control, which is what the motivation for the NanoRisc project springs from. The NanoRisc will be a general-purpose embedded processor providing a specific service in an application as described in [WongVa2004].

The task of designing a processor core as described in the original task had the aim of producing a controller that could replace some of the larger finite state machines (FSMs) currently used for hardware sequencing and protocol control in the Chipcon transceivers. Microprocessors and hardware modules both have the capability to implement most any digital logic function. There are, however, unique advantages and disadvantages to either approach.

- **Ease of implementation** – It was thought that it would be easier to program an FSM as software rather than in a hardware description language (HDL). In terms of testing, it is not necessarily more work to verify a processor running a program than to verify the FSM implementing the same logic.
- **Size** – When evaluating the implementation size of a processor versus a hardware module, it is important to count the memory requirements of microcontroller which grows with complexity. On the other hand, a processor re-uses its logic resources, while the need for logic resources in a hardware implementation increases with complexity. A processor with memory is not expected to be smaller than the hardware state-machine implementing the same logic.
- **Performance** – The sequential execution of instructions in a processor entails a degradation of performance with increased complexity. A hardware module will not necessarily in the same way experience performance degradation with increased complexity because algorithms can be implemented in parallel. This leads to a faster but more expensive implementation. To counterpart to this in a software implementation would be to use several microcontrollers working in parallel to increase performance.
- **Flexibilty –** A microcontroller implementation is more flexible and makes it easier to make changes and fix bugs after initial production. An error in a large hardware FSM causing incorrect behavior discovered after initial production can be very expensive. Fixing it would demand several man-months of work plus the full set of masks to be remade at considerable cost. Alternatively, a manual manipulation of the routing could be performed in some cases, but it is very difficult work. The advantage of using a processor is that its program memory can be changed by changing only one of the masks. Errors in its behavior can thus be changed easily and one can in some cases use the processor to make workarounds for errors elsewhere on the chip. In a larger perspective, a software program is easier to upgrade and to differentiate from a previous design.

An ideal approach is usually to have a microcontroller embedded on a chip where the complex, non-timing crucial control functions can be implemented on the microcontroller, while timing critical or data path functions are implemented in hardwired logic [Xilinx2004].

There are several processor cores on the market today designed for SoC integration, and re-use of such a module would have to be considered for an application such as those mentioned above. The Chipcon engineering group is familiar with design reuse. Both an implementation of the 8051 microcontroller and the Cambridge Consultants Xap1 processor has successfully been used as IP cores in Chipcon designs. The 8051 is found as a user programmable microcontroller chips such as CC1010 and CC2420, while the Xap1 processor was used as a controller in an ASIC project. A discussion of possible integration of a processor IP core is found in chapter 4.

# 3 Requirement Specification

The requirements specification is based on the original task and a series of meetings with Chipcon representatives in January 2005. They represent what Chipcon believes the processor should be for it to be suitable for integration into a future Chipcon transceiver.

1. **Load/store architecture.** This requirement arose from the original task. It implies that data processing should only be done on data in registers.
2. **Simple, orthogonal, high density instruction set.** This requirement arose from the original task. It is understood that the instruction set can not be entirely orthogonal due to its load/store nature, but consistency should be strived for. Fixed size instructions are wanted, but other solutions could be explored.
3. **Data processing capabilities.** This requirement arose from the meetings. The processor should include an ALU with logical and arithmetic operations and should support some shift and rotate scheme.
4. **8x8 Multiplier.** This requirement arose from the meetings. An 8x8 hardware multiplier should be implemented.
5. **Stack.** This requirement arose from the meetings. A stack should be implemented in hardware or software to allow nested function calls and facilitate a C compiler.
6. **16-bit memory interface, no paging.** This requirement, decided in the meetings, is an extension of a requirement in the original task. The processor should be able to interface a 16-bit memory bus, and have a shared, byte addressable address space for data and program. Paging should not be used, as it complicates programming.
7. **A parameterizeable sized I/O space with bit-operations.** This requirement arose partly from the original task and was extended in the meetings. It is important the processor should be able to interface with peripherals without having to memory map these. This is to be implemented by the use of I/O ports, and the number and width of these ports should be parametrizeable. Efficient bit-operations such as "set", "clear" and "test" should be able to operate on these ports.
8. **Simple integrated interrupt controller.** This requirement arose from the original task. The processor should be able to respond promptly to external events by the use of interrupts.
9. **Small footprint (2K-5K).** Perhaps the most important requirement arose from the original task. As size directly influences production cost, the size of the controller should be kept at a minimum to keep it attractive compared to dedicated hardware solutions.
10. **Power consumption of < 25 uW/MHz in the 0.18 um process excluding memories.** This requirement arose from the original task.

# 4 State of the Art

This chapter will give an overview of the current small embedded processor in the market today. The discussion of examples will be limited to architectures available as intellectual property (IP) modules that can be licensed for use in ASIC/ASSP production.

## *4.1 Design Reuse*

Reuse of hardware cores has become increasingly popular over the last few years. The need to close the gap between design sizes and engineer productivity has lead the ASIC industry to adopt the concept of design reuse from software development. A multitude of vendors selling intellectual IP modules for system integration has emerged.

The reuse of processor IP cores is well known and has been available for years through established vendors as ARM [ARM] and MIPS [MIPS] [Rosenberg1999]. Perhaps one of the main reasons behind the success of processor IP cores is the availability of tools. Developing a processor architecture from scratch requires that one also builds up an arsenal of tools for programming and debugging to make the processor usable in practice. Many processor IP vendors have avoided this by providing IP cores that are functionally equivalent to well known and established processor architectures, enabling the tools such as compilers, assemblers and simulators available for these architectures to be reused. Important in this respect is also the fact that the competency of the programmers can be reused when such an approach is chosen.

As hardware systems grow increasingly in complexity, the need to increase engineer productivity grows with it. [Bouldin] suggests that design reuse might save 70% of the design effort compared to the original development. This is a great motivation for design reuse.

Another motivation for design reuse is the brand-name recognition and consumer trust that has been built up by major IP vendors. It is harder to convince an engineer that his application would be easy to write, debug and run on your unknown processor than on the newest ARC supporting the Thumb2 instruction set for example. For Chipcon this becomes a big issue if the processor should ever be targeted for user programming. As long as it is an internal integrated firmware processor, the Chipcon-internal recognition of the processor is all that counts.

There are, however, certain pitfalls associated with hardware design reuse. One of the more obvious is to choose a design that does not turn out to satisfy specifications and constraints [Bolado2003]. Public information on IP cores, however is limited, and it can be difficult to find a processor that fulfills all requirements [Salminen2004]. An alternative is to modify an IP which is almost suited to fulfill all requirements. Most processor core IP's are delivered as soft IP cores, that is, they are described in a hardware definition language. As opposed to physical design, netlists or RTL models, this is the most easily modifiable IP format. The problem, however, is that according to [McCorquodale] verification represents at least 50% of the design cycle for IP components, and if modifications are made, the design will have to be re-verified. In this respect, modified IP is not suited to close the gap between the vast amount of logic that can be put on a chip and the amount of logic an engineer can design [Rosenberg1999].

## 4.2 Comparison of IP Processor Cores

This section describes and evaluates a number of processor IP cores available in the market today. The cores selected for study are generally their vendor's smallest cores with an emphasis on low power consumption. As stated by [Salminen2004], however, IP comparisons are hard and tedious because public information is limited and product briefs are inaccurate and unclear. The IP cores to evaluate were found through conversations with engineers at Chipcon, using the search at design-reuse.com and searches on general-purpose search engines on the internet. Key numbers on these processors are summarized in Table 1.

**Table 1 - IP processor cores**

| *Core* | *Data Width* | *Area [kilogates]* | *Max. Freq. [MHz]* | *Power Consumption [mW/MHz]* | *Performance [MIPS/MHz]* |
|---|---|---|---|---|---|
| *Arc Classic86 [ARCx86]* | 16b | 22 | 80 | - | 0.08 |
| *ArcLite [ArcLite]* | 8b | 3,5 | 160 | - | - |
| *ARM Cortex-M3 [ARMCortex-M3]* | 32b | 33 | 100 | 0.12 | 1.2 (DMIPS) |
| *MC8051 [Salminen2004]* | 8b | 10 | 100 | - | - |
| *OpenRISC 1200 [Salminen2004]* | 32b | 25 | 150 | - | - |
| *PicoBlaze [PicoBlaze]* | 8b | - | - | - | - |
| *Xap1 [Xap1]* | 16b | 3 | - | ~0.45 | 0.18 (DMIPS) |
| *Xap2 [Xap2]* | 16b | 12 | 100 | > 0.45 | 0.37 (DMIPS) |

### 4.2.1 Performance

Most of these processors use pipelining to some extent. Pipelining is the splitting of instructions into steps that are executed on one clock cycle each. It enables the processor to greatly increase its throughput because it can run at higher clock speeds and in most cases complete one instruction each cycle. Several instructions are being executed at one time, and it requires logic to handle precise interrupts, minimize branch penalties, data dependency stalls etc. In addition it requires registers to store results between each stage of the pipeline [Hennesy1996]. This logic may account much of the area consumed by some of these processors, but on the other hand it is what enables for example the OpenRISC to obtain such high operating frequencies.

The measure of relative performance of microprocessors is difficult. Measures like MHz, MIPS or benchmarks do not necessarily reflect how a processor would perform doing your task. The user is referred to [Hennesy1996] for a thorough discussion on the metrics of processor performance. In practice, however, for small processor cores, the Dhrystone benchmark is the most commonly used measurement of performance. The Dhrystone benchmark is a synthetic benchmark, invented in 1984 which contains no floating point operations. The output from the benchmark is the number of Dhrystones per second (number of iterations of the main code loop per second), but the most frequently used metric is the Dhrystone MIPS (DMIPS). It is a measure of the Dhrystone processor relative to the Dhrystone performance of a DEC VAX processor [Weiss2002]. DMIPS are generally seen as being proportional to the clock speed, and the more useful measure of DMIPS/MHz is usually reported. There were, however, no specific requirements as to performance for this project, so no processors are excluded based on it.

### 4.2.2 Area

Based on the requirements put forth in the task for this thesis, most of the above mentioned processors can be ruled out for this application based on simple metrics. One of the requirements is that an implementation of the processor core should be less than 5000 gates. This requirement has a great impact on the cost of the solution, as the gate count is directly proportional to die area and the die area to the fourth power is proportional to the cost of the chip [Hennesy1996]. Even though the ARM Coretex-M processor can compete with many 8-bit architectures in terms of size, the stringent size requirement for this project rules out all the 32-bit processors [Wong2004]. The Xilinx PicoBlaze is a small 8-bit controller especially designed to achieve a small footprint in the Xilinx field programmable gate arrays (FPGA), but its size when implemented in a 18nm ASIC process has not been found. This leaves the 8-bit ArcLite and the 16-bit Xap1 for consideration.

When considering the area consumed by a particular processor, it is also important to consider the size of the required program memory i.e. the code size. Small code size has been an important selling point for the AVR architecture for example. The size of each instruction word and the number of instruction words to implement a program is important. The size of the instruction word is usually bounded from below by the number of instructions needed and the size of the immediate values necessary. Many CISC architectures use a variable-size instruction word to allow large immediate values while avoiding a general increase in instruction word size. The RISC architecture of the Xap2, however, uses a "PREFIX" instruction to load a part of an immediate value into a temporary register which is used to extend the immediate field of the next instruction. Its effect is similar to that of a variable-size instruction word, but with a simpler instruction set.

### 4.2.3 Power Consumption

Power consumption is very important in Chipcon products. Due to their wireless nature, they are often used in battery driven applications. Power consumption is measured in watts, but it is highly dependent on the frequency of operation, so it is common to find the power consumption of a processor in mW/MHz. The requirement for power consumption of < 25 uW/MHz given in the task is for the processor core without memory. In practice, however, for an architecture which demands large amounts of memory or frequent memory accesses, the power consumption of the memory surpasses that of the core. It is therefore important that a low-power design also has sufficient registers to avoid frequent memory accesses. The power consumption of an architecture varies with the process in which it is implemented. For the processors where the power consumption was reported, the process was not always mentioned, and it will be assumed that the numbers are for the popular 18um process. The power consumption of the Xap1 processor is reported as 2.25 mW [Xap1]. Interestingly enough, the frequency of operation is not reported, and assuming a frequency of operation of 50 MHz, this gives 45 uW/MHz, which exceeds the specified value. In addition, the Xap1 only has 4 16-bit registers available in an accumulator architecture, which requires frequent memory accesses. The Xap1 must therefore be ruled out based on power consumption.

### 4.2.4 Ease of Programming

An important but unspecific criteria for a processor is that that it is easy to program. The availability of good compilers for high-level languages as for example "C" is important when writing large programs. All the above mentioned processors except the PicoBlaze implement architectures for which "C" compilers are available. For small control applications where code-size and speed is crucial, it is often necessary to program the processor in assembly language. Assembly language is a programming language in which each instruction generally corresponds to one machine language instruction on the architecture which the assembly language is for. In contrast to higher level languages, an assembly language reflects all the particularities of the ISA of the target machine. Assemblers are available for all the above mentioned architectures. The assembler is a computer program which translates a program written in assembly language to machine language. It is intended to close the semantic gap between the expressions that are close to the human way of representing statements, and the primitive operations of a processor [Clements]. The relationship between the above mentioned entities is described in Figure 1.

```
High Level Language  ->  Assembly Language  ->  Machine Language
                 Compiler              Assembler
```

**Figure 1 - Code hierarchy**

When programming a processor in assembly, it is important for the ease of coding that the architecture bit-width is greater or equal to the size of the data types being operated on. If the bit-width of the architecture is less than the size of the data, several instructions have to be used to perform an operation on the data, and the assembly programming becomes a complex task. This is a great disadvantage of the 8-bit architectures, and especially the PicoBlaze, as it has no high-level language compiler to hide this complexity.

Perhaps the greatest advantage of buying an IP core as opposed to developing a new solution is the availability of tools. In addition to the compilers, many of the above mentioned architectures have editors, debuggers and instruction set simulators available, which simplifies programming and debugging.

### 4.2.5 Architecture

One of the greatest disadvantages of the ArcLite is its interrupt handling. It implements interrupts by inserting an interrupt instruction into it's pipeline. The interrupt handling routine latency is the time from an external interrupt signal change until the first useful instruction of the interrupt routine is executed. For the ArcLite this time is 12 cycles in the worst case. In addition to this, high priority interrupts can not interrupt lower priority interrupt routines in so-called nested interrupts. This means that the worst case interrupt service routine latency for a high priority interrupt is in fact 12 cycles plus the time it takes to finish a lower priority interrupt routine. This makes implementing control applications with any degree of real-time demands very difficult [ArcSupport].

One of the major disadvantages of the PicoBlaze is that it is only capable of addressing 1024 bytes of program memory, while a typical application for a firmware processor could easily be 4Kbytes in size. This rules out the PicoBlaze.

## Part II

# 5 Architecture Specification

This chapter describes the processor from the perspective of the user, or the "Architecture" as described by Blaauw and Brooks [Blaauw1997]. This architecture specification was developed as a part of this project based on the requirements given in the task description (see section 1.1) and the subsequently developed requirements specification (see chapter 3). A discussion on many of the aspects of the NanoRisc design described in this chapter is found in chapter 10. The NanoRisc name was given as part of the task description and has become, as the name imposes, a relatively simple RISC processor. It features 13 16-bit general registers, a dedicated stack pointer, up to 32 I/O ports and a 16-bit memory interface. It features single cycle execution of all instructions that do not read from memory. It is a load/store architecture which means that logical and arithmetic operations can only be performed on data stored in the registers, however, efficient load/store and stack instructions makes moving data between memory and registers fast. A simple overview of the architecture is given in Figure 2.

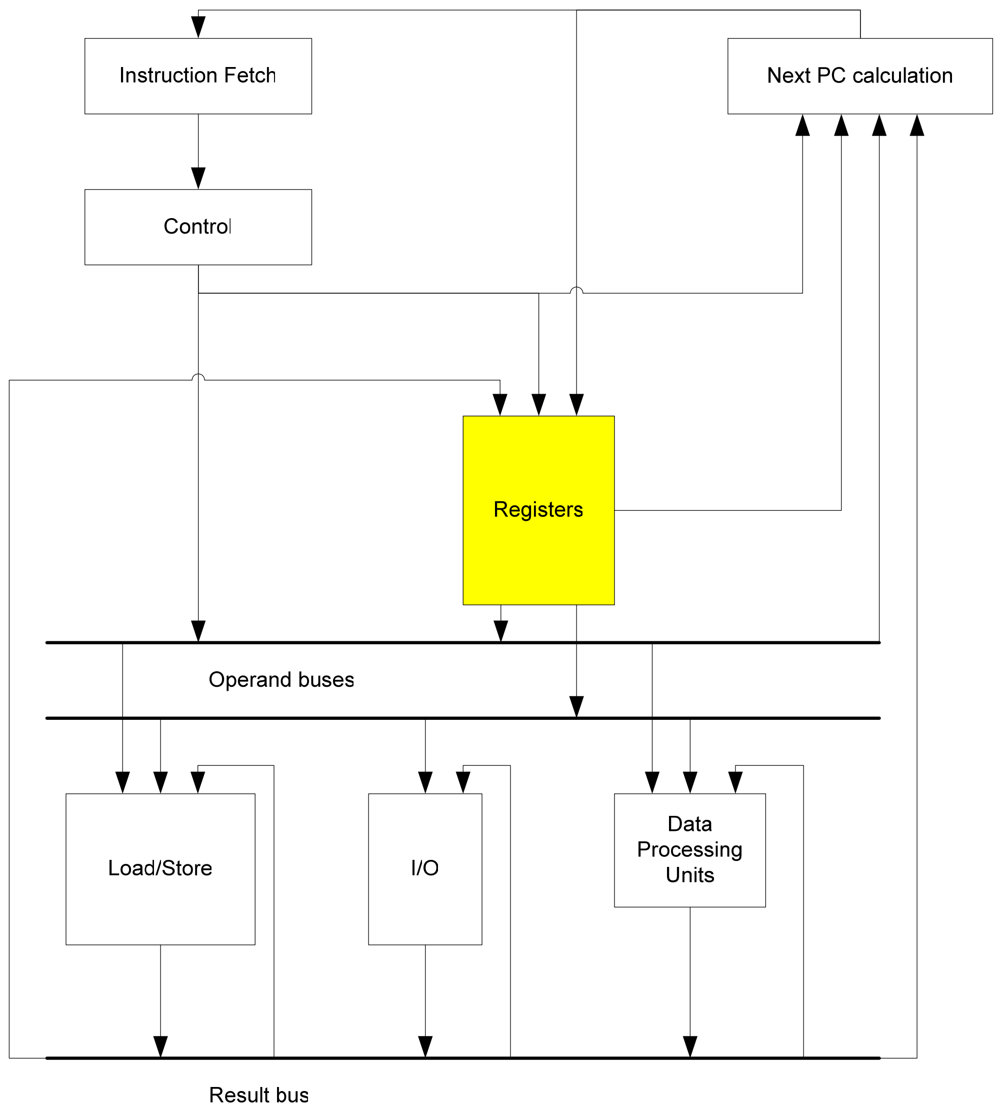


**Figure 2 - NanoRisc simple overview**

Before going into detail on the functionality of the processor, an overview of the nomenclature used will be given.

## 5.1 Nomenclature

**Special registers:**

SR: Status Register

- Halt: Halt Flag
- IRQ: Interrupt Request Flag
- IE: Interrupt Enabled Flag
- V: Two's complement overflow indicator
- N: Negative Flag
- Z: Zero Flag
- C: Carry Flag

PC: Program Counter
SP: Stack Pointer

**General 16-bit registers:**
R0-R12: General registers

**Memory spaces**
M: Data memory
IO: I/O ports

**Bit change indications:**
“-“: Not changed
“*”: Changed
“1”: Set
“0”: Cleared

**Instruction registers and operands:**
Rd: Destination (and source) register in the Register File
Rs Source register in the Register File
K: Constant data
k: Constant address

**Bytes**
LSB: Least significant byte
MSB: Most significant byte

## *5.2 Address Space Overview*

There are three different memory spaces in the NanoRisc. The first memory space is the register memory space. There are 16 addressable registers. The first 13 are general registers, followed by the 16-bit stack pointer, the 7-bit status register, and the up to 15-bit program counter (see Figure 3).

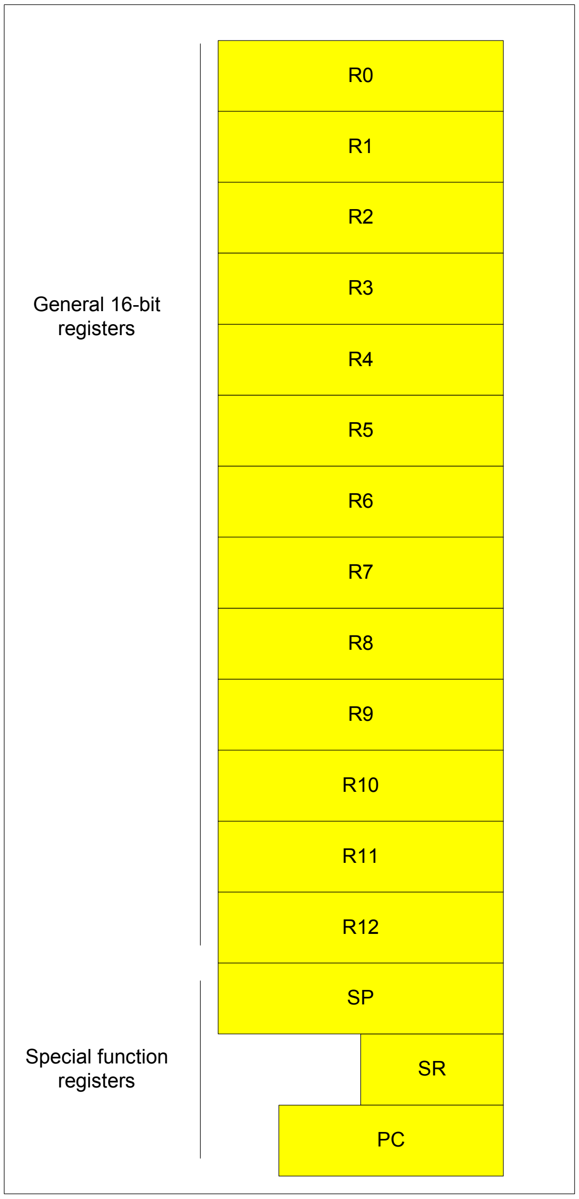


**Figure 3 - Logical register layout**

The second address space is the I/O address space. It is a 5-bit address space where the first 16 addresses are for input ports, and the last 16 addresses are for output ports. The third address space is the combined program and data memory space. It is a 16-bit memory space of which the user decides its composition of RAM and ROM. It is implemented in a manner that is impartial to the big endian/little endian problem. Only aligned word-accesses are allowed, and for byte operations, the byte read or written is read or written to and from the LSB of the register. The correct arrangement of bytes must be handled in an external module.

## 5.3 Instruction Addressing Modes

Due to the load/store nature of the architecture, the NanoRisc instruction set is not orthogonal when it comes to addressing modes. Each instruction supports only one addressing mode, and they are described for each instruction in Appendix A.

## 5.4 Constants

The constant field of the immediate instructions of the NanoRisc ranges from 2 to 8 bits while the architecture is 16-bits. To allow 16-bit immediate values in the branch, logical, arithmetic and load/store instructions, the "pre" instruction is included in the instruction set (see Appendix A). When the "pre" instruction precedes any of the above mentioned instructions that take a "pre" instruction, the constant field of the "pre" instruction is concatenated with the constant field of the instruction being executed to form a 16-bit constant field. The least significant bits of the "pre" instruction form the most significant bits of the constant being used in the instruction. The least significant bits of this constant is the immediate field of the instruction word of the current instruction.

## 5.5 Instruction Fetch

Instructions are fetched on each clock cycle with the exception of the first cycle of an instruction that reads from memory, or when waiting for a busy data memory. The instruction read from the instruction memory is executed in that same cycle. Which address to be loaded next is calculated and selected on each cycle, which voids the need for branch prediction.

## 5.6 General Registers

The NanoRisc contains 13 16-bit general registers without any dedicated role. These registers can be used in all operations where a source or destination register is required. There is no dedicated accumulator. The general registers are referenced in the assembler as R0 through R12, but aliases can be created for better clarity of code (see section 7.1.1.2).

## 5.7 Special registers

The special registers of the NanoRisc each have a dedicated role in the processor. They comprise the stack pointer (SP), status register (SR) and program counter (PC) and are referenced respectively as R13 through R15. These registers can be used in all operations where a source register is required, and with the exception of the program counter, they can also be used as destination registers. When a special register is used as destination register for an operation, this will override any other attempt to modify the contents of the register in that cycle. Note that for the registers that are not 16-bits, when used as a destination register, the result will be truncated.

### 5.7.1 SP

The stack pointer is a 16-bit register which points to the top of the top of the system stack located in the data memory. The stack pointer must be initialized to the desired address of the stack before it is used. Note that all stack operations are word operations, and that the stack pointer should then be initialized to an even address. Stack operations employ a pre-decrement, post-increment scheme in which the stack pointer is decremented before writing a data object to its location (push) and incremented after reading an item from its location (pop). The operations that influence the stack are the following push/pop pairs:

- "push"/"pop"
- "call"/"ret"
- interrupt/"reti"

The instructions that operate on the stack pointer are summarized in Table 2.

**Table 2 - Stack instructions**

| *Mnemonic* | *Description* |
|---|---|
| call | Indirect Call |
| calli | Push PC on the stack, jump to relative |
| pop | Pop a value from the stack and put it in Rd |
| push | Push the contents of Rd onto the stack |
| ret | Jump to the PC popped from the stack + 2 bytes |
| reti | Jump to the PC popped from the stack |

The stack can also be manipulated manually by using the stack pointer as the destination register of an operation, but it is then important to remember that the stack pointer should only point to even addresses. Figure 4 shows an example of stack use. The contents of the register R0 is pushed onto the stack, and then popped to register R1.

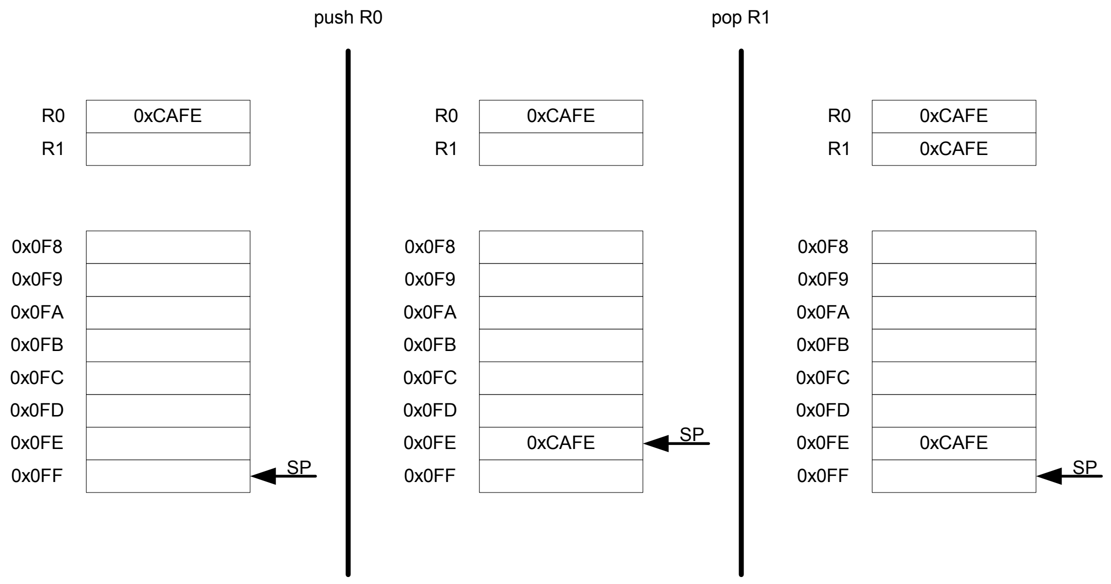


**Figure 4 - Stack use**

## 5.7.2 SR

The 8-bit status register contains the flags and status bits of the NanoRisc. The high nibble contains the bits that control various aspects of the processor while the lower nibble contains the flags that are set based on the result of a logical, arithmetic or shift operation. Which bits are affected by the different instruction is shown in Appendix A. Using the status register as destination register in an operation which writes to register takes priority over the automatic updating of flags by the operation being executed. The register layout is shown in Figure 5, and the individual bits and their behavior when not written to as a register are described below.

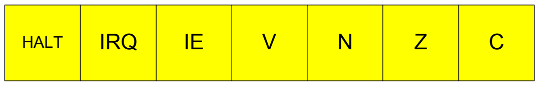


**Figure 5 - Status register**

- HALT – Halt execution. This bit is set when the processor should halt execution. No instructions are executed while this bit is set.
- IRQ – Interrupt request. This bit is set on every clock cycle when the interrupt request line is held high, and cleared when it is low.
- IE – Interrupt enable. When this bit is set, it masks out any interrupt request causing it not to be acknowledged.
- V – Overflow. This bit is set when the result of an arithmetic operation overflows the signed variable range.
- N – Negative. This bit is set whenever the result of an operation is negative.
- Z – Zero. This bit is set whenever the result of an operation is zero.
- C – Carry. This bit is set whenever an operation produces a carry.

### 5.7.3 PC

The program counter contains the program memory word address of the instruction currently being executed. Its width is fully configurable depending of the program memory size in the implementation. A zero is concatenated with the PC to make up the program memory byte address of the instruction (see Figure 6).

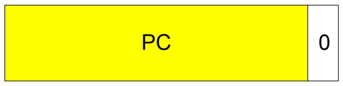


**Figure 6 - Program address**

The PC can not be more than 15-bits wide which would make a 16-bit byte address. A program memory address of 11-bits for example, enables the NanoRisc to address 2K instruction words, or 4K bytes of program memory. The PC is by default reset to -1, but the reset value can be changed by the user before synthesis and set to the address of the first instruction in the program minus one. The PC is written on each cycle of execution except when waiting for a memory operation (see 5.10). A manual write to PC by using it as the destination register of an operation is not allowed, however the “jmp” instruction provides much of the same functionality (see Appendix A).

## *5.8 Data Processing*

The NanoRisc implements a comprehensive set of data processing instructions enabling the user to efficiently process data in the registers. In keeping with the load/store architecture philosophy, the data processing instructions all operate on values stored in the registers.

### 5.8.1 Arithmetic

The arithmetic instructions are provided to enable calculations on both signed and unsigned quantities stored in the registers. All arithmetic instructions set the flags in the status register as if a signed operation was performed. However, due to the principles of two's complement arithmetic, the same instruction can be used to calculate a valid result for both a signed operation and an unsigned operation. The distinction between signed and unsigned operations is made with the conditional branch instructions (see section 5.11). Table 3 gives an overview of the arithmetic instructions implemented in the NanoRisc.

**Table 3 - Arithmetic instructions**

| *Mnemonic* | *Description* |
|---|---|
| add | Add |
| addc | Add with Carry |
| addci | Add Immediate with Carry |
| addi | Add Immediate |
| cmp | Compare |
| cmpi | Compare Immediate |
| sub | Subtract |
| subc | Subtract with Borrow |
| subci | Subtract Immediate with Carry |
| sxt | Sign-extend LSB into MSB |
| zxt | Zero-extend LSB into MSB |

There is no instruction implemented to negate a register in one cycle, because it would require another multiplexer on the input to the ALU in addition to the control logic. Figure 7 shows how this can instead be done in two cycles employing the principle of two's complement.

```
inv R0
addi R0, 1
```

**Figure 7 – Negate**

### 5.8.2 Logical

The logical instructions perform logical bitwise operations between the contents of two registers. They enable amongst other things the setting, clearing and toggling of individual bits. Table 4 gives an overview of the logical instructions of the NanoRisc.

**Table 4 - Logical instructions**

| *Mnemonic* | *Description* |
|---|---|
| and | Logic AND |
| andi | Logic AND Immediate |
| inv | Invert Rd |
| mov | Copy register content |
| or | Logic OR |
| ori | Logic OR Immediate |

| | |
|---|---|
| tst | Logic Test |
| tsti | Logic Test Immediate |
| xor | Logic XOR |
| xori | Logic XOR Immediate |

The architecture does not have a dedicated function to clear the contents of a register. The same result can, however be produced by xor'ing the register with itself as shown in Figure 8. This will correctly set the zero flag of the status register.

```
xor R0, R0
```

**Figure 8 - Clear register**

### 5.8.3 Multiplication

The architecture is capable of single cycle 8x8 unsigned multiplication. The result is stored in one of the 16-bit registers. To achieve signed multiplication, the user must give the result the sign indicated by the xor of the sign bits of the multiplier and multiplicand. An overview of the NanoRisc multiplication instructions is given in Table 5.

**Table 5 - Multiplication instructions**

| *Mnemonic* | *Description* |
|---|---|
| mul | Unsigned Multiplication of LSB |
| muli | Unsigned Multiplication Immediate of LSB |

### 5.8.4 Shift and Rotate

The NanoRisc processor provides full shift and rotate functionality. Between 0 and 15 shifts can be performed in either direction in a single cycle, shifting in '0's, '1's, the value of the carry bit or the bits shifted out. A summary of the shift and rotate instructions is given in Table 6.

**Table 6 - Shift and rotate instructions**

| *Mnemonic* | *Description* |
|---|---|
| rol | Rotate left |
| ror | Rotate Right |
| slc | Carry Shift Left |
| sll | Logic Shift Left |
| sra | Arithmetic Shift Right |
| src | Carry Shift Right |
| srl | Logic Shift Right |

The NanoRisc does not include a specific instruction to swap the LSB with the MSB of a register. This effect can instead be achieved by rotating the register 8 places to the left or to the right.

## *5.9 I/O*

The NanoRisc Input/Output ports allow connection to peripherals or other NanoRisc processors, and resemble the external ports of the 8051 architecture. The advantage of keeping a separate set of I/O ports as opposed to memory mapped I/O is the possibility to perform single cycle bit-operations on output which is important when implementing bit-banging or timing critical control functions. Bit-banging is the action of transmitting data by emulating the protocol in software without the use of a task-specific hardware controller.

The processor contains a configurable number of addressable I/O ports. The width of the I/O ports is configurable from 1 to 16 bits. There are always equally many input ports as there are output ports with a maximum of 16 of each. The port layout with the maximum number of ports is described in Figure 9.

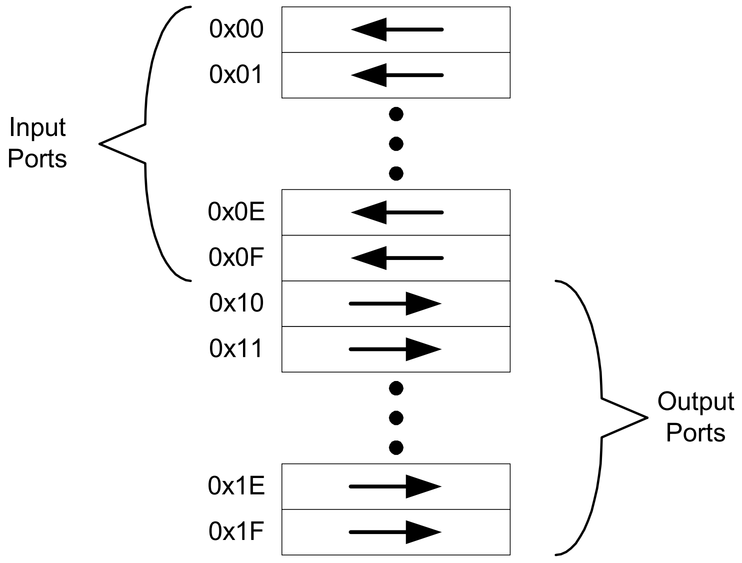


**Figure 9 - I/O port layout**

Read operations can be performed on either input or output ports. If the width of the I/O port is less than 16-bit, the value is zero-extended when read into a register. Operations that change the value of a port can only be performed on output ports. An attempt to change the value of an input port will be ignored by the processor. The NanoRisc I/O instructions are summarized in Table 7.

**Table 7 - I/O instructions**

| *Mnemonic* | *Description* |
|---|---|
| iobc | Clear bit in I/O out register |
| iobs | Set bit in I/O out register |
| iosc | Set bit in I/O out register to value of carry flag |
| iotg | Toggle bit in I/O in register |
| iots | Test bit in I/O in register |
| rdio | Read from I/O port |
| rdioi | Read from I/O port immediate address |

|  |  |
|---|---|
| wrio | Write to I/O port |
| wrioi | Write to I/O port immediate address |

## 5.10 Memory

The program and data memories of the NanoRisc share address space, however they each have a separate memory bus going out of the NanoRisc (see 5.15). This enables the user to choose the most sensible solution for memory implementation depending of the demands of the application. A sensible solution in an application where the NanoRisc functions as a coprocessor to another microcontroller could be to implement the program and data memories in the same memory. This would require an arbiter between the program and data memory busses, the CPU and potentially other units using the same RAM (Random Access Memory) (see Figure 10). The CPU would then load the program data for the NanoRisc into it's RAM on startup. It is important that the NanoRisc program memory read would always be prioritized in such a setting.

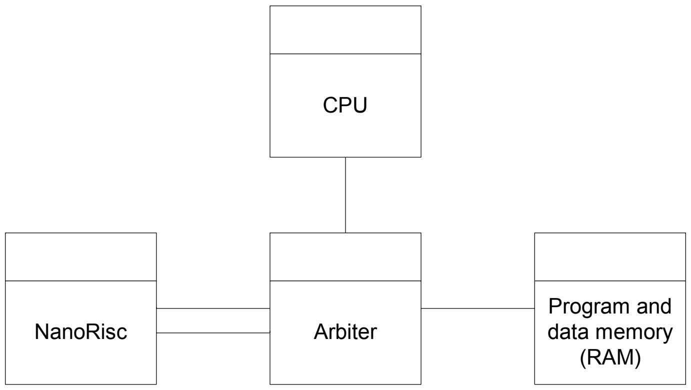


**Figure 10 - Single memory solution**

When the NanoRisc is used as a standalone implementation of a control function, it will in most cases be simpler and more economical to implement the program memory as a separate ROM (Read Only Memory) (see Figure 11). This configuration, however, does not allow the NanoRisc to read constants from its own program memory without an arbiter on the data bus with access to the ROM.

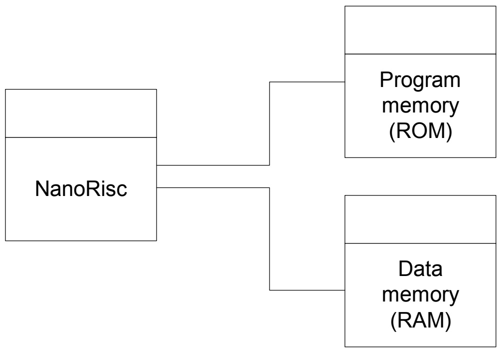


**Figure 11 - Separate memories solution**

Other configurations are along the same lines are of course possible. It should be noted, however, that the NanoRisc reads from program memory on practically every cycle and that these reads have to be prioritized.

### 5.10.1 Program Memory

The program memory contains the instruction words of the program. After a reset, the NanoRisc will always start reading from the "PC_RESET_VAL" address plus one word, so the program data should always be placed at this address. The memory is byte-addressed, and instruction words are always read from even addresses. Because a read instruction word is not registered internally before it is used, there are certain requirements as to the delivery of the program data. The program data must be a synchronous RAM or ROM. The access time (time after a rising clock edge before data is ready on the lines) should be relatively low in order to achieve a high operating frequency. If the program memory receives a read request from the NanoRisc which it can not fulfill on the next cylcle, it should deassert the "clk_enable" signal to stall the processor for one cycle.

### 5.10.2 Data Memory

The data memory should be implemented in a synchronous RAM. The memory is byte addressed and there are operations to read and write both bytes and words. The LSB of a word is always on an even address. When reading a byte, the MSB of the result is ignored which gives freedom when implementing the memory. The physical memory unit in which the data memory is implemented can be shared with the program memory of the NanoRisc or other units. If, due to contention, a data memory request from the NanoRisc cannot be fulfilled on the next clock cycle, the memory wait signal (see section 5.15) should be asserted by the memory. This will cause the processor to wait until the signal is deasserted and the request has been fulfilled. The timing diagram in Figure 12 shows the sequence of events when the NanoRisc receives the memory wait signal at a load operation.

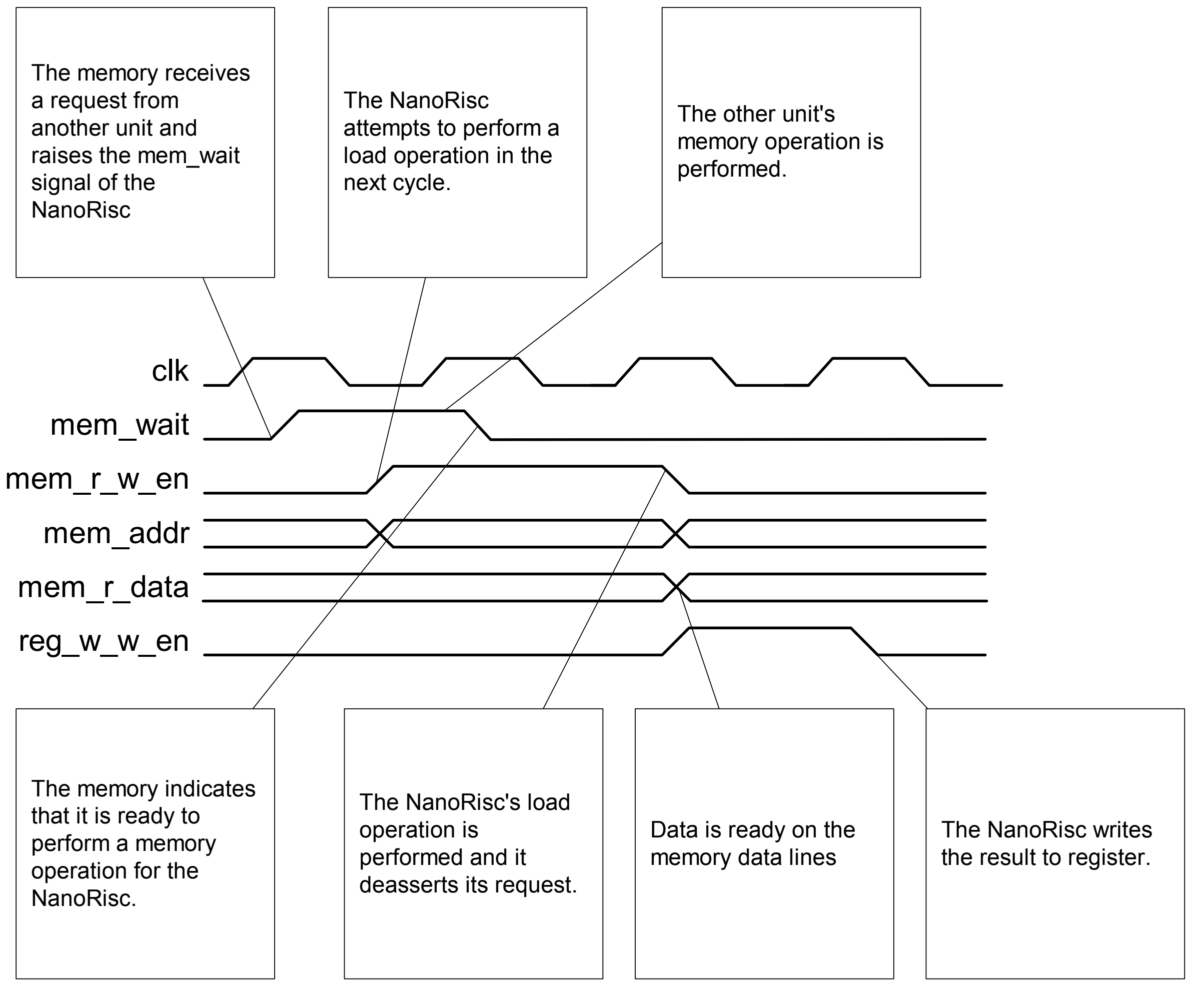

The memory receives a request from another unit and raises the mem_wait signal of the NanoRisc
The NanoRisc attempts to perform a load operation in the next cycle.
The other unit's memory operation is performed.
clk
mem_wait
mem_r_w_en
mem_addr
mem_r_data
reg_w_w_en
The memory indicates that it is ready to perform a memory operation for the NanoRisc.
The NanoRisc's load operation is performed and it deasserts its request.
Data is ready on the memory data lines
The NanoRisc writes the result to register.

**Figure 12 - Memory wait timing diagram**

The instructions that operate on the data memory are summarized in Table 8.

**Table 8 - Memory instructions**

| *Mnemonic* | *Description* |
|---|---|
| ld | Load byte/word |
| lda | Load byte/word with inc/dec |
| ldo | Load byte/word with offset |
| st | Store byte/word |
| sta | Store byte/word with inc/dec |
| sto | Store byte/word with offset |

In addition to these, the stack operations described in 5.7.1 also operate on the data memory.

## 5.11 Program Flow

The default program flow is to execute next the instruction located after the current instruction in the program memory. This default flow can be overridden by either of two separate events; acknowledging an interrupt or change of flow by an instruction, where the interrupt takes priority. A description of the interrupt handling is given in section 5.12. Change of flow by instruction is a single cycle operation, and this voids the need for branch prediction. Table 9 describes the instructions that potentially change the flow of the program execution.

**Table 9 - Change-of-flow instructions**

| *Mnemonic* | *Description* |
|---|---|
| beq | Branch if zero flag is set |
| bges | Branch if greater than or equal signed |
| bgeu | Branch if greater than or equal unsigned |
| blts | Branch if less than signed |
| bltu | Branch if less than unsigned |
| bn | Branch if negative |
| bne | Branch if zero flag is not set |
| call | Indirect Call |
| calli | Push PC on the stack, jump to relative |
| jmp | Indirect jump |
| jmpi | Unconditional branch |
| ret | Jump to the PC popped from the stack + 2 bytes |
| reti | Jump to the PC popped from the stack |

## 5.12 Interrupt

The interrupt functionality allows the NanoRisc to respond promptly to external asynchronous events without having to poll continuously for them. This means that the processor is free to perform other useful work while it is waiting for an external event. An interrupt is triggered by an interrupt request (IRQ) signaled by the assertion of the "irq" input line on the NanoRisc (see section 5.15). This signal is sampled on each rising clock to set the "IRQ" flag in the status register. The interrupt will be considered for acknowledgement if the "interrupt enable" (IE) flag of the status register is set. There is no rollback functionality in the NanoRisc, so all instructions have to be executed atomically. This means that an interrupt will neither be acknowledged while a load instruction is waiting for the result, nor between the execution of a "pre" instruction and the instruction that is to use the "pre" value.
When an interrupt is acknowledged, the instruction read from memory on that cycle is not executed, instead the PC is pushed onto the stack and the next instruction to be loaded is the instruction at address of the interrupt vector (see Figure 13). The interrupt vector is an input to the NanoRisc(see section 5.15).

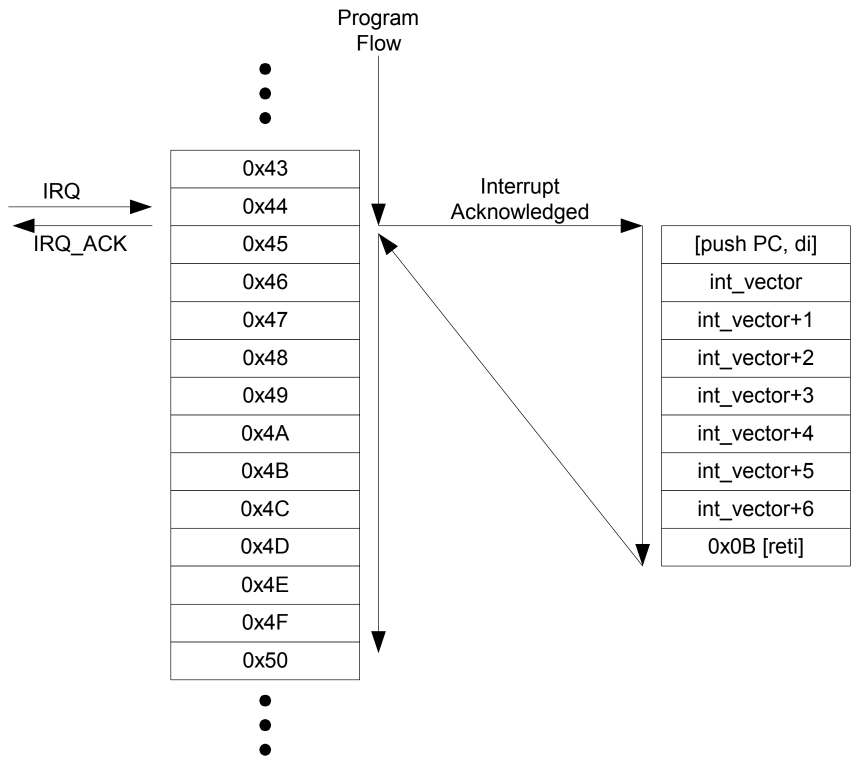


**Figure 13 - Interrupt sequence**

The maximum time between the rise of the IRQ line and the execution of the first instruction of the interrupt handler is two cycles. Usually the first instruction of the interrupt handler should be to push the status register to the stack in order to preserve the state at which the interrupt was acknowledged. When an interrupt is acknowledged, the "irq_ack" signal is set high for one clock cycle, and the interrupt enable (IE) flag of the status register is cleared to avoid unwanted nesting of interrupts. To enable nested interrupts, the "ie" instruction should be run by the interrupt handler. Nested interrupts means that an interrupt handling routine can be interrupted to start another interrupt handling routine. The "reti" instruction will atomically return to the instruction which was about to be executed when the interrupt was acknowledged and enable interrupts by setting the IE flag of the status register. In some rare cases one might want to use an interrupt routine to disable interrupts. In this case one can replace the "reti" instruction by a decrement of the PC on the stack followed by a "ret" instruction which does not enable interrupts.

Two levels of priorities for interrupts may be achieved in the NanoRisc by the use of the "ie" instruction. The nature of the interrupt should be determined by software, and if it is a low priority interrupt, the "ie" instruction should be run to enable a high priority interrupt to interrupt the routine. This simple interrupt handling will be insufficient for some applications. In those cases, an external interrupt module such as a programmable interrupt controller (PIC) [Bolado2003] can be used to provide extended interrupt functionality as described in Figure 14.

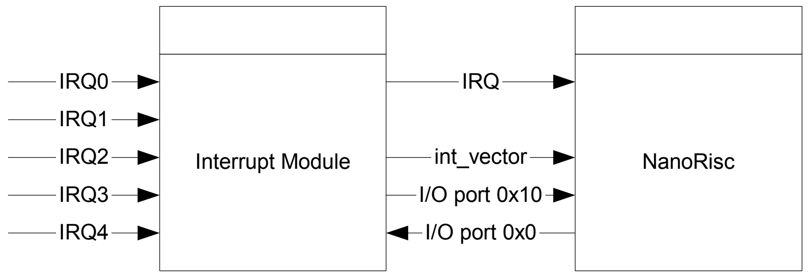


**Figure 14 - External interrupt module**

Several interrupt request lines may be connected to the interrupt module. The module prioritizes the interrupts and forwards an IRQ to the NanoRisc. The NanoRisc will, as one of the first operations of the interrupt handler, read the identity of the current interrupt from the I/O lines. Interrupt priorities may be changed dynamically by the NanoRisc by communicating with the interrupt module over the I/O lines. Alternatively, the interrupt module can dynamically set the interrupt vector depending on the identity of the interrupt.

The architecture of the NanoRisc also allows software interrupts or “traps”. A trap can be implemented by manually setting the “IRQ” flag of the status register, for example by using the emulated raise interrupt (“ri”) instruction (see Appendix A). Please note that the NanoRisc does not make a distinction between software and hardware interrupts, and that a software interrupt is also acknowledged using the external “irq_ack” line.

## 5.13 Halt

The halt functionality of the NanoRisc enables it to be entered into a power-saving mode. The “sleep” input line (see 5.13) is sampled on every rising clock edge into the “halt” bit of the status register. When the halt bit is set, the state of the processor is frozen, and no more instructions are performed. The “halt” bit also drives the “is_sleeping” output line. When the “is_sleeping” signal goes hight, the “clk_en” signal can be driven low, causing the registers of the NanoRisc to stop sampling, further reducing the power consumption. To wake the NanoRisc, the “clk_en” signal must be driven high, and an IRQ must be signaled. The processor will then continue execution where it left off or handle the interrupt if the interrupt enable bit of the status register is set. Another alternative for waking the processor is to reset it, in which case the state of the processor is reset and execution starts from address 0 (see section 5.14).

## 5.14 Reset

A reset of the processor is performed by holding the “reset_n” line low for at least one clock cycle (see section 5.15). A reset will set the status register to 0, which means among other things that interrupt is initially disabled. The PC is by default reset to -1, but the reset value can be changed by the user before synthesis and set to the address of the first instruction in the program minus one. The program memory read lines “p_rdata” should be reset to 0. The I/O output registers are reset to 0. The other registers of the NanoRisc are not affected by a reset.

## *5.15 Interface*

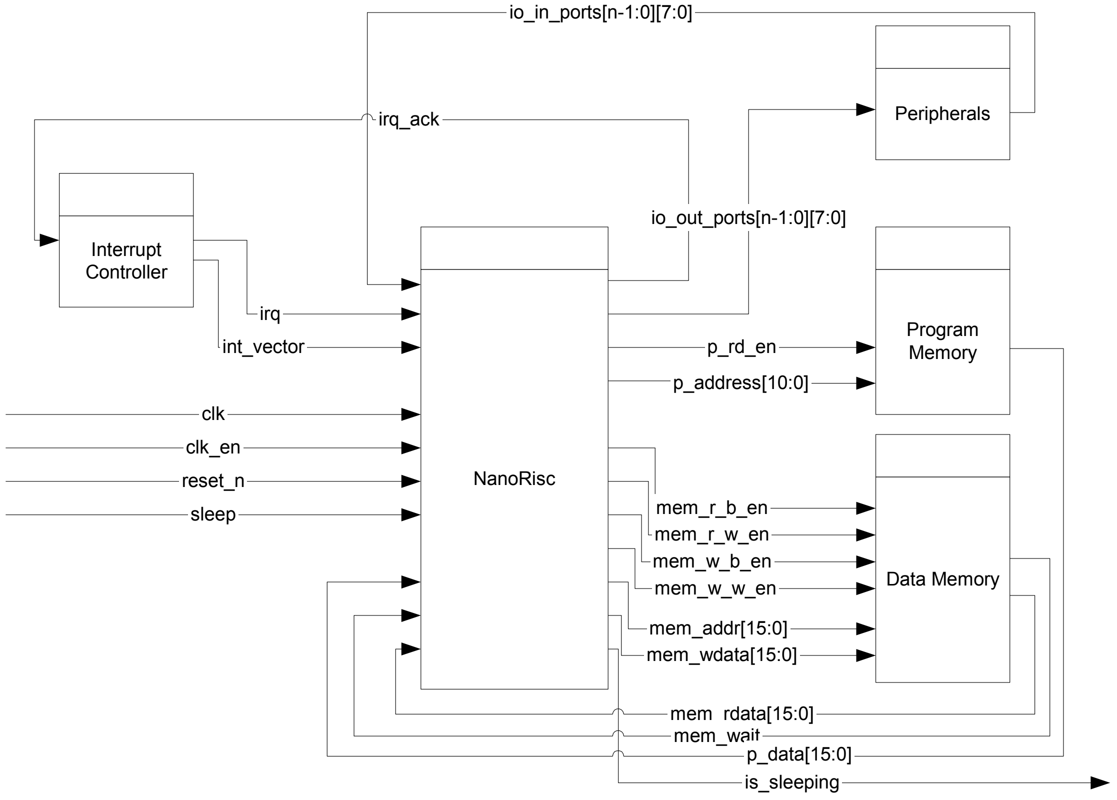


**Figure 15 - Top-level interface**

An overview of the top-level interface signals to the NanoRisc is shown in Figure 15. A detailed description of each signal is given in Table 10.

**Table 10 - Top-level interface signal descriptions**

| | | |
|---|---|---|
| io_in_ports [n-1:0][7-0] | in | **I/O input ports.** It consists of n ports of byte width. Valid data should be present on these lines when reading from the corresponding I/O port. |
| irq | in | **Interrupt request line.** Sampled on each rising clock to set the IRQ flag in the status register. Should be deasserted as soon as the "irq_ack" is asserted. |
| int_vector | in | **Interrupt vector.** These lines contain the address of the interrupt handler. |
| clk | in | **Clock input.** All elements of the NanoRisc CPU are clocked from the rising edge of this signal. |
| clk_en | in | **Clock enable signal.** This signal gates the clock signal to all registers and should be deasserted when the processor is sleeping to save power. |
| reset_n | in | **Reset.** This signal is active low and should be deasserted for at least one clock cycle. It resets the CPU to the state described in section 5.14. |
| sleep | in | **Sleep.** This signal will when asserted for at least one clock |

| | | |
|---|---|---|
| | | cycle cause the "halt" flag of the status register to be asserted and the processor to enter the sleep state. |
| p_rdata[15:0] | in | **Program data.** The valid instruction word corresponding to the address on the "p_address" lines should be present on these lines after each positive clock edge where "p_rd_en" has been asserted. See section 5.2 for maximum allowable delay. |
| mem_wait | in | **Memory wait.** This signal should be asserted whenever the data memory is busy and will be unable to fulfill a request from the NanoRisc processor on the next clock cycle. |
| mem_rdata[15:0] | in | **Data memory read data.** This signal should contain the data as indicated by the address on the "mem_addr" on the cycle after one of the "mem_r_b_en" or "mem_r_w_en" signals are asserted. |
| irq_ack | out | **Interrupt request acknowledge.** This signal is asserted as soon as an interrupt request has been acknowledged. See section 5.12 for a description of the event sequence. |
| io_out_ports [n-1:0][7:0] | out | **I/O output ports.** It consists of n ports of byte width. These ports always reflect the contents of the output registers of the processor, and can be read at any time. |
| p_rd_en | out | **Program memory read enable.** This signal is asserted whenever a word should be read from the program memory. |
| p_address[10:0] | out | **Program memory address.** This byte address indicates which word which is to be read from the program memory. |
| mem_r_b_en | out | **Data memory read byte enable.** This signal is asserted whenever a byte should be read from the data memory. |
| mem_r_w_en | out | **Data memory read word enable.** This signal is asserted whenever a word should be read from the data memory. |
| mem_w_b_en | out | **Data memory write byte enable.** This signal is asserted whenever the LSB should be written from the "mem_wdata" lines to the data memory. |
| mem_w_w_en | out | **Data memory write word enable.** This signal is asserted whenever a word should be written from the "mem_wdata" lines to the data memory. |
| mem_addr[15:0] | out | **Data memory address.** This byte address indicates which part of the data memory should be read or written. |
| mem_wdata[15:0] | out | **Data memory write data.** This signal contains valid data to be written to memory whenever the "mem_w_b_en" or the "mem_w_w_en" signals are asserted. |
| is_sleeping | out | **Is sleeping.** When this signal is asserted, the processor has entered the sleep mode and will not execute any new instructions until an interrupt request is received. |

## 5.16 Instruction Set

The machine code instruction set of the NanoRisc consists of 63 16-bit instructions. A summary of the instruction set showing how many cycles of execution is required for each instruction and a brief description of each is given in Table 11.

**Table 11 - Instruction set summary**

| *Mnemonic* | *Cycles* | *Description* |
|---|---|---|
| add | 1 | Add |
| addc | 1 | Add with Carry |
| addci | 1 | Add Immediate with Carry |
| addi | 1 | Add Immediate |
| and | 1 | Logic AND |
| andi | 1 | Logic AND Immediate |
| beq | 1 | Branch if zero flag is set |
| bges | 1 | Branch if greater than or equal signed |
| bgeu | 1 | Branch if greater than or equal unsigned |
| blts | 1 | Branch if less than signed |
| bltu | 1 | Branch if less than unsigned |
| bn | 1 | Branch if negative |
| bne | 1 | Branch if zero flag is not set |
| call | 1 | Indirect Call |
| calli | 1 | Push PC on the stack, jump to relative |
| cmp | 1 | Compare |
| cmpi | 1 | Compare Immediate |
| inv | 1 | Invert Rd |
| iobc | 1 | Clear bit in I/O out register |
| iobs | 1 | Set bit in I/O out register |
| iosc | 1 | Set bit in I/O out register to value of carry flag |
| iotg | 1 | Toggle bit in I/O in register |
| iots | 1 | Test bit in I/O in register |
| jmp | 1 | Indirect jump |
| jmpi | 1 | Unconditional branch |
| ld | 2 | Load byte/word |
| lda | 2 | Load byte/word with inc/dec |
| ldi | 1 | Load Immediate |
| ldo | 2 | Load byte/word with offset |
| mov | 1 | Copy register content |
| mul | 1 | Unsigned Multiplication of LSB |
| muli | 1 | Unsigned Multiplication Immediate of LSB |
| nop | 1 | No-operation |
| or | 1 | Logic OR |
| ori | 1 | Logic OR Immediate |
| pop | 2 | Pop a value from the stack and put it in Rd |
| pre | 1 | Load 10-bit PRE register |
| push | 1 | Push the contents of Rd onto the stack |
| rdio | 1 | Input |
| rdioi | 1 | Read from I/O port |
| ret | 2 | Jump to the PC popped from the stack |
| reti | 2 | Jump to the PC popped from the stack |

| | | |
|---|---|---|
| rol | 1 | Rotate left |
| ror | 1 | Rotate Right |
| slc | 1 | Carry Shift Left |
| sll | 1 | Logic Shift Left |
| sra | 1 | Arithmetic Shift Right |
| src | 1 | Carry Shift Right |
| srl | 1 | Logic Shift Right |
| st | 1 | Store byte/word |
| sta | 1 | Store byte/word with inc/dec |
| sto | 1 | Store byte/word with offset |
| sub | 1 | Subtract |
| subc | 1 | Subtract with Borrow |
| subci | 1 | Subtract Immediate with Carry |
| sxt | 1 | Sign-extend LSB into MSB |
| tst | 1 | Logic Test |
| tsti | 1 | Logic Test Immediate |
| wrio | 1 | Output |
| wrioi | 1 | Write to I/O port |
| xor | 1 | Logic XOR |
| xori | 1 | Logic XOR Immediate |
| zxt | 1 | Zero-extend LSB into MSB |

The encoding of the instruction words is shown in Figure 16.

| | Bits | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Mnemonic | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| nop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | RESERVED | | | |
| ret | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | RESERVED | | | |
| reti | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | RESERVED | | | |
| push | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | Rd | | | |
| pop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | Rd | | | |
| zxt | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | Rd | | | |
| sxt | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | Rd | | | |
| inv | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | Rd | | | |
| jmp | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | Rd | | | |
| call | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | Rd | | | |
| rdio | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Rs | | | | Rd | | | |
| wrio | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | Rs | | | | Rd | | | |
| mov | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | Rs | | | | Rd | | | |
| or | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | Rs | | | | Rd | | | |
| xor | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | Rs | | | | Rd | | | |
| and | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | Rs | | | | Rd | | | |
| tst | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | Rs | | | | Rd | | | |
| mul | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | Rs | | | | Rd | | | |
| muli | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | U4 | | | | Rd | | | |
| add | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | Rs | | | | Rd | | | |
| sub | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | Rs | | | | Rd | | | |
| addc | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | Rs | | | | Rd | | | |
| subc | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | Rs | | | | Rd | | | |
| cmp | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | Rs | | | | Rd | | | |
| calli | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | U8 | | | | | | | |
| rdioi | 0 | 0 | 0 | 1 | 0 | 0 | 0 | U5 | | | | | Rd | | | |
| wrioi | 0 | 0 | 0 | 1 | 0 | 0 | 1 | U5 | | | | | Rd | | | |
| iosc | 0 | 0 | 0 | 1 | 0 | 1 | 1 | U5 | | | | | Bitnum | | | |
| iobs | 0 | 0 | 0 | 1 | 1 | 0 | 0 | U5 | | | | | Bitnum | | | |
| iobc | 0 | 0 | 0 | 1 | 1 | 0 | 1 | U5 | | | | | Bitnum | | | |
| iots | 0 | 0 | 0 | 1 | 1 | 1 | 0 | U5 | | | | | Bitnum | | | |
| iotg | 0 | 0 | 0 | 1 | 1 | 1 | 1 | U5 | | | | | Bitnum | | | |
| pre | 0 | 0 | 1 | 0 | 0 | 0 | U10 | | | | | | | | | |
| sta | 0 | 0 | 1 | 0 | 0 | 1 | size | d/i | Rs | | | | Rd | | | |
| lda | 0 | 0 | 1 | 0 | 1 | s/z | size | d/i | Rs | | | | Rd | | | |
| sto | 0 | 0 | 1 | 1 | 0 | size | U2 | | Rs | | | | Rd | | | |
| sll | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | K4 | | | | Rd | | | |
| srl | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | K4 | | | | Rd | | | |
| sra | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | K4 | | | | Rd | | | |
| slc | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | K4 | | | | Rd | | | |
| src | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | K4 | | | | Rd | | | |
| rol | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | K4 | | | | Rd | | | |
| ror | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | K4 | | | | Rd | | | |
| beq | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | S8 | | | | | | | |
| bne | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | S8 | | | | | | | |
| blts | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | S8 | | | | | | | |
| bges | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | S8 | | | | | | | |
| bltu | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | S8 | | | | | | | |
| bgeu | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | S8 | | | | | | | |
| bn | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | S8 | | | | | | | |
| jmpi | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | S8 | | | | | | | |
| st | 0 | 1 | 0 | 0 | 1 | size | U6 | | | | | | Rd | | | |
| ld | 0 | 1 | 0 | 1 | s/z | size | U6 | | | | | | Rd | | | |
| ldo | 0 | 1 | 1 | 0 | s/z | size | U2 | | Rs | | | | Rd | | | |
| ldi | 0 | 1 | 1 | 1 | S8 | | | | | | | | Rd | | | |
| ori | 1 | 0 | 0 | 0 | S8 | | | | | | | | Rd | | | |
| xori | 1 | 0 | 0 | 1 | S8 | | | | | | | | Rd | | | |
| andi | 1 | 0 | 1 | 0 | S8 | | | | | | | | Rd | | | |
| tsti | 1 | 0 | 1 | 1 | S8 | | | | | | | | Rd | | | |
| addi | 1 | 1 | 0 | 0 | S8 | | | | | | | | Rd | | | |
| addci | 1 | 1 | 0 | 1 | S8 | | | | | | | | Rd | | | |
| subci | 1 | 1 | 1 | 0 | S8 | | | | | | | | Rd | | | |
| cmpi | 1 | 1 | 1 | 1 | S8 | | | | | | | | Rd | | | |

**Figure 16 - Instruction Encoding**

## 5.17 The NanoRisc Assembly Language

The NanoRisc specific assembly language is based closely on the NanoRisc instruction set. It consists of a set of assembly instructions enabling the user to produce all possible machine code instructions in the NanoRisc instruction set. Each assembly instruction consists of a mnemonic followed by a possibly empty comma separated list of arguments.

Certain machine code instructions are so frequently used in combination with a fixed set of arguments that though they are not implemented as separate instructions in hardware, they are given a separate assembly instruction. While most instructions in the NanoRisc assembly language correspond directly to a machine code instruction, these pseudo instructions are emulated by machine code instructions with fixed values for one or more of the arguments. The purpose of these instructions is to ease programming and improve readability of a program.

Other features of the assembly language and the functionality of the assembler are described in section 7.1.

# 6 Implementation

This chapter describes the implementation of the NanoRisc processor. Implementation is defined by Blaauw and Brooks [Blaauw1997] to be the aspects of a design the user does not see. The data flow diagram shown in Figure 17 shows the data flow between the main entities of the design, and also indicates which module they are implemented in. Many of the features shown in this diagram will be discussed below, but the reader is referred to the source code in Appendix B for the details on the implementation.

The implementation of the NanoRisc is based on a centralized principle where the instruction is decoded and all control signals are set in the processor control unit (PCU) module. The multiplexers and registers that implement a unit, however, are implemented in their respective modules.

Multiplexers are generally described as "case" expressions and their control signals are of a specific type defined in the "pkg" package. This is done to improve readability both of code and of simulation results.

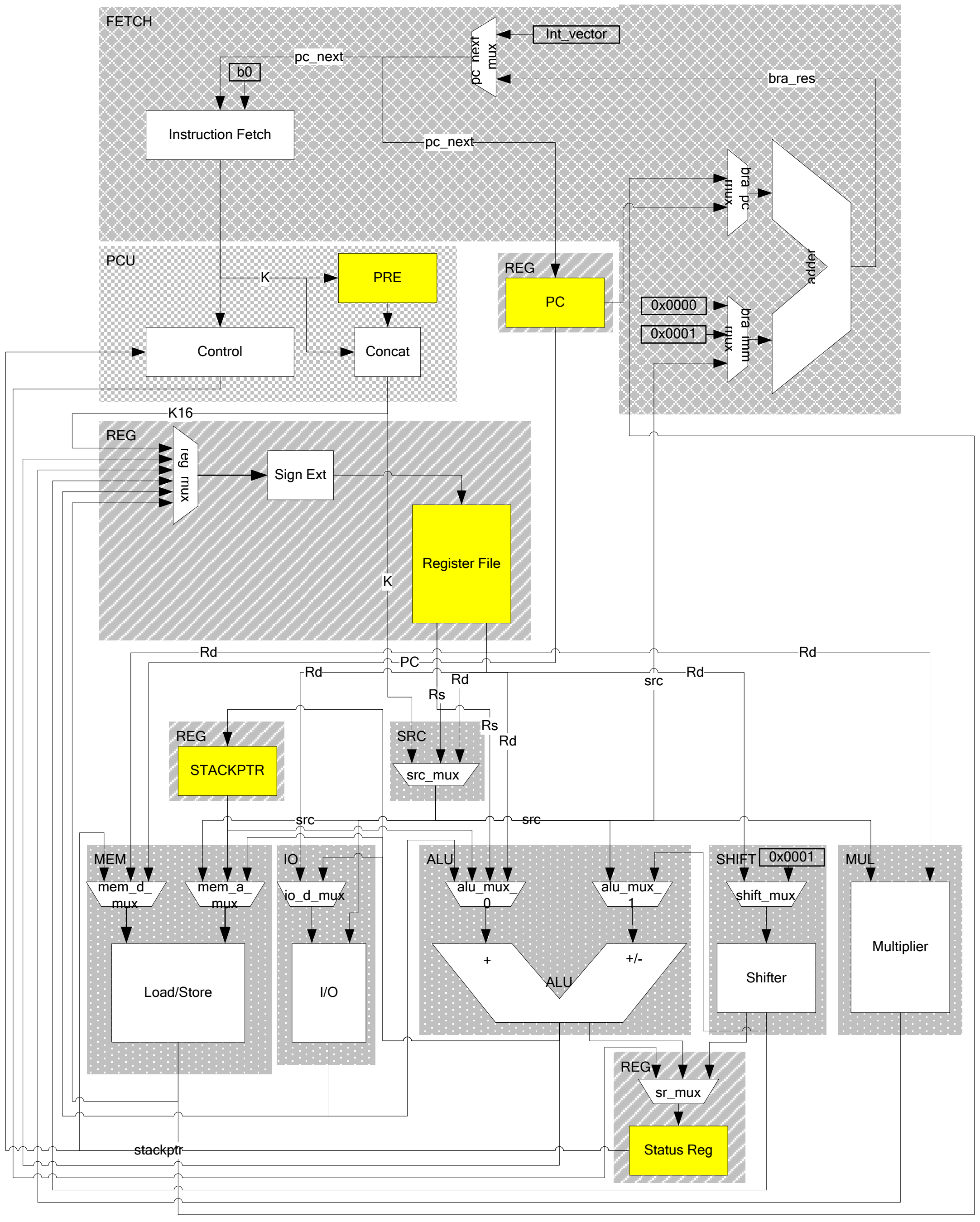


**Figure 17 - Data flow diagram**

An overview of the modules of the NanoRisc is given in Figure 18.

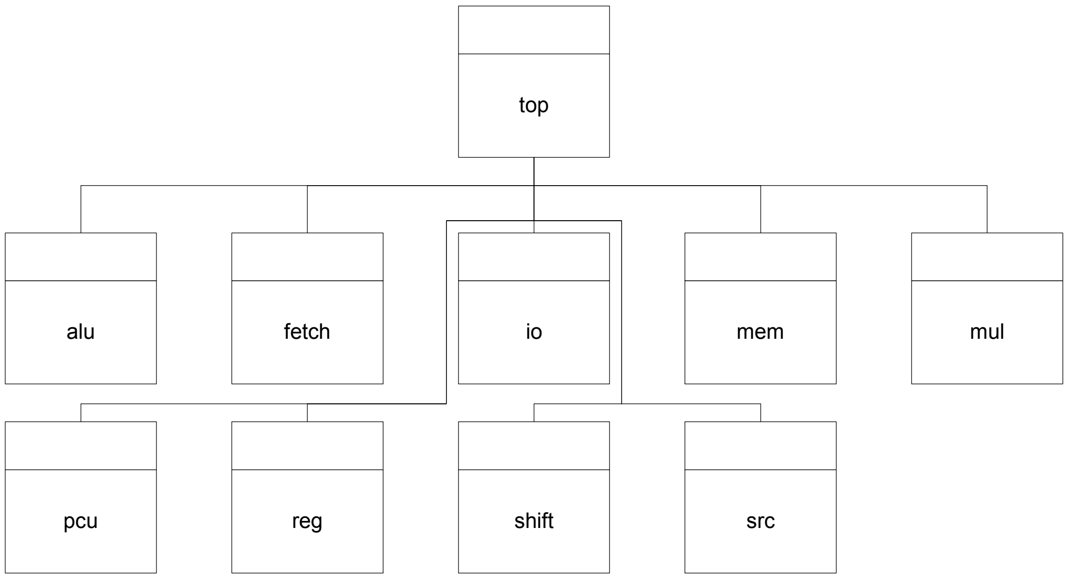


**Figure 18 - Module overview**

They are implemented as VHDL modules with entity and architecture in separate files. The source code for these modules is found in Appendix B, and a short description of each module follows below in alphabetical order.

## 6.1 Top

The main function of this module is to connect the signals of its sub-modules internally and the external interface as described in section 5.15. It also implements the wake-on-interrupt functionality as described in section 5.13.

## 6.2 ALU

The arithmetic logical unit is the main unit for data processing in the NanoRisc. An overview of the ALU structure is given in Figure 19.

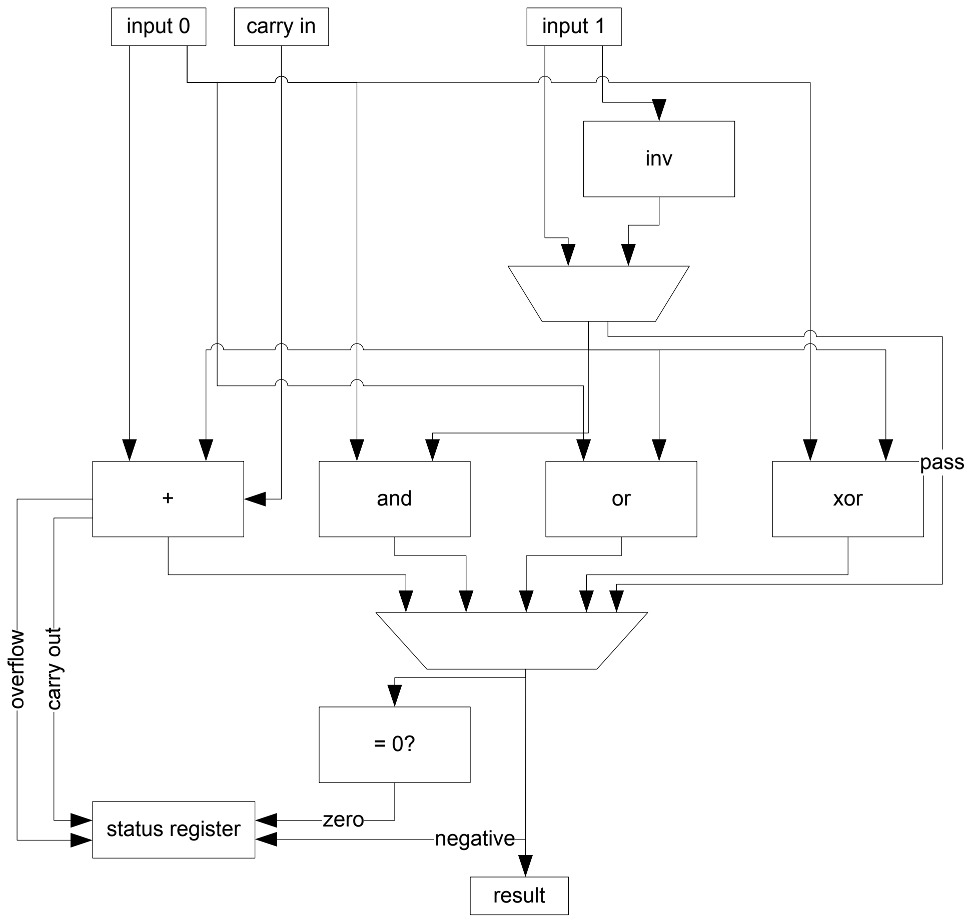


**Figure 19 - ALU overview**

It has two 16-bit operand inputs of which the second can be inverted. It contains a 16-bit carry propagate adder with optional carry-in to perform addition and subtraction operations. The result of the adder also sets the carry and overflow signals to the status register module. The overflow signal is asserted when the carry-in of the msb xor the carry-out of the msb is 1 as explained in [Hennesy1996]. It contains 16-bit logical modules to perform and, or and xor operations. In addition to the results from the adder and logical units, the multiplexer which chooses the final result also has the possibility of choosing the inverted or not inverted input 1. The zero and negative signals to the status register are calculated from the result chosen by this multiplexer.

The adder was written manually from “and”, “or” and “xor” gates. The reason why a Design Ware component instantiated by the “+” operator was not used, is that the carry out from the second most significant bit is needed for the calculation of the overflow bit. An alternative would have been to directly instantiate full-adder elements from the standard cell library, but this would tie the adder to a specific library. On the other hand, using the “+” operator would when used with a good Design Ware library allow the synthesis tool to choose between different adder implementation. Although a simple ripple-carry adder will always be smaller than a carry-look-ahead adder at low speeds, a carry-look-ahead adder might be preferable at high speeds [Smith1999]. Another implementation of the adder should then be considered when synthesizing the NanoRisc for high speeds.

## 6.3 Fetch

This unit generates the address of the next instruction to be executed. This can be one of five main choices:

- The current PC + 1
- The current PC offset by an immediate amount
- The current PC offset by the result from the "src" module (see section 6.10)
- A value read from memory optionally incremented by one.
- The interrupt vector input port value

The choice of what calculation is to be used to determine the next instruction to be executed is decided by the PCU (see section 6.7). It bases its decision on the type of the current instruction, the content of the status register and any interrupt requests. The "fetch" module also calculates the program memory address as described in section 5.7.3.

## 6.4 I/O

This module contains the I/O output registers and controls the reading and writing of I/O ports. There is only one set of I/O address lines, so an output register can be both read and written on the same cycle, but one can not read one port and write another on the same cycle. Bit operations on I/O output registers are performed by passing the current value of the register through the ALU and writing it back on the same cycle. The result of a read from an I/O port is always zero-extended to a 16-bit word.

## 6.5 Mem

The memory module controls the reading and writing of the data memory. It selects the address at which to read or write and the data to be written.

## 6.6 Mul

This module contains the 8x8 multiplication unit of the NanoRisc. It is generated by using the infix operator '*' which produces a multiplier from the Synopsys DesignWare library at synthesis. Chipcon currently only has a license for the most basic of the Design Ware libraries which does not include fast multipliers. As with the adder, a different implementation of the multiplier such as the Booth multiplier should be considered when synthesizing the NanoRisc for high speeds [Smith1999].

## 6.7 PCU

The control unit (PCU) sets the control signals for all modules in the design. It decodes the instruction, sets the control signals and selects the correct address and constant fields from the instruction word.

### 6.7.1 Instruction Decoding

The type of the instructions is identified by a set of nested "case" clauses who evaluate the variable length op-codes of the instructions. For the purpose of instruction decoding, all shift and branch instructions are grouped into the types "iShift" and "iBranch" respectively. The decoding into specific shift and branch instructions is delayed to reduce code size. When an instruction word does not correspond to any of the instructions in the instruction set, it is given the type “iUnknown”, and is executed as a “nop” instruction.

### 6.7.2 Interrupt Handling

An interrupt will be acknowledged if all of the following conditions are met:

- The interrupt request and interrupt enable flags of the status register are set
- The “pre” bit is not set
- The processor is not in the load cycle state (see section 6.7.5).

When an interrupt is acknowledged, the PCU will set control signals to push the SP to the stack, clear the interrupt enable flag of the status register and load the instruction at the address indicated by the interrupt vector input port. The instruction read from memory on this cycle is decoded, but its execution is overridden by the setting of signals to acknowledge the interrupt.

### 6.7.3 Branch Control Unit

The branch control unit is implemented as a function that takes the status register and the condition code as an argument and returns whether or not to branch. The condition code is taken from the three lowest bits (PD0-2) of the relative branch instruction word. The decoding is shown in Figure 20.

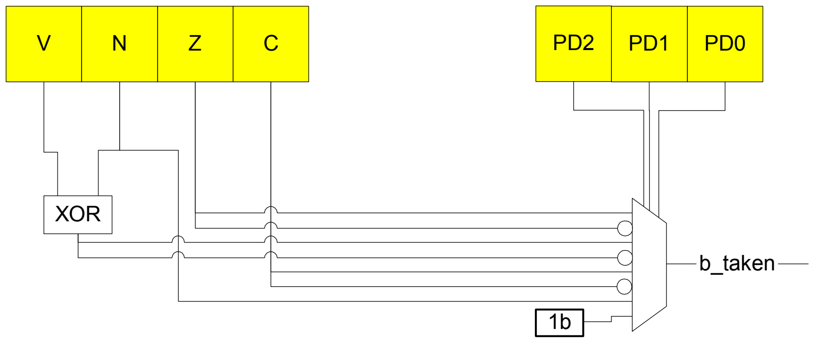


**Figure 20 - Branch control unit**

### 6.7.4 PRE register

The “pre” instruction is implemented by loading its constant value into a 10-bit PRE register located in the PCU module and setting the “pre” bit of the PCU module. Two sets of constants are generated for each constant type; one where the constant field of the executing instruction is sign or zero extended to 16-bit, and one where the PRE register is concatenated with the constant field to form a 16-bit constant. An instruction which takes a “pre” value will determine which of these constants to use based on the status of the “pre” bit. Any such instruction will also clear the “pre” bit.

### 6.7.5 Load Cycle

The control unit contains a 1-bit state machine used to extend any memory load operation to two cycles. In the first cycle, the memory address is calculated and set up, and in the second cycle, called the load cycle, the result is written to register and a new instruction is loaded.

### 6.7.6 Setting Control Signals

The control signals controlling the multiplexers and registers of the design are all set in this module. The default values of the control signals correspond to a "nop" instruction. For all other instructions, the control signals are set to produce the signal path which will give the desired result.

## *6.8 Reg*

This module implements the special and general registers of the NanoRisc. They are implemented as flip-flops. The only flip-flops that have reset functionality are the ones that implement the program counter and status register. The general registers are implemented as an array of 16-bit registers.
The module has two read ports and one write port, but only two address lines, so when reading two register any register to be written must be one of those who are also read. In addition, all special registers can be updated on each clock cycle. Writing to a general register is done by addressing the register and asserting "reg_w_w_en". A special register can also be written in this way, and this will override any other update of the register on that clock cycle. Reading a register is done by addressing it on one of the two sets of address lines, which will put its value on the corresponding read port. In addition, the values of the special registers are out ports on the module.

## *6.9 Shift*

The shifter is implemented as one signal path for shifting to the left and one signal path for shifting to the right. The data flow for the left shifter is shown in Figure 21. It consists of four levels of shifting. The first level shifts by 8, the second by 4, the third by 2 and the fourth by 1. What is shifted in depends on the values of the rotate enable ("rot_en") and carry enable ("carry_en") signals as shown in the diagram. The "rot_en" signal is asserted if the shift type as set by the PCU is "sRot", and the "carry_en" signal is asserted if the shift type is "sC" and the carry flag of the status register is asserted. For each level, a multiplexer chooses between the output from the previous level and the most significant bits of the output from the previous level concatenated with the bits to be shifted in. The multiplexer chooses the first input when the corresponding bit of the shift amount is '0' and the second input when it is '1'. The right shifter is similar to the left shifter, but also includes the possibility of arithmetic shift.

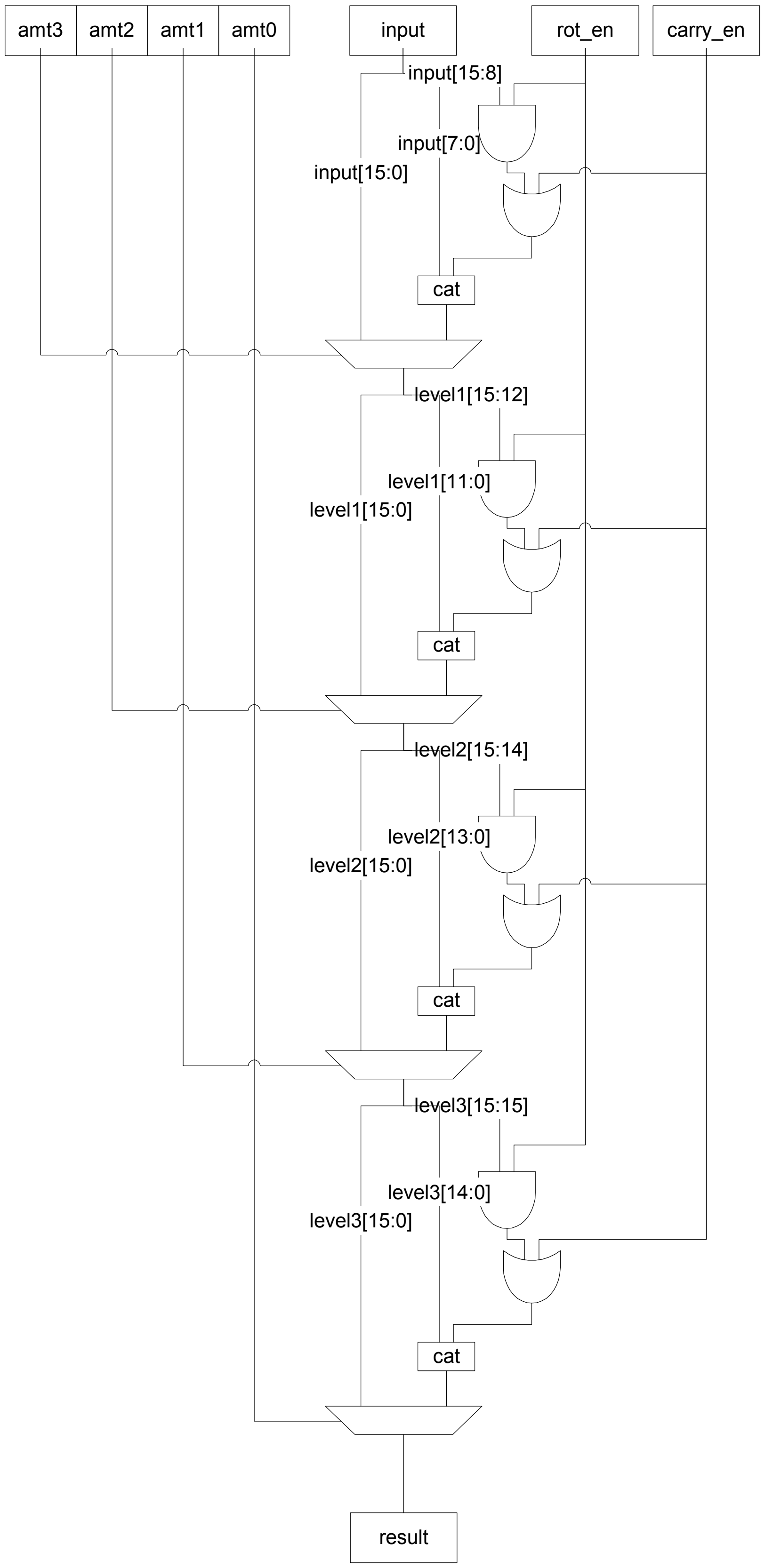


**Figure 21 - Left shifter**

The shift module is used as a mask and constant generator for the ALU. For bit-operations on I/O registers, the bit-number is used as the amount, and the input is set to 1, thereby generating a mask which can be used by the ALU. It is similarly used to generate constants for incrementing the stack pointer and memory address by 1 or 2.

## 6.10 Src

This small module contains only one multiplexer which chooses the source operand “src” for many of the functional units. It chooses between the immediate value, the “Rd” register value or the “Rs” register value.

# 7 Tools

The existence of a comprehensive set of tools is crucial to the success of any microcontroller core. The task specified two tools to be made; an assembler and an instruction set simulator (ISS). Several toolsets for processor tools generation exist such as LisaTek by Coware [LisaTek], and CGEN [CGEN]. The cost of LisaTek starts at $50,000 which is too much for this project's $0 budget. CGEN is free, but is still in a sort of beta stage, and as the developer in this project is unfamiliar with SCEME programming, it was considered too much of a risk. The tools were therefore developed without the use of such a toolset.

## *7.1 Assembler*

The NanoRisc assembler translates a program written in the NanoRisc assembly language into NanoRisc machine code. This section will first describe the prominent features of the assembler, and go on to describe their implementation. Finally, a brief introduction to its command line interface is given.

### 7.1.1 Features

In addition to direct translation between assembly language and machine code, the assembler includes a number of features to improve programmability.

#### 7.1.1.1 Labels

To ease the use of the immediate branch and call instructions, the possibility of defining and referencing labels is included in the assembler. A label is defined by a line in the assembly code with a label name followed by a colon (see Figure 22).

```
labelname:
```

**Figure 22 - Label definition syntax**

A label represents the address of the first instruction that follows it. To refer to a label, the label name can be inserted for any immediate branch offset argument whether it is relative or absolute.

#### 7.1.1.2 Defines

Allowing the programmer to define aliases for registers and constants greatly improves programmability and readability. In the NanoRisc assembler, this is implemented by the ".def" directive (see Figure 23).

```
.def alias value
```

**Figure 23 - Define syntax**

The value part of this directive can be either a constant value or a register name. A register alias can be used in the place of any register name as an argument to an instruction. A value alias can be used in the place of any immediate value. The alias must be defined prior to its use. An attempt to use a value alias in the place of a register name in the argument list of an instruction or vice versa will produce an error.

#### 7.1.1.3 Includes

The possibility to include assembly source code from several files in a program is very helpful for larger programs or when using library functions. The ".include" directive is a simple way to include code from other files without the need for a full linker application. In contrast to a linker, however, it features only a flat scope which may produce conflicts between labels and alias definitions. The include directive can be used at any point in the program, and allows 10 levels of nesting. The syntax is described in Figure 24.

```
.include "filename"
```

**Figure 24 - Include syntax**

The filename is the relative path and filename of the file to be included. When the assembler encounters an ".include" directive, it will immediately start parsing the contents of the included file. Once the included file is parsed to the end, parsing of the including file will resume from the point of the include directive.

#### 7.1.1.4 Logical and Arithmetic Operations

To improve programmability and readability, the capability of evaluating constant expressions containing constant numbers and aliases was implemented. The following operators are supported:

- Plus "+"
- Minus "-"
- Multiply "*"
- Divide "/"
- Unary minus "-"
- And "&"
- Or "|"
- Not "~"

Parentheses are used to group expressions.

#### 7.1.1.5 Constant Interpretation and Automatic "pre" Insertion

The assembler accepts constants on the decimal form such as 79 and hexadecimal form such as 0x4F. All constants entered in the assembler are interpreted to be signed 16-bit words. Thus the hexadecimal value 0xFF will be interpreted as 255 while 0xFFFF will be interpreted as -1. If the programmer enters constant values as an argument to an instruction that are too large to be represented in the instructions constant field, the assembler will attempt to split the value up by inserting a "pre" instruction containing the most significant bits before the instruction. If the type of instruction does not support the use of preceding “pre” instructions or if there is already an immediately preceding “pre” instruction, an error will be generated. Hence, if the programmer attempts to enter a value that is too large to be represented by the instruction and a preceding “pre” instruction, the assembler will first insert the first “pre” instruction, and then attempt to insert another which will cause it to report an error stating that a “pre” instruction already precedes the instruction.

#### 7.1.1.6 Comments

The ability to include comments in the source code greatly increases the programmer’s possibilities for making the program easier to read and understand. All text following a semicolon up to the end of the line will be interpreted by the NanoRisc assembler as comments, and will not be parsed (see Figure 25). C-style line comments (“//”) are also supported.

```
; comment
```

**Figure 25 - Comment syntax**

### 7.1.2 Implementation

The assembler program is written in Microsoft Visual C++ using the lexer generator Flex and parser generator Bison to generate the lexer and parser. The plug-in integrating Flex and Bison into Visual C++ was made by FG-Soup. Though Flex and Bison do not make up a state of the art assembler generator, they were chosen because the developer was familiar with their use. The full source code for the assembler can be found in Appendix D.

#### 7.1.2.1 Lexer

The lexer reads the input stream searching for sequences of characters matching the patterns accepted in the programming language it is reading (tokens). The lexer is called by the parser (see below) and returns a token with corresponding value, an error or end of file (EOF). The NanoRisc assembler lexer recognizes all instruction mnemonics, directives, register names, alias identifiers, constant values and operators as tokens. It skips all white space and comments.

The most complex feature of the lexer is the handling of the “.include” directive. When an “.include” directive is recognized by the lexer, the first thing it does is to create an object representing the state of the file currently being read, and pushing it to the 10 items deep include file stack. It then takes the filename to be included and calculates its relative path with respect to the current directory. It opens the file, creates a new input stream, and starts reading from the new file. When EOF is encountered, if there are more files in the include stack, it pops the file information from the stack and continues reading from the point where it encountered the “.include” directive. If there are no more files in the include stack, this means that the end of the program has been encountered, and the parsing is terminated.

#### 7.1.2.2 Parser

The parser uses a description of the syntax of the programming language to identify constructs of the language that give meaning. The parser in the NanoRisc assembler sees a program as a sequence of program lines. Each program line can be an instruction, a label definition or a directive.

An instruction line is identified as an instruction mnemonic followed by a list of arguments of the correct type. The arguments to instructions are register names or aliases, constant expressions or labels. Register and constant aliases are immediately resolved by looking them up in the symbol table. If they are not already defined, an error is generated. With labels, however, this approach would cause forward references to labels to produce an error. Instead, when a label name is encountered as an argument to an instruction, if the label is not already in the symbol table, the symbol table entry is created but marked as undefined and the instruction created will point to the symbol table entry. If the label is already in the symbol table, the created instruction will point to the corresponding symbol table entry. When an instruction has been correctly identified, the parser uses the "addInstr" method of the global "NrProgram" object "thisProgram" to add an instruction of the appropriate type with the appropriate values for its arguments. The current filename and line number is also stored with the added instruction.

When a program line has been identified as a label definition, if the label already exists in the symbol table and is not already marked as defined, it is marked as defined and the current program address is stored with it. If the label is not already defined in the symbol table, the symbol table entry is created, marked as defined and the program address is stored with it.

When a ".def" directive is identified, a symbol table entry is created describing whether it is a register or a constant alias, and what its value is.

Constant expression evaluation in the parser is done by parsing the expression, evaluating it, and replacing it with the resulting constant value. The operations for evaluating the sub-expressions are implemented using the corresponding standard "C" integer operations on the constant values.

#### 7.1.2.3 Symbol Table

The symbol table is a class which holds a list of entries connecting an identifier used in the program to a type and a value. It is used by the parser who adds new entries and looks up the types and values of existing entries. The symbol table class also ensures that no duplicate entries are allowed.

#### 7.1.2.4 Program Class

The program class is the class that holds all the instructions in the programs after they have been parsed. It contains methods to add instructions and generate code. The process of generating code starts by resolving the label references. The symbol table entries for each label contains the address of the instruction they are pointing to, and after resolving the label reference, the symbol table entry contains a pointer to the "NrInstruction" object

it is pointing at. When the label references have been resolved, the branch offset values of all branch immediate instructions are calculated. The assembler then goes on to check if all constant values can be represented by the constant fields in their instruction. Where it is found not to be the case, a "pre" instruction is inserted before the instruction containing the most significant bits of the value. When a "pre" instruction is inserted, two things happen: all branch immediate offsets have to be recalculated because the relative address of the target instruction might have changed, and then it follows that the branch immediate value of some instruction might have become so large or small as to require a "pre" instruction. Thus, after having inserted one or more "pre" instructions, the assembler recalculates the branch offsets and checks again if any "pre" instructions are needed. This process continues until no "pre" instructions are needed.
When no more "pre" instructions are needed, the assembler goes on to calculate the instruction word for each instruction. All opcodes are stored in an array indexed by the instruction type. For each instruction, the corresponding opcode is or'ed with the values of the variable fields to form the instruction word.

The program class is also responsible for writing the program to file. The method "writeProgram" writes information about the program to file depending on the desired format. Currently, the only supported format prints one line for each instruction containing the instruction word in hexadecimal ASCII character format followed by the word address of the instruction, the originating filename and line number.

### 7.1.3 Program Use

The program is implemented as a Windows command-line executable with syntax as seen in Figure 26.

```
nr_asm input_file [output_file]
```

**Figure 26 - Assembler command-line syntax**

If the "output_file" argument is not supplied, it defaults to the name of the input file with its extension changed to ".hex". When no arguments are supplied, a short text describing the syntax of the program is displayed.

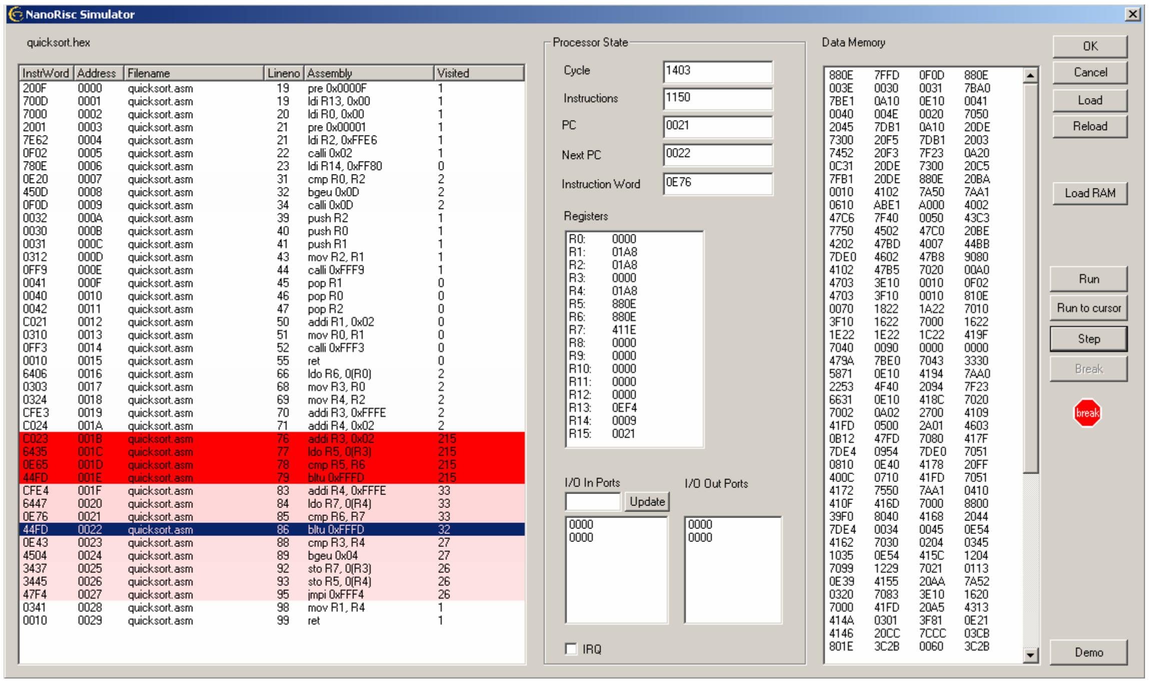
## *7.2 ISS*

The instruction set simulator (ISS) simulates the behavior of the NanoRisc processor at an instruction accurate level. That is to say, after the execution of any one instruction in a program, its status is the same as that of the processor running with the same input. Because the instructions of the NanoRisc with few exceptions are single cycle, the ISS is very close to cycle accurate. The NanoRisc ISS was written as a debugging tool for programmers that can also be useful for optimization of code.

### 7.2.1 Features

Perhaps the greatest feature of the NanoRisc ISS is its speed. In comparison with an RTL simulation, running a program on the ISS consumes negligible amounts of time. Its graphical user interface (GUI) also provides the programmer with a simple way to supervise and control the simulation. Below is an overview of the ISS features and Figure 27 is a screenshot of the application.

- Hexadecimal view of the data memory with the target of the stack pointer highlighted
- Possibility to load data memory contents
- Reload button to quickly restart simulation
- I/O view with the possibility of setting I/O input ports
- IRQ checkbox
- Register overview
- Cycle counter, program counter and current instruction word clearly displayed
- Full disassembler
- Code view showing instruction word, program address, filename, line number, assembly code and the number of times it has been executed for each instruction in the program
- Highlighting of current instruction and color coding of most visited instructions
- “Run”, “Step” and “Run to cursor” modes with possibility to break execution at any point.



quicksort.hex

| InstrWord | Address | Filename | Lineno | Assembly | Visited |
|---|---|---|---|---|---|
| 200F | 0000 | quicksort.asm | 19 | pre 0x0000F | 1 |
| 700D | 0001 | quicksort.asm | 19 | ldi R13, 0x00 | 1 |
| 7000 | 0002 | quicksort.asm | 20 | ldi R0, 0x00 | 1 |
| 2001 | 0003 | quicksort.asm | 21 | pre 0x00001 | 1 |
| 7E62 | 0004 | quicksort.asm | 21 | ldi R2, 0xFFE6 | 1 |
| 0F02 | 0005 | quicksort.asm | 22 | calli 0x02 | 1 |
| 780E | 0006 | quicksort.asm | 23 | ldi R14, 0xFF80 | 0 |
| 0E20 | 0007 | quicksort.asm | 31 | cmp R0, R2 | 2 |
| 450D | 0008 | quicksort.asm | 32 | bgeu 0x0D | 2 |
| 0F0D | 0009 | quicksort.asm | 34 | calli 0x0D | 2 |
| 0032 | 000A | quicksort.asm | 39 | push R2 | 1 |
| 0030 | 000B | quicksort.asm | 40 | push R0 | 1 |
| 0031 | 000C | quicksort.asm | 41 | push R1 | 1 |
| 0312 | 000D | quicksort.asm | 43 | mov R2, R1 | 1 |
| 0FF9 | 000E | quicksort.asm | 44 | calli 0xFFF9 | 1 |
| 0041 | 000F | quicksort.asm | 45 | pop R1 | 0 |
| 0040 | 0010 | quicksort.asm | 46 | pop R0 | 0 |
| 0042 | 0011 | quicksort.asm | 47 | pop R2 | 0 |
| C021 | 0012 | quicksort.asm | 50 | addi R1, 0x02 | 0 |
| 0310 | 0013 | quicksort.asm | 51 | mov R0, R1 | 0 |
| 0FF3 | 0014 | quicksort.asm | 52 | calli 0xFFF3 | 0 |
| 0010 | 0015 | quicksort.asm | 55 | ret | 0 |
| 6406 | 0016 | quicksort.asm | 66 | ldo R6, 0(R0) | 2 |
| 0303 | 0017 | quicksort.asm | 68 | mov R3, R0 | 2 |
| 0324 | 0018 | quicksort.asm | 69 | mov R4, R2 | 2 |
| CFE3 | 0019 | quicksort.asm | 70 | addi R3, 0xFFFE | 2 |
| C024 | 001A | quicksort.asm | 71 | addi R4, 0x02 | 2 |
| C023 | 001B | quicksort.asm | 76 | addi R3, 0x02 | 215 |
| 6435 | 001C | quicksort.asm | 77 | ldo R5, 0(R3) | 215 |
| 0E65 | 001D | quicksort.asm | 78 | cmp R5, R6 | 215 |
| 44FD | 001E | quicksort.asm | 79 | bltu 0xFFFD | 215 |
| CFE4 | 001F | quicksort.asm | 83 | addi R4, 0xFFFE | 33 |
| 6447 | 0020 | quicksort.asm | 84 | ldo R7, 0(R4) | 33 |
| 0E76 | 0021 | quicksort.asm | 85 | cmp R6, R7 | 33 |
| 44FD | 0022 | quicksort.asm | 86 | bltu 0xFFFD | 32 |
| 0E43 | 0023 | quicksort.asm | 88 | cmp R3, R4 | 27 |
| 4504 | 0024 | quicksort.asm | 89 | bgeu 0x04 | 27 |
| 3437 | 0025 | quicksort.asm | 92 | sto R7, 0(R3) | 26 |
| 3445 | 0026 | quicksort.asm | 93 | sto R5, 0(R4) | 26 |
| 47F4 | 0027 | quicksort.asm | 95 | jmpi 0xFFF4 | 26 |
| 0341 | 0028 | quicksort.asm | 98 | mov R1, R4 | 1 |
| 0010 | 0029 | quicksort.asm | 99 | ret | 1 |

Processor State

| | |
|---|---|
| Cycle | 1403 |
| Instructions | 1150 |
| PC | 0021 |
| Next PC | 0022 |
| Instruction Word | 0E76 |

Registers

```
R0:   0000
R1:   01A8
R2:   01A8
R3:   0000
R4:   01A8
R5:   880E
R6:   880E
R7:   411E
R8:   0000
R9:   0000
R10:  0000
R11:  0000
R12:  0000
R13:  0EF4
R14:  0009
R15:  0021
```

I/O In Ports

```
0000
0000
```

I/O Out Ports

```
0000
0000
```

IRQ

Data Memory

```
880E  7FFD  0F0D  880E
003E  0030  0031  7BA0
7BE1  0A10  0E10  0041
0040  004E  0020  7050
2045  7DB1  0A10  20DE
7300  20F5  7DB1  2003
7452  20F3  7F23  0A20
0C31  20DE  7300  20C5
7FB1  20DE  880E  20BA
0010  4102  7A50  7AA1
0610  ABE1  A000  4002
47C6  7F40  0050  43C3
7750  4502  47C0  20BE
4202  47BD  4007  44BB
7DE0  4602  47B8  9080
4102  47B5  7020  00A0
4703  3E10  0010  0F02
4703  3F10  0010  810E
0070  1822  1A22  7010
3F10  1622  7000  1622
1E22  1E22  1C22  419F
7040  0090  0000  0000
479A  7BE0  7043  3330
5871  0E10  4194  7AA0
2253  4F40  2094  7F23
6631  0E10  418C  7020
7002  0A02  2700  4109
41FD  0500  2A01  4603
0B12  47FD  7080  417F
7DE4  0954  7DE0  7051
0810  0E40  4178  20FF
400C  0710  41FD  7051
4172  7550  7AA1  0410
410F  416D  7000  8800
39F0  8040  4168  2044
7DE4  0034  0045  0E54
4162  7030  0204  0345
1035  0E54  415C  1204
7099  1229  7021  0113
0E39  4155  20AA  7A52
0320  7083  3E10  1620
7000  41FD  20A5  4313
414A  0301  3F81  0E21
4146  20CC  7CCC  03CB
801E  3C2B  0060  3C2B
```

**Figure 27 – ISS screenshot during execution of quicksort**

### 7.2.2 Implementation

The NanoRisc ISS is written in MS Visual C++ using the Microsoft Foundation Class (MFC) library for window handling. The simulator itself is implemented in the “NrSimulator” class, which is controlled by a GUI class called “SimulatorDlg” (see Appendix E for the source code). Both the GUI and simulator classes also use the disassembler for the NanoRisc assembly language implemented in the “NrDisassembler” class. The “CListCtrlEx” control is a list control developed by Chipcon and should not be considered as a part of the work done for this thesis.

#### 7.2.2.1 Simulator

The simulator class “NrSimulator” contains the entire state of the processor core, the data memory and the program memory. It has methods to execute instructions, and set and retrieve processor state. It keeps the contents of the program and data memories in arrays of words. The values of the registers and I/O ports are stored similarly. It also keeps an array of unsigned integers the size of the number of words in the program memory where each integer signifies how many times the corresponding instruction word has been executed.

When an instruction is executed, the simulator first checks the status register for interrupt request or if the execution has been halted. If neither is the case, it uses the “findInstrType” method of the disassembler to determine what type of instruction it is. Depending on the type of instruction, a sequence of operations is performed on the variables containing the state of the processor to reflect the changes of the executed instruction. Typically it will involve a change to the program counter and an arithmetic operation whose result is reflected in the status register and destination register, or a memory or I/O operation.

#### 7.2.2.2 GUI

The GUI class contains the controls that display information to the user and receives input from the user. It also implements functions to load a new program or memory file. The functions handling the pushing of the buttons “Run” or “Run to cursor” start new threads from where the simulator methods to execute instructions are called. This is to allow the thread which runs the GUI window to continue to receive user input while the simulation is running. This enables the “Break” button to function as intended.

#### 7.2.2.3 Disassembler

The disassembler performs the opposite job of the assembler. It takes the instruction words as input and produces assembly code. This simple disassembler performs a direct translation of each instruction word individually. It will not necessarily reproduce the assembly code that produced the instruction word in the first place because it does not generate labels, aliases, macros or emulated instructions. The purpose is simply to be able to view the instruction word in a more human readable form.

# Part III

# 8 Test

The testing of the NanoRisc was performed in RTL simulation using ModelSim 6.0 by Mentor. The goal of the testing was twofold:

- Ensure correct functionality of the processor
- Make sure no logical errors were made in the design of the instruction set

To meet the first goal, a VHDL test bench and a NanoRisc program designed to cover the full functionality of the processor were made. To ensure that the second goal was met, a set of real-life applications were made and run on the processor.

## *8.1 Test Bench*

The test bench was written in non-synthesizable VHDL, and simulates a typical system surrounding the NanoRisc. It is implemented by a top module which instantiates the NanoRisc, a program RAM, a data RAM, an I/O input queue and an I/O output queue (see Figure 28 and Appendix C for source code).

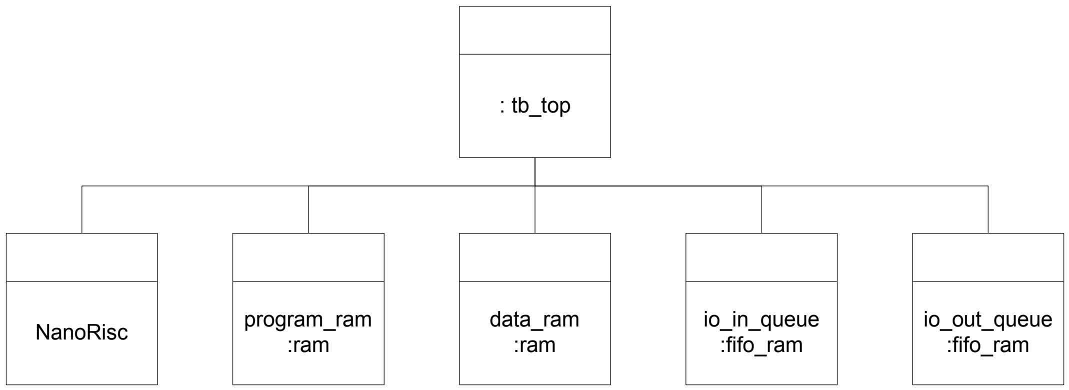


**Figure 28 - Testbench overview**

The test bench can load content into the RAMs on startup, and dump their contents to file when the simulation ends. The path of the program to run is thus specified in the "P_LOAD_NAME" generic. The VHDL functions to read data from file into a bit-array were re-used from a previous Chipcon project and should not be considered as a part of this project. The test bench has functions to halt and wake the processor, to simulate data-ram contention, to implement a data transfer handshake protocol over the I/O lines and to generate interrupts. Each simulation starts by resetting the processor, but which of the above mentioned functions are run subsequently is determined by the values of the corresponding boolean generics with a "_PROC" suffix. An example of an interrupt is shown in Figure 29 where the interrupt is acknowledged at the yellow line, and the interrupt vector is 0x0004.

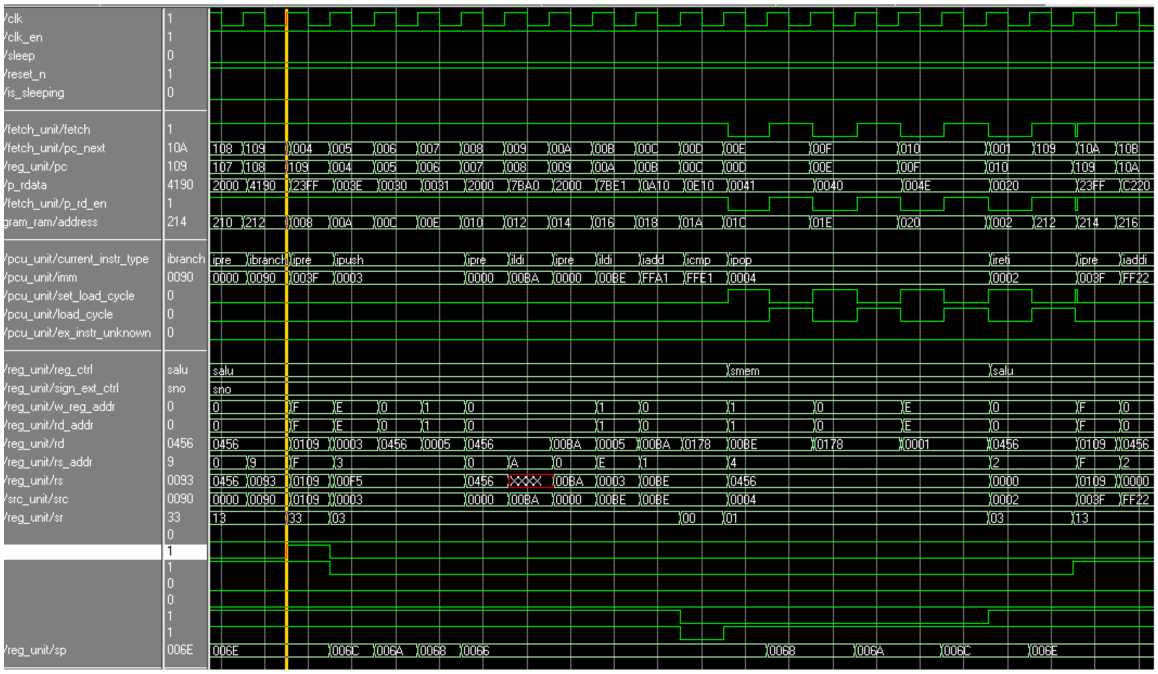


**Figure 29 – Interrupt simulation**

An example of the sleep functionality is shown in Figure 30. The NanoRisc enters the sleep mode when the "is_sleeping" signal goes high. The red bar represents a time-gap while the yellow bar is where the processor is wakened from the sleep mode by an interrupt.

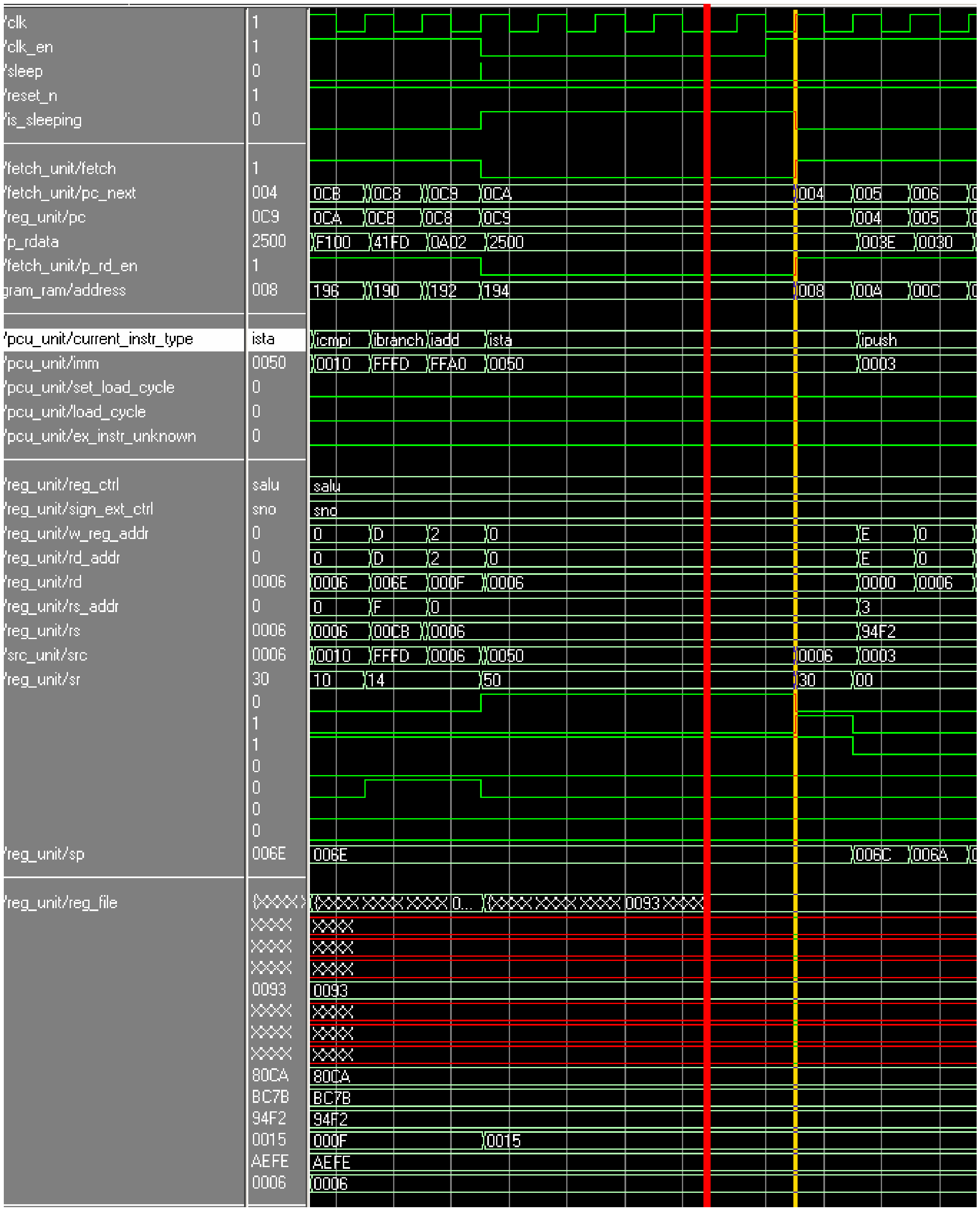


**Figure 30 – Sleep functionality**

## *8.2 Coverage program*

The coverage program was written systematically with the intent to cover as much of the NanoRisc functionality as possible in one program. It sets up and implements self-checking tests of each instruction in the instruction set. The example in Figure 31 shows a test of the “sto” and “ldo” instructions with preceding “pre” instructions.

```
    ;; Sto with PRE and Ldo with PRE
    ldi R5, 0x0004
    ldi R6, 0xFFDE
    sto.b R6, 30(R5)
    ldo.bz R7, 30(R5)
    ldo.bs R8, 30(R5)
    cmp R8, R6
    bne ERROR
    zxt R6
    cmp R7, R6
    bne ERROR
```

**Figure 31 - Example instruction test**

In addition to checking the functionality of each instruction, the test bench simulates data-memory contention, interrupts and halt/wake events during the execution of the program. The self-checking functionality is implemented by testing the results of the operation and jumping to the "ERROR" label if the result was not as expected. The results have also undergone manual inspection. The program consists of approximately 500 lines of assembly code and achieved 98.84% statement coverage. The remaining 10 uncovered statements have been thoroughly inspected and deemed not to contain any errors.

## 8.3 Quicksort Program

The quicksort program sorts an array of words in the data RAM using the quicksort algorithm as described in [Cormen2001]. The quicksort algorithm was implemented as a part of the testing because it tests recursive function calls, conditional branches and load/store functionality in a real-life application. The results of the testing were uplifting.

## 8.4 CRC Program

The cyclic redundancy check algorithm (CRC) is an algorithm used to generate checksums used to detect error in transmission or storage of data. The CRC-16 variant implemented here produces a 16-bit checksum to be appended to its data. It is used for example in the Universal Serial Bus (USB) protocol. The CRC program implemented reads the data from which to generate the checksum as bytes from an I/O port using a handshake protocol. The checksum is calculated upon the reception of each byte. When it is told that all data has been received, it starts transmitting the two generated CRC bytes over the I/O lines using the same handshake protocol as when receiving the data. This program tests the I/O functionality and data processing. An interesting property of the CRC algorithm is that a CRC checksum calculated from data with its checksum appended to it will give a zero checksum. This property was used to verify the correctness of the algorithm implemented. Performance wise, it would have been much more effective had the I/O port width been 16-bit. Still, it calculates and transfers 16-bit CRC from 40-bits of data in 500 cycles.

## 8.5 16-bit Multiplication

A “multiply16” function was implemented which performs a software shift-add multiplication of two 16-bit numbers and stores the result in two general registers. It tests the code size for the NanoRisc. The NanoRisc is able to perform a 16x16 unsigned multiplication in 112 cycles and with a code size of 12 words, while the AVR for example, uses more than 153 cycles with a code size of 14 words [AVR1997]. In terms of speed, however, the NanoRisc is capable of performing a 16x16 multiplication in 19 cycles using 19 words of code size by the use of four 8x8 multiplications, shifting and adding.

## 8.6 SPI

To test the bit-banging capabilities of the NanoRisc, a set of functions to implement the Serial Peripheral Interface (SPI) master were written. SPI is a loose standard for a full-duplex serial bus that is slow but cheap. It uses four lines:

- “sck” – clock
- “miso” – master in slave out data line
- “mosi” – master out slave in data line
- “nss” – not slave select

For details on the protocol, the reader is referred to [Kalinsky2002].

The implementation for the NanoRisc has a clock signal with 50% duty cycle, and still achieves a throughput of 454 KB/s at 10 MHz each way while the AVR for example achieves only 444KB/s at the same speed with a skewed duty cycle [Atmel2002]. The great advantage of the NanoRisc is the shift into carry and set I/O bit to the value of carry (“iosc”) instructions in combination to implement efficient bit-banging, and testing showed that this worked flawlessly.

# 9 Results

This chapter describes the results achieved when synthesizing and testing the NanoRisc. The results are summarized in Table 12.

**Table 12 – Results overview**

| | |
|---|---|
| Area | 4362 gates |
| Power Consumption | 0.04 mW/MHz |
| Performance | 0.9 MIPS/MHz |
| Max Frequency of Operation with minimum area | 50 MHz |

## 9.1 Synthesis

The synthesis of the design was done using the Synopsys Design Compiler with the Virage Logic TSMC 0.18um FSG DUS Standard Cell Library. The synthesis was done at two clock speeds: 25 MHz for low power applications and 100 MHz for high performance applications. The results below are reported for these two frequencies separately, however, in keeping with [Salminen2004], both power and performance are reported for each frequency. The synthesis scripts were constructed by modifying standard Chipcon synthesis scripts.

Due to the lack of an instruction register, the timing of the arrival of program data is crucial to the design. To get a timing model for the program memory, a high-speed single-port synchronous diffusion ROM of 2048x16 bits was generated by an Artisan ROM generator. In a typical process with typical conditions, it has an address setup time of 0.31 ns and an access time of 1.29 ns. The address setup time is the time the program address has to be stable on the lines before the rising clock edge, and the access time is the time from the rising clock until the read data is stable on the data lines. These constraints were conveyed to the synthesis tool using the "set_output_delay" and "set_input_delay" statements respectively.

The design compiled is the NanoRisc core with 13 general registers, 2 8-bit I/O ports in each direction and an 8x8 multiplier and a 10-bit program counter. The synthesis was optimized for speed.

#### 9.1.1.1 Area

Clock gating is first and foremost a power-saving technique which consists of gating out the clock of a unit which is not used. In this design, it consists of turning off the clock for arrays of registers that are not to sample new data. This approach can save power while minimizing clock skew [Wakerly2000]. Design Vision studies of the registry control unit revealed that instead of gating away the clock to the last 8-bits of each register when writing just a byte, the synthesis had inserted a 2-multiplexer in on the input of these 8-bits. This multiplexer chose between current flip-flop output value and the new input value. Gating the clock would simply be an "and" gate in comparison. The clock gating was inserted using parameters to the synthesis tool, and it turned out not only to save power, but also area in the register module.

In the register module, the reading of the registers is currently implemented by a 16-bit multiplexed bus. An implementation using a tri-state read bus could have used a smaller area. A tri-state bus is a bus where an output connected to the bus can be either in a low, high or high impedance (Hi-Z) state. In the Hi-Z state, the output behaves like it is not connected to the bus. It is thus very important that only one output on a line is in one of its non-Hi-Z states at one time [Wakerly2000]. The advantage is that a tri-state bus read solution takes less space, but on the other hand it uses more power and is harder to debug than a multiplexed bus. It is due to this last point that Chipcon has a strict policy against the use of internal tri-state buses, and hence, it has not been used in this design.

Another possibility for reducing the size of the register module would be to use latches in stead of flip-flops to implement the registers. A latch is typically less than one half the size of a flip-flop. While a D flip-flop only updates its value on the rising edge of a clock, the D latch will change its value as long as its control signal is asserted [Wakerly2000]. The main problem with a latch implementation is the lack of possibility for scan testing. The idea of scan testing is to be able to drive the registers data input while a scan enable signal is asserted. By doing this, the contents of the registers can be changed by the user by shifting in a test pattern, the clock is then run for one edge, and we shift out the result and compare against the expected. The contents of the registers are then read out and compared to expected results. This is one of the main test methods at Chipcon, and therefore all registers have scan capabilities and the latch implementation was ruled out.

The area of a design will often become greater if it is to be run at greater speeds. This is due to timing constraints that need to be met, and the drive strengths of the components have to be increased to reduce propagation delays. A unit with greater drive strength takes more space in silicon. This explains the rather significant increase in area by some modules when going from 25 MHz to 100 MHz operating frequency as shown in Table 13.

**Table 13 - Area**

| Module | 25MHz Area | 100MHz Area |
|---|---|---|
| alu | 5069 | 6886 |
| fetch | 1071 | 1084 |
| io | 1853 | 1864 |
| mem | 1152 | 1014 |
| mul | 3626 | 5337 |
| pcu | 7389 | 12719 |
| reg | 13380 | 13788 |
| shift | 4055 | 4619 |
| src | 711 | 1390 |
| Total Area | 38306 | 48701 |
| Gatecount | 4 362 | 5 546 |

The synthesis for 100 MHz gave a significant increase in the size of the ALU and multiplier modules. It is probable that one would be able to achieve a smaller design at 100 MHz by using for example a carry look-ahead adder (CLA) or Booth multiplier [Smith1999]. These are larger modules in terms of the number of logic elements used, but they have shorter critical paths, and hence, the drive strengths of the logic elements might not have to be as great. This might in turn reduce the area consumed by the modules. At 25 MHz, the simple adder and multiplier used are fast enough and should be kept as they use a minimum amount of area.

### 9.1.2 Timing

Slack is the difference between the maximum amount of time a signal can use to be propagated along a signal path for the signal to still arrive on time and the actual amount of time it takes. If the slack for a path is positive, it means that the timing goal for that path is met, while if it is negative, the goals have not been met [Synopsys2004].

At 25 MHz, the path with the lowest slack is a path where the data is required after 40.42 ns and arrives after 20 ns giving a slack of 20.42 ns. This path is from a program memory data line, through the PCU, through the I/O unit, through the ALU and into a general register. This is a fairly large slack, and shows that the frequency of operation could be increased significantly without having to make changes to the drive strengths. In fact, further synthesis showed that the NanoRisc could operate at 50 MHz without increasing the area (see synthesis report summary in Appendix F).

At 100 MHz, the design was optimized and drive strengths increased until the critical path had a slack of zero. This path is from a program memory line, through the PCU, through the shifter, through the ALU and to the status register. Any further increase in the operating frequency would lead to a larger area for the design.

### 9.1.3 Power

An estimate of power consumption in running mode, excluding memories, was calculated using Power Compiler by Synopsys. The Power Compiler estimate is based on an estimate of the switching activity in the processor, and should therefore only be considered as ballpark figure. The switching activity was estimated to 50% except for the clock which switches every cycle and the clock enable signal which is always high. When running at 25 MHz and 1.98V, the power consumption of the processor is in the order of 0.9 mW. When running at 125MHz, the power consumption of the NanoRisc is the order of 4.8 mW.

In the halted mode, no signals are toggling, and the power consumption is deemed only to be the leakage power [Salminen2004]. The Power Compiler estimate for the NanoRisc design is a leak power of $5.5*10^6$ pW at 25 MHz and $7.5*10^6$ pW at 100 MHz.

## *9.2 Performance*

As shown thoroughly by John L. Hennessy and David A. Patterson in “Computer Architecture: A Quantitative Approach”, the measurement of relative performance of two processors is difficult. A direct comparison of clock speeds does not take into account the effective number of instructions being completed in those clock cycles. The number of million instructions per second (MIPS) is a popular metric that can be very misleading when comparing processors running different instruction sets because some architectures have instructions that perform in on instruction what another architecture would use several instructions to do [Hennesy1996]. An example of this is the "decrement and branch" instruction found in many architectures which performs in one instruction what another architecture would use both a decrement and a branch instruction to do. However, when comparing load/store RISC instruction set architectures, they are more comparable with respect to the amount of work being performed per instruction, and hence, the MIPS measurement becomes more indicative of performance than when comparing a RISC architecture to a CISC architecture for example.

The NanoRisc 25 MHz version is chosen for the calculation of MIPS because it has low size, reasonable power consumption and is the frequency at which it most probably will be implemented. MIPS can be expressed as:

$$\text{MIPS} = \frac{\text{Instruction Count}}{\text{Execution Time} \times 10^6} = \frac{\text{Instruction Count} \times \text{Clock Rate}}{\text{CPU Clock Cycles} \times 10^6}$$

The cycles per instruction (CPI) is a measure of how effective the cycles of a computer are. It is given as:

$$\text{CPI} = \frac{\text{CPU Clock Cycles}}{\text{Instruction Count}}$$

This gives:

$$MIPS = \frac{ClockRate}{CPI \times 10^6}$$

There are two reasons why the Instruction Count deviates from the number of CPU Clock Cycles in the NanoRisc and gives a CPI greater than 1:

- Memory reads take two cycles
- "pre" instructions are not counted as instructions

The CPI can be calculated by running a program in the ISS and using the values for "Cycle" and "Instructions". The test programs for CRC calculation and Quicksort sorting (see chapter 0) were used as representative programs for the calculation of CPI. A normal run of the CRC calculation of 4 bytes of data gave a CPI of 1.004 while the memory intensive quicksort program gave a CPI of 1.226. Weighting the two applications equally, this gives an average CPI of 1.115. At 25 MHz, this gives a MIPS value of 22.42. This again gives 0.9 MIPS/MHz.

The performance of a microcontroller core, however, is much more than speed. It is usually a function of speed, size and power consumption. These factors should be weighted according to the application when deciding which microcontroller core to use [Vogelin2002]. A common measure of microcontroller performance is how fast it is compared with its power consumption. The NanoRisc will give approximately 0.05 mW per MIPS, 20000 MIPS/Watt or 0.04 mW/MHz.

# 10 Discussion

This chapter discusses some of the more interesting problems encountered in the design and implementation of the NanoRisc processor and its tools.

## *10.1 Instruction set*

The instruction set of the NanoRisc is the result of over a month of intense work. The instruction set is the aspect of a processor that is most visible to the user, and hence it is important to bring the user in on decisions concerning its design. Through deliberation and meetings with the potential users of the processor, the NanoRisc instruction set was defined. Some of Van de Goor's [VanDeGoor1989] fundamental characteristics of a good architecture used as a basis for the first part of the discussion: consistency, completeness and open-endedness.

### 10.1.1 Consistency

The following quote from Carney's *Introduction to Symbolic Logic* as quoted in [Dorp1994] can be inspiring for the designer of an ISA: *"Consistency is essential; soundness is needed; completeness is most desirable, though not always obtainable; independence is obtainable, aesthetically pleasing, and can reduce one's labors in metalogic"*

The instruction set of the NanoRisc is not as orthogonal and "aesthetically pleasing" as one could have hoped. It contains 13 different instruction formats, and different instructions for all three address spaces. By consistency Van de Goor means freedom from irregularity and that a system is consistent if partial knowledge of the system permits you to predict other things about the system [Clements2004]. The NanoRisc instruction set is despite all efforts, however, not as consistent as could be desirable for an instruction set. It is, however, highly optimized for code compactness and only 909 out of 65536 possibilities are not exhausted. The high number of instruction formats is reflected in that the PCU represents about 20% of the total design area. This was found to be acceptable, as a more orthogonal and consistent design would mean that certain instructions take up more space than necessary, leaving less room for large immediate values. According to numbers for the Vax architecture found in [Hennesy1996], 35% of all instructions in integer programs are immediate instructions. Integer programs are the type of programs meant to be run on the NanoRisc. For the integer program/benchmark "gcc", an immediate field of 4-bits would be sufficient to represent the immediate value in 50% of the immediate instructions, while an immediate field of 7-bits is sufficient in 70% of the immediate instructions. This means that 20% more of the immediate instructions would need "pre" instructions if the immediate field was 4-bits instead of 8-bits. This would further result in an approximately 7% decrease in code size by having 8-bit immediate values instead of 4-bits. Such an decrease in code size probably makes up for the any reduction in the size of the PCU one would see as a result of a more orthogonal instruction set with 4-bit immediate values. As [Clements2004] stated it: "maximizing the bang-per-buck runs counter to consistency". Though there are many instruction formats, the immediate values are to the largest extent possible aligned to avoid extra multiplexers which would further increase the size of the PCU. While the instruction set is clearly not orthogonal in terms of addressing modes, it is orthogonal in the sense that there are not many combinations of instructions with which to achieve a specific result.

The architecture could be called orthogonal in register addressing, however as none of the general registers have special functions. Features like R0 = 0 is better for separate operand and destination register ISAs because it would enable one to load an immediate by running “addi R1, 0, value”. The special registers of the NanoRisc are, however, mapped to the same address space as the general registers. This is done to avoid separate “move special register” instructions.

### 10.1.2 Completeness

The term “completeness” as used by Van de Goor with respect to instruction sets is that a class of instruction contains all its members. The NanoRisc ISA includes all primitive arithmetic operations except division and “subtract immediate”. Division is not included because it is rarely useful in the control-oriented applications that are to be implemented in the NanoRisc, and requires a large amount of area. The “subtract immediate” instruction is not included because it can be emulated by an “add immediate” using the negative value of the immediate constant, thus saving space in the instruction set. The branch conditions used are the same as in the Texas Instruments MSP430 and are “complete” in the sense that they allow all useful comparisons on integer numbers.

### 10.1.3 Open-endedness

The criteria of open-endedness means that the architecture is capable of future expansion and development. This criteria is not immediately fulfilled by the NanoRisc as only 909 possibilities are left unused in the instruction set. It is, however, readily modifiable. The I/O functionaliy can be removed in applications where one, for example is forced to use memory-mapped I/O, and the freed up space in the instruction set could be used for new instructions. The drawback, however, is that the design would have to be verified and the tools would have to be modified accordingly.

### 10.1.4 Portability

Portability of existing programs is often an important issue when designing new instruction sets as exemplified by Intels x86 architecture[1]. This, however, was not an issue for Chipcon, as the NanoRisc will not replace another processor for which the programs are already written, but hardware. The design of the NanoRisc is therefore free of compatibility compromises.

### 10.1.5 I/O Addressing

The addressing of the I/O ports was a difficult issue to conclude on. The simplest aspect would be to interpret the address as an input port for read operations and interpret it as an

[1] Intel has built each new architecture on top of the old architecture to maintain compatibility (8086, 80286, 80386, 80486, Pentium) [Clements]. This compatibility has not come without cost, and many would argue that Intel should have scrapped compatibility years ago to rid itself of past mistakes. More about the architecture can be found at http://en.wikipedia.org/wiki/X86.

output port for write operations. The complicating aspect is that the output ports should be readable in order to be able to be able to perform arithmetic operations on its contents without having to mirror the contents elsewhere. This means that one either has to have separate instructions for reading the output register or one would have to add an extra bit to the address of the port to indicate if it is an input port or output port. A separate instruction for reading an output port would have saved a bit in each I/O instruction but it would add two instructions to the instruction set by introducing separate versions of "rdio" and "iots". In terms of space in the instruction set, the separate instructions would have been more economical, but in the spirit of simplicity, the extended address approach was chosen. Any write operation to an input port is now ignored.

### 10.1.6 Decrement and Branch

An instruction that was considered for inclusion in the instruction set is the "decrement and branch" instruction. It would decrement a register and branch if the result of the decrement operation is not zero. This operation is perhaps not entirely in keeping with the RISC philosophy, but it would save one instruction in many loops, and could make a significant difference on performance. It turned out, however, that it would not be possible to include this instruction as it would have to include a branch target, and a register address which would consume 4096 possibilities, and would not fit in the instruction set. Load and store with auto-increment/decrement are perhaps not entirely in keeping with the RISC philosophy either, but were included because they have a great potential for reducing the number of instructions needed in a loop and fit in the instruction set.

### 10.1.7 Status Flags

The overflow, carry, zero and negative flags of the status register and the corresponding condition codes are similar to the schemes used in the MSP430, the AVR and many other architectures. It was chosen mainly because it defers the decision on whether an arithmetic operation was signed or unsigned until the conditional branch evaluation. It means that the condition code has one extra bit to indicate if it should evaluate the result of the previous arithmetic operation as signed or unsigned. This adds one non-opcode bit to 8 branch instructions of 11 non-opcode bits. Setting the signed/unsigned value for each arithmetic operation would demand 1 extra non-opcode bit on 8 instructions of 12 non-opcode bits and 8 instructions of 8 non-opcode bits which would use far more space in the instruction set.

### 10.1.8 Byte Operations

The instruction set provides byte versions of load and store instructions to allow one byte to be changed in memory and allow any one byte read from memory to be correctly placed in a register by one instruction. There are, however, no byte versions of arithmetic, logical and shift instructions. This is partly because it would increase the complexity of the architecture but mostly because it would take up too much space in the instruction set as data processing of bytes can be easily implemented by sign or zero extending them to words.

### 10.1.9 Immediate Memory Addressing

Paging as discussed in the requirements specification refers to the technique of specifying a current "page" or portion of memory by storing the current most significant bits of the address in a special register. By using paging, one only has to specify the least significant bits for each memory access within the same page. The disadvantage is that it complicates programming because the programmer constantly needs to consider the contents of the paging register and when to change it. As paging was not an option, another scheme had to be used to allow direct addressing of the full 16-bit memory space. One option is to use variable size instructions, which means in this case that a memory access instruction with direct addressing would be a 32-bit instruction with a 16-bit immediate field. The disadvantage of this is that the most significant bits of the address would have to be repeated for each access to the memory even when they are within the same page. The prefix instruction approach chosen in the NanoRisc is very similar, and has the same disadvantage. However, it simplifies the instruction set as the "pre" instruction can be applied to all instructions with an immediate value without having to use different opcodes for long instructions.

### 10.1.10 General

The impression of the NanoRisc after using it to implement the programs as discussed in chapter 0, is that it is complete and efficient. It is complete because it implements the necessary primitive arithmetic and logical and branch operations. It is efficient in terms of code compactness because it allows variable sized immediate values while limiting the need for "pre" instructions. Its efficiency in terms of speed is not so much defined by the instruction set because it is a RISC instruction set. However, the powerful load/store with increment/decrement will enable the user to save valuable cycles in a tight loop.

## *10.2 Architecture*

During the development of the architecture of the NanoRisc, many difficult decisions and trade-offs had to be made. It was sometimes difficult to predict the exact implications of a decision on the performance, programmability and area of the NanoRisc, but they were all analyzed thoroughly, and some of those analyses follow in this section.

### 10.2.1 Bit-width

One of the most important dilemmas encountered in the design of the NanoRisc was to decide the bit-width of the architecture. A wider bit width enables more data to be processed or moved in one operation and simplifies programming, while a narrower bit-width reduces the size of the design and may potentially increase the frequency of operation. Early gate-count estimations without taking into account control logic showed that an 8-bit implementation would take around 2900 gates, a 16-bit implementation would take 3300 gates and a 32-bit implementation would take 4200 gates. It was decided that a 16-bit architecture would be sufficient to handle most data types used in the intended applications such as 16-bit memory addresses, packet lengths and 16-bit CRC while still staying comfortably within the required gate-count limits.

### 10.2.2 Instruction Read

One of the less common solutions in the NanoRisc is the complete absence of pipelining and the lack of an instruction register. The instruction word is executed directly from the data lines on the program memory bus. The idea for this spurred from reading the data sheet for the ROM to be used. Access times indicated that the program data would always be stable on the lines well in time to propagate through the NanoRisc before the next positive clock edge. The advantage of this is that by using a synchronous ROM, there will be no need for prefetching logic, and hence there will be no branch penalties. The downside is that the design will have to run with an about 2 ns longer clock cycle than it if the instruction had been prefetched. The first application of the NanoRisc will be a packet control function in an upcoming Chipcon transceiver which is specified to run at a 36 MHz clock, while the NanoRisc can run at up to 50 MHz with minimum area. This means that in the foreseeable future, the speed of the NanoRisc will be adequate without an instruction register.

### 10.2.3 ROM

There are mainly three different types of ROM available for an implementation of the NanoRisc processor:

- *Diffusion ROM* – This is a very cheap ROM in terms of bit-density. It is located in one of the lower layers of the wafer so that it is rather expensive and time-consuming to change its contents in production.
- *Metallic ROM* – This is still relatively cheap, but not as cheap as diffusion ROM. It is located on the top layer of the wafer, and its contents can be changed even in the late stages of production.
- *One-Time Programmable (OTP) ROM* – This is expensive in terms of area, but can be programmed once after production. Allows user programming of ROM.

A diffusion ROM was chosen as a model when determining constraints for the synthesis as this was believed to be the most probable ROM type to be chosen for an implementation.

### 10.2.4 Memory Space

The NanoRisc has a possibility of sharing memory space between the program and data memories as discussed in section 5.10. The Von Neumann architecture is an architecture in which a single storage unit is used to hold both program and data. In a predominantly single-cycle execution processor like the NanoRisc, the memory becomes a true bottleneck because the processor will attempt to fetch an instruction word from the memory on almost every cycle, leaving little vacant capacity for memory data operations. This speaks to the advantage of a Harvard architecture with separate memory busses and physically separate memories for program and data [Arnold2001]. In addition to this, ROM is cheaper and consumes less power than RAM, and since data memory has to be implemented in RAM, it would be better to have program memory separate so that it can be implemented in a ROM.

### 10.2.5 Load Data Hazard

A potential data and structural hazard is introduced when reading from data memory. When setting up the address in one cycle, the data will not be ready before the next cycle. The affected instructions are the load instructions, the return instructions and the pop instruction. For the load instructions, for example, if one allowed a new instruction to start executing in this second cycle, one would have a data hazard because it may attempt to read the destination register of the memory read which has not yet been updated. A structural hazard arises because the instruction might attempt to use the register file write port that the read from memory needs to write the result of the read. Possible solutions to these hazards are:

- The compiler (or assembler) inserts "nop"s after memory reads where necessary. This might save cycles where there are no hazards.
- The processor always executes a "nop" after a memory read while the result is being stored. This will save program memory compared to the above mentioned solution, but will stall the processor for one cycle for each memory read.
- A bit in the memory read instruction sets whether or not it should use two cycles. This saves program memory, but uses instruction set space. Hazards are dependent on the nature of the following instruction, and will have to be analyzed by the assembler.
- A register locking mechanism will avoid data hazards by locking a register for reads when before the contents have been updated. This will require complicated logic and 1 extra bit per register. A separate register write port will have to be used to avoid the structural hazard.
- Perform any auto-increment/decrement on first cycle, and write load result in second cycle. The disadvantage is that a memory read always uses two cycles, and that a 1-bit state machine is required to avoid loading a new instruction on the second cycle. It does, however, avoid both hazards.

The latter approach was chosen for the NanoRisc because it requires a minimum of logic, does not increase program size and does not complicate the instruction set or assembler. It will always use two cycles for a load, but there are no stringent requirements on speed for the NanoRisc. A data memory cache could reduce most memory reads to one cycle, but it is not considered for the NanoRisc because it will be too expensive.

### 10.2.6 Stack

The call stack, where the program counter is stored on a call to a function, could be implemented either as registers in the processor, or in memory. An implementation in registers is faster as each return instruction would only take one cycle, but it would be more expensive due to the registers used. In addition, the depth of the stack would be limited by the depth chosen at synthesis, while its depth could be modified in software if it were implemented memory. Following this argumentation, the stack is implemented in memory in the NanoRisc with a dedicated stack pointer register.

### 10.2.7 Shifter

A full barrel shifter with rotate and "shift in from carry" capabilities has been implemented in the NanoRisc. This shifter takes 460 gates. Many small microprocessors implement shifters that can only shift one place in either direction. Given the rather large cost of the full barrel shifter, this would probably also have been the case with the NanoRisc had it not been for the I/O bit operations. The I/O bit operations use the shifter to generate bit-masks. An advantage of a full barrel shifter that is very useful for some of the intended applications of the NanoRisc is that it can shift a bit field from anywhere in a word to the least significant bits of the word in only one clock cycle. This can save many cycles in packet processing applications. A couple of things are sub-optimal in the way the shift instructions are implemented in the NanoRisc. Shifting in from carry can be useful, especially when implementing shift registers. However, shifting in from carry will in most cases only make sense when shifting by only one place. In the instruction set, consistency is chosen over instruction space in this case. Another option would of course be to implement a 16-bit carry register, allowing fast shifting of bits in two registers to handle word-misaligned bit-fields. This was, however, deemed too expensive compared to its usefulness. It should be noted as well that rotate right or left is only differentiated by the value the carry bit is set to, and it could be argued that one of the instructions could be dropped, however this would impair the functionality of rotating through a bit in the I/O space.

### 10.2.8 Registers

The general registers of the NanoRisc could either be implemented in a register file or as flip-flops. The general registers would only need one write port as there is only one destination register in each operation and the loads with auto-increment/decrement are implemented over two cycles. Such a single-port register file will in most cases be smaller than the corresponding dual-port register file. It turned out, however that for the relatively small register file needed for the NanoRisc, the single-port register file generated by the Artisan register file generator was actually larger than the dual-port register file. According to Chipcon engineers, this is a symptom of inefficient arrangement of the bit-arrays in such small register files. Due to the overhead of reading and writing logic, the area consumed by the register file surpassed the area needed to implement the registers with flip-flops. Had the register needs of the NanoRisc been twice as high, however, the register file would have been significantly smaller.

### 10.2.9 Interrupt

The interrupt scheme of the NanoRisc is as required "simple" but it is also highly flexible and extendible. An alternative scheme that was considered was to not automatically acknowledge interrupts, but rather implementing an interrupt acknowledge instruction. In this scheme, low priority interrupts would acknowledge the interrupt in the beginning of its handling routine, and high priority interrupts would not acknowledge the interrupt before it was done with the interrupt handler. However, this requires an extra instruction and an extra cycle for each interrupt handler, so it was not implemented. It could be argued that the status register should be pushed to the stack automatically similarly to the PC. This would require one more cycle to acknowledge interrupts and another state machine in the PCU. It was therefore not implemented.

The decision of big- or little-endian memory organization and handling of misaligned word accesses are the responsibilities of an external module as it is dependent on the memory being used. For most Chipcon applications, however, a simple module which ignores the lowest bit of the address on word access and uses a big-endian scheme would be suitable.

#### 10.2.10 General

Due to the requirement of low gate-count, the NanoRisc architecture implements a very limited functionality in some areas. However, it can perform all register-register and I/O operations in a single cycle, branch without penalty and write to memory in a single cycle while still running at 50 MHz.

## 10.3 Tools

As mentioned previously, the availability of high quality tools is very important to the success of a microprocessor. Even though the NanoRisc is only intended for in-house use, the existence of good tools for programming and debugging is an important criterion when project managers decide whether or not to use the controller. Programming and debugging tools are also very helpful in the testing of the microcontroller core design itself.

The assembler and the instruction set simulator are in most cases sufficient for writing small and simple applications. This describes most of the target applications of the NanoRisc. However, there is little in the NanoRisc architecture that would make it difficult to use it for more complex applications. One such application could perhaps be an on-chip implementation of the Zig-Bee MAC. It is currently being run on the Chipcon development boards using an AVR ATMega128 using about 24k of program memory. For such an application, other development tools would be needed as will be discussed in section 10.5.

## 10.4 Testing

The testing of the NanoRisc was limited to conceptual testing on the ISS and RTL simulation. Though these tests allowed for verification of much of the functionality of the NanoRisc, it would also have been useful to test the NanoRisc in hardware. An FPGA implementation of the NanoRisc with memories and peripherals was deemed too time consuming for this project. There is little to suggest that hardware testing of the NanoRisc should prove difficult:

- It has scan-enabled registers for debugging.
- It is fully synchronous to one clock edge (except resets).
- It contains no internal tri-states as they are difficult to debug.

## 10.5 Future Work

At the end of this project, there is still a lot of work to be done both on the processor and the tools before the NanoRisc could be included in one of Chipcon's projects.

### 10.5.1 Processor

The design should be verified and tested in hardware. This could be done in an FPGA. A debug interface should be designed for the NanoRisc. This could be as simple as allowing instructions to be run from a debug module without incrementing the program counter.

All branches and calls are implemented as PC relative jumps. This was done for consistency and to avoid an extra multiplexer in the "fetch" unit. In using the instruction set, it has, however, been found that when the relative indirect branches and calls are not handled by the compiler, it is in some cases rather difficult to maintain consistent values for them. Indirect branches and calls should probably be absolute in future implementations of the NanoRisc.

There is an inconsitency in the instruction set of the NanoRisc. As argued previously, the subtract immediate ("subi") instruction was not included because it could be emulated by the add immediate instruction ("addi"). However, the subtract immediate with borrow ("subci") instruction was included even though it could be emulated by the add immediate with carry ("addci") instruction by inverting the second input. Removing the "subci" instruction from the instruction set could free up space for future expansions of the instruction set.

The NanoRisc does not implement exceptions. This is mainly because were deemed not very useful when there was no operating system. However, for debugging of programs, it would be useful to be informed about any undefined instructions, stack pointer overflows, or unaligned memory accesses. This could be implemented as some sort of exception which would halt execution at least in simulation. With the instruction set being as compact as it is, however, the probability of a random word being an unknown instruction rather than being interpreted as a valid instruction is only 3.3%.

The data on the program memory data lines now give the state of the processor. This may potentially cause problems during the load cycles and on the cycle after a reset. A one-bit state machine should be implemented which indicates if the data on the program lines are valid or not. This would ensure correct behavior of the processor more independently of the behavior of the program memory module.

### 10.5.2 Tools

For assembly programming to be effective also for larger programs, the NanoRisc assembler should probably be expanded to include support for macros. Macros is a set of rules for text replacement. In assembly languages they are usually simple and often implemented by a preprocessor. They can be used to simplify assembly programming by simplifying the syntax of commonly used constructions. They can also be expanded to include conditional and loop statements to enable more powerful generation of code. The Gnu assembly pre-processor ("gasp") is now deprecated, but could be used to implement powerful macros [Pesch1994]. The assembler should also be expanded to use the GNU BFD library. This library provides a canonical interface to virtually all widely used output file formats such as "a.out", COFF, ELF etc. [BFD]. This would provide compatibility with linkers, debuggers etc.

The instruction set simulator does not today include any support for simulating the processor's interaction with peripherals other than memory. When using the simulator to debug an application which includes the NanoRisc, it is necessary to be able to make software modules to emulate peripherals. A system for connecting plug-ins should be implemented, perhaps using the Microsoft COM or .Net interfaces for flexibility. It should be possible to connect peripheral modules to both I/O ports and as memory-mapped modules. They should also be able to raise interrupts and set the processor in the sleep mode. Such a system would be a great help in verifying systems on a high level.

One of the first things that should be done for the NanoRisc is to write a "C" compiler. "C" is the de-facto standard for writing high level language programs for small microprocessors. The NanoRisc architecture is very "C" compiler friendly with a large number of general registers, and stack functionality. A frame pointer can be implemented in one of the general registers. The writing of a "C"-compiler was considered done as a part of this project, but writing it from scratch would then not be an alternative due to the time-limitation. Several toolsets for generating compilers were considered. The most widespread open-source compiler framework today is GCC (http://gcc.gnu.org). It is a framework for writing compilers who output assembly code. However, [Bolado2003] estimates a port of GCC to the OpenRISC architecture, for example, to 4 man-months. This would clearly not be possible within the time frame of this project, but is has been proposed as a project for a student at NTNU for this fall. See [Stallman1998] for an introduction to how to make a successful GCC port. The free, retargetable "C" compiler LCC (http://www.cs.princeton.edu/software/lcc/) was also considered, but it would also be too time-consuming to get operational. A C-- (http://www.cminusminus.org/) compiler vas also considered for the NanoRisc, but the project does not seem to have made much progress lately, and it is doubtful that this will ever become a standard.

## 10.6 Summary of Requirements Conformity

This section sums up how the NanoRisc conforms to the requirements specification (see chapter 3).

1. **Load/store architecture.** The NanoRisc implements a load/store architecture.
2. **Simple, orthogonal, high density instruction set.** It turns out that orthogonality and high density are rather conflicting properties of an instruction set. The NanoRisc instruction set is relatively simple and of high density, but it is not orthogonal.
3. **Data processing capabilities.** The NanoRisc implements a relatively complete set of primitive arithmetic and logical functions.
4. **8x8 Multiplier.** An 8x8 hardware multiplier and corresponding instructions are included in the NanoRisc.
5. **Stack.** The NanoRisc implements a stack in memory with corresponding instructions and a dedicated stack pointer register.
6. **16-bit memory interface, no paging.** The NanoRisc has a flexible interface to two 16-bit memory busses, and has a shared, byte addressable address space for

data and program. The entire address space can be addressed with immediate instructions without the use of paging.

7. **A parameterizeable sized I/O space with bit-operations.** The NanoRisc implements a parametrizable set of I/O ports of parametrizable width. Its instruction set includes powerful single-cycle bit-operations on these ports.
8. **Simple integrated interrupt controller.** The simple interrupt controller of the NanoRisc enables it to respond promptly to external events. It is flexible, but requires an external interrupt controller in most cases.
9. **Small footprint (2K-5K).** The minimum area implementation of a NanoRisc with a typical configuration takes about 4.5 Kgates.
10. **Power consumption of < 25 uW/MHz in the 0.18 um process excluding memories.** The power consumption of the NanoRisc is estimated to 4 uW/MHz in the 0.18 um process.

# 11 Conclusion

The study of the IP processor cores available in the market today revealed that existing cores did not provide the combination of size and performance required by Chipcon in this application. The NanoRisc turned out to be a well-performing processor who satisfies all the requirements. The design is less than 4500 gates and runs at 50 MHz, but its most impressive feature is that it performs all register to register and branch instructions in a single clock cycle without branch penalties. It will be able to provide Chipcon with the processing capabilities they need to run firmware in their future designs. The tools developed are stable and provides sufficient features for the writing and debugging of small programs. Though it needs some further work and testing, it was decided by Chipcon that the NanoRisc should be included as a firmware processor in an upcoming Chipcon transceiver.

# Bibliography

[ArcLite]
ArcLite 8-bit RISC Core
Accessed 21.04.2005
http://www.arc.com/upload/download/F1220.0_ARClite_4-4-03_FINAL.pdf

[ArcSupport]
Interrupt Latency
Accessed 29.04.2005
https://support.arc.com/techres/ArcSolveview.asp?id=262&product=43&release=&category=&keyword=&

[Arcx86]
x86 Series
Accessed 21.04.2005
http://www.arc.com/configurablecores/legacy/x86.html

[ARM]
www.arm.com

[ARMCortex-M3]
ARM Cortex-M3
Accessed 22.04.2005
http://www.arm.com/products/CPUs/ARM_Cortex-M3.html

[Arnold2001]
Ken Arnold
Embedded controller hardware design
LLH Technology Publishing, 2001

[AVR1997]
Multiply and Divide Routines
Application Note AVR200, 04.07.1997
Accessed 14.4.2005
http://www.people.cornell.edu/pages/mlk24/boom/code/avr200.asm

[Atmel2002]
Software SPI Master
Accessed 12.4.2005
http://www.atmel.com/dyn/resources/prod_documents/DOC3041.PDF

[BFD]
BFD
Accessed 14.05.2005
http://sourceware.org/binutils/docs-2.16/bfd/index.html

[Blaauw1997]
Gerrit A. Blaauw, Frederick P. Brooks, Jr.
Computer Architecture Concepts and Evolution
Addison Wesley 1997

[Bolado2003]
M. Bolado, J. Castillo, H. Posadas, P. Sánchez, E. Villar, C. Sánchez, P. Blasco, H. Fouren
Using Open Source Cores in Real Applications
Accessed 17.04.2005
http://www.escet.urjc.es/~jcastillo/paperdcis.pdf

[Bouldin]
http://microsys6.engr.utk.edu/ece/nasa03-bouldin.pdf

[CGEN]
http://sources.redhat.com/cgen

[Chipcon]
Chipcon
http://www.chipcon.com

[Clements2004]
Alan Clements
What is Computer Architecture?
Accessed 5.4.2004
http://www-scm.tees.ac.uk/users/a.clements/arch/arch1a.htm

[Cormen2001]
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
Introduction to Algorithms, 2nd edition
The MIT Press, 2001

[Dorp1994]
Hendrik Dick van Dorp
On an Extension of Functional Languages for Use in Prototyping, 22 december 1994
Accessed 5.4.2004
http://doc.utwente.nl/fid/1372

[Hennesy1996]
John L. Hennessy and David A. Patterson
Computer Architecture: A Quantitative Approach
Morgan Kaufman 1996

[Kalinsky2002]

David Kalinsky and Roee Kalinsky
Introduction to Serial Peripheral Interface
http://www.embedded.com/story/OEG20020124S0116

[LisaTek]
http://www.coware.com/products/lisatek.php

[McCorquodale]
Michael S. McCorquodale, Eric D. Marsman, Robert M. Senger, Fadi H. Gebara,
Matthew R. Guthaus, Daniel J. Burke, and Richard B. Brown
Microsystem and SoC Design with UMIPS
Accessed 21.04.2005
http://www.eecs.umich.edu/~mmccorq/research/mccorquodaleVLSI03.pdf

[Microsoft]
Microsoft
http://www.microsoft.com

[MIPS]
www.mips.com

[Nordic]
Nordic Semiconductor
http://www.nvlsi.no/

[Pesch1994]
Roland Pesch
GASP, an assembly preprocessor
http://www.ugcs.caltech.edu/info/binutils/gasp_toc.html

[Rosenberg1999]
Larry Rosenberg
What"IP"really means, 8/5/1999
Accessed 21.04.2005
http://www.edn.com/article/CA46033.html

[Salminen2004]
E. Salminen, K.Kuusilinna, and T.D. Hämäläínen
Comparison of Hardware IP Components for System-on-Chip
International Symposium on System-on-Chip, November 2004, Tampere, Finland, pp. 69-73

[Smith1999]
Douglas J. Smith
HDL Chip Design
Doone Publications

[Sony]
Sony PlayStation
http://www.playstation.com/

[Stallman1998]
Richard M. Stallman
Using and Porting GNU CC, 28 February 1998
Accessed 12.1.2005
http://hal.csd.auth.gr/thelug/faqs/gcc/gcc.html

[Synopsys2004]
Design Compiler Tutorial Using Design Vision
Synopsys Online Documentation (SOLD), 2004
Available at "http://mediadocs.synopsys.com"

[VanDeGoor1989]
A. J. van de Goor
Computer architecture and design
Addison-Wesley Longman Publishing Co., Inc., 1989

[Vogelin2002]
Stephen A. Voegelin
Early power estimates guide IP selection, 06.05.2002
Accessed 15.4.2005
http://www.eetimes.com/news/design/features/showArticle.jhtml?articleId=16505177&k
c=4235

[Wakerly2000]
John F. Wakerly
Digital Design: Principles & Practices, Third edition
Prentice Hall, 2000

[Weiss2002]
Alan R. Weiss
Dhrystone Benchmark, 01.09.2002
Accessed 22.04.2005
http://ebenchmarks.com/download/ECLDhrystoneWhitePaper2.pdf

[Wong2004]
William Wong
32-Bit Architecture Challenges 8-Bit Processors, 18.09.2004
Accessed 17.04.2005
http://www.elecdesign.com/Articles/ArticleID/8903/8903.html

[WongVa2004]

Stephan Wong, Stamatis Vassiliadis, and Sorin Cotofana
Embedded Processors: Characteristics and Trends, 03.2004
Accessed 5.4.2004
http://ce.et.tudelft.nl/publicationfiles/884_14_EP-CE-TR-2004-03.pdf

[Xap1]
Xap1 ASIC Processor
http://www.cambridgeconsultants.com/PDFs/asic/ASICs-SB-007.pdf

[Xap2]
Xap2 ASIC Processor
http://www.cambridgeconsultants.com/PDFs/asic/ASICs-SB-009.pdf

[Xilinx2004]
PicoBlaze 8-bit Embedded Microcontroller User Guide
June 10th 2004
Accessed 05.01.05
http://www.xilinx.com/bvdocs/userguides/ug129.pdf

[ZigBee]
The ZigBee Alliance
www.zigbee.org