University of Applied Sciences
Fachhochschule Offenburg

PIPELINING CONTROL OF A 32 BIT MICROPROCESSOR

David Berner

Advisor:
Prof. Dr.-Ing. Dirk Jansen
Reviewer:
Prof. Dr.-Ing. Werner Reich

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

ASIC Design Center
Badstrasse 24
77656 Offenburg, Germany
mail@davidberner.de

August 2002

Abstract

The ANTARES processor is a 32 bit RISC core for embedded systems developed at the Fachhochschule Offenburg. This work evaluates the existing architecture in order to extend it to work in pipelined mode. It describes the concept and the implementation of a suitable pipelining design and evaluates the architectural changes in order to integrate it. Adding pipelining to the ANTARES will speed up the overall performance of the system by two – three times.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Many thanks go to Prof. Dr.-Ing. Dirk Jansen who has been a great advisor to me. He contributed a lot of ideas and personal effort into the ANTARES project. It is just great to work with a professor who is not only willing to teach, but also to learn things.

Special thanks also go to Sujan Pandey, my faithful ANTARES fellow and friend.

I also like to thank the all the guys in the lab who supported me, Ingrid Lange, the secretary who kept many boring tasks away, Christoph Bohnert who always had an open ear when I needed someone to talk about some difficulties, and Simon King for proof-reading.

Finally I should thank Mathilda who assisted me in so many ways.

Chapter 1

Introduction

In the year 2002 a microprocessor does not really live up to its name any more. In terms of performance, complexity, and power consumption the word *Micro* just does not sound right at all. In the Formula 1 league of processors, the Intel Pentium 4, the AMD Athlon or the IBM Power 4 with speeds around 2 GHz, 1000 MFlops, nearly 500 pins, and 70 Watts rather make us think of terms like maxi, ultra or mega. Only maybe the size justifies the traditional naming? Not quite. 200 mm^2 and more make them look rather macro compared to the 12 mm^2 of the Intel 4004.

Is this the end of all "real" microprocessors?

No. The rapid growing market of mobile applications, emerging technologies such as pervasive and ubiquitous computing still demand for the original virtues of microprocessors: Small, cheap, strictly low-power while still offering high performance. The stars of this world of embedded systems such as the ARM7 or the Motorola DragonBall do not have the same prestige as the Pentium or Athlon racers. Nevertheless they do important work in your cell phone, personal digital assistant, car, mp3 player, camera, and washing machine, to name a few possibilities.

Though embedded systems are used in an unrelated field, many of the concepts developed for high performance processing can be translated to embedded

systems. These very same high-level concepts that often cause high-end processor designs to become significantly bigger, can - when applied with different parameters - help developers of embedded system cores to make these even more compact. One of these elementary concepts is **pipelining**.

The ASIC Design Center at the Fachhochschule Offenburg, University of Applied Sciences has conducted considerable research and development in the field of embedded systems. Within this research lay the emergent need for a better performing processor core than the Design Center has. The Design Center decided therfore to develop its own core. This RISC core for embedded systems was called ANTARES and will be optimized for running C-programs. It soon became clear that in order to optimize the system this processor should employ an instruction pipeline.

This project is about development and implementation of a pipeline concept that matches the given architecture and the evaluation of the architecture to find out what changes the architecture has to undergo in order to integrate it

Chapter 2

Pipelining

The pipelining principle is widely known outside high technology. Most modern-day production lines work in this way. Everyday examples can incorporate the basics of pipelining. A common example is the laundry process.

2.1 Laundry Example

In the laundry example we assume, that there are several loads of laundry to be washed. For each load of laundry we have to perform three steps (Figure 2.1, Figure 2.2).

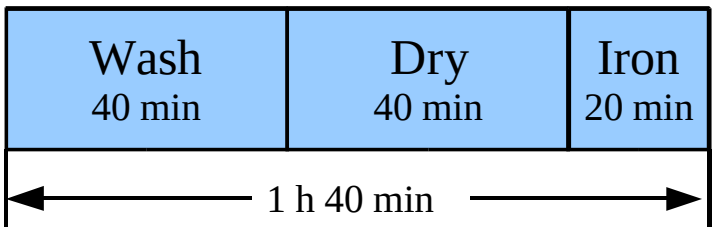

Figure 2.1. Timing for Sequential Laundry

The whole laundry process from dirty washing to clean and ironed takes about 1 hour 40 minutes. Anyone can see that there is still some room for improvement. Since the washing machine is not used during the drying process, after we have put its contents into the dryer, we can refill it straight away. While both

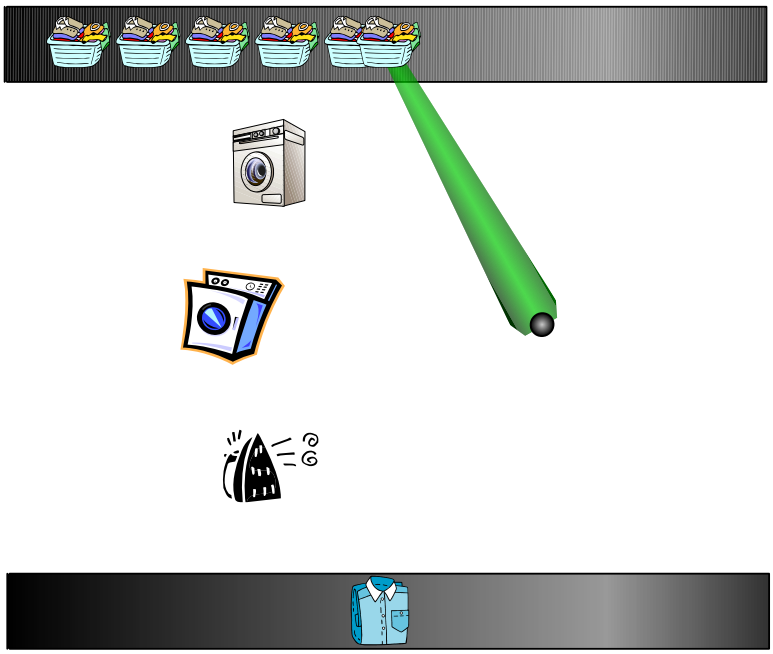

Figure 2.2. Schematic of Sequential Laundry Process

machines are busy we can start the ironing for another load that has just been taken out of the dryer. Figure 2.3 shows the schematic for this process.

The optimal throughput of this process would be one load every 40 minutes. This is the case when all units are busy. At the start and the end of the processing the throughput will naturally be somewhat smaller. When comparing the two laundry methods we must assume that the amount of loads to be processed is fixed. For four consecutive loads, a non-pipelined laundry method would take 6 hours 40 minutes, whereas a pipelined version would need only 3 hours 40 minutes to process the same amount.

2.2 Pipelining and Computers

The pipelining mechanism for computers dates back to the late 1950s. At the beginning of the 1980s it was still a technique solely uses in supercomputers. In the beginning of the 1990s pipelining made its way down to the microprocessors and

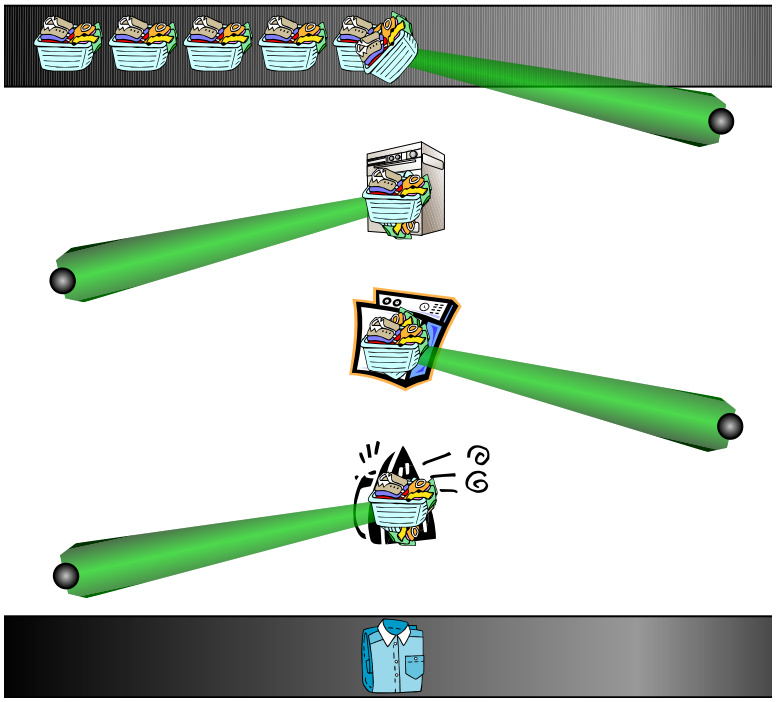

Figure 2.3. Pipelined Laundry Process

had a great success there [HP02]. Since then there is no serious high-performance microprocessor design that does not take advantage of it. For the Intel processors, the 80486 was the first using an instruction pipeline [Bre97].

In the microprocessor branch, pipelining means exploiting instruction level parallelism. Each instruction, even the most simple, consists of several stages. An instruction must firstly be read from memory (instruction fetch cycle, IF), it must then be decoded (instruction decode cycle, ID) and then has one or many execution steps (execution cycle, EX). These stages will use different units of the processor. A pipelined processor therefore utilizes these units concurrently and so increases the processor throughput. Multiple instructions overlap simultaneously.

Obviously the ultimate goal of pipelining is speed a processor up at little extra cost and power consumption. But what is the actual speedup of a pipelined design? For good results, the different pipeline stages should be balanced, i.e. they all should take about the same amount of time. Each stage should take one

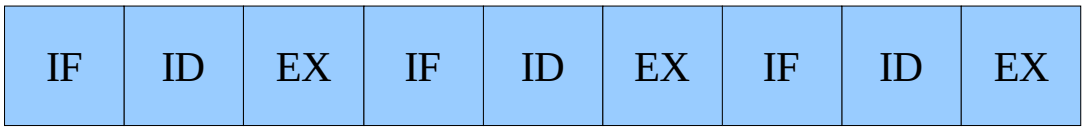

Figure 2.4. Execution without Pipelining

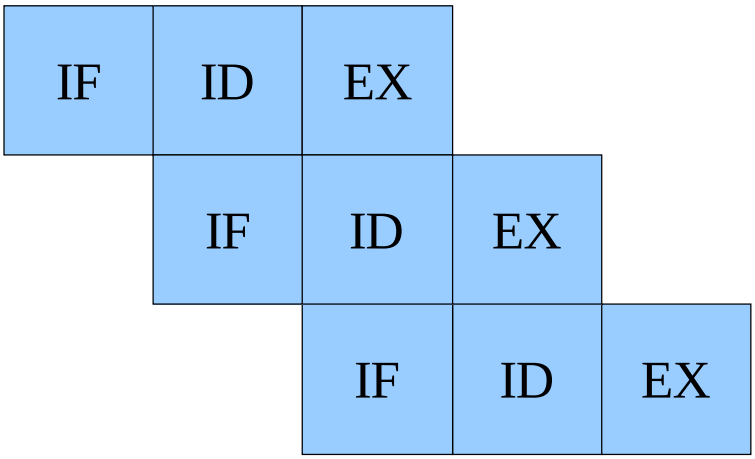

Figure 2.5. Ideal Pipelined Execution

processor cycle (which is the case for most designs). With these assumptions, we get an ideal

$$time\ per\ instruction = \frac{time\ without\ pipelining}{number\ of\ stages}$$

.

The throughput of the system will be n times higher, where n is the number of stages. That means the system will be n times faster. Of course practically this is never the case as the stages are not perfectly balanced.

Since one pipeline stage usually takes one cycle, the number of cycles per instruction (CPI) is a good way of measuring pipeline performance. The target of each pipelined design is an ideal CPI of 1.0. This can only be reached when all instructions consist of n stages and execution is uninterrupted. The instructions in most of today's designs differ in the number of their stages. It is obvious, that a 16-bit instruction can be processed faster than a 32-bit instruction. Another

impediment in pipelining is that not all instructions are really independent of each other. Dependencies between instructions can occur in many ways and will cause problems for optimal pipelined execution. The real speedup of a pipeline can be calculated as follows:

$$a_{pipelining} = \frac{execution\ time_{unpipelined}}{execution\ time_{pipelined}} = \frac{cycle\ time_{unpipelined} * \emptyset CPI_{unpipelined}}{cycle\ time_{pipelined} * \emptyset CPI_{pipelined}}$$

Where $cycle\ time_{pipelined} = cycle\ time_{unpipelined} + t_{overhead} + t_{balance}$.

Overhead time is the time needed for additional pipeline control and balance time is the time needed for balancing pipeline stages of different length.

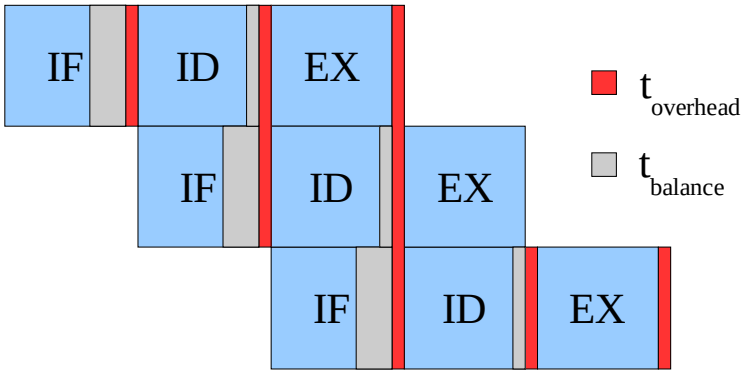

Figure 2.6. Real Pipelined Execution

2.3 Hazards

The problems that can occur during pipelining are called "pipelining hazards". They become more critical and more difficult to handle the longer the pipeline becomes. This is the reason why the number of pipeline stages must be limited to ensure efficient processing [HP90]. This section will describe briefly the basic types of hazards that can occur and how they are dealt with.

2.4 Structural Hazards

Structural hazards are a result of bottlenecks within the architecture of the system. They occur when the systems resources are inadequate for the demand. A system with only one data bus for both instructions and data alike (as in the ANTARES) will produce a structural hazard each time an instruction tries to load or store a value and the pipeline wants to fetch the next instruction in the same cycle. One possible solution for this problem is to introduce separate memories and buses for both instruction and data alike. Then both actions would be performed independently but at the same time. However, a designer does not always wish to avoid structural hazards by duplicating hardware. As a result he would often get a highly redundant system whose units are again idle for most of the time. The speed of the system will increase, but also at the same time costs and power consumption will rise.

The basic approach of pipelining is rather to use existing units more efficiently. So when a structural hazard occurs, we have to wait while one instruction is executed until the needed resources are free. This mechanism requires the insertion of wait states or NOPs. So long as pipeline wait states are intermittent, the system will still run close to optimal and the resources will be used much more efficiently. When there are many structural hazards (e.g. when each instruction requires two memory accesses), it may make sense to duplicate a unit in order to increase the usage of the other units.

2.5 Control Hazards

The pipelining principle is based on the assumption, that there is a steady flow of instructions. Of course, there are instructions that, by definition, interrupt this flow. Any kind of jump or function call or return instruction may want to resume operation at another location in the code. This causes a problem, since the processor will fetch the next instruction while decoding. Figure 4.6 on page 42 shows an example of a control hazard in the ANTARES processor.

The jump instruction may execute an instruction while the one that just has been fetched from memory. What has to be done is a so called "pipeline flush". The processing of instructions that have been issued after the branch must be discarded. The pipeline is then stalled until the target of the jump is known. While a pipeline is "stalled", no new instructions are issued. Once the target of the branch is known, the next instruction can be fetched and normal operation can resume (Figure 4.7). Depending on the length of the pipeline these control hazards can be quite expensive in terms of execution time. Because of that, there has been substantial research done in order to minimize the delay caused by branches in the code (penalty minimization). One very popular method is branch prediction. A certain logic tries to guess whether the branch is going to be taken or not and according to that result the code at the branch target or the code after the branch instruction is read. Another method is to start executing both code segments, and discard the one that is not used. Both of these solutions require important additional units. These result in more expensive development, production and increase power consumption. Since these approaches will give significant speedup only for longer pipelines where the branch target is known before the end of the branch instruction, it does not make sense to use them in a lightweight system like ANTARES.

Another, more simple solution is to assume, that the branch will be always taken, or always not taken. Depending on the system architecture and on the type of programs executed, there may be more taken branches or more branches that are not taken. For a three-stage pipeline as in the ANTARES (Section 4) the only alternative is to assume "branch not taken". This requires no additional units and will not cause any delay for about 50% of the branches.

Function calls and returns are - of course - always taken. There we have to look for other ways to optimize the resulting delay (Section 4).

2.6 Data Hazards

The third type of hazards that can happen are data hazards. These are the result of unresolved data dependencies between instructions. For example an instruction needs an argument for calculation which is not yet loaded from memory because the preceding load operation is still unfinished (Figure 4.9). In this case the original order of read and writes to the operands changes because of the overlap of instructions. Data hazards is the type of hazard, that is the most difficult to handle. They may result from a special combination of instructions while all other combinations work fine. If they are not detected, the execution will continue as if everything were ok, but it will produce a wrong result - which will be detected only much later or perhaps never.

Because of this, the detection of data hazards is crucial to the proper functioning of a processor design. In order to do this, some unit has to check if the argument of an instruction matches the target of a previously issued instruction that has not completed yet. If this is the case, it has to wait until the result is complete (Figure 4.10). The different types of data hazards can be classified as follows.

2.6.1 Read After Write (RAW)

This is the most common type of hazard. An instruction tries to read a result before a previous instruction has written its updated value (Figure 2.7). It always occurs when we have several execution cycles preceding a STORE instruction or if the result of an ALU operation is available only one cycle after it actually has been generated.

2.6.2 Write After Read (WAR)

An instruction tries to write a value, before the previous value can be read by another instruction (Figure 2.8). This can happen for combinations like a LOAD instruction after some long arithmetic instruction.

	t1	T2	T3	T4	T5	T6
STORE	INF	IND	DST1	DST2		
LOAD		INF	IND	DFT1	DST2	

Figure 2.7. Read After Write Example

	t1	T2	T3	T4	T5	T6
LOAD	INF	IND	DFT1	DFT2		
STORE		INF	IND	DST1	DST2	

Figure 2.8. Write After Read Example

2.6.3 Write After Write (WAW)

An instruction will tentatively write a value before a previous instruction has written at the same location (Figure 2.9). The newer value will be overwritten. This type of hazard occurs when there are several write cycles for one instruction or where an "out of order" execution is used.

2.6.4 Forwarding

For some architectures like the DLX [Mär01] the result of an ALU operation is only written to the register in the cycle after the last execution cycle (write back cycle). This architecture would produce RAW data hazards for all consecutive ALU operations that use a result of the preceding instruction. Obviously this would impede pipelining operation in an significant way. The most common approach around this, is to add some *forwarding logic* [HP96] that is able to put

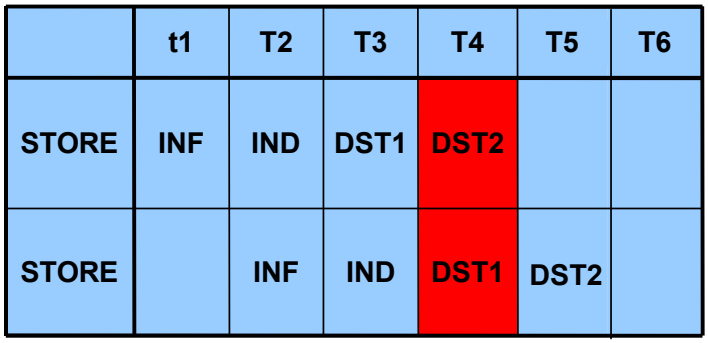

Figure 2.9. Write After Write Example

the result of the ALU directly into the input of the ALU when required. For the ANTARES such logic is not required since in any case it can use the result of a previous operation right away.

2.7 Implementation Approach

Knowing what pipelining is and what problems you are facing when dealing with it is one thing. Resolving them is another. So we have to look what different approaches for implementations are available. One is unfortunately confronted by the pauicity of solutions that can be realistically implemented. There is considerable literature available about advanced pipelining methods such as forwarding, out of order execution, branch prediction, dynamic scheduling, and speculation but actual solutions are not easy to find. Some implementations use registers to keep the information for the processing and then just shift the contents of the registers in order to advance in the pipeline [Mes96] [Web97]. This is a very simple solution and it avoids the trouble of having one Finite State Machine (FSM) with an incredibly high number of states, but it adds absolutely no intelligence to the system. There have to be special units that detect all exceptions and hazards that can occur and change the flow of processing accordingly. Again this can become very complicated, inconsistent and difficult to modify once implemented.

Also many solutions, especially the older ones, attempt to resolve the problem on a very low level. They transfer it into microcode and are designing the actual

schematic. While this may lead to a very efficient solution in terms of mm^2, it will become very inefficient in terms of design time, readability and flexibility. ASIC Design Center was seeking more a solution that matches the high level design ability of VHDL in order to make a design that is simple, consistent but still lightweight.

2.8 Summary

The pipelining principle permeates many everyday processes. It is well known for computers and there are only few basic problems that can occur, though many ways how to handle them exist in theory. The approach in overcoming pipelining hazards is elementary linked to the basic architecture of the system. There is great scope for improvement. Costs, performance, speed, power, size, flexibility, and manageability are the most important ones but many of them would suggest totally different approaches for the pipeline design. A pipelining design therefore always represents a compromise.

Also the handling of pipelining hazards become more and more complex with the length of the pipeline. Obviously the more steps you have, the more exceptions can occur and the greater the necessity for control. For the ANTARES processor the most important design goals are power consumption and manageability. Of course the other goals still have a certain role. Section 4 will evaluate more accurately what the right option for this case is.

Chapter 3

Antares

The ANTARES is a microprocessor developed in the ASIC Design Center at the University of Applied Sciences in Offenburg. ANTARES means **AN**SI C **Ta**rgeting **R**educed Instruction Set Core for **E**mbedded **S**ystems". This name is quite self explanatory, though you might wonder about the term "ANSI C Targeting" in it. Of course many new processors can be programmed in the C language. Even for embedded systems the higher level languages are becoming more and more important since the costs in making larger programs in an acceptable time-frame are becoming bigger and bigger. The special feature of ANTARES is that the instruction set architecture was developed in a way, so that C-programs would compile in a very compact manner and therefore would run very efficiently. The specialty of this approach was not to make a smart little processor, optimize it for its assembly code and then make a C-compiler for it and try to optimize the compiler for the platform. In the ANTARES project we tried to make the platform that was optimized for the C language in the first place.

Another reason for this order of proceeding becomes apparent when we look at what "designing a new processor core" entails. Figure 3.1 shows all the components of the ANTARES system. We can see, that the actual VHDL core is just one component of many. To have, for example, a simulator that works reliably and can be used comfortably is nowadays just as important. A general tendency is, that the software part will become more and more important while the hardware part becomes rather easier to design since the tools are getting better.

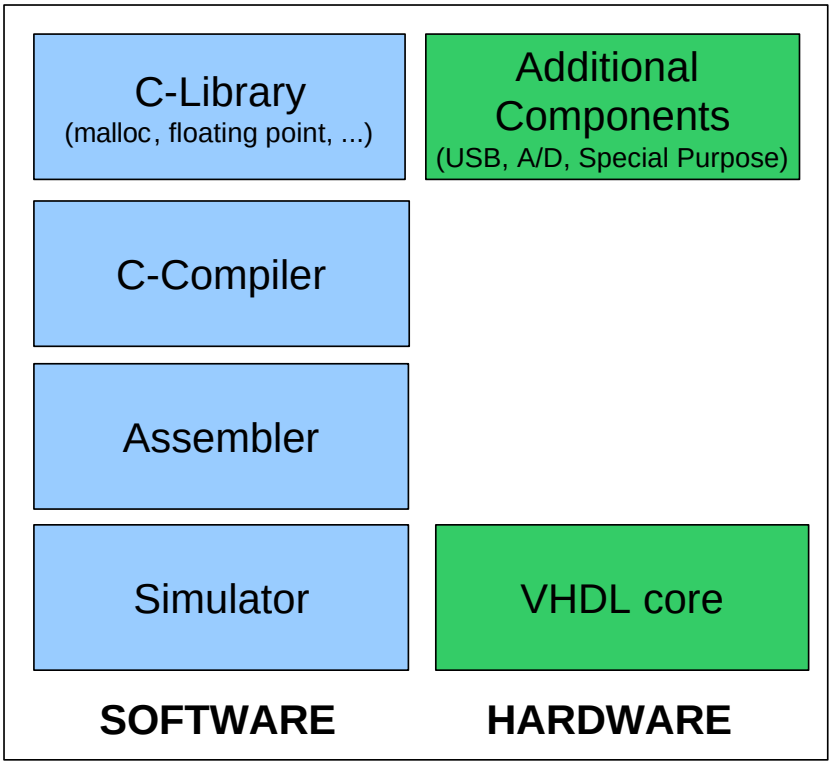

Figure 3.1. Components of the ANTARES System

Without having high level design tools like VHDL the world of microprocessors would look totally different today.

Since the software part has become more important it does not make sense any more to first design the hardware and then tailor all the software for it. Rather, it is better to start thinking about the software side and then try to implement a hardware that supports it in an efficient way.

As the subject of this thesis is the pipelining, we will focus on the VHDL part anyway. This chapter will provide some background about the actual architecture of the ANTARES as we found it. It will not try to explain all the details, these can be obtained at [PB01] and [Jan00] but only the most important facts and what we think is necessary for pipelining.

3.1 Architecture

When the architecture for the ANTARES was chosen, the focus was always on making a small, balanced system, suitable for embedded systems. Since the bulk of the targeted applications will have to handle 16 bit data, most optimizations will go in that direction. Above that, 32 bit should be still natively supported. The major decisions then were to have a 16 bit data bus and a strict 16 bit instruction format. In order to be able to address large memories without translation a 32 bit address bus is used. The 16 bit ALU as well as the register file with six 16 bit registers plus 32 bit indexed and stack registers are rather compact.

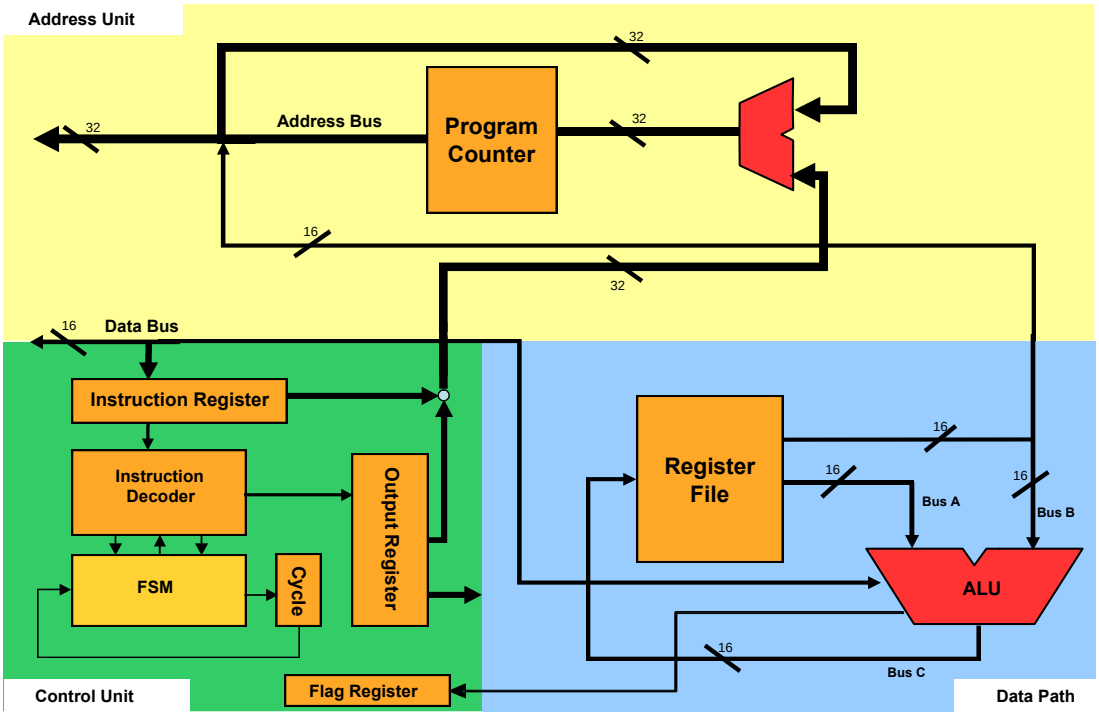

Figure 3.2. The Antares Architecture

Figure 3.2 shows a simplified schematic diagram of the overall ANTARES architecture. It may look heterogeneous at first glance, having a 16 bit data bus, 32 bit addressing, 16 and 32 bit registers and supporting both 16 and 32 bit instructions. When we try to validate the architecture on the background of a compact core for embedded systems though, it all fits together quite well and

forms altogether a compact and fast system. Of course we can see right away some restrictions such as the 16 bit ALU, the small number of registers and the 16 bit data bus. But these will only be apparent for 32 bit operation which is not the focus of our endeavors. For 16 bit operation however, these "restrictions" can in fact be seen as advantages, since they result in less code, less memory requirements, a smaller chip and therefore also less power consumption.

3.2 Instruction Set Design

The system architecture comprises the most critical decisions to take and often acts as a starting point. However this can not be regarded in isolation as the architecture is directly linked to the instruction set as well as the compiler. Actually we could regard the instruction set as the basis of the whole system, software and hardware. Virtually every behavior of the system is based in the instruction set definition.

One crucial decision was to maintain a strictly 16-bit instruction format. This makes the instruction set design much more challenging since it fundamentally restricts the coding space. However when we will succeed in making an efficient and complete instruction set that adheres to a 16 bit format, we will gain far greater efficiency. The whole of instructions are separated into six categories:

3.2.1 Instruction Coding

1. Type A: Instructions without extensions
   These instructions take no argument. These are the most simple instructions, mostly having to do with setting, resetting and checking flags. There is coding space for 256 instructions, but only 12 are actually used.

2. Type B: Instructions with 8 bit immediate field
   These instruction have an 8 bit immediate argument, but do not address any register. An example is *CLA 08Ah*. There is coding space for eight instructions and all of it is used.

3. Type C: Instructions working with one register
   These are unary instructions like all kind of shifts and compare zero. Again there is coding space for 256 instructions, but only 40 instructions belong to this category.

4. Type D: Instructions with two registers addressed
   This category combines all instruction with two register arguments like *ADD R1, R2*. There is space for 32 instructions. Currently we have 30 instructions of this type, the two remaining slots are reserved for 16 and 32 bit division which may be eventually implemented in hardware at some point.

5. Type E: Instructions with one register and an 8 bit immediate field
   In this group are most of the immediate instruction like *LDI 0A4h ABl*. The entire available space of 16 instructions is used.

6. Type F: Instructions with long offset
   This last instruction type carries a 12 bit immediate field. Therefore the coding space is so small, that there are only four instructions that can fit into this group. They are absolute and relative jumps (JMP and JPR), a subroutine call (CAL) and a load (LEA). Since in ANTARES all addresses are aligned, we do not have to include the last bit of the address in the coding. It is always zero. Therefore our 12 bit immediate field is effectively a 13 bit address where the LSB is implicitly zero. This enables us to address a quite large address space directly without having to use the prefix mechanism (Section 3.2.3).

Table 3.1 shows the instruction coding of all the categories. The "X" stands for the actual coding space for the instructions, the "R" means the coding of a register argument and "N" signifies the immediate information. The remaining bits are the Huffman coding for the instruction type. We can see, that Type E and F use the most expensive codes since they have to carry 14 bits of information. Overall, the coding space is used very efficiently. Only in Type A and D is some space left for further instructions. Table A.3 in Appendix A lists all instructions with their coding, syntax, functionality, and description.

Type	**Number Inst.**	**15 14 13 12**	**11 10 9 8**	**7 6 5 4**	**3 2 1 0**
A	256	0 0 0 0	0 0 0 0	X X X X	X X X X
B	8	0 0 X X	X 0 0 1	N N N N	N N N N
C	256	0 0 R R	R 0 1 0	X X X X	X X X X
D	32	0 0 R R	R 0 1 1	X X X X	X R R R
E	8	0 1 R R	R X X X	N N N N	N N N N
	8	1 0 R R	R X X X	N N N N	N N N N
F	4	1 1 X X	N N N N	N N N N	N N N N

Table 3.1. Instruction Coding

3.2.2 Operation Types

The instruction coding does already make a certain division of the instructions into different categories. While working with the system it makes sense to make a subdivision into categories that are a bit more differentiated. We created 16 categories of instructions and call them operation types. The operation type of an instruction gives information about the basic type such as LOAD, STORE, or CONTROL but also contains information about the number of registers used or the type of data that is handled (word or long word). Table 3.2 gives an overview over all operation types along with a short explanation. The operation type of an instruction is detected by the control unit and this information is used in the other units in order to make faster comparisons. For most evaluations it is sufficient to know the type of the operation, we do not always have to decide based upon the instruction itself.

Operation type	Description
CONTROL	for control operation.
INDXS1	push byte or word in the stack.
INDXF1	pop byte or word from the stack.
INDXS2	push long in the stack.
INDXF2	pop long from the stack.
REGREG0	16-bits ALU operation (one 16-bits register).
REGREG1	32-bits ALU operation (one 32-bits register).
REGREG2	16-bits ALU operation (two 16-bits registers).

REGREG3	32-bits ALU operation (two 32-bits registers).
LOAD0	loading byte or word or long from mem. with direct addressing mode.
LOAD1	loading byte or word from mem. with index and stack relative addressing mode.
LOAD2	loading long from mem. with index and stack relative addressing mode.
STORE0	storing byte or word or long in mem. with direct addressing modes.
STORE1	storing byte or word in mem. with index and stack relative addresing modes.
STORE2	storing long in mem. with index addressing mode.
JUMP	jump absolute and jump relative with condition.

Table 3.2: Categorization of Operation Types

3.2.3 Prefix Mechanism

There are two different ways of using immediate operations. When the 8 or 12 bit immediate value is enough we do not get any problems. In some cases (especially for addressing) we even can add the LSB implicitly, which lets us effectively use 9 or 13 bit respectively. Obviously we need another way to do the same operation for all cases when this is still insufficient. This kind of problem is, of course, an issue of the overall architecture, but it has to be resolved within the instruction set definition. We need instructions that allow us to address the entire address space.

The solution is to load a prefix register beforehand with some value and concatenate this one with the actual immediate constant. There are three different prefix operations: LPR, LPL, and LPH. Each of them takes an 8 bit value and

Figure 3.3. Composition of the 32 bit Prefix Register

positions it in a certain prefix register location. Figure 3.3 show the composition of the prefix register. So with the help of these three commands plus the immediate field, we can use an entire 32 bit number. In order to make an absolute jump to the upper regions of the 32 bit address space we have to use four instructions. This may seem quite inefficient, but as the bulk of all jumps and addressing operations will be close by, this is entirely practical. The closer a jump target is, the less processor cycles will be wasted. That also means that close addressing operations will be very fast, which is much more important.

The insertion of the prefix commands is transparent for the programmer as soon as he uses the C language. The assembler in combination with the linker automatically inserts these instructions whenever it is needed.

3.3 Data Path

The data path is the central processing part of the core. For ANTARES it is kept rather compact. Its basic components are the ALU and the registers. The ALU is connected to the registers via two 16 bit input busses (A-bus and B-bus) and one 16 bit output bus (C-bus). The data bus is connected to the ALU. It is read via the C-bus and written via the A-bus. Later (Section 4) we will see that this is one major hinderance for pipelining. The B-bus is also connected to the addressing unit via a multiplexer. Figure 3.4 shows the overall schematic.

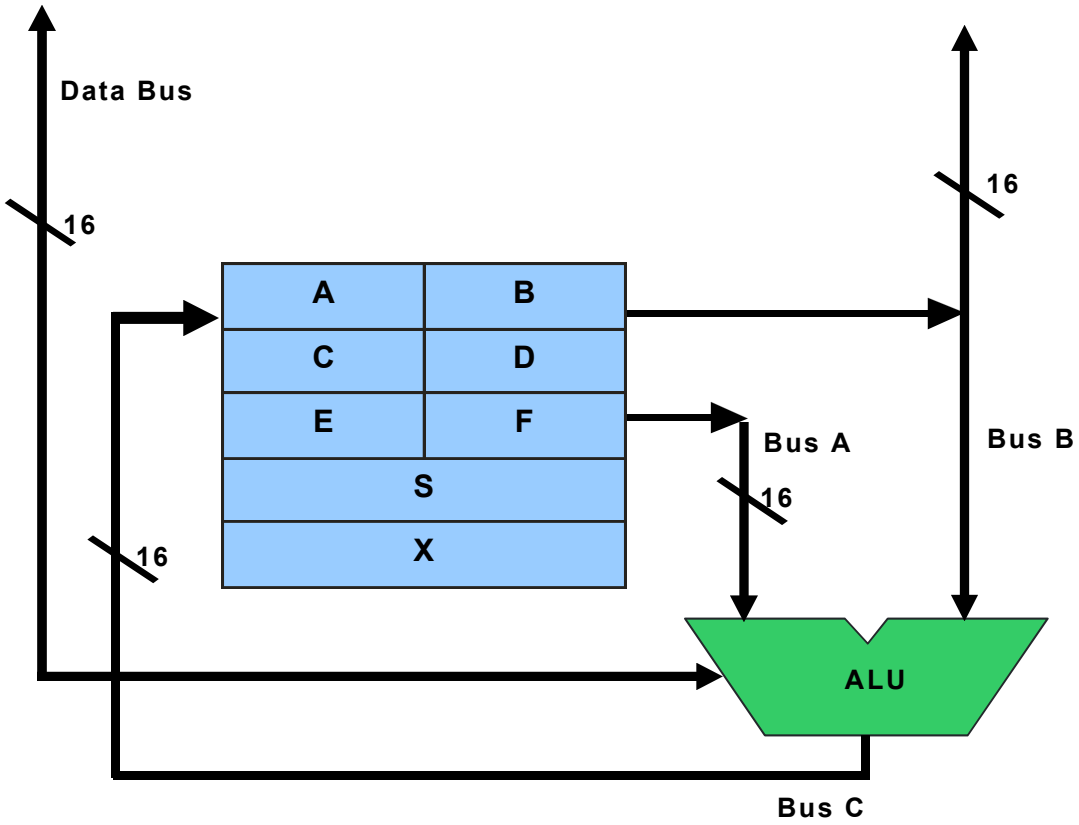

Figure 3.4. Data Path

3.3.1 ALU

As described before, the ALU is a fully 16 bit ALU. A 32 bit ALU would have been too costly in terms of area and is not really needed for our targeted applications. We can extend the ALU to 32 bit without having to change either the instruction set, nor the assembler or compiler in the future. The idea is that we later have a whole family of processors that are all ANTARES compatible but contain exactly the functionality that is needed for the specific application. The whole software part, as a major part of the overall system, would not have to be altered for any of these.

The basic functions of the current ALU are listed in Table 3.3. It includes all major arithmetic, logic and unary functions. We even included a hardware 16 bit multiplication function. The result of this multiplication will be a 16 bit

Arithmetic	**Logic**	**Unary**
C = A + B	C = A & B	C = 0
C = A – B	C = A | B	C = A
C = A × B	C = A ⊕ B	C = B
C = A + 1	C = A ≫ B	C = –A
C = A – 1	C = A ≪ B	C = !A

Table 3.3. Main ALU Functions

integer. This behavior reflects more or less the C behavior and therefore it is not only pragmatic but also efficient. A hardware multiplier will not be needed in all applications, but it is necessary, especially for handling floating points. Also the multiplication units are not too big so we can afford to insert it by default.

As for the arithmetic functions, there is no division included. This is because hardware division is quite a big unit and not needed very frequently. Since we reserved coding space in the instruction definitions for the division instructions we might, however, add them later as an optional unit.

A unit that is desirable for the logic operations would be a barrel shifter. It is not included in the current design since we would rather keep it smaller than complete with all possible functions. However, it is already clear, that for extensive floating point processing a barrel shifter would increase the system speed. This version of the processor, will be able to do floating point processing, but it is not particularly well-suited for it. The barrel shifter could also become an optional unit in a later revision.

3.3.2 Registers

The ANTARES processor has six 16 bit registers and a 32 bit indexed and stack register. This means that registers are a precious good for the ANTARES. Of course it is always nice to have many registers available, but since we only have 16 bit instruction coding, we cannot afford more that eight registers. Eight registers can be coded with three bits. For operations with two operands this means that

we need six bits for coding. If we use more than eight registers we would have to use four bits for coding and this would exceed the possibilities of the 16 bit instruction coding.

In order to still be able to use the registers efficiently, they are all general purpose registers except the 32 bit stack register. Then we have six 16 bit registers (ABl, ABh, CDl, CDh, EFl, EFh) which can also be used as three 32 bit registers (AB, CD, EF). The last is the index register (X) which is a 32 bit general purpose register.

The number of registers available is small, so we might get some swapping overhead for complex expressions. This is a sacrifice necessary for the small chip size and the compact code. However for most of the common code, it should be sufficient and not impair performance.

3.4 Addressing

The Addressing unit is responsible for the correct setting and incrementation of the program counter and for writing the correct value onto the address bus. Figure 3.5 shows a detailed schematic diagram of the addressing unit. It gives a minimal structure consisting only of a 32 bit register, a 32 bit adder and some multiplexers. In the figure and the following detailed description, we already assume a 32 bit B-bus. In the first implementation the B-bus was 16 bit, but since it is clear now that we will change it to 32 bit, it does not make sense to explain the old version. The 32 bit B-bus simplifies things tremendously and speeds up the address unit . We can classify the functionality of the addressing unit into two tasks: program counter and address bus, and categorize the different modes that are possible.

3.4.1 Program counter

The program counter carries always the address of the current instruction that is being processed. At the end of each instruction it is incremented and its value

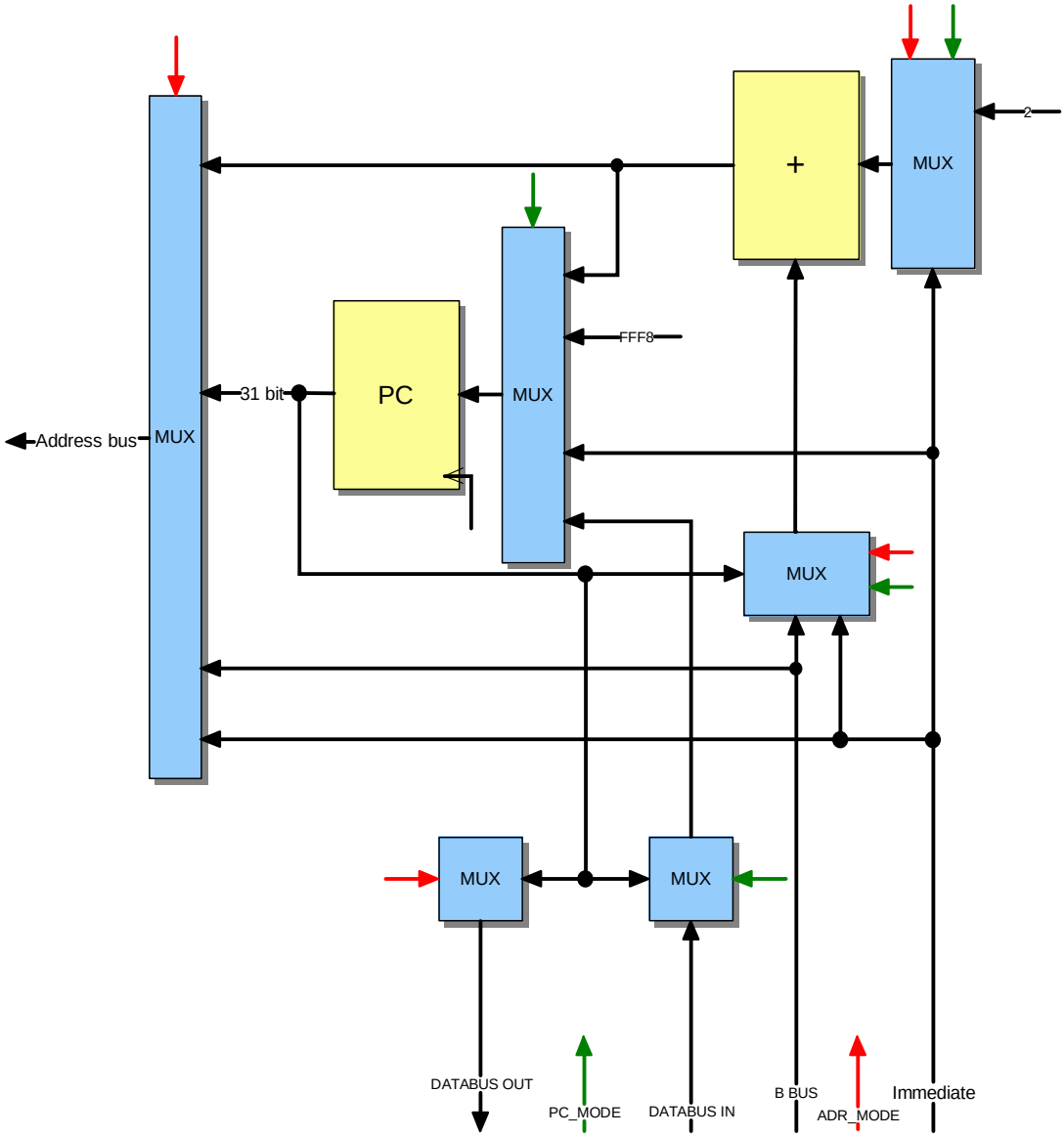

Figure 3.5. Address Unit

is put to the address bus in order to fetch the new instruction word. Apart from this gerneral operation there are instructions that directly modify the program counter. These are all instructions that change the normal flow of code such as jumps, functions calls and returns, and the reset. The different operating modes for the program counter are:

1. Increment
   Increment the program counter by two. This is the standard action in order to proceed to the next instruction

2. Same value
   The program counter keeps its current value. For any cycle where no instruction ID fetched or any other kind of manipulation takes place, the program counter has to keep the current value.

3. Reset
   The reset instruction (RST) sets the program counter back to the BIOS start address (0FFF8h)

4. Immediate
   The value of the immediate field is written into the program counter for all absolute jumps and calls (JMP, CLA). When using the prefix register, this may be a full 32 bit value.

5. Immediate + PC
   For relative jumps and calls (JPR, CAL) the value of the immediate is added to the program counter. The immediate value can also be negative in order to be able to make jumps in both directions.

6. Low word from data bus
   Writes the value of the data bus to the low word of the program counter. Whenever the Program Counter is restored from memory, for example after a function call, we need to write the value of the databus to the program counter. Since the data bus is 16 bit and the program counter is 32 bit wide, there are two modes: One for the low word and one for the high word.

7. High word from data bus
   Write the value of the data bus to the high word of the program counter. See above.

3.4.2 Addressing Modes

The way the value on the address bus is composed depends upon the type of the instruction and the cycle. There are seven possible ways of setting the address bus. The decision on which possibility is taken, is made within the control unit. Then it just gives a signal to the address unit that indicates the current mode.

1. Program Counter
   The program counter is written to the address bus before an instruction fetch can be performed.

2. B-bus
   The value of the B bus is put on the address bus for standard indexed operations like LDX, STX, and LXL. In this case, the value of a register contains the address of the data of interest.

3. B-bus and Increment
   This also indicates an indexed operation, but the actual address of the register is incremented in order to address the next word. This mechanism enables us to make long load and store instructions such as LXL and SXL. Since the data bus is only 16 bit wide, we need two addressing cycles. First we address the lower word and then the higher word.

4. Immediate
   The value of the immediate is written to the address bus for the absolute load and store instructions such as LDA and STA. For these instructions the immediate carries the address information. In combination with the prefix register we can address the full 32 bit address range.

5. B-bus + Immediate
   This type is needed for stack relative addressing. For these cases, we load the value of the stack register via the B-bus and then have to add the value of the immediate to get the target address.

6. Low PC word to data bus
   Writes the low word of the program counter to the data bus. This does not

really write a value onto the address bus, but onto the data bus. Since the result is the same (that we have a value on the databus) we count it among the addressing modes anyway. Besides the coding of the control signals for the address unit is made more compact and consistent. This is what happens, when we store the content of the program counter to memory for example for a function call (CAL, CLA).

7. High PC word to data bus
   Writes the high word of the program counter to the data bus. Same as above, only that the high word of the program counter is saved. Since the data bus is 16 bit and the program counter 32 bit we need two cycles for this.

3.5 Control

The controlling component is very important for any processor. For many designs it is the most complex part of the design. The intruduction of pipelining will in no way change this fact. Pipelining is basically just a way of smart controlling. The non pipelined version of the control unit consists of the following units:

- Instruction register
  In the instruction fetch cycle it reads the current value of the data bus. It keeps this value for the decode logic during all later cycles of that instruction .

- Decode logic
  The decode logic takes the value from the instruction register and then decodes it. There are actually two parts in decoding:

  - Instruction decoding
    Depending upon the input of the instruction register this unit will deliver the instruction, its type, the values of the register arguments and the immediate field if applicable.
  - Cycle decoding
    This unit takes the instruction and the type from the first decode stage

and then evaluates the cycle output from the FSM in order to assign the source and target register values on a per cycle basis.

- FSM
  The Finite State Machine takes the instruction and the operation type from the decode logic and gives as an output the cycle information.

- Flag register
  Here the status of all the flags are stored in a register. Its information is needed for the decoding and will be altered when requested.

- Output registers
  The output registers store all information needed from other units such as the ALU, addressing unit, and memory. They contain information about the current instruction, its type, the current source and target registers, the value of the immediate, and the read or write signal for the memory.

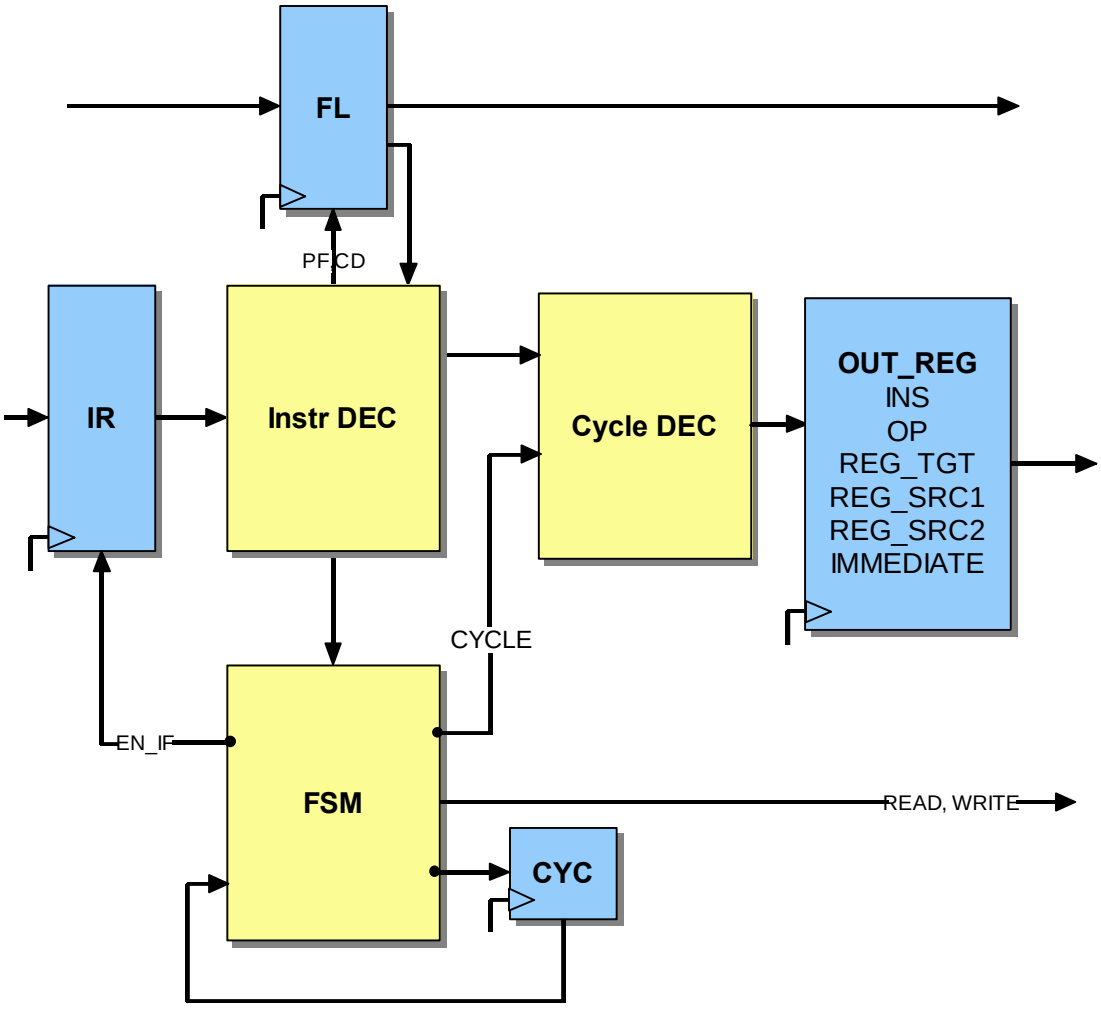

Figure 3.6. Non-pipelined Control Unit

Chapter 4

Conception

Having evaluated the many different pipelining concepts for different architectures we should use for the ANTARES processor, we must specify firstly the implementation properties that should be incorporated into the core. Using these characteristics we can then finalize the concept. In Figure 4.1 we can see that there are different aspects that are important for our approach. Although some of them point in a similar direction, it is impossible to satisfy all requirements without finding a best-fit compromise.

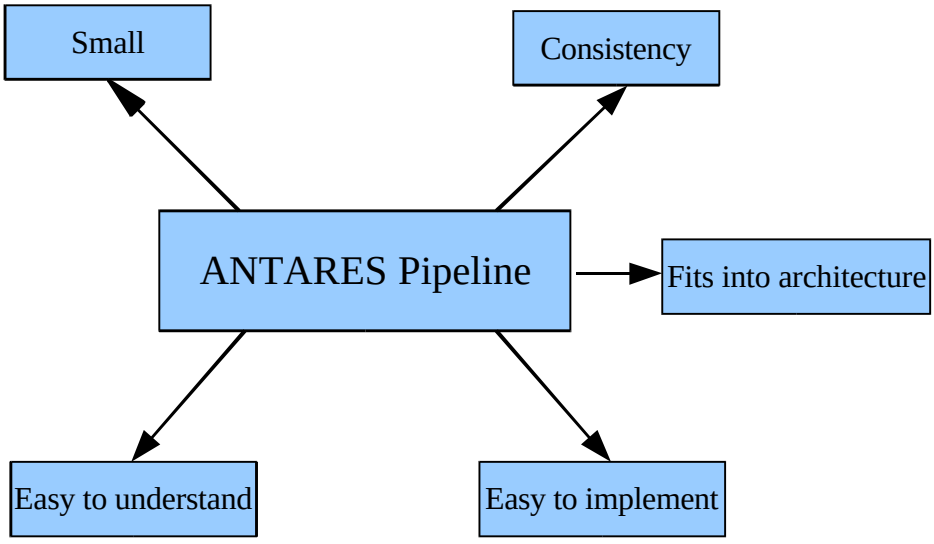

Figure 4.1. ANTARES Pipeline Requirements

In the old control unit [PB01], the FSM was the unit that controlled the actual state of the processor. Since pipelining can be seen as an intelligent way of changing between states in different units, we could imagine some sort of FSM that also takes over all the pipelining control. Since the decisions in an FSM are made in a very transparent way, the solution would be consistent and easy to understand. This concept would probably require the least number of changes in the current architecture, so this is the natural choice. The size of this implementation is not easy to estimate beforehand, but there is no apparent reason to believe that it would become too cumbersome.

4.1 The Finite State Machine (FSM)

4.1.1 Single Mode

The existing FSM (Figure 4.2) consists of 15 states:

State name	Description
INF	Instruction fetch
IND	Instruction decode
EXC0	Execute 0
EXC1	Execute 1
EXC2	Execute 2
EXC3	Execute 3
DFT0	Data fetch 0
DFT1	Data fetch 1
DFT2	Data fetch 2
DST0	Data store 0
DST1	Data store 1
DST2	Data store 2
STP	Stop
ADC0	Address calculation 0
ADC1	Address calculation 1

Table 4.1. States of the Original FSM

When extending it to pipelined mode, they will be unchanged, but wait states must be added. The FSM will go into a wait state each time it cannot proceed

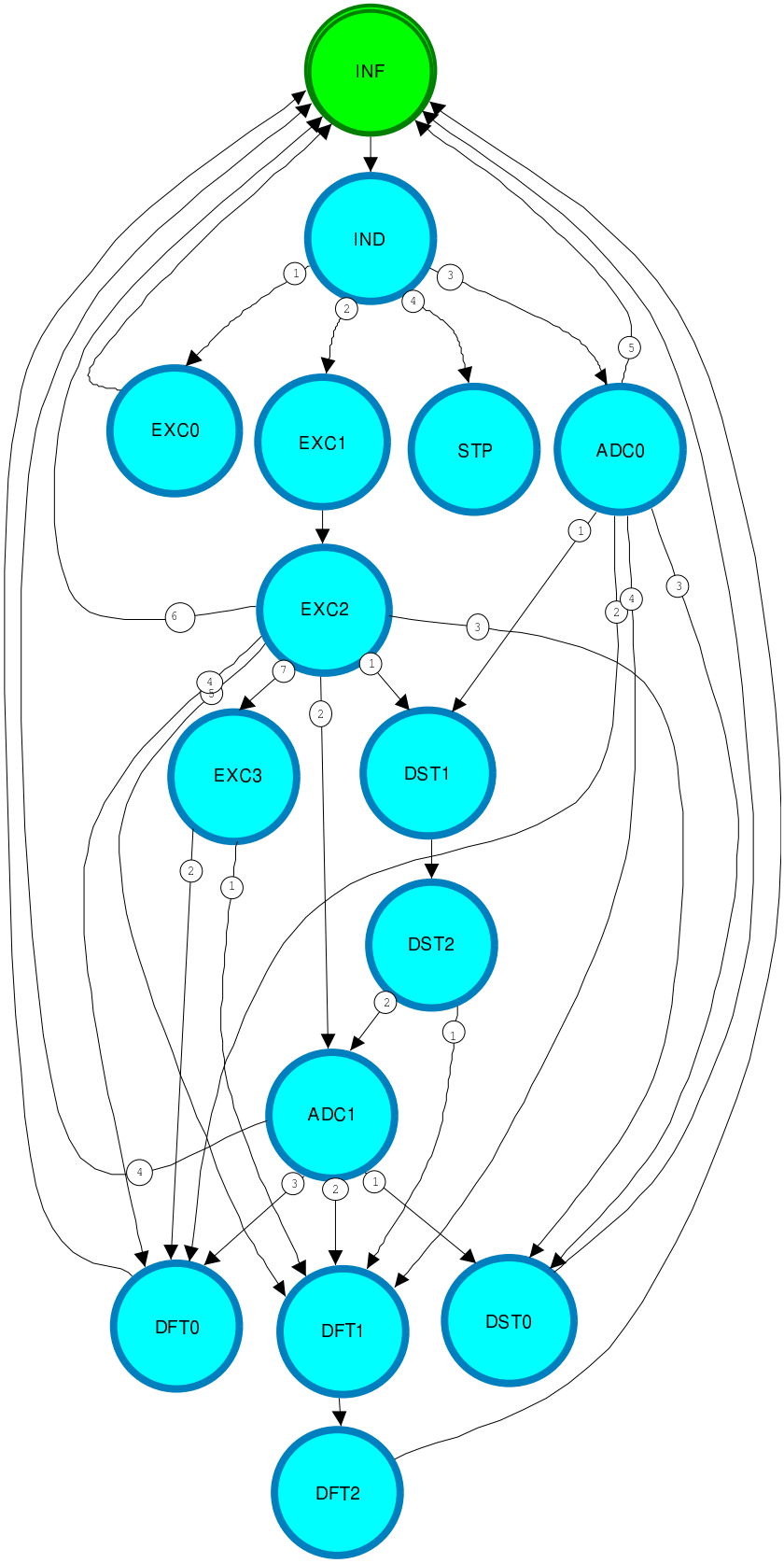

Figure 4.2. Original FSM

to the next state when there is a resource conflict. Some cycles have to be executed consecutively because they subdepend, and use the same resources. This is termed control restrictions and there must not be any wait state inserted. Control restrictions occur for:

- EXC1, EXC2, EXC3
- DST1, DST2
- DFT1, DFT2

In addition, it does not make sense to wait between INF and IND because we will always be able to decode the instruction right after we fetched it.

Considering these restrictions we find out that the insertion of five wait states (W1-W5) would be sufficient. The insertion of these additional states makes the schematic diagram of the FSM more complex. It bevomes more confusing since we also must add transitions to and from the wait states. The complete FSM for pipelined mode is shown in Figure 5.2 on page 54. In order to make it work properly we have to set the appropriate conditions to the transitions to and from the wait states.

4.1.2 Resources

A more critical part of pipeline design is the definition of resources. The definition of the number and type of resources will directly determine the number and type of pipeline stages we will have. This again inherently depends on the architecture of the processor.

Our premise being that the changes in the architecture should be as small as possible, we should first have a closer look at the ANTARES architecture, evaluate the current units and their partitioning and then check if it makes sense to introduce additional units in terms of cost/benefit.

The resources we discover in the current architecture are:

- RAM/ROM
- Address bus
- Data bus
- ALU
- A-bus
- B-bus
- C-bus
- Decode logic
- Program counter

Now we can determine which instructions use which of these nine resources. A transformation of the resulting table into transition conditions would then sufficiently describe the resource restrictions and handling.

State	RAM	Address	Data	PC	ALU	A	B	C	Dec
INF	X	X	X						
IND									X
EXC0		X		X	O	O	O	O	
EXC1		O			X	X	O	X	
EXC2		X		X	X	X	O	X	
EXC3		X					X		
DFT0	X	X	X	X				X	
DFT1	X	X	X	X				X	
DFT2	X	X	X	X				X	
DST0	X	X	X	X		X			
DST1	X	X	X			X			
DST2	X	X	X	X		X			
ADC0		X		X					
ADC1		X		X					
STP									

Table 4.2. Resource usage of the States (X:all cases, O:some cases)

Evaluating the resource usage of the current design (Table 4.2), we can see that the resources are used in a way which makes it very difficult to do pipelining. In order to define pipeline stages, we have to find states whose resources are absolutely independent from others. Right now this would be the case only for IND. Taking the current design as a base, we would get a two stage pipeline, where only IND can be done in parallel to any other state. Assuming an average of four stages per instruction, we would get an ideal CPI of 3.0 with pipelining as opposed to 4.0 with no pipelining. A maximum speedup of 33% sounds initially quite good, but after factoring in penalties for branches, other hazards and instructions of different lengh, there will be little overall gain.

The major drawback of this design is that the memory cycles can not be used independently from the ALU cycles. There are two reasons for this:

1. The Program Counter is increased at the end of each instruction and its value is given to the address bus.

   This is an impediment for pipelining in general, since after the end of an instruction there is not always immediately another instruction issued.

   There are several ways to solve this. We could apply both, i.e. increase the Program Counter and write its value onto the address bus at the beginning of the INF cycle. This would create the greatest resource independence, but we might generate timing problems while performing a 32bit addition before the actual data can be addressed. Another solution would be to increment the Program Counter at the end of each INF cycle and write its value onto the address bus at the beginning instead. In this case we would have less timing problems to worry about, however for operations working with the Program Counter such as relative jumps or function calls, we have to keep track of the changes done in advance to the Program Counter.

2. The data bus is read via the C-Bus and written via the A-Bus
   This can be prevented by connecting the data bus directly to the register file. For the data store cycles we then have to add some multiplexing logic,

so that the register file can write two values simultaneously, one from the data bus and one from the C-Bus. Of course data hazard detection would prevent them from writing to the same register in the same cycle.

The new resource usage can be seen in Table 4.3 with these changes. This resource distribution now permits a much better pipeline design. We can clearly separate three resources:

1. Memory,
   comprising RAM/ROM, address bus, data bus and program counter
2. The instruction decode logic
3. ALU,
   comprising ALU, A bus, B bus and C bus

State	**RAM**	**Address**	**Data**	**PC**	**ALU**	**A**	**B**	**C**	**Dec**
INF	X	X	X	X					
IND									X
EXC0		O		O	O	O	O	O	
EXC1		O			X	X	O	X	
EXC2		O		O	X	X	O	X	
EXC3		X					X		
DFT0	X	X	X	X					
DFT1	X	X	X	X					
DFT2	X	X	X	X					
DST0	X	X	X	X					
DST1	X	X	X						
DST2	X	X	X	X					
ADC0		X		X					
ADC1		X		X					
STP									

Table 4.3. Improved Resource usage of the States (X:all cases, O:some cases)

We are now able to build up a three stage pipeline with an ideal CPI of 1.0 for three-stage instructions and 2.0 for four-stage instructions. That means a maximum speedup of 100% for four-stage instructions and 200% for three stage

instructions. Even if we sacrifice some processing time due to hazards and mixing of instructions with different length, the gain is remarkable.

4.1.3 Multi Mode

We now have an FSM that respects resource restrictions. One FSM characteristic is that it will always be in one certain state. In order to build up a three stage pipeline, we should use three FSMs. This is the fundamental idea. If each FSM respects the given resource restrictions, it will work absolutely self contained. We only have to establish a priority hierarchy for all FSMs over the resources. Figure 4.3 shows the basic schematic diagram of the multi mode FSM control.

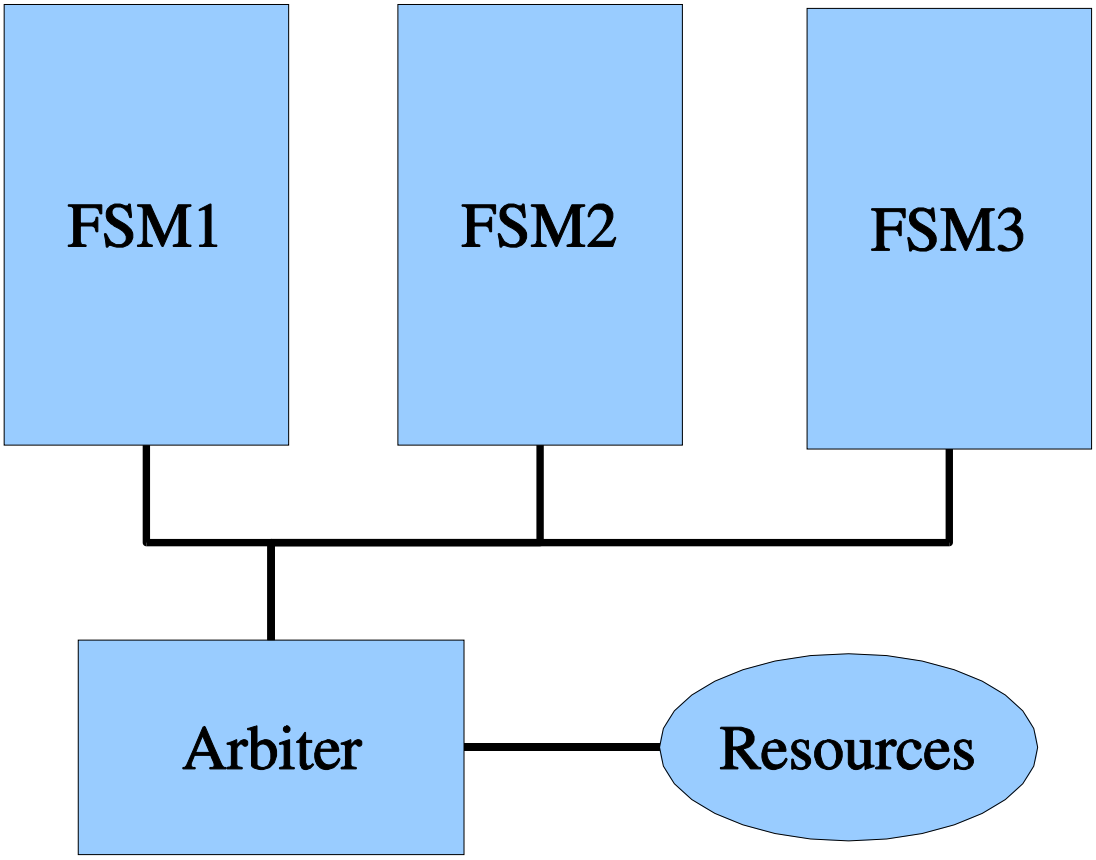

Figure 4.3. Schematic with several FSMs

The introduction of the multi mode FSM is a very simple, but powerful pipelining solution. Basically, it uses the FSM from the non-pipelined version,

adds wait states and appropriate transition conditions. It is then easy to make as many instances as there are pipeline stages and connect them to a unit that controls the resource priorities. We can appreaciate the power of the concept when it comes to changes to the architecture. They will only require changes in the FSM-design. Since we have multiple instances of the same FSM, this step will update the systems entire pipelining behavior. Another selling point is that any person who understands the function of one FSM can make changes to the system.

4.2 The Arbiter

Our design so far will be able to run several instructions in parallel. In Section 2.3 we took a look at the different types of hazards that can occur during pipelining and it seems, that hazard handling is a major part of all pipelining designs. So far we did not explicitly look at this problem in the ANTARES pipeline, but now it is time to do so.

All hazards that have occured, or are about to occur, must be detected and dealt with. In the multi mode FSM design we introduced a unit that controls the priorities of the resource handling. This unit could be extended in order to be able to detect and deal with all types of hazards. We call this unit an arbitration unit or arbiter, since it adjudicates which FSM has priority over the resources and can decide which FSM to halt.

Now let's take a look at the different types of hazards that can occur during program execution and check what steps we have to make in order to avoid them or to handle them.

4.2.1 Structural Hazards

Structural hazards cannot be avoided. Since they originate from the structure of the system itself, we could only get around them if we duplicated the corre-

sponding parts of the system. This is not always an option. In the ANTARES architecture, structural hazards could occur during pipelined operation when one instruction stores something to memory, while simultaneously the processor wants to fetch the next instruction (Figure 4.4). Looking at our design concept we realize, that we can already handle these kinds of hazards directly within the FSM. We introduced wait states with transition conditions into the FSM design. These will take effect whenever certain resources are busy. So if a structural hazard such as a double memory access or a double execution is about to occur, it will transparently be dealt with by the multi mode FSM (Figure 4.5) without affecting any other units.

	t1	t2	t3	t4	t5	t6	t7	T8
STL	INF	IND	DST1	DST2				
ADL		INF	IND	EXC1	EXC2			
SBL			INF	IND	EXC1	EXC2		
ORI				INF	IND	EXC0		

Figure 4.4. Structural Hazard

4.2.2 Control Hazards

Control hazards occur whenever discontinuities take place in the order of code execution. This is the case for all jumps and function calls. Control hazards cannot be prevented, so we have to control what happens when they emerge. There are several basic methods of doing this:

1. After the detection of the jump, all instructions that were issued after

	t1	t2	t3	t4	t5	t6	t7	T8
STL	INF	IND	DST1	DST2				
ADL		INF	IND	EXC1	EXC2			
SBL			STALL	STALL	INF	IND	EXC1	EXC2
ORI				STALL	STALL	INF	IND	EXC0

Figure 4.5. Solved Structural Hazard

the jump instruction are discarded. Execution will only resume after the program counter is modified.

2. If it is clear that the jump must be taken (function call or return), it will directly fetch the new memory address. This works only, if we "know" in advance that there is going to be a jump. We would need a longer pipeline, which would increase the latency of the system.

3. If it is unclear whether the jump should be taken or not, we could fetch and decode both instructions, the one after the jump and the one at the jump target. Then we just discard the "wrong" value and start working with the new one as soon as it is known whether the jump has to be taken or not. In addition to the longer pipeline we would need a second decoding unit and an additional value from the memory this means the memory bandwidth will increase.

4. As an alternative to the previous possibility, instead of fetching both instructions, we could guess if the jump is going to be taken or not and fetch only that value. This solution is called "branch prediction". It would also imply a longer pipeline latency while still having the risk, that the wrong guess was made.

	t1	t2	t3	t4	t5	t6	t7
CZE ABH	INF	IND	EXC0				
JMP 0FA3h		INF	IND	ADC0			
ADD CDL,CDH			INF	IND	EXC0		
ANA ABH,EFL				INF	IND	EXC0	

Figure 4.6. Control Hazard

	t1	t2	t3	t4	t5	t6	t7
CZE ABH	INF	IND	EXC0				
JMP 0FA3h		INF	IND	ADC0			
~~ADD CDL,CDH~~ SBL AB,EF			INF	STALL	INF	IND	EXC1
ORA ABH,EFL				STALL	STALL	INF	IND

Figure 4.7. Solved Control Hazard

The ANTARES jump instructions are very short (only three cycles). This is because we - in accordance with the RISC concept - do not make any comparisons within the instruction but do it beforehand in a separate comparison instruction. Because of this, the loss incurred if a branch has to be taken is so small that it does not make sense to accept the greater drawbacks of more complex control hazard handling. Instead we assume that the jump is not going to be taken and if is, we will lose only two cycles (Figure 4.6 and Figure 4.7).

Function calls and return instructions are a bit more complicated, since here you must write or read a value from the stack, modify the stack pointer and the

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
CAL 0AF2h	INF	IND	EXC1	EXC2	DST1	DST2	ADC1			
ADL CD,EF ~~SUB ABh,EFh~~		INF	STALL	STALL	STALL	STALL	STALL	INF	IND	EXC0
ADD CDL,CDH			STALL	STALL	STALL	STALL	STALL	STALL	INF	

Figure 4.8. Solved CALL Control Hazard

program counter. For CLA and CAL we therefore lose six cycles before being able to return to the normal pipelined mode (Figure 4.8).

4.2.3 Data Hazards

Data hazards occur whenever there are unresolved data dependencies between instructions. Usually this is the case when an instruction wants to use the result of a preceding instruction that has not yet been completed. These cases have to be detected and we should insert pipeline interlocks in a way that ensures the execution of an instruction only starts when all its arguments are determined.

In the ANTARES architecture it is much less likely, that data hazards will occur than is the case in many other standard RISC architectures like the MIPS [HP02] or the DLX [HP94] since there is no write back cycle to the register file. That means, there is no need to make any forwarding construct because as soon as the result of the operation is calculated, it will be available to succeeding instructions. This is another reason why the overall design is simple, small and efficient.

The detection of a data hazard is quite straightforward. After an instruction has been decoded, we have to check if any of the two source registers is identical to any of the target registers of the other two pipeline stages. This can be done quite efficiently using comparators. In Section 5 we will discuss the actual solu-

tion in greater detail.

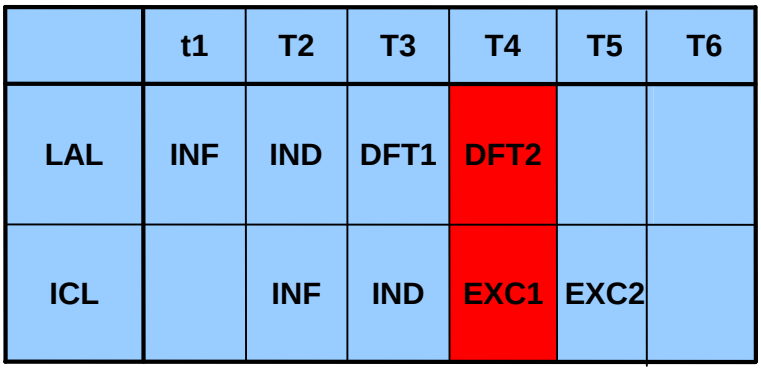

Figure 4.9. Data Hazard

Figure 4.10. Solved DATA Hazard

The question is: in which cases does a data hazard occur? First of all, we can state that it will definitely not occur for any three cycle instruction like ADD, MOV, or SAR. Most of the other instructions could provoke data hazard. Figure 4.9 and Figure 4.10 show the occurrence of a data hazard for a long load and a succeeding long increment. We can see that the value that is incremented is still the old value since the load has not finished yet. This example also shows that there is more room for improvement. If we ensure, that the lower byte of the result from a long operation is produced first and that the lower byte is used first in calculation as well, we could in for these cases, resume the calculation one cycle earlier. In the example in Figure 4.9 that would mean, that there is no data

hazard at all. Using this mechanism we could avoid most of the data hazards for four cycle instructions. Examining all ANTARES instructions we find that there remain only few instructions that are likely to produce a data hazard:

1. Some load instructions
   LEA, LBX, LDX, LDS, LXL, LSL, PPF, POP, PPL

2. Long Comparisons
   CPL, CZL, CML, CGL, CQL, CEL, CNL

3. Some special instructions
   PIN, PSH, PSL

For some of the other operations (mostly store instructions) it is still possible to generate a data hazard, but not very likely. There are not many cases where it makes sense to read a value, that just has been stored to memory. These are: STL, SBX, STX, SBS, STS, SXL, SSL, PSF, and POT.

It turns out, that only 29 instructions out of 112 can possibly produce a data hazard. Only 19 of them are likely to do so. This demonstrates the enormous power of the ANTARES design. While handling these hazards, we could even think about optimizing the case of a conditional jump combination. We could take the jump already during the second execution cycle of the comparison, based on the state of the conditional flag at that time. If the conditional flag does not change during the second execution cycle, we just keep on going without any hazard ocurring. If it does change, we perform the address calculation cycle of the jump again. This will drastically reduce the probability of causing extra delay between a long comparison and a jump. Of course this is an advanced pipelining feature, which should only be included in the implementation when all basic functions are working.

4.3 Modifications to Other Units

Up to now out topic has been the ANTARES pipelining concept. The major part of the pipelining logic concerns the control unit. However, there are still other

modifications to be done in the overall system that can be summarized here.

4.3.1 Mandatory Changes

First of all we should investigate critical changes, later then we can look at other changes and optimizations that could or should be taken in order to make everything smooth, fast, and consistent.

- Instruction Fetch
  The most important changes affect the instruction fetch. In the previous design, the Program Counter is incremented in the last cycle of an instruction and in the same cycle its value is written to the address bus. However, in the pipelined mode we have to increment the program counter at the end of each instruction fetch and write its value to the address bus at the start. In order to do this, we must first of all make some important modifications to the address unit. As the program counter now will always be 4 bytes "ahead", we have to take this into account when conducting program counter relative operations such as relative jumps (JPR) or relative function calls (CAL). This will also generate the need for small but important changes in the data path.

- Connect data bus to register file
  This is another important change in the architecture. Without which, the speedup by pipelining will become negligible. Just by connecting the data bus to the register file instead of to the ALU, write instructions to memory can be performed parallel to execution cycles.

- Add a second write port to the register file
  In order to make an execution instruction completely independent from a data instruction, we have to be able to read value from memory the execution unit is used. Since both actions request a write to the registers, a certain logic must be added that allows two writes to the registers in one cycle. The arbiter will prevent the scenario where both writes are performed on the same register.

4.3.2 Optimizations

Without the above changes, pipelined operation will be of little benefit to ANTARES. As we worked with the system, we discovered that many things could speed the system up, or make it more consistent, or both.

- 32 bit B-bus
  If the B-bus is made 32 bit wide, it will allow ANTARES to put a 32 bit register value to the Program Counter or address bus within one cycle. This will reduce the number of cycles for some important instructions like PUSH, POP, RETURN, and all the indexed operations. Since having an equal number of stages per instruction makes for efficient pipelining, this change will enhance the design better in this direction. It will make the execution of subroutines and complex calculations like for floating points much more smoothly.

- Dedicated bus to the addressing unit
  In addition to making the B-bus 32 bit wide, it would speed up pipelined execution siginifcantly if we introduced a dedicated D-bus instead. This 32 bit bus would connect directly to the control unit, removing any dependencies between addressing and data path access. The effort to implement this change it not very important, also the increase in size is not significant. On a process with five or six metal layers, an additional bus does not result in much surface on the die. The gain on the other hand will be significant. Also it will simplify the control, reducing resource restrictions.

- Control instructions with only two cycles
  When the incrementation of the Program Counter and the addressing is done in an INF cycle, we realize that all control instructions no longer need an execution cycle. All the required actions happen within the control unit and can be done during the instruction decode. This will not result in an important general speedup, since all other instructions are longer, but it might reduce the number of structural hazards that could occur for the execution unit.

- Use carry for faster stack modifications

Incrementing or decrementing the stack takes two cycles since we have to perform a 32 bit addition with a 16 bit ALU. In most cases a 16 bit addition would be sufficient. This would reduce the number of cycles for these instructions by one. By evaluating the carry that has to be calculated in the first cycle it can be easily detected whether this is the case or not. If no carry occurs, we have nothing to add to the higher byte and therefore can do other calculations in that time. We will again have faster pushs and pops, and the function calls and returns will be sped up.

- Another optimization, not directly linked to the pipelining is the extension of the register file. If we experience shortage of the number of registers, we could think of a way to increase their number without having to increase the number of bits for coding and therfore could still adhere to the 16 bit instruction format. As for each instruction the size of the source and target registers are known, we could introduce six additional 32 bit registers with the same names as the 16 bit registers. We would then have the registers A, B, C, D, E, and F both as a 16 bit and a 32 bit version. The control unit could do the decoding to the correct size with no additional effort and the performance of the system for large expressions or floating point operation would increase significantly. Of course this change would mean quite some additional size of the chip, but depending on the target application, it can be worth to consider.

- Pipelining Optimizations
  Apart from these general changes for speedup there are many small changes in the pipelining control itself that could optimize the processing. Basically these are optimizations for particular instructions such as load/store or some long operations. With smart usage of the resources we can still do some fine tuning here and maybe reduce some important cycles. But these changes directly depend on the other, more important ones. It makes sense to evaluate them in detail only once the rest of the system has been adapted.

4.4 Summary

Having different pipeline designs in mind and considering the special architectural properties of the ANTARES, we can propose a lightweight pipeline concept. It will give considerable speedup to the processor while still keeping the core small, ensuring it will remain suitable (and even more than before) to low-power applications.

However we dicsovered that integrating the pipeline into the core cannot be done without performing some small but profound changes to the existing architecture. Since we will have to modify the architecture anyway it makes sense to do this in tandem with the other modifications that will speedup, purify, and simplify the overall design.

Chapter 5

Implementation

5.1 Designing with VHDL

Having a good concept is all fine and dandy, but one can only appreciate its potency once implementation starts. As the complexity of today's systems increases the level of abstraction within the implementation and design process will also increase. While some designs today are implemented with classical VLSI tools and microprogrammed controls [GAS90], we choose VHDL as a high level description language. The price of a slightly larger design is very small in comparison to the gain in productivity. Using VHDL in all parts of the design allowed us to explore completely new possibilities and approaches.

This chapter will therefore describe the VHDL implementation of the pipeline. VHDL is an excellent tool, which enables relatively comfortable handling of complex problems and large designs. We will not go into the details of the language here, but some knowledge of it would be of benefit [IEE88, Ten95]. The progress of implementation can be very nicely visualized with the Y-chart from Gajski (Figure 5.1). A behavioral description will help to describe the system just that more easily. Once finished, we can introduce the structure. Portions of this structure can then again be described in the behavioral domain in order to transform them later into a structural description. The implementation process will eventually come to the center. There we have a description of the design at all levels. The physical domain is represented by the synthesis. This represents the masks,

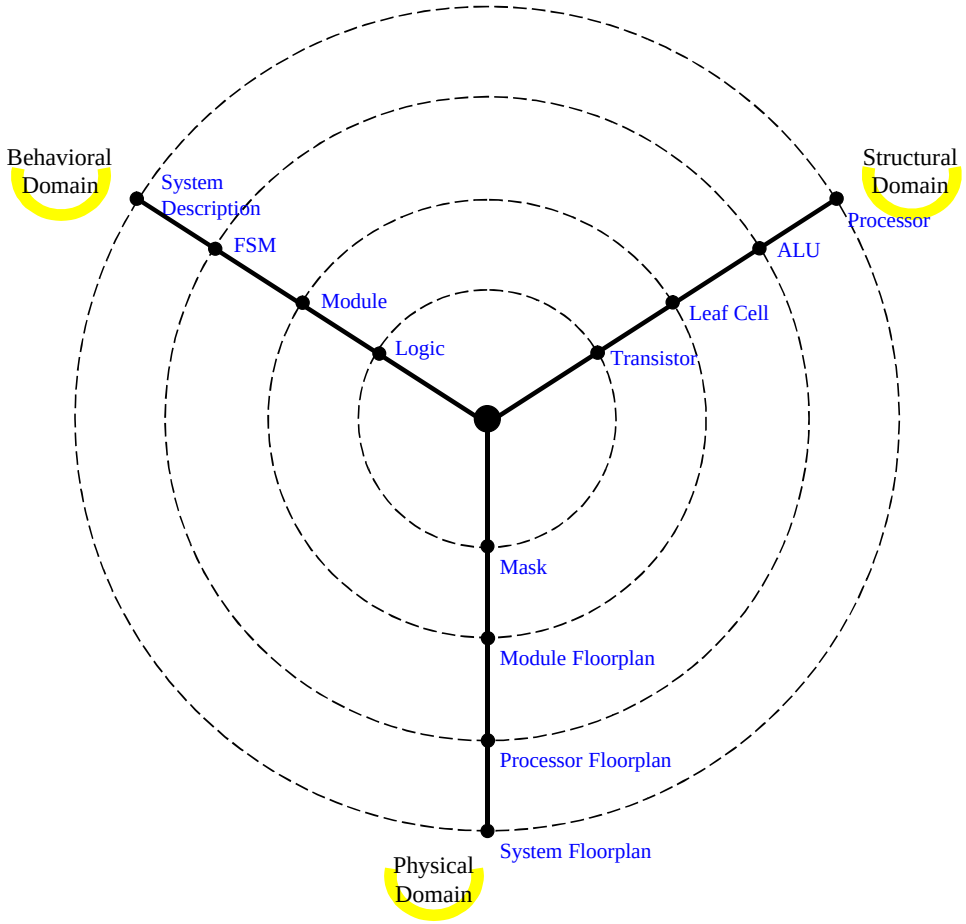

Figure 5.1. The Gajski-Y

placements, and routing again on all levels.

VHDL fits very well into this process since it allows both a behavioral as well as a structural description of a system. This makes the design process more consistent and reduces the risk of errors resulting from conversion or transcription problems.

When a complex new system is implemented, there is always the question of where to begin. It is not always so easy to decide which part should be implemented first. One could start with the interfaces or begin with a skeleton structure of the whole system. For ANTARES we found it quite natural to start

from the core and add layers until we finally reached the outside. This has the advantage, that the function can be verified at a very early stage and formal problems as well as semantical errors can be detected as they occur.

5.2 FSM

The FSM, being at the core, was the first thing to be implemented. Using the existing FSM we first separated the logic of the FSM from actions. We grouped all the transition conditions into one state-process, and grouped all the actions that occur in certain states into another execution process. Both of these processes are completely independent from the clock, so we added a small clocked process that would trigger the transitions. We switched from a single process FSM to this tree process design because it is much easier to deal with. In a three process FSM you can modify a certain part or check a certain functionality much faster. All the registered information is put into the clocked process so it is much easier to keep control over the FSM. A single process design (when built up correctly) can be a little bit faster than a three process design, but the complexity of our FSM made the three process design the model of our choice.

Since a graphical presentation of a problem is often much more comprehensive than a textual one, we drew the actual state diagram in *Mentor FPGA Advantage*. After adding all the transition conditions into the graph, we were able to generate VHDL code from the graph. This code was able to replace the state-process of our FSM. Having an automatic path from a graphical view down to the code helped us visualize the problem immensely. Now we can make modifications in the graph, generate the code and then directly check the result in simulation.

5.2.1 Add Wait States

In order to make this FSM work in pipelined mode, wait states must be added. It should always be possible to change into a wait state between two states that use different resources. There is no need to insert a wait state between two states

that use the same resource (e.g. EXC1 and EXC2), nor does it make sense to insert a wait state when there is no other state using this resource (like the IND). Before inserting the wait states we should rearrange the states in a way that ensures that all transitions proceed from top to bottom, except for the ones that return to the INF state (Figure 4.2). We should always keep this structure when inserting wait states as it will ultimately give us the number of FSMs that need to be inserted.

In order to add resource wait states, we have to define firstly the resources and then which state is using which resource. According to Table 4.3, we define three resources and assign each state to the appropriate resources:

1. Memory:
   INF, DFT0, DFT1, DFT2, DST0, DST1, DST2, ADC0, ADC1, EXC3
2. Decode: IND
3. ALU:
   EXC0, EXC1, EXC2, EXC3
4. no resource: STP

This is a simplified version of Table 4.3, but it will work for most cases. Only for some instructions like indexed operations will we have to make some special handling, since they also use memory in the execution cycles. Now we can insert a wait state whenever a resource change occurs. This is the case for:

- W1
  after IND
  before EXC0, EXC1, and ADC0
- W2
  after EXC2 and IND
  before DST1

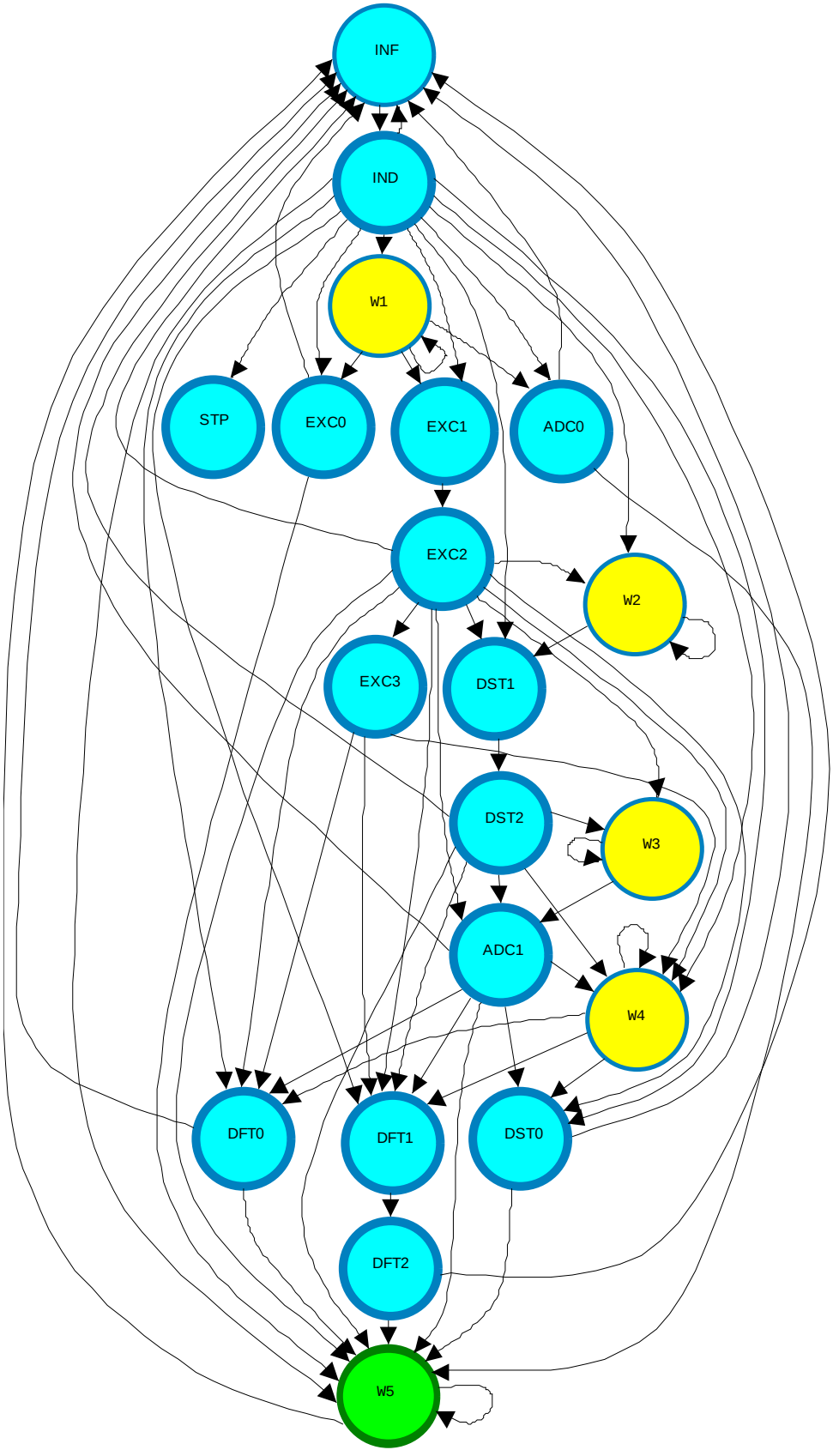

Figure 5.2. FSM with Wait States

- W3
  after EXC2 and DST2
  before ADC1

- W4
  after DST2, EXC3, EXC2, ADC1, and IND
  before DFT0, DFT1, DFT2

- W5
  comes after the last cycle of each instruction
  before instruction fetch

These five wait states are enough to resolve all resource restrictions. There are some levels in the FSM where we did not insert a wait state. This is either when a resource hazard can never occur (such as that between instruction fetch and instruction decode) or if the resource is captured for several consecutive cycles and should not passed to another instruction. The latter is the case for the pairs EXC1+EXC2, DST1+DST2, and DFT1+DFT2. For these combinations the resource has to stay with the same instruction.

5.2.2 Insert Transitions

Figure 5.2 now shows the FSM with our five wait states. It looks now much more complicated than Figure 4.2 even though we added only five states. The main reason for this is the number of transitions. Each additional state added several transitions to the FSM. If we add the correct conditions, the FSM will solve all possible resource restrictions.

The transition conditions can be added quite systematically. The condition to reach the wait state is exactly the same as that to reach the next state, only that the resource condition is added (e.g. and memory=used). The priority of this transition has to be higher than the unrestricted one, then it will first check to see if the resource is available, and only then evaluate the condition to proceed. To get out of the FSM is exactly the same condition to get in. If the only way

out points to one state, it is enough to evaluate the availability of the resource. If there are several input states and several output states - that means the state is handling more than one resource, we have to be careful with the conditions. All the conditions rely on the instructions and on the operation type. But since for some operation types we have different cycles, it can happen that two instructions go to a wait state based on the operation type, but look for different resources. So for these cases we have to look carefully at the conditions.

5.2.3 Entity Interface

Now we have all the functionality, we should think about the external interface of the unit. We have to define what inputs and outputs we really need. A simple and orthogonal interface always simplifies the usage of a system.

Figure 5.3. Interface of the FSM Module

Figure 5.3 shows the final interface of the FSM. We tried to make the interface very small and transparent. Our FSM has six inputs and two outputs. Most importantly, the next_st is the output that indicates the next state, depending on

the current_st input. Then it takes the name of the current instruction (instr_dec) and its type (op) in order to evaluate the transition conditions. It receives the status of the allocated resources with the signals *mem* and ALU and so decides whether to enter a wait state or not.

These signals would be enough to operate the FSM correctly, and ensure that it respects the given resources. For pipelined operation, we added a enable signal that orders the FSM to re-evaluate the next state and a ready signal as an output which becomes true once the value of next state is determined.

5.2.4 Simulation

The functioning of the FSM is essential to the system. In order to verify the functionality of the FSM design we conducted a simulation. The screen shot of the simulation in Figure 5.4 shows its most important functionality. At the beginning, the system is in reset mode. As current_st is held on wait state 5, the system does not advance. Once we release the reset, the FSM starts proceeding. We also can see, that the next state variable goes into a wait state as soon the resource is reserved. At this point, the next state is already determined and will be effective at the clock's next rising edge. The lower two signals show the resource usage. The resource is marked as used (true) as soon as the FSM detects that it will be needed in the next cycle. The resource is released at the beginning of the next cycle i.e., when the actual cycle is using the resource and the calculation of the next state can start.

Obviously, for non-pipelined operation we do not need the resource display. What we can see is how often the resource is actually idle. In this example we are using mainly instructions that access memory. Still we see that for some cycles the memory resource is idle. The execution unit, however, is idle for most of the time. This is exactly where we can get improvements with the help of pipelining.

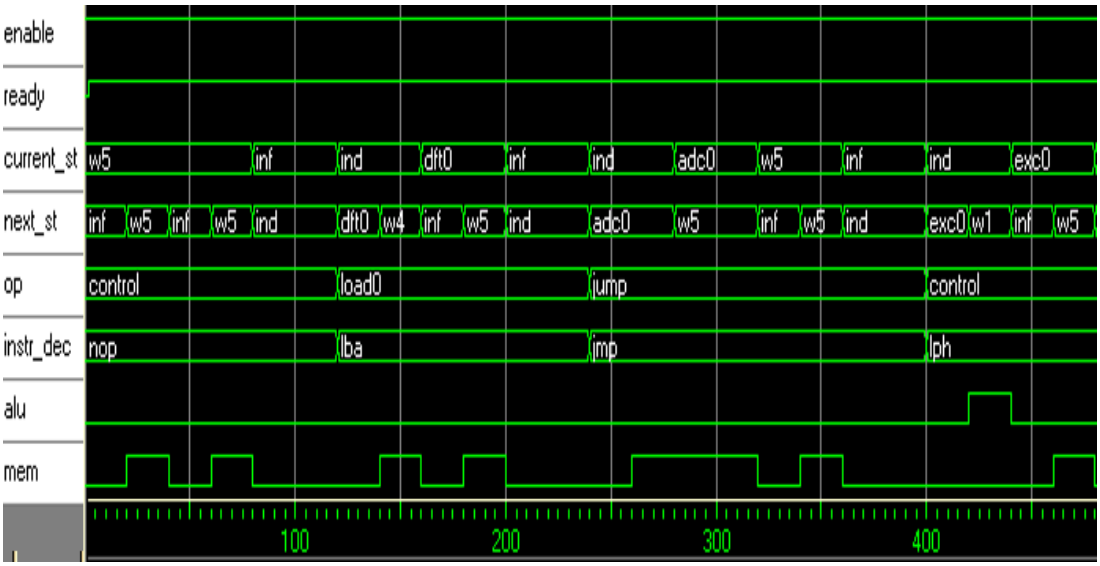

Figure 5.4. Simulation of the FSM

5.3 Arbiter

The arbiter is one of the key components of pipelining. It should connect the single FSMs, allocate the priorities for the resources and give a defined order of proceeding when a hazard or exception occurs. In Figure 5.5 we can see the actual wiring of the arbiter and its basic components. It consists of three instances of the FSM (Section 5.2), a "nextstate" process and a "set_io" process. The nextstate process is a major part of the control system. It takes the outputs of the FSMs, and in return provides them with the recent resource usage and the current state as an input. The set_io process takes the states of the FSMs as an input and assigns the resulting output to the exterior of the entity.

5.3.1 *nextstate* Process

The FSMs themselves do not have a clock. They work quasi-asynchronously and react to input changes such as current state, instruction, and resources. The nextstate process gives an enable signal to one FSM at a time. Only after it receives the ready signal from this FSM, can we be sure that the recalculation of the result has been completed and we can then enable the next FSM. Once

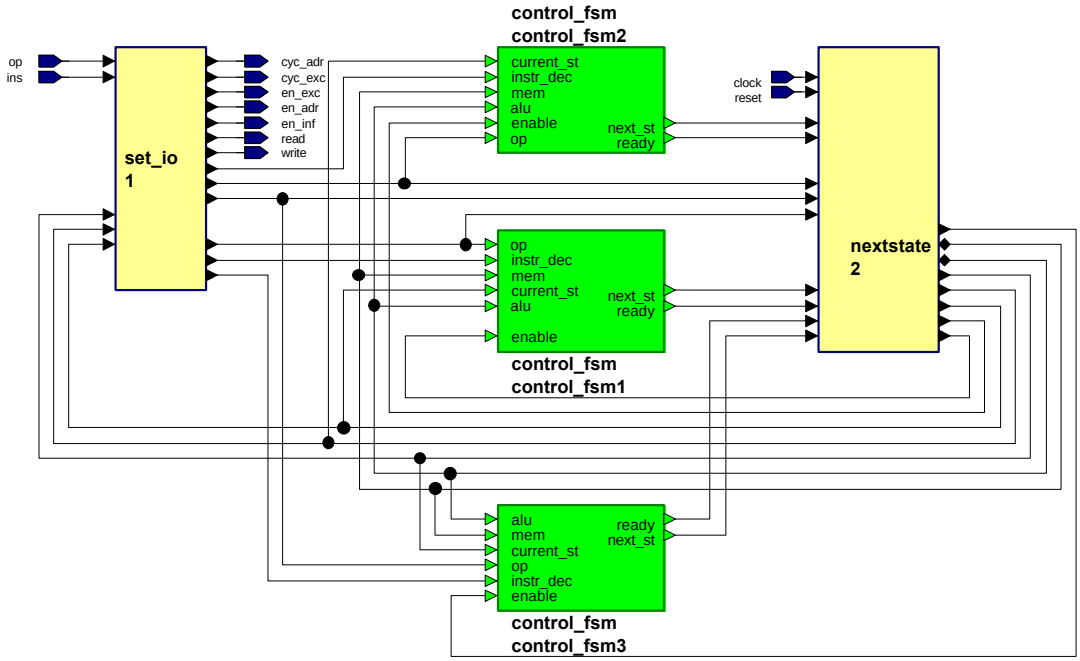

Figure 5.5. Schematic of the Multi Mode FSM

one FSM reserves a resource it will no longer be available for the other FSMs that evaluate their states later. This resource handling is the reason why the calculation of the next state can not be done in parallel by all of the FSMs. In order to still be able to perform this action within one clock cycle, we use this quasi-asynchronous structure, where within one clock cycle each FSM will be asked for its result sequentially.

It is not easy to determine the ordering of the priority distribution. If for example, a long load operation starts, we should not interrupt it by issuing another load or instruction fetch. One solution would be to always give priority to the instruction that was issued first. This does not cover all cases though, because some instructions are longer than other. Therefore we really have to ensure that an instruction, for example in DFT1 state, can immediately go to DFT2. Since we did not insert any wait state between DFT1 and DFT2 the instruction would even proceed to DFT2 even if the memory resource is blocked.

The solution is, to make several rounds in the resource handling. In a first round, all "must" resources are distributed, in a second round the remaining resources can be allocated. Resource constraints exist for EXC2, EXC3, DFT2, and DST2. The system basically behaves in a way, so that once a resource is taken by a FSM, it is kept as long as needed. Once these constraints are satisfied, the priority of resource distribution is no longer critical. The easiest solution (i.e. that the order is always the same) will be fine. The result will be, that the first FSM only goes into a wait state in order to satisfy a resource constraint. The second will have to wait more often since the first always has priority over the resource. The third only will get a chance to perform when neither of the other FSMs is requesting the needed resource.

Since control hazards are not evaluated in the FSM itself, the arbiter must take care of this too. The implementation is easy. Whenever an operation provokes a control hazard, it can be determined after the decode, if the jump has to be taken or not. If it is not taken, nothing has to be done, if it is taken, or if there is a CALL or return instruction which will always trigger a control hazard, the previous fetched operation is discarded and the FSM goes into wait state W5. Once the branch finishes, the FSM will go back to normal operation. This action also ocurred in the first run of the resource distribution.

The multi mode concept of pipelining control works out quite well. Resource handling is done by each FSM almost transparently for the arbiter, which must only deal with a few exceptions. The other hazards can also be dealt with relative comfort. One problem with this implementation however, is the fact, that the runtime quasi-asynchronous resource checking of the states may become critical. The evaluation of the FSM, from enabling to ready must be performed six times within the one cycle. As the FSM is quite compact we estimate, however, that this will not result in any problems and that this nextstate process will not lie on the critical path of the system. A memory access or some of the ALU cycles will probably reqire much more time.

5.3.2 *set_io* Process

The second process was made in order to handle of the input and output signals correctly. It is some sort of multiplexing logic without a clock that sends the input signals to the correct FSM and sends the FSM output signals to the correct entity outputs.

We receice the instruction and the operation type from the decode logic as input signals. These signals should only reach the corresponding FSM during the instruction decode cycle. For the rest of the instruction, they will remain the same for this FSM.

Not all the output signals of the FSMs are relevant for the outside. The set_io process only sends the cycle and the instruction of that FSM to the outside that is in some execution cycle. It also determines the enable signals for the ALU, addressing unit, and instruction register. It will also generate the read and write signals for the memory.

5.3.3 Entity Interface

The interface of the arbiter is, again, made minimal, so that it only sends signals essential for the outside. At present it has four input and seven output pins.

In addition to the instruction and the type, which we already needed for the FSM, the arbiter will read the clock as an input. The value of the current cycle of the FSMs is held in registers and therefore a clock is needed. Also, there is a reset signal in order to give the whole control a defined state at system startup.

The list of outputs represents the enable signals for the ALU, addressing, and instruction register plus the cycle type of the ALU- and addressing-cycle. Using the instruction information, the control unit will be able to decide what actions have to be taken. Whenever there is a cycle accessing memory, an appropriate read or write signal will also be generated by the arbiter. These can be connected directly to the memory.

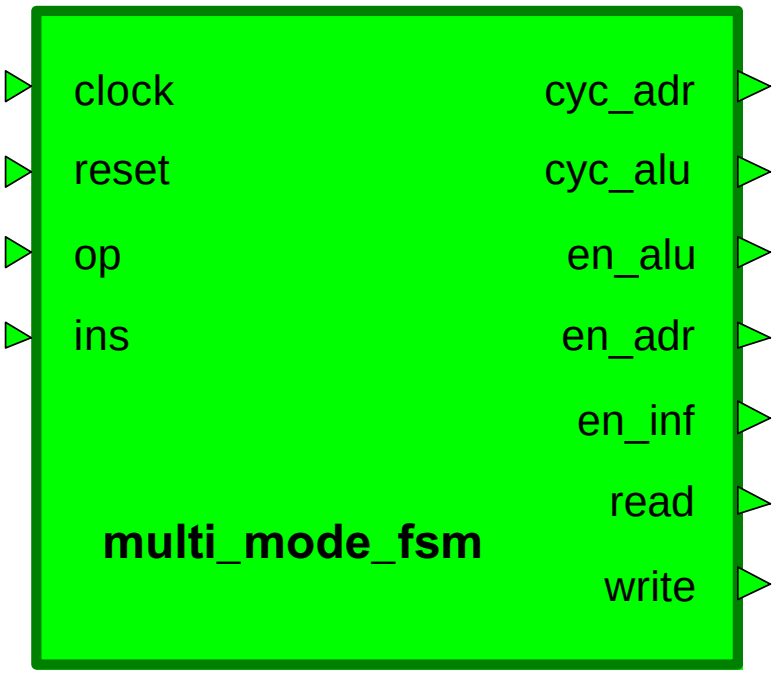

Figure 5.6. Interface of the Multi Mode FSM

5.3.4 Simulation

Though the structure is relatively straightforward, the arbiter and the state machine instances form a rather complex system. Functionality must be checked on every level. VHDL simulation with "Modelsim" from Mentor Graphics is a convenient tool to help discover any kind of errors in the design.

Figure 5.7 shows an excerpt of such a VHDL simulation. It checks the general system functioning. For each state machine we can see the instruction, operation, the current cycle, and the anticipated next cycle. All FSMs start off from the W5 state. We can see that for FSM1 the next_state signal goes to INF right from the beginning, blocking the memory resource, while the others stay in W5. Only when the reset releases will cycle1 advance to the INF state. At the same time, FSM2 is reserving the memory for the next cycle in order to be able to fetch an instruction there.

In order to find out the quality of the pipelining, we can check the enable sig-

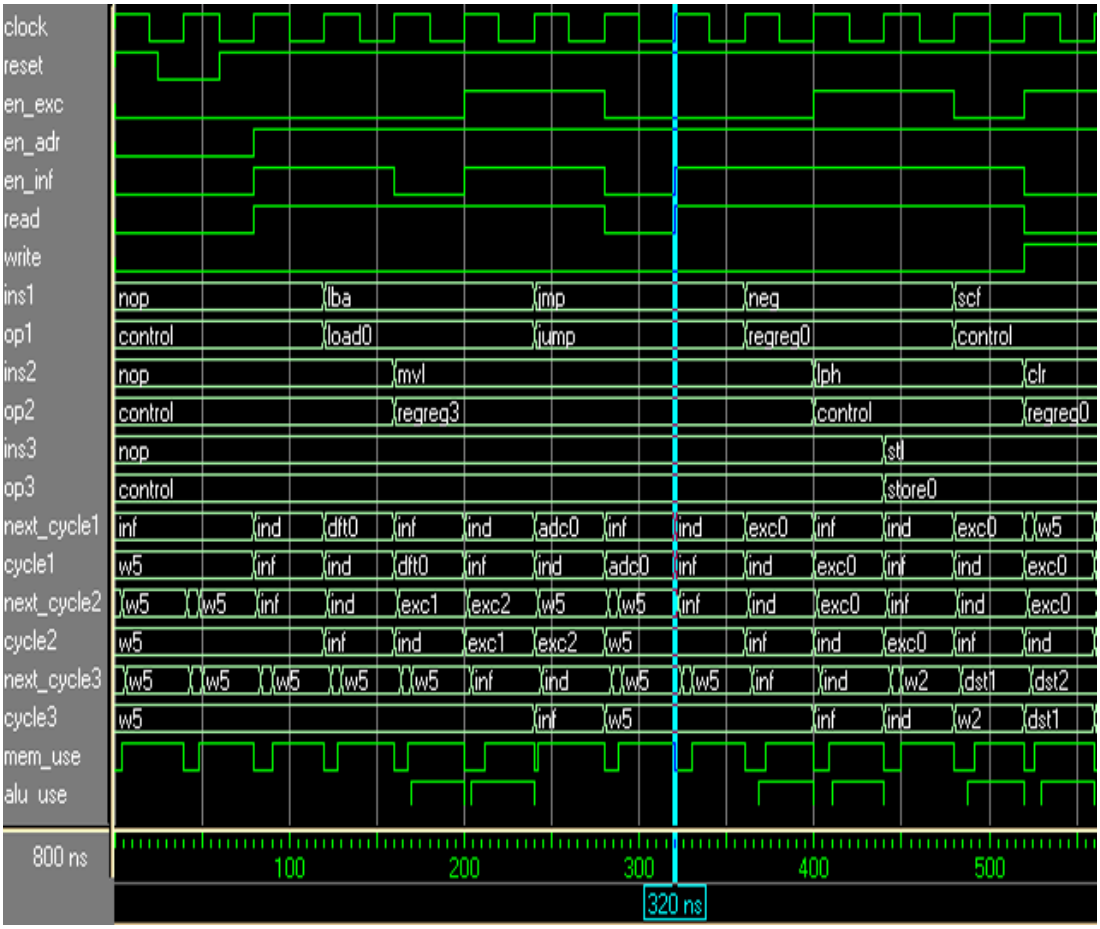

Figure 5.7. Simulation of Multi Mode FSM

nal of the address unit (en_adr). Anytime when the addressing unit or memory is busy, this signal is set to TRUE. In this simulation we can see, that once the signal turns to TRUE after the reset, it actually never goes down again. This also tells us, that for this combination of instruction, the memory resource will limit the speed. Looking at the signal en_exc tells us about the actual usage of the execution unit. We can see that even though the memory sets a fixed limit, for 50% of the time the execution unit is kept busy.

Another way of checking the performance and quality of the pipelining implementation is the number of wait states that occur. The first FSM will only go one time into a wait state. This is when FSM3 is in DST1 and must be advanced

to DST2, keeping the memory busy. FSM2 spends some more time in wait cycles while FSM3 has a greater incidence of wait states. This simulation sequence shows of course a somewhat more critical combination, as it uses many memory instructions and contains a taken branch. We can compare the behavior of this system to a system without pipelining we can take by looking at Figure 5.8. This simulation has exactly the same sequence of instructions as for Figure 5.7, only that we have sequential execution. If we measure the time taken from when the NEG instruction is issued to the end of the SCF instruction, we can see a significant difference. Execution without pipelining takes 550 ns while the pipelined control will do the same within 200 ns. This means a speedup of 175%.

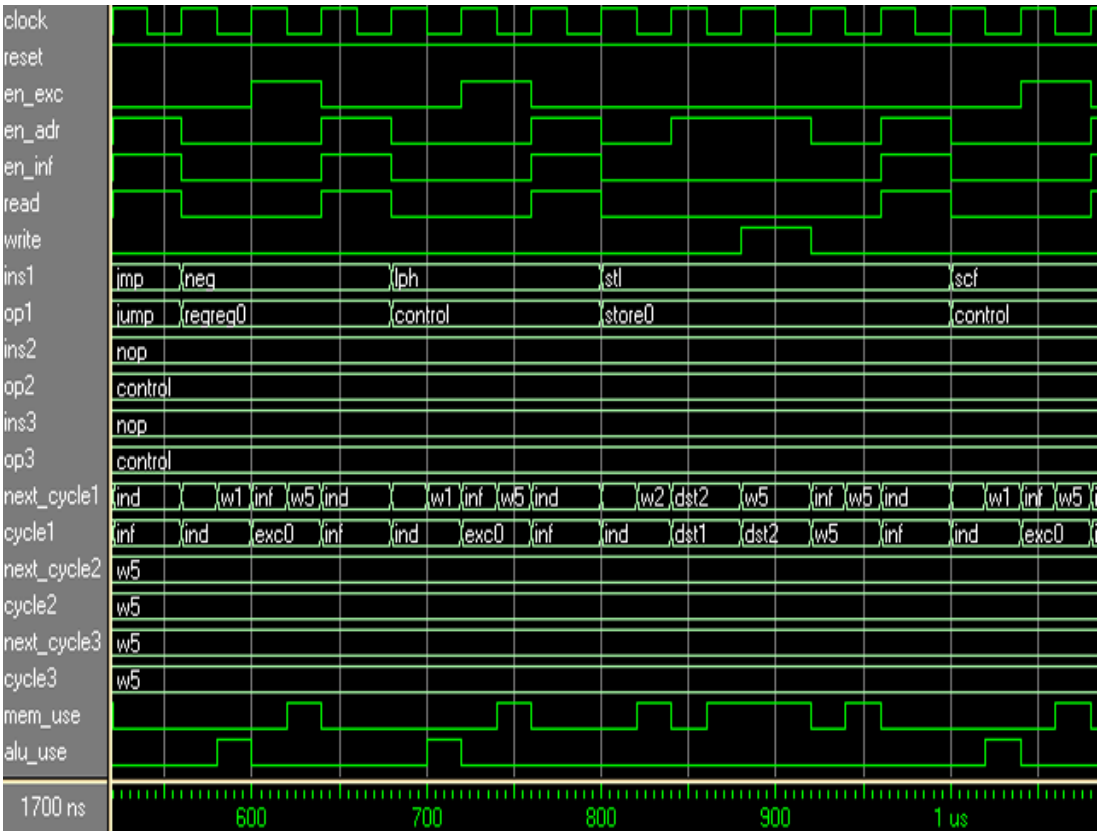

Figure 5.8. Serial Simulation for Comparison

5.4 Control Unit

When we evaluate the existing control unit (Figure 3.6) for pipelined operation, we can see, that there will have to be substantial changes. What we can use is the instruction decoding part. The cycle decoding must be replaced since we can be in serveral different cycles at the same time. We also have to change the external interface since the data path and the addressing unit will operate concurrently on different instructions. In order to minimize the inevitable changes done to the other units, we have developed a control unit schematic diagram (Figure 5.9).

In this version, the decoding no longer depends upon the cycle. The result of the decode unit is sent directly into the registers. We have now two main register parts, one register containing the values needed for the execution unit (EXC_REG) and one register containing the values needed for the addressing unit (ADR_REG). The execution register contains the instruction, its type, two source, one target register, and the immediate value (if applicable). For all long operations, the data path expects the source and target registers to change between the cycles. This is because the ALU works with 16 bit registers, even when the operations are 32 bit. This task has been done in advance by the cycle decode part. What we do now is insert a sort of multiplexer behind the execution register, which, depending on the cycle and the instruction, will send the correct target and source registers to the execution unit.

The register for the addressing unit contains primarily the control signals for the addressing and program counter modes. It also contains the source and target registers as well as the value of the immediate field.

The read and write signals for the memory control are directly generated in the multi mode FSM. They are sent unbuffered to the memory. The values linked directly to the instruction have to remain in the registers for the instruction's duration. This is done by the enable signals from the arbiter. They become true whenever the decode cycle detects an instruction of the appropriate type. Then the results of the decoding like instruction type, source and target registers,

and immediate field are stored into the registers and will not change until that instruction finishes and another instruction of that type is issued. The control unit tries to bundle as much intelligence as possible, by reducing the other units to their "stupid" functionality. This makes the design transparent and easy to modify.

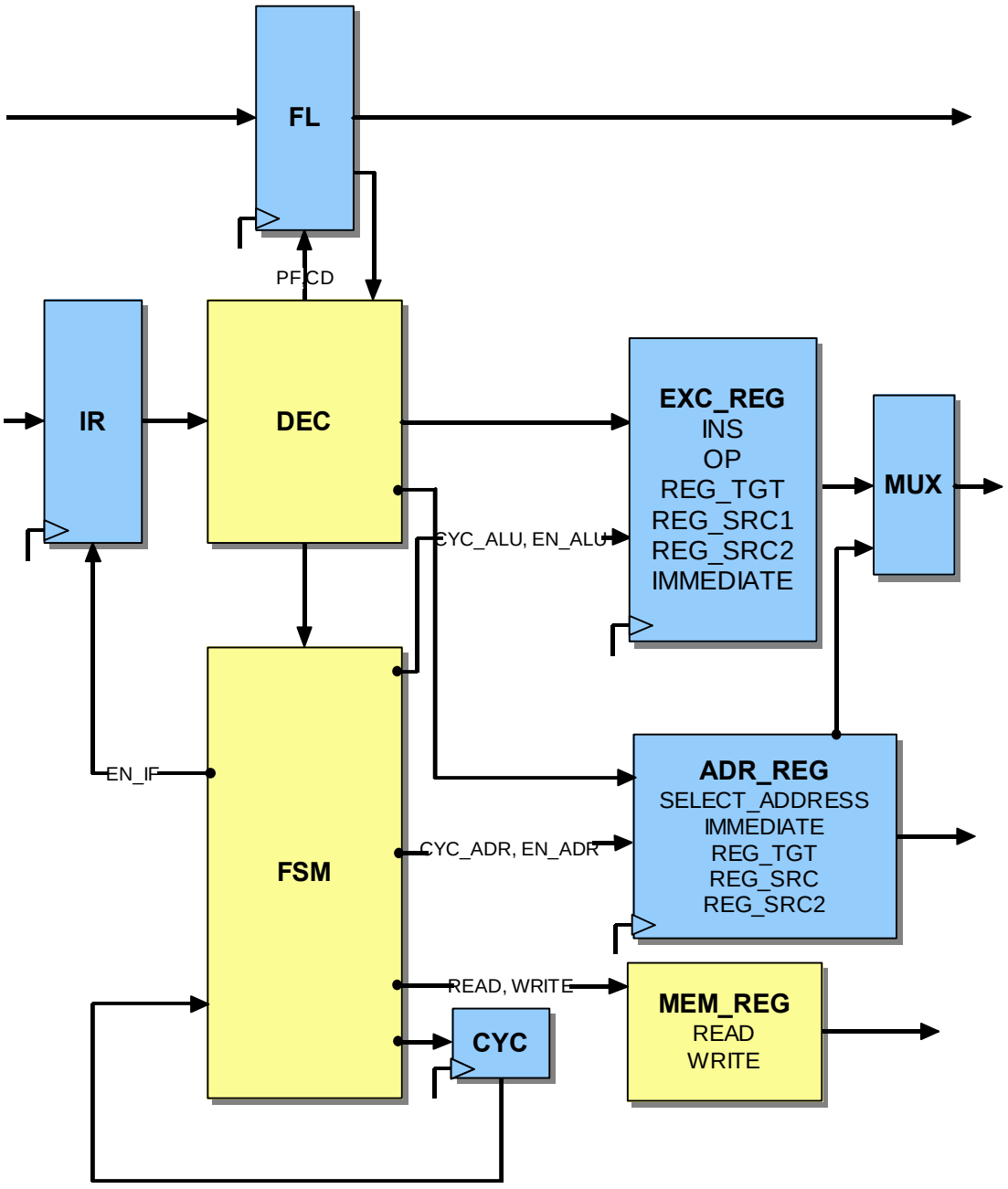

Figure 5.9. Pipelined Control Unit

Chapter 6

Outlook

We have not yet reached the very final version of the ANTARES. There are still a number of steps to be taken before it is running reliably together with the pipelining logic.

Before any changes to the actual architecture can be made, the existing non-pipelined design must be thoroughly debugged, validated, and tested. At the time of writing we are about to put a first version on FPGA. Then we will be able to systematically test and verify all components and functions.

The architectural changes discussed in Section 4.3.2 can be implemented when a valid design is finished. This can be done step by step. Connecting the data bus directly to the register file and introducing a dedicated D-bus from the register file to the addressing unit will reduce the more critical bottlenecks that will arise during pipelined operation. Then the addressing unit must be rewritten in a way, so that it only depends upon the signals delivered by the control unit. These changes will also require reworking of the FSM, the addressing and parts of the execution unit as they will reduce the number of cycles needed for many instructions. Indexed addressing, in particular will become much faster.

After these basic but essential changes we can initiate some other changes that, while not absolutely necessary, enhance the design. Some of them are more local and do not require a lot of effort to implement such as changing the number

of cycles for control instructions and the evaluation of the carry flag during stack pointer incrementation. Other options such as the extension of the register file, will necessitate more global changes within the assembler, compiler, and simulation environment. However, they are worth considering since they will increase access to a whole range of new applications that would otherwise run too slow.

Only when the architecture is fixed, when the length of the instructions is known and when we removed the major dependencies and bottlenecks, can we start to rework the pipelining control. Adjusting the FSM, checking control hazards and at last detecting and handling data hazards will be the major steps. Meticulous test procedures between these steps and, in particular, at the end will conclude the overall process.

Since the changes are made from a fully functional design, we can simultaneously develop application and peripheral units for ANTARES. Since the external interface and timing is not altered at all, we will be able to use these results as soon as the pipelined advanced core becomes available.

Chapter 7

Conclusion

When developing the ANTARES microprocessor core, the focus was on making a core that is compact, easy to handle, open for extensions, fast, small, and strictly low-power. The introduction of the pipelining concept is just another step in this direction.

In this project we developed a pipeline concept that fits into the current architecture. The high-level approach of using several independently operating Finite State Machines that are coordinated by a superordinated unit proved to be valid and could be verified in simulation. We evaluated the existing architecture in order to integrate this concept successfully into ANTARES. Even though the concept closely matches the system, it still requires some changes. In addition to the mandatory changes, we illustrated other possible changes in order to avoid critical bottlenecks.

Once the pipelined ANTARES is running silicon, it will not only be an example of an interesting design or another finished university project, it will represent a serious alternative for commercial microprocessor cores and maybe two years from now it will operate one of your little portable devices without your realizing it.

Appendix A

Antares32 Instruction Set

A.1 Explanations of Abbreviations

R1, R2 = 16 bit Registers: ABl, ABh, CDl, CDh, EFl, EFh
R(1&2) = 32 bit Registers: AB, CD, EF, X, S
FL = Flagregister (8 bit)
N12 = NNN
rrr = 3-bit Register Address
+/-N11 = signed N11
NNN0 = NNN shift left 1
NNN00 = NNN shift left 2
+/-N7 = signed N8
PC = Program counter
M(XXX) = Memory addressed by XXX
PR = Prefixregister (32 bit)

A.2 Register Indices

16 bit Register	ABl	ABh	CDl	CDh	EFl	EFh		
32 bit Register	AB		CD		EF		S	X
Index	000	001	010	011	100	101	110	111

Table A.1. Register Indices

A.3 Flag-Register

Bit	Flag	Description
7	**Pd**	Power Down Mode, Busses in High Impedance, low clock
6	**Se**	Sense, Hardware Sense Line
5	**Ip**	Interrupt pending
4	**Ie**	Interrupt enable
3	**Pf**	Prefix Flag
2	**Cd**	Conditional Jump Flag
1	**OV**	Overflow
0	**Cy**	Carry

Table A.2: Flag Register

All addresses are byte-counts. Constants and addresses with sign (+/–N7, +/–N11) are processed sign-extended.

A.4 Instructions

Type A : Instructions without extensions					
0000000000000000	HLT	Nil	1 ⇒ FL.Pd	Pd	Enter Power Down Mode
0000000000000001	DIS	Nil	0 ⇒ FL.IE	Ie	Disable Interrupt
0000000000000010	ENI	Nil	1 ⇒ FL. IE	Ie	Enable interrupt
0000000000000110	RCF	Nil	0 ⇒ FL.Cd	Cd	Reset Condition Flag
0000000000000111	SCF	Nil	1 ⇒ FL.Cd	Cd	Set Condition Flag
0000000000001000	CSE	Nil.	Se ⇒ FL.Cd	Cd	Copy Sense to Cd
0000000000001001	COV	Nil	OV⇒ FL.Cd	Cd	Compare if Overflow
0000000000001010	CCY	Nil	CY⇒ FL.Cd	Cd	Compare if Carry set
0000000000001100	PSF	Nil	S - 2 ⇒ S Fl ⇒M(S)[7:0]	-	Push Flags on Stack
0000000000001101	PPF	Nil	M(S)[7:0] ⇒ Fl S+2 ⇒ S	Fl	Pop Flags from Stack

Bin./Hex code	Mnem	Exten	Function	Flags	Description
0000000000001110	NOP	Nil	Empty cycle	Nil	No Operation
0000000000001111	RST	Nil	0000FFF8 ⇒ PC, 00000000 ⇒ PR, 00 ⇒ FL	Fl	Reset PC
Type B : Instructions with 8-bit vector on Bus					
00000001|NN	PIN	N8	IO(NN) ⇒ A	-	Input of IO, NN on D-Bus
00001001|NN	POT	N8	A ⇒ IO(NN)	-	Output to IO, NN on D-Bus
00010001|NN	SWI	N8	PC ⇒ M(S) , M(NN0) ⇒ PC, S – 4 ⇒ S	-	Software interrupt, PC on Stack, Vector addr. in NN
00011001|NN	LPR	N8	NN⇒PF[15:8]	Pf	Load Prefix Register short direct
00100001|NN	CLA	N8	S – 4 ⇒ S PC⇒M(S), NN0 ⇒ PC if Pf: PF[31:8]&NN ⇒PC, 0 ⇒ Pf,	Pf	Subroutine call absolute
00101001|NN	RET	N8	M(S) ⇒PC; S+NN ⇒S If Pf: PF[31:8]&NN⇒S 0 ⇒ Pf	Pf	Return from subroutine, correct stack pointer,With prefix
00110001|NN	LPH	N8	NN ⇒ PF[31:24]	Pf	Load Segment high byte
00111001|NN	LPL	N8	NN ⇒ PF[23:16]	Pf	Load Segment low byte
Type C: Instructions working on one register					
00rrr010|00	CLR	R1	0 ⇒ R1	Fl	Clear registers
00rrr010|01	INV	R1	Inv(R1) ⇒ R1	-	Invert bit wise
00rrr010|02	NEG	R1	Inv(R1) + 1 ⇒ R1	-	Two's complement of R1
00rrr010|03	INC	R1	R1 + 1 ⇒ R1	Cy,Ov	Increment
00rrr010|04	DEC	R1	R1 – 1 ⇒ R1	Cy,Ov	Decrement
00rrr010|05	SAR	R1	R1 >>1, R1[14]⇒R1[15] R1[0]⇒Cy	Cy	Shift Arithmetic Right
00rrr010|06	SAL	R1	R1[14:0] <<1, R1[15]=R1[15], Cy⇒R1[0]	-	Shift Arithmetic Left

Bin./Hex code	Mnem	Exten	Function	Flags	Description
00rrr010|07	SLR	R1	R1>>1, 0⇒R1[15]	-	Shift Logical Right
00rrr010|08	SLL	R1	R1<<1, 0 ⇒R1[0]	-	Shift Logical Left
00rrr010|09	SRC	R1	R1[15:1]⇒R1[14:0], Cy⇒ R1[15]	-	Shift log. right with Carry
00rrr010|0A	SLC	R1	R1[14:0]⇒R1[15:1], Cy⇒R1[0]	Cy	Shift log. left with Carry
00rrr010|0B	IVC	R1	Inv(R1)+Cy⇒R1	Cy	Invert with Carry
00rrr010|0C	SWH	R1	R1[7:0] ⇒ R1[15:8]	-	Swap to high byte
00rrr010|0D	SWL	R1	R1[15:8] ⇒ R1[7:0]	-	Swap to low byte
00rrr010|0E	GFL	R1	FL⇒R1[7:0] 0⇒R1[15:8]	-	Get FL to R1
00rrr010|0F	SFL	R1	R1[7:0]⇒FL	all FL	Set R1 to FL
00rrr010|10	CPT	R1	Parity(R1) ⇒ Cd	Cd	Calculate Parity of R1
00rrr010|11	CPS	R1	If (R1>0): 1⇒Cd	Cd	Compare positive
00rrr010|12	CZE	R1	If (R1=0):1⇒Cd	Cd	Compare if zero
00rrr010|13	CBI	R1	If R1[7]: FF⇒R1[15:8]	-	Convert Byte to Int
00rrr010|14	CMI	R1	If (R1<0):1⇒Cd	Cd	Compare if negativ
00rrr010|15	CIL	R(1&2)	If R1[15]:FFFF ⇒ R2 Else 0000 ⇒ R2	-	Convert Int to long
00rrr010|23	JPX	R(1&2)	If Cd : M(R1) ⇒ PC	-	Jump conditional indexed
00rrr010|24	CLX	R(1&2)	S–4⇒S PC⇒ M(S), M(R1) ⇒ PC	-	Call indexed
00rrr010|25	PSH	R1	S – 2 ⇒ S, R1 ⇒ M(S),	-	Decrement Stack Pointer, R1 on Stack
00rrr010|26	POP	R1	M(S) ⇒ R1, S + 2 ⇒ S	-	Get R1 from Stack, Increment Stack Pointer
00rrr010|27	PSL	R(1&2)	S–4⇒S, R(1&2) ⇒ M(SI)	-	Decrement Stack Pointer long, Push combined R(1&2) on Stack

Bin./Hex code	Mnem	Exten	Function	Flags	Description
00rrr010|28	PPL	R(1&2)	M(S)⇒R(1&2) S + 4 ⇒ S	Cy, Cd, Ov	Pop Data Long to combined R(1&2), Increment Stack Pointer long
00rrr010|30	CRL	R(1&2)	0 ⇒ R(1&2)	All Fl	Long clear
00rrr010|31	IVL	R(1&2)	Inv(R(1&2)) ⇒ R(1&2)	Fl	Invert bitwise long
00rrr010|32	NGL	R(1&2)	Inv(R(1&2))+1 ⇒ R(1&2)	Fl	Negate long
00rrr010|33	ICL	R(1&2)	R(1&2)+1 ⇒ R(1&2)	Fl	Increment long
00rrr010|34	DCL	R(1&2)	R(1&2)-1 ⇒ R(1&2)	Fl	Decrement long
00rrr010|35	ARL	R(1&2)	R(1&2)>>1, R2[15]⇒R2[14] R2[0]⇒Cy	Cy	Arithmetic right shift long, keep sign
00rrr010|36	ALL	R(1&2)	R(1&2)<<1, R2[15]⇒R2[15], R1[15]⇒R2[0], 0 ⇒ R1[0]	-	Arithmetic left shift long, keep sign
00rrr010|37	LRL	R(1&2)	R(1&2)>> 1, Cy⇒ R2[15], R2[0]⇒R1[15]	Cy	Logical right shift long with carry
00rrr010|38	LLL	R(1&2)	R(1&2) << 1, Cy ⇒ R1[0], R1[15]⇒R2[0]	Cy	Logical left shift long with carry
00rrr010|39	CPL	R(1&2)	If R(1&2) > 0: 1⇒ Cd else 0 ⇒ Cd	Cd	Compare positive long
00rrr010|3A	CZL	R(1&2)	If R(1&2) = 0: 1⇒ Cd else 0 ⇒ Cd	Cd	Compare on Zero long
00rrr010|3B	CML	R(1&2)	If R(1&2) < 0: 1⇒ Cd else 0 ⇒ Cd	Cd	Compare on Minus long
Type D : Instructions with two Registers addressed					
00rrr011|0|0rrr	MOV	R1, R2	R1 ⇒ R2	-	Transfer
00rrr011|0|1rrr	MVL	R(1&2) R(3&4)	R(1&2) ⇒ R(3&4)	-	Move long
00rrr011|1|0rrr	ADD	R1, R2	R2 + R1 ⇒ R2	Cy, Ov	Add

Bin./Hex code	Mnem	Exten	Function	Flags	Description
00rrr011|1|1rrr	ADL	R(1&2), R(3&4)	R(1&2)+R(3&4)⇒R(3&4)	Cy, Ov	Add long
00rrr011|2|0rrr	LXL	R(1&2), R(3&4)	M(R1&R2)[31:0]⇒R(3&4)	-	Load long indexed
00rrr011|2|1rrr	ANL	R(1&2), R(3&4)	R(1&2) AND R(3&4)⇒ R(3&4)	-	AND long
00rrr011|3|0rrr	SUB	R1, R2	R2 – R1 ⇒ R2	Cy, Ov	Substract
00rrr011|3|1rrr	SBL	R(1&2), R(3&4)	R(3&4)-R(1&2) ⇒ R(3&4)	-	Substract long
00rrr011|4|0rrr	SXL	R(1&2), R(3&4)	R(3&4)⇒M(R1&R2)[32:0]	-	Store long indexed
00rrr011|4|1rrr	ORL	R(1&2), R(3&4)	R(1&2) OR R(3&4) ⇒ R(3&4)	-	OR long
00rrr011|5|0rrr	ANA	R1, R2	R2 AND R1 ⇒ R2	-	Logical and bit wise
00rrr011|5|1rrr	CGL	R(1&2), R(3&4)	If (R(1&2) > R(3&4)) 1 ⇒ Cd, else 0⇒ Cd	Cd	Compare Greater long
00rrr011|6|0rrr	ORA	R1, R2	R2 OR R1 ⇒ R2	-	Logical or bit wise
00rrr011|6|1rrr	CQL	R(1&2), R(3&4)	If (R(1&2) ≥ R(3&4)) 1 ⇒ Cd, else 0⇒ Cd	Cd	Compare Greater or Equal long
00rrr011|7|0rrr	XRA	R1, R2	R2 XOR R1 ⇒ R2	-	Logical xor bit wise
00rrr011|7|1rrr	XRL	R(1&2), R(3&4)	R(1&2) XOR R(3&4) ⇒ R(3&4)	Fl	XOR long
00rrr011|8|0rrr	CEQ	R1, R2	(R1 = R2) ⇒ Cd=1	Cd	Compare if equal
00rrr011|8|1rrr	CEL	R(1&2), R(3&4)	If (R(1&2) = R(3&4)) 1⇒ Cd, else 0⇒ Cd	Cd	Compare Equal long
00rrr011|9|0rrr	CNE	R1, R2	(R1 != R2) ⇒Cd=1	Cd	Compare if non equal
00rrr011|9|1rrr	CNL	R(1&2), R(3&4)	If (R(1&2) <> R(3&4)) 1⇒ Cd, else 0⇒ Cd	Cd	Compare Non Equal long
00rrr011|A|0rrr	CGT	R1, R2	(R1 > R2) ⇒Cd=1	Cd	Compare if greater
00rrr011|A|1rrr	LBX	R(1&2), R3	M(R1&R2)[7:0] ⇒ R3	-	Load byte indexed

Bin./Hex code	Mnem	Exten	Function	Flags	Description
00rrr011|B|0rrr	CGE	R1, R2	(R1 $\geq$ R2) $\Rightarrow$Cd=1	Cd	Compare if greater-equal
00rrr011|B|1rrr	LDX	R(1&2), R3	M(R1&R2)[15:0] $\Rightarrow$ R3	-	Load word indexed
00rrr011|C|0rrr	MPY	R1, R2	(R1*R2)[15:00] $\Rightarrow$ R2	Ov	Unsigned Multiply Short
00rrr011|C|1rrr	MYL	R(1&2), R(3&4)	(R(1&2)*R(3&4))[31:0]$\Rightarrow$ R(3&4)	Ov	Multiply long unsigned
00rrr011|D|0rrr	MSY	R1, R2	(R1*R2)[15:00] $\Rightarrow$ R2	Ov	Signed Multiply
00rrr011|D|1rrr	MSL	R(1&2), R(3&4)	(R(1&2)*R(3&4))[31:0]$\Rightarrow$ R(3&4)	Fl	Multiply long signed
00rrr011|E|0rrr	DIV	R1, R2	R1/R2 $\Rightarrow$ R2	Ov	Division, not implemented
00rrr011|E|1rrr	SBX	R(1&2), R3	R3[7:0] $\Rightarrow$ M(R1&R2)[7:0]	-	Store byte indexed
00rrr011|F|0rrr	DVL	R(1&2), R(3&4)	R(1&2)/R(3&4)$\Rightarrow$ R(3&4)	Ov	Long Division, not implemented
00rrr011|F|1rrr	STX	R(1&2), R3	R3 $\Rightarrow$ M(R1&R2)[15:0]	-	Store word indexed
Type E : Instructions with one register and an immediate 8 bit constant					
01rrr000|NN	LDI	+/- N7,R1	NN$\Rightarrow$R1[7..0], Sign(NN)$\Rightarrow$R1[15..8] If Pf : PF[15:8]&NN$\Rightarrow$R1, 0$\Rightarrow$Pf	Pf	Load an eight bit constant with sign extension, 16 bit constant with prefix,
01rrr001|NN	ADI	N8, R1	R1+NN$\Rightarrow$R1 If Pf:PF[15:8]&NN+R1=R1 0$\Rightarrow$Pf if R1=X or S:32 bit add	Cy,Ov, Pf	Add pos 8-bit constant, add 16 bit constant with prefix flag set, reset prefix flag, For X or S-register: 32 bit add
01rrr010|NN	SBI	N8, R1	R1+NN$\Rightarrow$R1 If Pf: PF[15:8]&NN+R1$\Rightarrow$ R1; 0 $\Rightarrow$ Pf if R1=X or S:32 bit add	Cy,Ov, Pf	Sub pos 8-bit constant, sub 16 bit constant with prefix flag set, reset prefix flag For X or S register: 32 bit substract operation

Bin./Hex code	Mnem	Exten	Function	Flags	Description
01rrr011|NN	ANI	N8, R1	R1[7:0] AND NN ⇒ R1, 0 ⇒ R[15:8] if Pf: (PF[15:8]&NN) AND R1 ⇒ R1 0 ⇒ Pf	Pf	Bit wise AND with 8-bit constant, bitwise and with 16 bit constant with prefix flag set, reset prefix flag
01rrr100|NN	ORI	N8, R1	R1[7:0] OR NN ⇒ R1, 0⇒R1[15:8] if Pf: (PF[15:8]&NN) OR R1 ⇒R1 0⇒Pf	Pf	Bit wise OR with 8-bit constant, bitwise OR with 16 bit constant with prefix flag set, reset prefix flag
01rrr101|NN	SBS	R1, +/-N8	R1[7:0]⇒ M(S+/-NN0)[7:0] if Pf: R1[7:0] ⇒ M(PF[31:8]&NN+S) 0 ⇒ Pf	Pf	Store byte local, with prefix, reset prefix flag
01rrr110|NN	LDS	+/-N8, R1	M(S+/-NN0) ⇒ R1 if Pf: M(PF[31:8]&NN+S) ⇒R1, 0⇒Pf	Pf	Load word local to R1, with prefix, reset prefix flag
01rrr111|NN	STS	R1, +/- N8	R1 ⇒ M(S+/-NN0) if Pf: R1 ⇒ M(PF[31:8]&NN+S) 0⇒Pf	Pf	Store R1 word to local, with prefix, reset prefix flag
10rrr000|NN	LBA	N8,R1	M(NN0) [7:0] ⇒ R1[7:0] if Pf: M[7:0](PF[31:8]&NN) ⇒R1[7:0], 0 ⇒ Pf	Pf	Load byte absolute

Bin./Hex code	Mnem	Exten	Function	Flags	Description
10rrr001|NN	LDA	N8,R1	M(NN0)⇒R1 if Pf: M(PF[31:8]&NN) ⇒R1, 0 ⇒ Pf	Pf	Load word absolute
10rrr010|NN	STA	R1,N8	R1⇒M(NN0) if Pf: R1 ⇒ M(PF[31:8]&NN), 0 ⇒ Pf	Pf	Store word absolute
10rrr011|NN	SBA	R1, N8	R1[7:0]⇒M[7:0](NN0) if Pf: R1[7:0] ⇒ M[7:0](PF[31:8]&NN), 0 ⇒ Pf	Pf	Store byte absolute , with prefix, reset prefix flag
10rrr100|NN	LAL	NN, R(1&2)	M(NN0)⇒R(1&2) if Pf: M(PF[31:8]&NN)⇒ R(1&2), 0 ⇒ Pf	Pf	Load long absolute
10rrr101|NN	STL	R(1&2), NN	R(1&2)⇒M(NN0) If Pf: R(1&2) ⇒ M(PF[31:8]&NN), 0 ⇒ Pf	Pf	Store long absolute
10rrr110|NN	LSL	+/-N8, R(1&2)	M(S+/-NN0)⇒R(1&2), if Pf: M(PF[15:8]&NN+S) ⇒R(1&2), 0 ⇒ Pf	Pf	Load long local
10rrr111|NN	SSL	R(1&2), +/-N7	R(1&2)⇒M(S+/-NN0), If Pf: R(1&2) ⇒ M(PF[15:8]&NN+S), 0 ⇒ Pf	Pf	Store long local
Type F: Instructions with long offset					

Bin./Hex code	Mnem	Exten	Function	Flags	Description
1100|NNN	LEA	N12	NNN0⇒X If Pf: PF[31:8]+NN ⇒ X, 0⇒ Pf	Pf	Load effective address absolute, with prefix, prefix flag reset
1101|NNN	JMP	N12	If Cd: NNN0⇒ PC, If Pf: PF[31:8]+NN ⇒PC, 0 ⇒ Pf	Pf, Cd	Jump absolute conditional, with prefix, prefix flag reset
1110|NNN	JPR	+/-N11	If Cd: NNN0 + PC ⇒ PC, If Pf: PF[31:8]+NN+PC ⇒PC, 0 ⇒ Pf	Pf, Cd	Jump relative conditional, with prefix, prefix flag reset
1111|NNN	CAL	+/-N11	S–4 ⇒ S PC+NNN0 ⇒ PC, PC⇒M(S), if Pf: PF[31:8]+NN+PC ⇒ PC, 0 ⇒ Pf	Pf	Decrement Stack, Subroutine call relative, PC on stack, with prefix, prefix flag reset

Table A.3: Instruction Set Definition

Appendix B

Operation Types and Cycles

Instruction	Cycle	Description of resources used
Operation type: CONTROL		
HLT	INF,IND,STP	cycle INF ⟶ address bus and data bus
DIS	INF,IND,EXC0	cycle IND ⟶ Instruction Decoder
ENI	INF,IND,EXC0	
RCF	INF,IND,EXC0	
SCF	INF,IND,EXC0	
CSE	INF,IND,EXC0	
COV	INF,IND,EXC0	
CCY	INF,IND,EXC0	
NOP	INF,IND,EXC0	
RST	INF,IND,EXC0	
LPR	INF,IND,EXC0	
LPH	INF,IND,EXC0	
LPL	INF,IND,EXC0	
Operation type: INDXS1		
PSF	INF,IND,EXC1,EXC2,DST0	cycle INF ⟶ address bus and data bus
PSH	INF,IND,EXC1,EXC2,DST0	cycle IND ⟶ Instruction Decoder
		cycle EXC1 ⟶ A bus, B bus and C bus
		cycle EXC2 ⟶ A bus, B bus and C bus
		cycle DST0 ⟶ address bus and data bus

Instruction	Cycle	Description of resources used
Operation type: INDXF1		
PPF POP	INF,IND,EXC1,EXC2,EXC3,DFT0 INF,IND,EXC1,EXC2,EXC3,DFT0	cycle INF ⟶ address bus and data bus cycle IND ⟶ Instruction Decoder cycle EXC1 ⟶ A bus and C bus cycle EXC2 ⟶ A bus, B bus and C bus cycle EXC3 ⟶ B bus cycle DFT0 ⟶ address bus and data bus
Operation type: INDXS2		
SWI CLA CLX PSL CAL	INF,IND,EXC1,EXC2,DST1,DST2,DFT1,DFT2 INF,IND,EXC1,EXC2,DST1,DST2,ADC1 INF,IND,EXC1,EXC2,DST1,DST2,DFT1,DFT2 INF,IND,EXC1,EXC2,DST1,DST2 INF,IND,EXC1,EXC2,DST1,DST2,ADC1	cycle INF ⟶ address bus and data bus cycle IND ⟶ Instruction Decoder cycle EXC1 ⟶ A bus, B bus and C bus cycle EXC2 ⟶ A bus, B bus and C bus cycle DST1 ⟶ address bus and data bus cycle DST2 ⟶ address bus and data bus cycle DFT1 & DFT2 ⟶ address bus and data bus cycle ADC1 ⟶ PC and immediate
Operation type: INDXF2		
RET PPL	INF,IND,EXC1,EXC2,EXC3,DFT1,DFT2 INF,IND,EXC1,EXC2,EXC3,DFT1,DFT2	cycle INF ⟶ address bus and data bus cycle IND ⟶ Instruction Decoder cycle EXC1 ⟶ A bus and C bus cycle EXC2 ⟶ A bus, B bus and C bus cycle EXC3 ⟶ B bus cycle DFT1 & DFT2 ⟶ address bus and data bus
Operation type: REGREG0		
CLR INV NEG INC DEC SAR	INF,IND,EXC0 INF,IND,EXC0 INF,IND,EXC0 INF,IND,EXC0 INF,IND,EXC0 INF,IND,EXC0	cycle INF ⟶ address bus and data bus cycle IND ⟶ Instruction Decoder cycle EXC0 ⟶ A bus, C bus and ALU

Instruction	Cycle	Description of resources used
SAL	INF,IND,EXC0	
SLR	INF,IND,EXC0	
SLL	INF,IND,EXC0	
SRC	INF,IND,EXC0	
SLC	INF,IND,EXC0	
IVC	INF,IND,EXC0	
SWH	INF,IND,EXC0	
SWL	INF,IND,EXC0	
CPT	INF,IND,EXC0	
CPS	INF,IND,EXC0	
CZE	INF,IND,EXC0	
CBI	INF,IND,EXC0	
CMI	INF,IND,EXC0	
CIL	INF,IND,EXC0	
Operation type: REGREG1		
CRL	INF,IND,EXC1,EXC2	cycle INF $\longrightarrow$ address bus and data bus
IVL	INF,IND,EXC1,EXC2	cycle IND $\longrightarrow$ Instruction Decoder
NGL	INF,IND,EXC1,EXC2	cycle EXC1 $\longrightarrow$ A bus, C bus and ALU
ICL	INF,IND,EXC1,EXC2	cycle EXC2 $\longrightarrow$ A bus, C bus and ALU
DCL	INF,IND,EXC1,EXC2	
ARL	INF,IND,EXC1,EXC2	
ALL	INF,IND,EXC1,EXC2	
LRL	INF,IND,EXC1,EXC2	
LLL	INF,IND,EXC1,EXC2	
CPL	INF,IND,EXC1,EXC2	
CZL	INF,IND,EXC1,EXC2	
CML	INF,IND,EXC1,EXC2	
Operation type: REGREG2		
GFL	INF,IND,EXC0	cycle INF $\longrightarrow$ address bus and data bus
SFL	INF,IND,EXC0	cycle IND $\longrightarrow$ Instruction Decoder
ADD	INF,IND,EXC0	cycle EXC0 $\longrightarrow$ A bus, B bus, C bus and ALU
MOV	INF,IND,EXC0	

Instruction	Cycle	Description of resources used
SUB	INF,IND,EXC0	
ANA	INF,IND,EXC0	
ORA	INF,IND,EXC0	
XRA	INF,IND,EXC0	
CEQ	INF,IND,EXC0	
CNE	INF,IND,EXC0	
CGT	INF,IND,EXC0	
CGE	INF,IND,EXC0	
LDI	INF,IND,EXC0	
ADI	INF,IND,EXC0	
SBI	INF,IND,EXC0	
ANI	INF,IND,EXC0	
ORI	INF,IND,EXC0	
Operation type: REGREG3		
MVL	INF,IND,EXC1,EXC2	cycle INF $\longrightarrow$ address bus and data bus
ADL	INF,IND,EXC1,EXC2	cycle IND $\longrightarrow$ Instruction Decoder
ANL	INF,IND,EXC1,EXC2	cycle EXC1 $\longrightarrow$ A bus, B bus, C bus and ALU
SBL	INF,IND,EXC1,EXC2	cycle EXC2 $\longrightarrow$ A bus, B bus, C bus and ALU
ORL	INF,IND,EXC1,EXC2	
CGL	INF,IND,EXC1,EXC2	
CQL	INF,IND,EXC1,EXC2	
XRL	INF,IND,EXC1,EXC2	
CEL	INF,IND,EXC1,EXC2	
CNL	INF,IND,EXC1,EXC2	
LEA	INF,IND,EXC1,EXC2	
Operation type: LOAD0		
LBA	INF,IND,DFT0	cycle INF $\longrightarrow$ address bus and data bus
LDA	INF,IND,DFT0	cycle IND $\longrightarrow$ Instruction Decoder
LAL	INF,IND,DFT1,DFT2	cycle DFT0 $\longrightarrow$ address bus, data bus and C bus
		cycle DFT1 $\&$ DFT2 $\longrightarrow$ address bus and data bus and C bus

Instruction	Cycle	Description of resources used
Operation type: LOAD1		
LBX LDX LDS	INF,IND,EXC1,EXC2,DFT0 INF,IND,EXC1,EXC2,DFT0 INF,IND,EXC1,EXC2,ADC1,DFT0	cycle INF ⟶ address bus and data bus cycle IND ⟶ Instruction Decoder cycle EXC1 & EXC2 ⟶ B bus cycle ADC1 ⟶ address_bus_int and immediate cycle DFT0 ⟶ address bus and data bus and C bus
Operation type: LOAD2		
JPX LXL LSL	INF,IND,EXC1,EXC2,DFT1,DFT2 INF,IND,EXC1,EXC2,DFT1,DFT2 INF,IND,EXC1,EXC2,ADC1,DFT1,DFT2	cycle INF ⟶ address bus and data bus cycle IND ⟶ Instruction Decoder cycle EXC1 & EXC2 ⟶ B bus cycle ADC1 ⟶ address_bus_int and immediate cycle DFT1 & DFT2 ⟶ address bus and data bus and C bus
Operation type: STORE0		
STA SBA STL	INF,IND,DST0 INF,IND,DST0 INF,IND,DST1,DST2	cycle INF ⟶ address bus and data bus cycle IND ⟶ Instruction Decoder cycle DST1 & DST2 ⟶ address bus and data bus and A bus
Operation type: STORE1		
SBX STX SBS STS	INF,IND,EXC1,EXC2,DST0 INF,IND,EXC1,EXC2,DST0 INF,IND,EXC1,EXC2,ADC1,DST0 INF,IND,EXC1,EXC2,ADC1,DST0	cycle INF ⟶ address bus and data bus cycle IND ⟶ Instruction Decoder cycle EXC1 & EXC2 ⟶ B bus cycle ADC1 ⟶ address_bus_int and immediate cycle DST0 ⟶ address bus and data bus and A bus
Operation type: STORE2		
SXL SSL	INF,IND,EXC1,EXC2,DST1,DST2 INF,IND,EXC1,EXC2,DST1,DST2	cycle INF ⟶ address bus and data bus cycle IND ⟶ Instruction Decoder cycle EXC1 & EXC2 ⟶ B bus

Instruction	Cycle	Description of resources used
		cycle DST1 & DST2 ⟶ address bus, data bus and A bus
Operation type: JUMP		
JMP JPR	INF,IND,ADC0 INF,IND,ADC0	cycle INF ⟶ address bus and data bus cycle IND ⟶ Instruction Decoder cycle ADC0 ⟶ PC and immediate

Table B.1: Operation Types and Cycles

Appendix C

VHDL Codes

C.1 Finite State Machine

```
------------------------------------------------------------------
------------------------------------------------------------------
--    ANTARES 32 bit Project                                    ---
--    .                                                         ---
--    Author:      David Berner                                 ---
--    CREATION:    14.06.2002                                   ---
--    MODULE:      Control FSM                                  ---
--    DESCRIPTION: Finite State Machine controlling the state   ---
--                 transitions for each instruction             ---
--    .                                                         ---
--    Modified:    26.08.2002                                   ---
------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use work.std_logic_arith.all;
use work.antares_definitions.all;

entity control_fsm is
  port(
    op         : in  operation_type;
```

```
    instr_dec  : in  instr_type;
    current_st : in  cycle_type;
    alu        : in  std_ulogic;
    mem        : in  std_ulogic;
    enable     : in  std_ulogic;
    ready      : out std_ulogic;
    next_st    : out cycle_type
    );

end control_fsm;

architecture control_fsm_behave of control_fsm is
  signal adc : std_ulogic;

  function ind_to_exc1(op : operation_type) return boolean is
    -- condition for transition from IND to EXC1 cycle
    variable tmp          : boolean;
  begin
    tmp := ((op = LOAD1) or
            (op = STORE1) or
            (op = LOAD2) or
            (op = STORE2) or
            (op = INDXS1) or
            (op = INDXS2) or
            (op = INDXF1) or
            (op = INDXF2) or
            (op = REGREG1) or
            (op = REGREG3));
    return (tmp);
  end ind_to_exc1;

  function ind_to_exc0(op : operation_type) return boolean is
    -- condition for transition from IND to EXC0 cycle
  begin
    return ((op = REGREG0) or (op = REGREG2) or (op = CONTROL));
```

```
  end ind_to_exc0;

  function ind_to_adc0(op : operation_type) return boolean is
    -- condition for transition from IND to ADC0 cycle
  begin
    return ((op = JUMP) or (op = LOAD0) or (op = STORE0));
  end ind_to_adc0;
  signal next_state       : cycle_type;

begin
  next_st <= next_state;

  ------------------------------------------------------------------
  nextstate : process (
    op,
    alu,
    current_st,
    instr_dec,
    mem,
    enable
    )
    ------------------------------------------------------------------
  begin

    if enable = '0' then
      -- when the enable goes to '0' we set ready to '0'
      -- when enable goes to one, ready will be set to '1'
      -- only after calculation of the new result.
      ready <= '0';

    else
      next_state <= current_st;

      -- this is the FSM-logic generated by FPGA-advantage
      -- Combined Actions
```

```
-- wait for 5 ns;
case current_st is
  when INF                          =>
    next_state                       <= IND;
  when IND                          =>
    if (((ind_to_exc0(OP) or ind_to_exc1(OP)) and
         ( alu = '1')) or
        (OP = JUMP and (adc = '1'))) then
      next_state                     <= W1;
    elsif ((instr_dec = STL) and (mem = '1')) then
      next_state                     <= W2;
    elsif (((instr_dec = STA) or (instr_dec = SBA) or
            (OP = LOAD0)) and (mem = '1')) then
      next_state                     <= W4;
    elsif (ind_to_exc0(OP)) then
      next_state                     <= EXC0;
    elsif (ind_to_exc1(OP)) then
      next_state                     <= EXC1;
    elsif (OP = JUMP) then
      next_state                     <= ADC0;
    elsif (instr_dec = HLT) then
      next_state                     <= STP;
    elsif (instr_dec = LBA or instr_dec = LDA) then
      next_state                     <= DFT0;
    elsif (instr_dec = LAL) then
      next_state                     <= DFT1;
    elsif (instr_dec = STA or instr_dec = SBA) then
      next_state                     <= DST0;
    elsif (instr_dec = STL) then
      next_state                     <= DST1;
    else
      next_state                     <= IND;
    end if;
  when EXC0                         =>
    if (mem = '1') then
```

```
    next_state                      <= W5;
  else
    next_state                      <= INF;
  end if;
when EXC1                          =>
  next_state                        <= EXC2;
when EXC2                          =>
  if (mem = '1' and (OP = INDXS2 or OP = STORE2)) then
    next_state                      <= W2;
  elsif (OP = INDXS2 or OP = STORE2) then
    next_state                      <= DST1;
  elsif (( instr_dec = LDS or instr_dec = LSL or
           instr_dec = STS or instr_dec = SBS)
         and adc = '1') then
    next_state                      <= W3;
  elsif (instr_dec = LDS or instr_dec = LSL or
         instr_dec = STS or instr_dec = SBS) then
    next_state                      <= ADC1;
  elsif (OP = INDXF1 or (op = INDXF2)) then
    next_state                      <= EXC3;
  elsif ((OP = INDXS1 or OP = STORE1 or OP = LOAD1 or
          OP = INDXF2 or OP = LOAD2) and ( mem = '1')) then
    next_state                      <= W4;
  elsif (OP = INDXS1 or OP = STORE1) then
    next_state                      <= DST0;
  elsif (OP = LOAD1) then
    next_state                      <= DFT0;
  elsif (OP = INDXF2 or OP = LOAD2) then
    next_state                      <= DFT1;
  elsif (mem = '1') then
    next_state                      <= W5;
  else
    next_state                      <= INF;
  end if;
when DST0                          =>
```

```
  if (mem = '1') then
    next_state                      <= W5;
  else
    next_state                      <= INF;
  end if;
when DST1                          =>
  next_state                        <= DST2;
when DST2                          =>
  if (((instr_dec = CLX) or (instr_dec = SWI)) and
      (mem = '1')) then
    next_state                      <= DFT1;
  elsif (( (instr_dec = CLA or instr_dec = CAL) and
           adc = '1')) then
    next_state                      <= W3;
  elsif ((instr_dec = CLA) or (instr_dec = CAL)) then
    next_state                      <= ADC1;
  elsif (((instr_dec = CLX) or (instr_dec = SWI)) and
         (mem = '1')) then
    next_state                      <= W4;
  elsif (mem = '1') then
    next_state                      <= W5;
  else
    next_state                      <= INF;
  end if;
when DFT1                          =>
  next_state                        <= DFT2;
when DFT0                          =>
  if (mem = '1') then
    next_state                      <= W5;
  else
    next_state                      <= INF;
  end if;
when DFT2                          =>
  if (mem = '1') then
    next_state                      <= W5;
```

```
  else
    next_state                      <= INF;
  end if;
when ADC1                          =>
  if ((OP = INDXF2 or OP = LOAD2 or OP = STORE1 or
       OP = LOAD1 or OP = INDXF1) and mem = '1') then
    next_state                      <= W4;
  elsif (OP = STORE1) then
    next_state                      <= DST0;
  elsif (OP = LOAD2) then
    next_state                      <= DFT1;
  elsif (OP = LOAD1) then
    next_state                      <= DFT0;
  elsif (mem = '1') then
    next_state                      <= W5;
  else
    next_state                      <= INF;
  end if;
when ADC0                          =>
  if (mem = '1') then
    next_state                      <= W5;
  else
    next_state                      <= INF;
  end if;
when STP                           =>
  next_state                        <= STP;
when W1                            =>
  if (((ind_to_exc0(OP) or ind_to_exc1(OP)) and
       ( alu = '1')) or
      (OP = JUMP and (adc = '1'))) then
    next_state                      <= W1;
  elsif (ind_to_exc1(OP)) then
    next_state                      <= EXC1;
  elsif (ind_to_exc0(OP)) then
    next_state                      <= EXC0;
```

```
    elsif (OP = JUMP) then
      next_state                        <= ADC0;
    else
      next_state                        <= W1;
    end if;
  when W5                              =>
    if (mem = '1') then
      next_state                        <= W5;
    else
      next_state                        <= INF;
    end if;
  when W2                              =>
    if (mem = '1') then
      next_state                        <= W2;
    elsif (mem = '0') then
      next_state                        <= DST1;
    else
      next_state                        <= W2;
    end if;
  when W4                              =>
    if (mem = '1') then
      next_state                        <= W4;
    elsif ((mem = '0') and (OP = STORE1 or OP = INDXS1 or instr_dec = STA
                            or instr_dec = SBA) then
      next_state                          <= DST0;
    elsif ((mem = '0') and (OP = INDXF2 or OP = LOAD2 or
            instr_dec = LAL or instr_dec = STL) then
      next_state                          <= DFT1;
    elsif ((mem = '0') and (OP = LOAD1 or OP = INDXF1 or
                            instr_dec = LBA or instr_dec = LDA) then
      next_state <= DFT0;
    else
      next_state <= W4;
    end if;
  when W3     =>
```

```
          if (adc = '1') then
            next_state <= W3;
          else
            next_state <= ADC1;
          end if;
        when EXC3   =>
          next_state   <= DFT0;
        when others =>
          next_state   <= W5;
      end case;
        ready        <= '1'after 2 ns;
        -- this tells the pipelined control that the FSM has finished
    end if;
  end process nextstate;
end control_fsm_behave;
```

C.2 Arbiter

```
-------------------------------------------------------------
-------------------------------------------------------------
--    ANTARES 32 bit Project                               ---
--    .                                                    ---
--    Author:      David Berner                            ---
--    CREATION:    15.06.2002                              ---
--    MODULE:      Arbiter with multimode FSM control      ---
--    DESCRIPTION: Control for pipelined operation         ---
--    .                                                    ---
--    Modified:    26.08.2002                              ---
-------------------------------------------------------------
-------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use work.std_logic_arith.all;
```

```
use work.antares_definitions.all;

entity control_pipe is
  port(
    clock   : in  std_ulogic;
    reset   : in  std_ulogic;
    op      : in  operation_type;
    ins     : in  instr_type;
    cyc_adr : out cycle_type;
    cyc_exc : out cycle_type;
    en_adr  : out std_ulogic;
    en_exc  : out std_ulogic;
    en_inf  : out std_ulogic;
    read    : out std_ulogic;
    write   : out std_ulogic
    );
end control_pipe;

architecture control_pipe_behave of control_pipe is

  component control_fsm
    port(
      op         : in  operation_type;
      instr_dec  : in  instr_type;
      current_st : in  cycle_type;
      alu        : in  std_ulogic;
      mem        : in  std_ulogic;
      enable     : in  std_ulogic;
      ready      : out std_ulogic;
      next_st    : out cycle_type
      );
  end component;

  signal cycle1      : cycle_type := W5;
  signal cycle2      : cycle_type := W5;
```

```
  signal cycle3      : cycle_type := W5;
  signal cycle1_int  : cycle_type := W5;
  signal cycle2_int  : cycle_type := W5;
  signal cycle3_int  : cycle_type := W5;
  signal next_cycle1 : cycle_type;
  signal next_cycle2 : cycle_type;
  signal next_cycle3 : cycle_type;

  signal op1 : operation_type := CONTROL;
  signal op2 : operation_type := CONTROL;
  signal op3 : operation_type := CONTROL;

  signal ins1 : instr_type := NOP;
  signal ins2 : instr_type := NOP;
  signal ins3 : instr_type := NOP;

  signal alu_use : std_ulogic := '0';
  signal mem_use : std_ulogic := '0';
  signal inf_use : std_ulogic := '0';

  signal done1 : std_ulogic := '0';
  signal done2 : std_ulogic := '0';
  signal done3 : std_ulogic := '0';

  signal en1 : std_ulogic := '1';
  signal en2 : std_ulogic := '1';
  signal en3 : std_ulogic := '1';

  signal ready  : std_ulogic := '0';
  signal ready1 : std_ulogic := '0';
  signal ready2 : std_ulogic := '0';
  signal ready3 : std_ulogic := '0';

  function uses_alu(state : cycle_type) return boolean is
-- checks if a cycle uses the ALU
```

```
  begin
    return ((state = EXC0) or (state = EXC1) or (state = EXC2)or
            (state = EXC3));
  end uses_alu;

  function uses_mem(state : cycle_type) return boolean is
-- checks if a cycle uses the addressing
  begin
    return ((state = DST0) or (state = DST1) or (state = DST2)or
            (state = DFT0) or (state = DFT1)or (state = dft2) or
            (state = adc0)or  (state = adc1)or (state = inf));
  end uses_mem;

  function mem_read(state : cycle_type) return boolean is
-- checks if a cycle reads from memory
  begin
    return ((state = DFT0)or (state = DFT1)or (state = dft2)or
            (state = inf));
  end mem_read;

  function mem_write(state : cycle_type) return boolean is
-- checks if a cycle writes to memory
  begin
    return ((state = DST0)or (state = DST1)or (state = DST2));
  end mem_write;



begin
-- instantiation of threee FSM's
  control_fsm1 : control_fsm
    port map(op         => op1,
             instr_dec  => ins1,
             current_st => cycle1,
             alu        => alu_use,
             mem        => mem_use,
```

```
            enable     => en1,
            ready      => ready1,
            next_st    => next_cycle1);

  control_fsm2 : control_fsm
    port map(op         => op2,
            instr_dec  => ins2,
            current_st => cycle2,
            alu        => alu_use,
            mem        => mem_use,
            enable     => en2,
            ready      => ready2,
            next_st    => next_cycle2);

  control_fsm3 : control_fsm
    port map(op         => op3,
            instr_dec  => ins3,
            current_st => cycle3,
            alu        => alu_use,
            mem        => mem_use,
            enable     => en3,
            ready      => ready3,
            next_st    => next_cycle3);

  set_io : process(cycle1, cycle2, cycle3)
  begin
-- it is determined where the input signals "op" and "ins" go.
-- they go to the FSM which is currently in the "IND" state
    if (cycle1 = IND) then
      op1  <= op;
      ins1 <= ins;
    elsif (cycle2 = IND) then
      op2  <= op;
      ins2 <= ins;
    elsif (cycle3 = IND) then
```

```
      op3  <= op;
      ins3 <= ins;
    end if;

-- here we set the appropriate output signals
-- for execution ...
    if uses_alu(cycle1) then
      cyc_exc <= cycle1;
      en_exc  <= '1';
    elsif uses_alu(cycle2) then
      cyc_exc <= cycle2;
      en_exc  <= '1';
    elsif uses_alu(cycle3) then
      cyc_exc <= cycle3;
      en_exc  <= '1';
    else
      en_exc  <= '0';
    end if;
-- ... and addressing
    if uses_mem(cycle1)or cycle1 = inf then
      cyc_adr <= cycle1;
      en_adr  <= '1';
    elsif uses_mem(cycle2)or cycle1 = inf then
      cyc_adr <= cycle2;
      en_adr  <= '1';
    elsif uses_mem(cycle3)or cycle1 = inf then
      cyc_adr <= cycle3;
      en_adr  <= '1';
    else
      en_adr  <= '0';
    end if;

-- tell the memory to read or write
    if (mem_read(cycle1) or mem_read(cycle2) or mem_read(cycle3)) then
      read  <= '1';
```

```
      write <= '0';
    elsif (mem_write(cycle1_int) or mem_write(cycle2)
          or mem_write(cycle3)) then
      write <= '1';
      read  <= '0';
    else
      read  <= '0';
      write <= '0';
    end if;

-- enable the Instruction register
    if (cycle1 = INF) then
      en_inf <= '1';
    elsif (cycle2 = INF) then
      en_inf <= '1';
    elsif (cycle3 = INF) then
      en_inf <= '1';
    else
      en_inf <= '0';
    end if;

  end process set_io;

  nextstate : process(clock, reset, ready1, ready2, ready3)
  begin
    if rising_edge(clock) then
      -- going to the next cycle
      cycle1 <= cycle1_int;
      cycle2 <= cycle2_int;
      cycle3 <= cycle3_int;

      en1     <= '1';
      en2     <= '0';
      en3     <= '0';
      done1   <= '0';
```

```
    done2   <= '0';
    done3   <= '0';
    alu_use <= '0';
    mem_use <= '0';
    ready   <= '0';
  end if;

  if reset = '0' then
    -- reset all FSMs to waitstate 5
    cycle1 <= W5;
    cycle2 <= W5;
    cycle3 <= W5;

  end if;

  -- now the next cycle of the FSMs are determined starting with FSM1
  -- in a first run we give priority to DFT2, DST2 and EXC2
  if ready1 = '1' then
    en1                 <= '0';
    if ready = '0' then
      if (((next_cycle1 = EXC2)or(next_cycle1 = EXC3)and
           (alu_use = '0')))then
             cycle1_int <= next_cycle1;
             done1      <= '1';
             alu_use    <= '1';
           elsif (((next_cycle1 = DFT2 )or( next_cycle1 = DST2))and
                  (mem_use = '0')) then
             cycle1_int <= next_cycle1;
             mem_use    <= '1';
             done1      <= '1';
           elsif ((op1 = JUMP) and (next_cycle1 = ADC0)) then
             mem_use    <= '1';
             cycle1_int <= next_cycle1;
             cycle2_int <= W5;
             cycle3_int <= W5;
```

```
  ready      <= '1';
  en1        <= '1';
  done1      <= '1';
  done2      <= '1';
  done3      <= '1';
end if;
elsif (ready = '1')and (done1 = '0') then
  cycle1_int <= next_cycle1;
  if (uses_alu(next_cycle1)) then
    alu_use  <= '1';
  elsif (uses_mem(next_cycle1) or (next_cycle1 = INF)) then
    mem_use  <= '1';
  end if;
end if;
ready        <= '1';
en1          <= '1';

elsif ready2 = '1' then
  en2            <= '0';
  if ready = '0' then
    if ((next_cycle2 = EXC2) and (alu_use = '0')) then
      cycle2_int <= next_cycle2;
      alu_use    <= '1';
      done2      <= '1';
    elsif (((next_cycle2 = DFT2) or (next_cycle2 = DST2))
          and (mem_use = '0'))then
      cycle2_int <= next_cycle2;
      mem_use    <= '1';
      done2      <= '1';
    elsif ((op2 = JUMP) and (next_cycle2 = ADC0)) then
      mem_use    <= '1';
      cycle2_int <= next_cycle2;
      cycle3_int <= W5;
      cycle1_int <= W5;
      done1      <= '1';
```

```
        done2      <= '1';
        done3      <= '1';
      end if;

    elsif (ready = '1')and done2 = '0' then
      cycle2_int <= next_cycle2;
      if (uses_alu(next_cycle2)) then
        alu_use  <= '1';
      elsif (uses_mem(next_cycle2) or (next_cycle2 = INF)) then
        mem_use  <= '1';
      end if;
    end if;
    en3          <= '1';

  elsif (ready3 = '1')then
    en3            <= '0';
    if ready = '0' then
      if ((next_cycle3 = EXC2) and (alu_use = '0')) then
        cycle3_int <= next_cycle3;
        alu_use    <= '1';
        done3      <= '1';
      elsif (((next_cycle3 = DFT2) or (next_cycle3 = DST2))
             and (mem_use = '0')) then
        cycle3_int <= next_cycle3;
        mem_use    <= '1';
        done3      <= '1';
      elsif ((op3 = JUMP )and (next_cycle3 = ADC0)) then
        mem_use    <= '1';
        cycle3_int <=next_cycle3;
        cycle1_int<=W5;
        cycle2_int<=W5;
        done1<='1';
        done2<='1';
        done3<='1';
      end if;
```

```
            ready<='1'; -- start the second run
            en1<='1';
          elsif (ready = '1') and done3='0' and (en1='0') then
            cycle3_int   <= next_cycle3;
            if (uses_alu(next_cycle3)) then
              alu_use    <= '1';
            elsif (uses_mem(next_cycle3) or (next_cycle3 = INF)) then
              mem_use    <= '1';
            end if;
          end if;
        end if;

        end process nextstate;
        end control_pipe_behave;
```

C.3 Control Unit

```
-----------------------------------------------
-----------------------------------------------
--    ANTARES 32 bit Project               ---
--                                         ---
--    Author:      David Berner            ---
--    CREATION:    15.06.2002              ---
--    MODULE:      Control unit            ---
--    DESCRIPTION: Control unit top sheet  ---
--                                         ---
--    Modified:    24.07.2002              ---
-----------------------------------------------
-----------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use work.std_logic_arith.all;
```

```
use work.antares_definitions.all;

entity control_unit is
  port(
    clock         : in  std_ulogic;
    reset         : in  std_ulogic;
    data_bus      : in  signed (15 downto 0);
    cd_IN         : in  std_ulogic;     --condition flag in
    se_in         : in  std_ulogic;     --sense flag
    intr          : in  std_ulogic;
    cy_in         : in  std_ulogic;     --carry in
    cy_out        : out std_ulogic;     --carry out
    cd_out        : out std_ulogic;     --condition flag out
    ins_exc_out   : out instr_type;
    cyc_exc_out   : out cycle_type;
    op_exc_out    : out operation_type;
    immediate_exc : out signed (31 downto 0);
    ins_adr_out   : out instr_type;
    cyc_adr_out   : out cycle_type;
    op_adr_out    : out operation_type;
    immediate_adr : out signed (31 downto 0);
    reg_tgt_out   : out std_ulogic_vector (2 downto 0);
    reg_src1_out  : out std_ulogic_vector (2 downto 0);
    reg_src2_out  : out std_ulogic_vector (2 downto 0);
    PF_OUT        : out std_ulogic;
    read          : out std_ulogic;
    IO            : out std_ulogic;
    intr_ack      : out std_ulogic;
    write         : out std_ulogic
    );

end control_unit;

architecture control_unit_behave of control_unit is
  component control_pipe
```

```
  port(
    clock   : in  std_ulogic;
    reset   : in  std_ulogic;
    op      : in  operation_type;
    ins     : in  instr_type;
    cyc_adr : out cycle_type;
    cyc_exc : out cycle_type;
    en_exc  : out std_ulogic;
    en_adr  : out std_ulogic;
    en_inf  : out std_ulogic;
    read    : out std_ulogic;
    write   : out std_ulogic
    );
end component;

signal Instr_Reg                    : std_ulogic_vector(15 downto 0);
signal cycle_exc, cycle_adr         : cycle_type;
signal cyc_exc_int, cyc_adr_int     : cycle_type;
signal ins_int, instr_exc, instr_adr : instr_type;
signal reg_tgt, reg_src1, reg_src2  : std_ulogic_vector(2 downto 0);
signal reg_src1_adr, reg_src2_adr   : std_ulogic_vector(2 downto 0);
signal reg_src1_int, reg_src2_int   : std_ulogic_vector(2 downto 0);
signal reg_tgt_int, reg_tgt_adr     : std_ulogic_vector(2 downto 0);
signal immediate_int                : std_ulogic_vector(31 downto 0);
signal op_int, operat_exc, operat_adr : operation_type;
signal Default_src1_reg             : std_ulogic_vector(2 downto 0)
  := "000";
signal Default_src2_reg             : std_ulogic_vector(2 downto 0)
  := "001";
signal Default_tgt_reg              : std_ulogic_vector(2 downto 0)
  := "000";
signal flag_reg                     : std_ulogic_vector(7 downto 0);
signal PF_int, IE_int, Cd_int       : std_ulogic;
signal ov_int, se_int, CY_int       : std_ulogic;
signal ip_int                       : std_ulogic;
```

```
  signal pd_int                        : std_ulogic;
  signal read_int, write_int           : std_ulogic;
  signal en_exc                        : std_ulogic;
  signal en_adr                        : std_ulogic;
  signal en_inf                        : std_ulogic;
  signal pc_mode                       : pc_mode_type;
  signal adr_mode                      : adr_mode_type;
-- signal mem : std_ulogic := '0';
-- signal adc : std_ulogic := '0';
-- signal alu : std_ulogic := '0';

  function increment (incre_reg : std_ulogic_vector(2 downto 0))
    return std_ulogic_vector is
    variable func_output        : std_ulogic_vector(2 downto 0);
  begin
    case incre_Reg is

      when "000"  =>
        func_output := "001";
      when "001"  =>
        func_output := "010";
      when "010"  =>
        func_output := "011";
      when "011"  =>
        func_output := "100";
      when "100"  =>
        func_output := "101";
      when "101"  =>
        func_output := "101";
      when "110"  =>
        func_output := "110";
      when "111"  =>
        func_output := "111";
      when others =>
        func_output := "000";
```

```
    end case;
    return(func_output);
  end increment;

begin

  control_pipe1 : control_pipe
    port map(
      clock   => clock,
      reset   => reset,
      op      => op_int,
      ins     => ins_int,
      cyc_adr => cyc_adr_int,
      cyc_exc => cyc_exc_int,
      en_exc  => en_exc,
      en_adr  => en_adr,
      en_inf  => en_inf,
      read    => read_int,
      write   => write_int
      );

-------------------- INSTRUCTION DECODE  ------------------------------
  instr_dec_process : process(instr_reg)
  begin
---------------------------Type A Instructions  -----------------------

    if (Instr_Reg(15 downto 8) = "00000000" ) then
      reg_src1_int <= Default_src1_reg;
      reg_src2_int <= Default_src2_reg;
      reg_tgt_int  <= Default_tgt_reg;
      op_int       <= CONTROL;

      case Instr_Reg(7 downto 0) is
        when "00000000" =>
          ins_int <= HLT;
```

```
    pd_int  <= '1';

  when "00000001" =>
    ins_int <= DIS;
    ie_int  <= '0';

  when "00000010" =>
    ins_int <= ENI;
    ie_int  <= '1';

  when "00000110" =>
    ins_int <= RCF;
    cd_int  <= '0';

  when "00000111" =>
    ins_int <= SCF;
    cd_int  <= '1';

  when "00001000" =>
    ins_int <= CSE;
    cd_int  <= flag_reg(6);

  when "00001001" =>
    ins_int <= COV;
    cd_int  <= flag_reg(1);

  when "00001010" =>
    ins_int <= CCY;
    cd_int  <= flag_reg(0);

  when "00001100" =>
    ins_int      <= PSF;
    op_int       <= INDXS1;
    reg_src1_int <= "110";
    reg_src2_int <= "110";
```

```
          reg_tgt_int  <= "110";

        when "00001101" =>
          ins_int      <= PPF;
          op_int       <= INDXF1;
          reg_src1_int <= "110";
          reg_src2_int <= "110";
          reg_tgt_int  <= "110";

        when "00001110" =>
          ins_int <= NOP;

        when "00001111" =>
          ins_int       <= RST;
          immediate_int <= "00000000000000000000000000000000";
          PF_int        <= '0';
          IE_int        <= '0';
          Cd_int        <= '0';
          CY_int        <= '0';
          se_int        <= '0';
          ov_int        <= '0';
          ip_int        <= '0';
          pd_int        <= '0';

        when others =>
          ins_int <= NOP;

      end case;

    end if;


    if Instr_Reg(15 downto 14) = "00" then

----------------------------Type B instructions  ----------------------
```

```
if Instr_Reg(10 downto 8) = "001" then
  reg_src1_int <= "110";
  reg_src2_int <= "110";
  reg_tgt_int  <= "110";

  case Instr_Reg(13 downto 11) is

    when "000" =>
      ins_int       <= PIN;
      op_int        <= LOAD1;
      immediate_int <= "00000000000000000000000000" &
                       instr_reg(7 downto 0);

    when "001" =>
      ins_int       <= POT;
      op_int        <= STORE1;
      immediate_int <= "00000000000000000000000000" &
                       instr_reg(7 downto 0);

    when "010" =>
      ins_int       <= SWI;
      op_int        <= INDXS2;
      immediate_int <= "00000000000000000000000000" &
                       instr_reg(7 downto 0) & '0';
      pf_int        <= '0';

    when "011" =>
      ins_int       <= LPR;
      op_int        <= CONTROL;
      pf_int        <= '1';
      immediate_int <= "0000000000000000" &instr_reg(7 downto 0)&
                       "00000000";

    when "100" =>
```

```
    ins_int         <= CLA;
    op_int          <= INDXS2;
    if flag_reg(3) = '1' then
      immediate_int <= immediate_int or
                       "00000000000000000000000"
                       & instr_reg(7 downto 0);
      pf_int        <= '0';
    else
      immediate_int <= "00000000000000000000000" &
                       instr_reg(7 downto 0) & '0';
    end if;

  when "101" =>
    ins_int         <= RET;
    op_int          <= INDXF2;
    if flag_reg(3) = '1' then
      immediate_int <= "00000000000000000000000" or
                       immediate_int & instr_reg(7 downto 0);
      pf_int        <= '0';
    else
      immediate_int <= "00000000000000000000000" &
                       instr_reg(7 downto 0);
    end if;

  when "110" =>
    ins_int       <= LPH;
    immediate_int <= instr_reg(7 downto 0) &
                     "00000000000000000000000";
    pf_int        <= '1';
    op_int        <= CONTROL;

  when "111" =>
    ins_int       <= LPL;
    op_int        <= CONTROL;
    immediate_int <= "00000000" & instr_reg(7 downto 0) &
```

```
                              "0000000000000000";
            pf_int        <= '1';

          when others =>
            ins_int <= NOP;
            op_int  <= CONTROL;

        end case;
      end if;

-------------------------Type C instructions  ------------------------

      if instr_reg(10 downto 8) = "010" then
        reg_src1_int <= instr_reg(13 downto 11);
        reg_src2_int <= instr_reg(13 downto 11);
        reg_tgt_int  <= instr_reg(13 downto 11);
        op_int       <= REGREG0;

        case instr_Reg(7 downto 0) is

          when "00000000" =>
            ins_int <= CLR;

          when "00000001" =>
            ins_int <= INV;

          when "00000010" =>
            ins_int <= NEG;

          when "00000011" =>
            ins_int <= INC;

          when "00000100" =>
            ins_int <= DEC;
```

```
when "00000101" =>
  ins_int <= SAR;

when "00000110" =>
  ins_int <= SAL;

when "00000111" =>
  ins_int <= SLR;

when "00001000" =>
  ins_int <= SL_L;

when "00001001" =>
  ins_int <= SRC;

when "00001010" =>
  ins_int <= SLC;

when "00001011" =>
  ins_int <= IVC;

when "00001100" =>
  ins_int <= SWH;

when "00001101" =>
  ins_int <= SWL;

when "00001110" =>
  ins_int <= GFL;

when "00001111" =>
  ins_int <= SFL;

when "00010000" =>
  ins_int <= CPT;
```

```
when "00010001" =>
  ins_int <= CPS;

when "00010010" =>
  ins_int <= CZE;

when "00010011" =>
  ins_int <= CBI;

when "00010100" =>
  ins_int <= CMI;

when "00010101" =>
  ins_int      <= CIL;
  op_int       <= REGREG0;
  reg_src1_int <= instr_reg(13 downto 11);
  reg_src2_int <= instr_reg(13 downto 11);
  reg_tgt_int  <= increment(instr_reg(13 downto 11));  --+'1'

when "00100011" =>
  ins_int <= JPX;
  op_int  <= LOAD2;

when "00100100" =>
  ins_int      <= CLX;
  op_int       <= INDXS2;
  reg_src2_int <= "110";
  reg_tgt_int  <= "110";

when "00100101" =>
  ins_int      <= PSH;
  op_int       <= INDXS1;
  reg_src2_int <= "110";
  reg_tgt_int  <= "110";
```

```
when "00100110" =>
  ins_int      <= POP;
  op_int       <= INDXF1;
  reg_src2_int <= "110";
  reg_tgt_int  <= "110";

when "00100111" =>
  ins_int      <= PSL;
  op_int       <= INDXS2;
  reg_src2_int <= "110";
  reg_tgt_int  <= "110";

when "00101000" =>
  ins_int      <= PPL;
  op_int       <= INDXF2;
  reg_src2_int <= "110";
  reg_tgt_int  <= "110";

when "00110000" =>
  ins_int      <= CRL;
  op_int       <= REGREG1;
  reg_src1_int <= instr_reg(13 downto 11);
  reg_src2_int <= increment(instr_reg(13 downto 11));  --+'1'
  reg_tgt_int  <= instr_reg(13 downto 11);

when "00110001" =>
  ins_int      <= IVL;
  op_int       <= REGREG1;
  reg_src1_int <= instr_reg(13 downto 11);
  reg_src2_int <= increment(instr_reg(13 downto 11));  ;
  reg_tgt_int  <= instr_reg(13 downto 11);


when "00110010" =>
```

```
  ins_int      <= NGL;
  op_int       <= REGREG1;
  reg_src1_int <= instr_reg(13 downto 11);
  reg_src2_int <= increment(instr_reg(13 downto 11));  ;
  reg_tgt_int  <= instr_reg(13 downto 11);

when "00110011" =>
  ins_int      <= ICL;
  op_int       <= REGREG1;
  reg_src1_int <= instr_reg(13 downto 11);
  reg_src2_int <= increment(instr_reg(13 downto 11));
  reg_tgt_int  <= instr_reg(13 downto 11);

when "00110100" =>
  ins_int      <= DCL;
  op_int       <= REGREG1;
  reg_src1_int <= instr_reg(13 downto 11);
  reg_src2_int <= increment(instr_reg(13 downto 11));
  reg_tgt_int  <= instr_reg(13 downto 11);

when "00110101" =>
  ins_int      <= ARL;
  op_int       <= REGREG1;
  reg_src1_int <= instr_reg(13 downto 11);
  reg_src2_int <= increment(instr_reg(13 downto 11));
  reg_tgt_int  <= instr_reg(13 downto 11);

when "00110110" =>
  ins_int      <= AL_L;
  op_int       <= REGREG1;
  reg_src1_int <= instr_reg(13 downto 11);
  reg_src2_int <= increment(instr_reg(13 downto 11));
  reg_tgt_int  <= instr_reg(13 downto 11);

when "00110111" =>
```

```
    ins_int      <= LRL;
    op_int       <= REGREG1;
    reg_src1_int <= instr_reg(13 downto 11);
    reg_src2_int <= increment(instr_reg(13 downto 11));
    reg_tgt_int  <= instr_reg(13 downto 11);

  when "00111000" =>
    ins_int      <= LLL;
    op_int       <= REGREG1;
    reg_src1_int <= instr_reg(13 downto 11);
    reg_src2_int <= increment(instr_reg(13 downto 11));
    reg_tgt_int  <= instr_reg(13 downto 11);

  when "00111001" =>
    ins_int      <= CPL;
    op_int       <= REGREG1;
    reg_src1_int <= instr_reg(13 downto 11);
    reg_src2_int <= increment(instr_reg(13 downto 11));
    reg_tgt_int  <= instr_reg(13 downto 11);

  when "00111010" =>
    ins_int      <= CZL;
    op_int       <= REGREG1;
    reg_src1_int <= instr_reg(13 downto 11);
    reg_src2_int <= increment(instr_reg(13 downto 11));
    reg_tgt_int  <= instr_reg(13 downto 11);

  when "00111011" =>
    ins_int      <= CML;
    op_int       <= REGREG1;
    reg_src1_int <= instr_reg(13 downto 11);
    reg_src2_int <= increment(instr_reg(13 downto 11));
    reg_tgt_int  <= instr_reg(13 downto 11);

  when others =>
```

```
            ins_int      <= NOP;
            op_int       <= CONTROL;
            reg_src1_int <= Default_src1_reg;
            reg_src2_int <= Default_src2_reg;
            reg_tgt_int  <= Default_tgt_reg;
        end case;


      end if;

-----------------------------Type D instructions  ----------------------

      if Instr_reg(10 downto 8) = "011" then
        op_int       <= REGREG2;
        reg_src1_int <= instr_reg(13 downto 11);
        reg_src2_int <= instr_reg(2 downto 0);
        reg_tgt_int  <= instr_reg(2 downto 0);

        case Instr_Reg(7 downto 3) is

          when "00000" =>
            ins_int <= MOV;

          when "00001" =>
            ins_int <= MVL;
            op_int  <= REGREG3;

          when "00010" =>
            ins_int <= ADD;
            op_int  <= REGREG2;

          when "00011" =>
            ins_int <= ADL;
            op_int  <= REGREG3;
```

```
when "00100" =>
  ins_int      <= LXL;
  op_int       <= LOAD2;
  reg_src1_int <= instr_reg(2 downto 0);
  reg_src2_int <= instr_reg(13 downto 11);
  reg_tgt_int  <= instr_reg(2 downto 0);

when "00101" =>
  ins_int <= ANL;
  op_int  <= REGREG3;

when "00110" =>
  ins_int <= SUB;
  op_int  <= REGREG2;

when "00111" =>
  ins_int <= SBL;
  op_int  <= REGREG3;

when "01000" =>
  ins_int      <= SXL;
  op_int       <= STORE2;
  reg_src1_int <= instr_reg(2 downto 0);
  reg_src2_int <= instr_reg(13 downto 11);
  reg_tgt_int  <= instr_reg(2 downto 0);

when "01001" =>
  ins_int <= ORL;
  op_int  <= REGREG3;

when "01010" =>
  ins_int      <= ANA;
  op_int       <= REGREG2;
  reg_src1_int <= instr_reg(2 downto 0);
  reg_src2_int <= instr_reg(13 downto 11);
```

```
    reg_tgt_int  <= instr_reg(2 downto 0);

  when "01011" =>
    ins_int <= CGL;
    op_int  <= REGREG3;

  when "01100" =>
    ins_int <= ORA;
    op_int  <= REGREG2;

  when "01101" =>
    ins_int <= CQL;
    op_int  <= REGREG3;

  when "01110" =>
    ins_int <= XRA;
    op_int  <= REGREG2;

  when "01111" =>
    ins_int <= XRL;
    op_int  <= REGREG3;

  when "10000" =>
    ins_int <= CEQ;
    op_int  <= REGREG2;

  when "10001" =>
    ins_int <= CEL;
    op_int  <= REGREG3;

  when "10010" =>
    ins_int <= CNE;
    op_int  <= REGREG2;

  when "10011" =>
```

```
    ins_int <= CNL;
    op_int  <= REGREG3;

  when "10100" =>
    ins_int <= CGT;
    op_int  <= REGREG2;

  when "10101" =>
    ins_int      <= LBX;
    op_int       <= LOAD1;
    reg_src1_int <= instr_reg(2 downto 0);
    reg_src2_int <= instr_reg(13 downto 11);
    reg_tgt_int  <= instr_reg(2 downto 0);

  when "10110" =>
    ins_int <= CGE;
    op_int  <= REGREG2;

  when "10111" =>
    ins_int      <= LDX;
    op_int       <= LOAD1;
    reg_src1_int <= instr_reg(2 downto 0);
    reg_src2_int <= instr_reg(13 downto 11);
    reg_tgt_int  <= instr_reg(2 downto 0);

  when "11000" =>
    ins_int <= MPY; op_int <= REGREG1;
  when "11001" =>
    ins_int <= MYL; op_int <= REGREG1;
  when "11010" =>
    ins_int <= MSY; op_int <= REGREG1;
  when "11011" =>
    ins_int <= MSL; op_int <= REGREG1;

  when "11101" =>
```

```
            ins_int <= SBX;
            op_int  <= STORE1;

          when "11111" =>
            ins_int <= STX;
            op_int  <= STORE1;

          when others =>
            ins_int      <= NOP;
            op_int       <= CONTROL;
            reg_src1_int <= Default_src1_reg;
            reg_src2_int <= Default_src2_reg;
            reg_tgt_int  <= Default_tgt_reg;
        end case;


      end if;  -- if type D
    end if;  -- if type "00"


---------------------------Type E1 instructions  ----------------------

    if Instr_Reg(15 downto 14) = "01" then
      op_int       <= REGREG2;
      reg_src1_int <= instr_reg(13 downto 11);
      reg_src2_int <= instr_reg(13 downto 11);
      reg_tgt_int  <= instr_reg(13 downto 11);

      case Instr_Reg(10 downto 8) is
        when "000" =>
          ins_int         <= LDI;
          if flag_reg(3) = '1' then
            immediate_int <= "0000000000000000" &
                             immediate_int(15 downto 8) &
                             instr_reg(7 downto 0);
```

```
    pf_int        <= '0';
  else
    immediate_int <= "00000000000000000000000000" &
                     instr_reg(7 downto 0);
  end if;

when "001" =>
  ins_int         <= ADI;
  if flag_reg(3) = '1' then
    immediate_int <= "0000000000000000" &
                immediate_int(15 downto 8) &
                     instr_reg(7 downto 0);
    pf_int        <= '0';
  else
    immediate_int <= "00000000000000000000000000" &
                     instr_reg(7 downto 0);
  end if;

when "010" =>
  ins_int         <= SBI;
  if flag_reg(3) = '1' then
    immediate_int <= "0000000000000000" &
                immediate_int(15 downto 8) &
                     instr_reg(7 downto 0);
    pf_int        <= '0';
  else
    immediate_int <= "00000000000000000000000000" &
                     instr_reg(7 downto 0);
  end if;

when "011" =>
  ins_int         <= ANI;
  if flag_reg(3) = '1' then
    immediate_int <= "0000000000000000" &
                immediate_int(15 downto 8) &
```

```
                    instr_reg(7 downto 0);
    pf_int        <= '0';
  else
    immediate_int <= "00000000000000000000000000" &
                    instr_reg(7 downto 0);
  end if;

when "100" =>
  ins_int         <= ORI;
  if flag_reg(3) = '1' then
    immediate_int <= "0000000000000000" &
                immediate_int(15 downto 8) &
                    instr_reg(7 downto 0);
    pf_int        <= '0';
  else
    immediate_int <= "00000000000000000000000000" &
                    instr_reg(7 downto 0);
  end if;

when "101" =>
  ins_int         <= SBS;
  op_int          <= STORE1;
  if flag_reg(3) = '1' then
    immediate_int <= "00000000000000000000000000" or
                    immediate_int & instr_reg(7 downto 0);
    pf_int        <= '0';
  else
    immediate_int <= "00000000000000000000000000" &
                    instr_reg(7 downto 0);
  end if;

when "110" =>
  ins_int         <= LDS;
  op_int          <= LOAD1;
  if flag_reg(3) = '1' then
```

```
            immediate_int <= "00000000000000000000000" or
                             immediate_int & instr_reg(7 downto 0);
            pf_int        <= '0';
          else
            immediate_int <= "00000000000000000000000" &
                             instr_reg(7 downto 0);
          end if;

        when "111" =>
          ins_int         <= STS;
          op_int          <= STORE1;
          if flag_reg(3) = '1' then
            immediate_int <= "00000000000000000000000" or
                             immediate_int & instr_reg(7 downto 0);
            pf_int        <= '0';
          else
            immediate_int <= "00000000000000000000000" &
                             instr_reg(7 downto 0);
          end if;

        when others =>
          ins_int      <= NOP;
          op_int       <= CONTROL;
          reg_src1_int <= Default_src1_reg;
          reg_src2_int <= Default_src2_reg;
          reg_tgt_int  <= Default_tgt_reg;

      end case;
    end if;

--------------------------------Type E2 instructions  -----------------

    if Instr_Reg(15 downto 14) = "10" then
      reg_src1_int <= instr_reg(13 downto 11);
      reg_src2_int <= instr_reg(13 downto 11);
```

```
reg_tgt_int  <= instr_reg(13 downto 11);

case Instr_Reg(10 downto 8) is

  when "000" =>
    ins_int         <= LBA;
    op_int          <= LOAD0;
    if flag_reg(3) = '1' then
      immediate_int <= "00000000000000000000000" or
                       immediate_int & instr_reg(7 downto 0);
      pf_int        <= '0';
    else
      immediate_int <= "00000000000000000000000" &
                       instr_reg(7 downto 0);
    end if;

  when "001" =>
    ins_int         <= LDA;
    op_int          <= LOAD0;
    if flag_reg(3) = '1' then
      immediate_int <= immediate_int or
                       "00000000000000000000000" &
                       instr_reg(7 downto 0);
      pf_int        <= '0';
    else
      immediate_int <= "00000000000000000000000" &
                       instr_reg(7 downto 0) & '0';
    end if;

  when "010" =>
    ins_int         <= STA;
    op_int          <= STORE0;
    if flag_reg(3) = '1' then
      immediate_int <= immediate_int or
                       "00000000000000000000000" &
```

```
                    instr_reg(7 downto 0);
    pf_int        <= '0';
  else
    immediate_int <= "00000000000000000000000" &
                    instr_reg(7 downto 0) & '0';
  end if;

when "011" =>
  ins_int         <= SBA;
  op_int          <= STORE0;
  if flag_reg(3) = '1' then
    immediate_int <= "00000000000000000000000" or
                    immediate_int & instr_reg(7 downto 0);
    pf_int        <= '0';
  else
    immediate_int <= "00000000000000000000000" &
                    instr_reg(7 downto 0);
  end if;

when "100" =>
  ins_int         <= LAL;
  op_int          <= LOAD0;
  if flag_reg(3) = '1' then
    immediate_int <= immediate_int or
                    "00000000000000000000000" &
                    instr_reg(7 downto 0);
    pf_int        <= '0';
  else
    immediate_int <= "00000000000000000000000" &
                    instr_reg(7 downto 0) & '0';
  end if;

when "101" =>
  ins_int         <= STL;
  op_int          <= STORE0;
```

```
    if flag_reg(3) = '1' then
      immediate_int <= immediate_int or
                       "00000000000000000000000" &
                       instr_reg(7 downto 0);
      pf_int        <= '0';
    else
      immediate_int <= "00000000000000000000000" &
                       instr_reg(7 downto 0) & '0';
    end if;

  when "110" =>
    ins_int         <= LSL;
    op_int          <= LOAD2;
    if flag_reg(3) = '1' then
      immediate_int <= "0000000000000000" &
                       immediate_int(15 downto 8) &
                       instr_reg(7 downto 0);
      pf_int        <= '0';
    else
      immediate_int <= "00000000000000000000000" &
                       instr_reg(7 downto 0);
    end if;

  when "111" =>
    ins_int         <= SSL;
    op_int          <= STORE2;
    if flag_reg(3) = '1' then
      immediate_int <= "0000000000000000" &
                       immediate_int(15 downto 8) &
                       instr_reg(7 downto 0);
      pf_int        <= '0';
    else
      immediate_int <= "00000000000000000000000" &
                       instr_reg(7 downto 0);
    end if;
```

```
        when others =>
          ins_int      <= NOP;
          op_int       <= CONTROL;
          reg_src1_int <= Default_src1_reg;
          reg_src2_int <= Default_src2_reg;
          reg_tgt_int  <= Default_tgt_reg;
      end case;

    end if;

------------------------------Type F instructions  --------------------

    if instr_Reg(15 downto 14) = "11" then
      reg_src1_int <= Default_src1_reg;
      reg_src2_int <= Default_src2_reg;
      reg_tgt_int  <= Default_tgt_reg;

      case instr_Reg(13 downto 12) is

        when "00" =>
          ins_int         <= LEA;
          op_int          <= REGREG3;
          reg_src1_int    <= "111";
          reg_src2_int    <= "111";
          reg_tgt_int     <= "111";
          if Flag_Reg(3) = '1' then
            immediate_int <= immediate_int(31 downto 8) &
                             instr_reg(7 downto 0);
            pf_int        <= '0';
          else
            immediate_int <= "00000000000000000000" &
                             instr_reg(11 downto 0) & '0';
          end if;
```

```
when "01" =>
  ins_int           <= JMP;
  op_int            <= JUMP;
  if Flag_Reg(2) = '1' then
    if Flag_Reg(3) = '1' then
      immediate_int <= immediate_int(31 downto 8) &
                       instr_reg(7 downto 0);
      pf_int        <= '0';
    else
      immediate_int <= "00000000000000000000" &
                       instr_reg(11 downto 0) & '0';
    end if;
  end if;

when "10" =>
  ins_int           <= JPR;
  op_int            <= JUMP;
  if Flag_Reg(2) = '1' then
    if Flag_Reg(3) = '1' then
      immediate_int <= immediate_int(31 downto 8) &
                       instr_reg(7 downto 0);
      pf_int        <= '0';
    else
      immediate_int <= "00000000000000000000" &
                       instr_reg(11 downto 0) & '0';
    end if;
  end if;

when "11" =>
  ins_int         <= CAL;
  op_int          <= INDXS2;
  reg_src1_int    <= "110";
  reg_src2_int    <= "110";
  reg_tgt_int     <= "110";
  if Flag_Reg(3) = '1' then
```

```
            immediate_int <= immediate_int(31 downto 8) &
                             instr_reg(7 downto 0);
            pf_int        <= '0';
          else
            immediate_int <= "00000000000000000000" &
                             instr_reg(11 downto 0) & '0';
          end if;

        when others =>
          ins_int <= NOP;
          op_int  <= CONTROL;
      end case;

    end if;
  end process instr_dec_process;

-------------------------- REGISTER DEFINITIONS  ----------------------
  instr_fetch : process(clock, en_inf)
  begin
    if rising_edge(clock) and en_inf = '1' then
      for i in instr_reg'range loop
        instr_reg(i) <= data_bus(i);
      end loop;
    end if;
  end process instr_fetch;

  exc_register : process(clock, en_exc)
  begin
    if rising_edge(clock) and en_exc = '1' then
      for i in immediate_int'range loop
        immediate_exc(i) <= immediate_int(i);
      end loop;
      instr_exc          <= ins_int;
      operat_exc         <= op_int;
      cycle_exc          <= cyc_exc_int;
```

```
    reg_tgt            <= reg_tgt_int;
    reg_src1           <= reg_src1_int;
    reg_src2           <= reg_src2_int;
  end if;
end process exc_register;

adr_register : process(clock, en_exc)
begin
  if rising_edge(clock) and en_exc = '1' then
    instr_adr          <= ins_int;
    operat_adr         <= op_int;
    cycle_adr          <= cyc_exc_int;
    reg_tgt_adr        <= reg_tgt_int;
    reg_src1_adr       <= reg_src1_int;
    reg_src2_adr       <= reg_src2_int;
    for i in immediate_int'range loop
      immediate_adr(i) <= immediate_int(i);
    end loop;
  end if;
end process adr_register;

read  <= read_int;
write <= write_int;

flag_register : process(clock)
begin
  if rising_edge(clock) then
    flag_reg(0) <= cy_int;
    flag_reg(1) <= ov_int;
    flag_reg(2) <= cd_int;
    flag_reg(3) <= pf_int;
    flag_reg(4) <= ie_int;
    flag_reg(5) <= ip_int;
    flag_reg(6) <= se_int;
    flag_reg(7) <= pd_int;
```

```
  end if;
end process flag_register;

cyc_exc_out <= cycle_exc;
ins_exc_out <= instr_exc;
ins_adr_out <= instr_adr;
op_exc_out  <= operat_exc;
op_adr_out  <= operat_adr;
cyc_adr_out <= cycle_adr;
CY_OUT      <= flag_reg(0);
cd_out      <= flag_reg(2);
PF_OUT      <= flag_reg(3);

mux_reg : process(reg_tgt, reg_src1, reg_src2)
begin
  reg_tgt_out  <= reg_tgt;
  reg_src1_out <= reg_src1;
  reg_src2_out <= reg_src2;

  if (cycle_exc = IND) then
    if ((instr_exc = PSH)or(instr_exc = POP)or(instr_exc = CLX)or
        (instr_exc = PSL)or(instr_exc = PPL)) then
      reg_src1_out <= reg_src2;
    end if;

  elsif (cycle_exc = EXC1) then
    if (operat_exc = REGREG1) then
      reg_src2_out <= reg_src1;
    elsif (operat_exc = REGREG3) then
      reg_tgt_out  <= increment(reg_tgt);
      reg_src1_out <= increment(reg_src1);
    elsif ((instr_exc = LBX)or(instr_exc = LDX)or(instr_exc = LXL))
    then
      reg_src1_out <= reg_src2;
      reg_src2_out <= increment(reg_src2);
```

```
    elsif ((instr_exc = SBX)or(instr_exc = STX)) then
      reg_src2_out <= increment(reg_src2);
    elsif (instr_exc = SXL) then
      reg_src1_out <= reg_src2;
      reg_src2_out <= increment(reg_src1);
    elsif ((instr_exc = SBS)or(instr_exc = STS)or(instr_exc = SSL))
    then
      reg_src2_out <= "110";
    elsif ((instr_exc = PSH)or(instr_exc = POP)or(instr_exc = CLX)or
           (instr_exc = PSL)or(instr_exc = PPL)) then
      reg_src1_out <= reg_src2;
    end if;

  elsif (cycle_exc = EXC2) then
    if ((instr_exc = LBX)or(instr_exc = LDX)or(instr_exc = LXL))then
      reg_src1_out <= reg_src2;
      reg_src2_out <= increment(reg_src2);
    elsif ((instr_exc = SBX)or(instr_exc = STX)) then
      reg_src2_out <= increment(reg_src2);
    elsif (instr_exc = SXL) then
      reg_src1_out <= reg_src2;
      reg_src2_out <= increment(reg_src1);
    elsif ((instr_exc = SBS)or(instr_exc = STS)or(instr_exc = SSL))
    then
      reg_src2_out <= "110";
    elsif ((instr_exc = PSH)or(instr_exc = CLX)or(instr_exc = PSL))
    then
      reg_src2_out <= reg_src1;
      reg_tgt_out  <= reg_src1;
    elsif ((instr_exc = POP)or(instr_exc = PPL)) then
      reg_src1_out <= reg_src2;
    end if;

  elsif (cycle_exc = DST1) then
    if (instr_exc = CLX) then
```

```
      reg_src2_out <= reg_src1_adr;
      reg_tgt_out  <= reg_src1_adr;
    elsif (instr_exc = PSL) then
      reg_src1_out <= increment(reg_src1_adr);
      reg_src2_out <= reg_src1_adr;
      reg_tgt_out  <= reg_src1_adr;
    elsif ((instr_exc = STL)or(instr_exc = SXL)or(instr_exc = SSL))
    then
      reg_src1_out <= increment(reg_src2_adr);
      reg_tgt_out  <= reg_src2_adr;
    end if;

  elsif (cycle_adr = DFT1) then
    if ((instr_exc = LAL)or(instr_exc = LSL)or(instr_exc = PSL)) then
      reg_src2_out <= reg_src1_adr;
      reg_tgt_out  <= increment(reg_src1_adr);
    elsif (instr_exc = LXL) then
      reg_src1_out <= reg_src2_adr;
      reg_tgt_out  <= increment(reg_tgt_adr);
    elsif (instr_adr = SWI)or(instr_adr = CLX)or(instr_adr = RET)
    then
      PC_MODE      <= DATA_LOW;
    end if;

  elsif (cycle_adr = DFT2) then
    if (instr_adr = SWI)or(instr_adr = CLX)or(instr_adr = RET) then
      PC_MODE <= DATA_HIGH;
    end if;

  elsif (cycle_exc = EXC3) then
    if ((instr_exc = POP)or(instr_exc = PPL)) then
      reg_src2_out <= reg_src1;
      reg_tgt_out  <= reg_src1;
    elsif ((instr_exc = PSH)or(instr_exc = CLX)or(instr_exc = PSL))
    then
```

```
        reg_src1_out <= reg_src2;
      end if;
    elsif (cycle_exc = EXC0) then
      if (instr_exc = RST) then
        PC_MODE      <= RST;
      end if;
    end if;
  end process mux_reg;
end control_unit_behave;
```

Appendix D

Stimuli Files

D.1 Finite State Machine

In order to check the FSM itself, we just disable all but one FSM in the arbiter. When only one FSM is operating, no hazard will occur. We can examine it seperately.

For the testing of the FSM there are two stimuli-files. The first just displays the basic signals, it will ge more easy to understand. The second displays more signals. It gives more detailes information but is more complex.

D.1.1 Basic Signals

```
#############################################################
# do -file for testing antares FSM with basic signals    #
#############################################################
restart -force -NOwave -NOlist -NOLOg -NOBreakpoint

# input signals

# output signals

add wave control_fsm1/enable
```

```
add wave control_fsm1/ready
add wave control_fsm1/current_st
add wave control_fsm1/next_st
add wave control_fsm1/op
add wave control_fsm1/instr_dec
add wave control_fsm1/alu
add wave control_fsm1/mem
#internal signals


view wave

force clock 1 0,0 20 -r 40
force reset 1 0,0 25, 1 60

force ins SCF 0
force op CONTROL 0
force ins CLR 70
force op REGREG0 70
force ins LBA 110
force op LOAD0 110
force ins MVL 150
force op REGREG3 150
force ins GFL 190
force op REGREG2 190
force ins JMP 230
force op JUMP 230
force ins CSE 270
force op CONTROL 270
force ins INV 310
force op REGREG0 310
force ins NEG 350
force op REGREG0 350
force ins LPH 390
force op CONTROL 390
```

```
force ins STL 430
force op STORE0 430


run 800
```

D.1.2 All Signals

```
#############################################################
# do -file for testing antares FSM with all signals      #
#############################################################
restart -force -NOwave -NOlist -NOLOg -NOBreakpoint

# input signals
add wave clock
add wave reset

# output signals
add wave en_exc
add wave en_adr
add wave en_inf
add wave read
add wave write

#internal signals
add wave ins1
add wave op1
add wave ins2
add wave op2
add wave ins3
add wave op3
add wave next_cycle1
add wave cycle1
add wave next_cycle2
add wave cycle2
```

```
add wave next_cycle3
add wave cycle3
add wave mem_use
add wave alu_use

view wave

force clock 1 0,0 20 -r 40
force reset 1 0,0 25, 1 60

force ins LBA 110
force op LOAD0 110
force ins MVL 180
force op REGREG3 180
force ins JMP 350
force op JUMP 350
force ins NEG 420
force op REGREG0 420
force ins LPH 600
force op CONTROL 600
force ins STL 780
force op STORE0 780
force ins SCF 900
force op CONTROL 900
force ins CLR 1100
force op REGREG0 1100
force ins GFL 1200
force op REGREG2 1200
force ins CSE 1300
force op CONTROL 1300
force ins INV 1400
force op REGREG0 1400
force ins SCF 1540
force op CONTROL 1540
```

```
run 1700
```

D.2 Arbiter

```
#############################################################
# do -file for testing antares arbiter                      #
#############################################################
restart -force -NOwave -NOlist -NOLOg -NOBreakpoint

# input signals
add wave clock
add wave reset

# output signals
add wave en_exc
add wave en_adr
add wave en_inf
add wave read
add wave write

#internal signals
add wave ins1
add wave op1
add wave ins2
add wave op2
add wave ins3
add wave op3
add wave next_cycle1
add wave cycle1
add wave next_cycle2
add wave cycle2
add wave next_cycle3
add wave cycle3
add wave mem_use
```

```
add wave alu_use

view wave

force clock 1 0,0 20 -r 40
force reset 1 0,0 25, 1 60

force ins SCF 460
force op CONTROL 460
force ins CLR 520
force op REGREG0 520
force ins LBA 110
force op LOAD0 110
force ins MVL 150
force op REGREG3 150
force ins GFL 590
force op REGREG2 590
force ins JMP 230
force op JUMP 230
force ins CSE 660
force op CONTROL 660
force ins INV 700
force op REGREG0 700
force ins NEG 350
force op REGREG0 350
force ins LPH 390
force op CONTROL 390
force ins STL 430
force op STORE0 430
force ins SCF 740
force op CONTROL 740

run 800
```

Bibliography

[Bre97] Barry B Brey. *The Intel Microprocessors 8086 ... Pentium Pro Processor*. Prentice Hall, New Jersey, 5th edition, 1997.

[GAS90] Randall L. Geiger, Phillip E. Allen, and Noel R. Strader. *VLSI Design Techniques for Analog and Digital Circuits*. McGraw-HILL, Singapore, 1990.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann, San Mateo, California, 1st edition, 1990.

[HP94] John L. Hennessy and David A. Patterson. *Rechnerarchitektur*. Vieweg&Sohn, Braunschweig/Wiesbaden, 1994.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann, San Mateo, California, 2nd edition, 1996.

[HP02] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann, San Francisco, California, 3rd edition, May 2002.

[IEE88] IEEE Std 1076-1987. *IEEE Standard VHDL Language Reference Manual*. Institute of Electrical and Electronical Engineers, New York, 1988.

[Jan00] Dirk Jansen. Architecture and Compiler for an ANSI C-targeting Reduced Instruction Set Core for Embedded Systems (ANTARES). Technical Report ICS-00-26, University of California, Irvine, July 2000.

[Mär01] Christian Märtin. *Rechner Architekturen*. Fachbuchverlag Leipzig, Munich, Vienna, 2001.

[Mes96] Frank Messicci. Erweiterung der DXLS um eine 5-stufige Pipeline. Technical report, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, Stuttgart, Germany, 1996.

[PB01] Sujan Pandey and David Berner. Development of a Microprocessor Core ANTARES. Technical report, Fachhochschule Offenburg, November 2001.

[Ten95] Klaus TenHagen. *Abstrakte Modellierung Digitaler Schaltungen*. Springer, Berlin, 1995.

[Web97] Lukas Weberruß. Erweiterung des DXLS-Prozessor-modells um eine 5-stufige Pipeline. Technical Report 1625, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, Stuttgart, Germany, July 1997.

I grant the University of Applied Sciences, Fachhochschule Offenburg the non-exclusive right to use this work for the University's own purposes and to make single copies of the work available to the public on a not-for-profit basis if copies are not otherwise available.

David Berner