Concert'02
Architecture Specification and Implementation

Johan Eriksson Thelin
Anders Lindström
Michael Nordseth

Chalmers University of Technology
2002

Abstract

This document describes a RISC CPU architecture, Concert'02. The Concert'02 architecture is based on an older specification, but has been updated for the intended host system.

We continue by implementing the architecture in VHDL. The implementation is called the JAM CPU core. It is a five-stage pipe-lined CPU core, with multi-cycle operations, forwarding and hazard checking.

The CPU has been tested in an actual FPGA, and proved to work properly. We have analysed the critical path, the synthesis timing and area report and current performance.

1.0 Table of Contents

2.0 The Concert'02 Architecture Specification

This chapter covers the Concert'02 computer architecture. Concert'02 is an adaptation of the Concert architecture [1] for an FPGA system. It describes a 32 bits RISC architecture with precise interrupts.

The changes that have been made can be divided into the following categories:

- Instruction set modifications
- Memory layout modifications
- System environment modifications

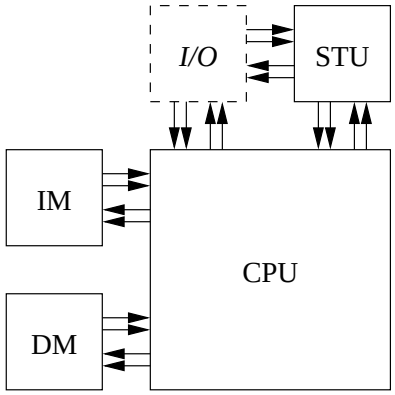
These changes are all discussed in detail in chapter 2.4.

2.1 System Overview

The Concert'02 system architecture can be seen in figure 1. It features four units outside the CPU core, the Timer and Syncronisation Unit, STU, the I/O unit and the two memories, IM and DM. The I/O unit has been left out as our lab system (see chapter 4.1) does not support parallell I/O to the outside world.

Communication between the CPU core and the units STU and I/O is handled through the PUT signal (described in chapter 2.1.2), the SYNCTRAP signal (described in chapter 2.1.1) and the processor status word, PSW, which is described in detail in chapter 2.2.2.

FIGURE 1. System Overview



2.1.1 The Timer and Syncronisation Unit

The STU manages the SYNCTRAP signal and the interrupt timer. The SYNCTRAP signal is used to indicate that an external interrupt has occured. This is indicated by setting the signal high during one clock cycle.

The interrupt timer is a 26 bits counter. The counter decreases its value by one each clock cycle. When it reaches zero it checks the value of the bit T in PSW (see table 1 for a description of the bits of PSW). If T is zero an interrupt is indicated through SYNCTRAP, and T is set to one in PSW. If T allready is one, no interrupt occurs.

When it reaches zero the counter is reloaded with the value TP from PSW in the most significant end, and ten zeroes in the least significant bits.

2.1.2 The I/O Unit

The I/O unit is supposed to handle asyncronous parallell communications with the outer world. As the lab system lacks the hardware needed for such a function, it has been left out but a possible interface is described below.

The communications with the outer world is managed through two 32 bits wide registers: IN and OUT. All communications are controlled by a handshaking protocol.

IN stores data gathered from four external busses, INA-D (INA=$IN_{0..7}$, INB=$IN_{8..15}$, etc.) For each group a status bit, IB_x, is held in PSW that indicates that new data is available. These bits are set as soon as a byte has been recieved. As long as the bit is set, no new data is accepted from the outside. The external handshaking is managed through the 8 bits IN_xREQ and IN_xACK.

OUT stores the data to be exported the the external busses OUTA-D (mapped as INA-B to the OUT register). Each byte has a status bit, OB_x, in PSW that indicates that new data to send is available. When each byte has been transmitted, the corresponding bit is set to zero. The external handshaking is managed though the 8 bits OUT_xREQ and OUT_xACK.

The CPU signals that new data for the OUT register is available through the PUT signal, the OUT register is thus updated independently of the PSW. This is to make sure that the OUT register is properly loaded one clock cycle before the PSW indicates that data is available.

2.1.3 Memory Layout

The Concert'02 is a Harward architecture and has two separate memories, one instruction memory and one data memory.

The memory on the lab system consists of two SRAM arrays with 8 KM68100C-15 ICs, each 512k x 8 bit large. The arrays are connected to form two 512 x 64 bit SRAM units, IM and DM. They are access by the CPU through an address buss 19 bits wide and a 64 bits data buss.

Every single SRAM IC on the system can be selected or deselected with a Chip Select, CS, signal and the two hole arrays with Output Enable, OE, and Write Enable, WE, signals. These signal are together refered as the control buss.

The CS signals makes it possible to write only half of the 64 bits, this is usefull since the CPU uses 32 bit words (discussed later).

2.2 Programming Model

2.2.1 Datatypes

The basic word used in Concert'02 is 32 bits wide. The bits are numbered 0 to 31 from the least significant bit to the most significant bit.

2.2.2 Visible State Variables

The instruction memory, IM, is a memory with the address space $0..2^{32}-1$, thus one address can be held in one word. All addresses in IM refer to a byte, but IM can only read aligned words (i.e. the two LSB is assumed to be zero). This memory is used to hold instructions.

The data memory, DM, is a memory with the address space $0..2^{32}-1$, thus one address can be held in one word. All addresses in DM refer to a byte, but DM can only read aligned words (i.e. the two LSB is assumed to be zero). This memory is used to hold data.

The register file, R, consists of 32 registers. Register 0 always hold the value zero. Registers 1 to 31 are general purpose and can be used to hold any value. Register 31 is used to hold the return address from traps and interrupts, but can be used to hold any value if this is taken under consideration.

The program counter, PC, always point to the instruction to be fetched and executed. It is used to address IM, and is thus assumed to be word aligned. If the instruction at address n is being executed, PC normaly points at the address n+4. However, when jumps, traps and branches are executed the PC is altered.

The input port, IN, is a 32 bits wide register holding data from Concert'02's external port.

The output port, OUT, is a 32 bits wide register holding data to be sent to Concert'02's external port.

The processor status word, PSW, is a 32 bits wide register holding status information concerning external communications, interrupts and the period time for the peridodic interrupt generator. The contents of the bits are described in table 1.

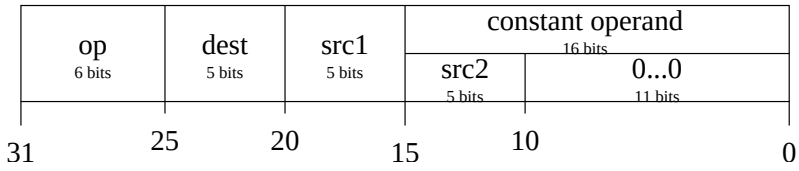
TABLE 1. PSW Contents

Bit(s)	Name	Meaning
0..3	IB[0..3]	Input port byte available flags
4..7	OB[0..3]	Output port byte available flags.
8	V	Arithmetic overflow flag (internal)
9	I	Illegal instruction flag (internal)
10	reserved	Reserved for future extensions (internal)
11..14	X[0..3]	External interrupt flags/user defined trap flags
15	T	Timer interrupt flags
16..31	TP	Timer period (divided by 1024 CLK cycles)

2.3 Instruction Set and Encoding

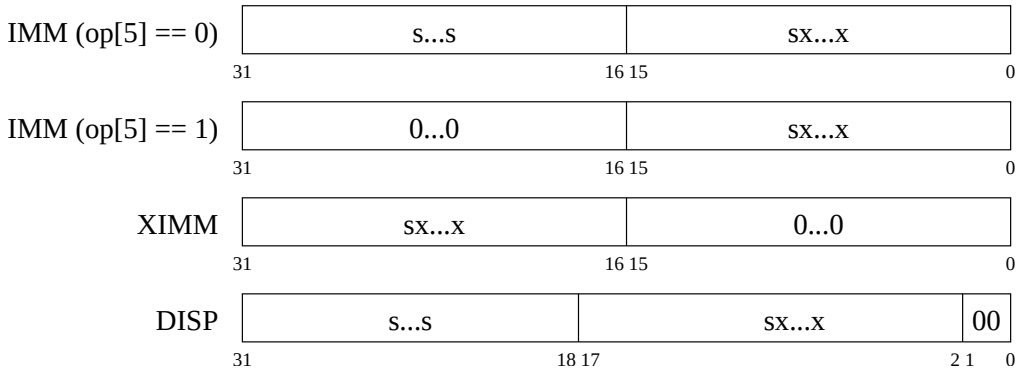
A Concert’02 instruction is always one word. The different parts of the word are shown below in figure 2. There are three five bits register references: src1, src2 and dest. The op field holds the six bits op-code of the instruction.

FIGURE 2. Instruction Word Layout



Concert’02 supports 22 different instructions. Each of these can be combined with one or more of the four different instuction modes: REGister that uses the src2 operand, IMMediate, eXtended IMMediate and DISPlaced that uses the constant operand. Figure 3 shows how the constant operand is extended from 16 to 32 bits in each mode and table 2 describes which instruction and instruction mode combinations that are available.

FIGURE 3. Immediate Format Interpretation



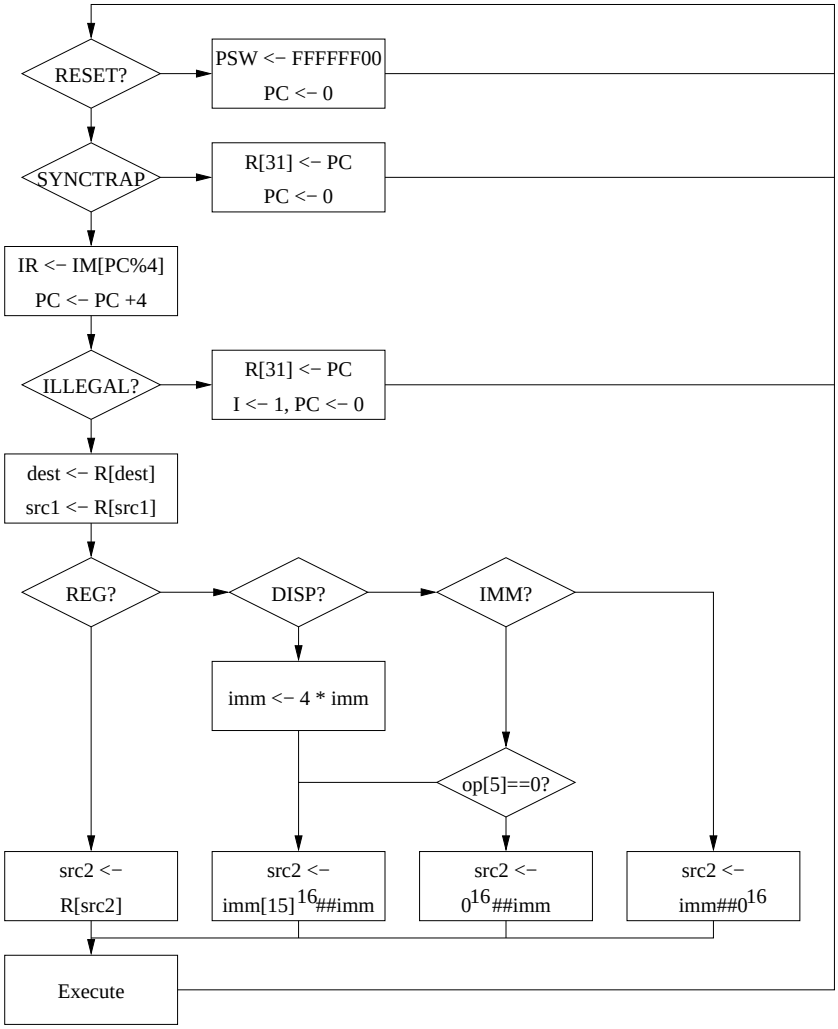
The instruction code encoding is shown in table 2. Please note that the different columns corresponds to the different instruction modes available.

TABLE 2. Concert'02 Instruction Set

op[1..0]	**00**	**01**	**10**	**11**
op[5..2]	**(REG)**	**(IMM)**	**(XIMM)**	**(DISP)**
0000	ADD	ADDI	ADDX	ADDD
0001	ADDV	ADDVI	ADDVX	ADDVD
0010	MUL	MULI		
0011	MULH	MULHI		
0100	SHZ	SHZI		BEQ
0101	SHS	SHSI		BNE
0110	CMP	CMPI	CMPX	CMPD
0111	JUMP		JUMPX	JUMPD
1000	SUB	GET		LW
1001	SUBV	PUT		SW
1010		TRAP		
1011	AND	ANDI	ANDX	
1100	OR	ORI	ORX	
1101	XOR	XORI	XORX	
1110	SET	SETI	SETX	
1111	RESET	RESETI	RESETX	

The basic interpretation algorithm can be seen in figure 4. In table 3 the outcome of each instruction is specified in detail.

FIGURE 4. Interpretation Algorithm



The table below shows the exact meaning of each instruction.

TABLE 3. Instruction Interpretation Algorithm

Mnemonic	Meaning	Function
ADD[I|X|D]	Add	dest <- src1 + src2
ADDV[I|X|D]	Add and trap on overflow	if(overflow(src1 + src2) && V==0) { R[31] <- PC, PC <- 0, V <- 1 } else dest <- src1 + src2
SUB	Subtract	dest <- src1 - src2
SUBV	Subtract and trap on overflow	if(overflow(src1 - src2) && V==0) { R[31] <- PC, PC <- 0, V <- 1 } else dest <- src1 - src2
CMP[I|X|D]	Compare (less than)	dest <- (src1 < src2) ? 1 : 0
MUL[I]	Multiply and return LSW	dest <- (src1 * src2)[0..31]
MULH[I]	Multiply and return MSW	dest <- (src1 * src2)[32..63]
AND[I|X]	Bitwise AND	dest <- src1 & src2
OR[I|X]	Bitwise OR	dest <- src1 | src2
XOR[I|X]	Bitwise XOR	dest <- scr1 ^ src2
RESET[I|X]	Read and reset status bits	dest <- PSW, PSW <- PSW & !src2
SET[I|X]	Read and set status bit	dest <- PSW, PSW <- PSW | src2
SHZ[I]	Shift in zero from left or right	dest <- (src2<0)? (0...0##src1>>-src2) : (src1##0...0<<src2)
SHS[I]	Shift in sign from left or right	dest <- (src2<0)? (src1[31]...src1[31]##src1>>-src2) : (src1##src1[31]...src1[31]<<src2
LW	Load word form data memory	dest <- DM((src1 + src2) % 4)
SW	Store word to data memory	DM((src1 + src2) % 4) <- dest
GET	Get data from IN port	dest <- IN, PSW <- PSW & !src2
PUT	Put data on OUT port	OUT <- src1, PSW <- PSW | src2
BEQ	Branch if equal	PC <- (dest == scr1) ? (PC+src2) : PC
BNE	Branch if not equal	PC <- (dest != src1) ? (PC+src2) : PC
JUMP[X|D]	Jump	dest <- PC, PC <- src1 + src2
TRAP	Jump to trap handler	R[31] <- PC, PC <- 0, PSW <- PSW | src2

2.3.1 Traps, Interrupts and Initialisation

After having recieved a RESET signal the processor starts in a predefined state. The PC is set to zero, and PSW to $FFFFFF00_{16}$ when the fetching and execution of instructions is started.

When the SYNCTRAP signal is raised or an overflow or illegal instruction is detected, a hardware interrupt occurs. When a hardware interrupt, or a software trap occurs the PSW is setup to indicate the source of the trap/interrupt and the execution is resumed from PC=0. The address of the interrupted instruction is stored in R31.

2.4 Changes from the Original Concert Specification

This section covers the changes between the original Concert specification and Concert'02.

2.4.1 Instruction set modifications

- The interpretation of the TRAP instruction has been slightly altered from the original Concert specification. The original definition was "dest <- PC, PC <- src1, PSW <- PSW | src2". The new definition removes the ability to store PC anywhere and to set PC to other values than zero.
- The PUT instruction uses src1 instead of dest to load the output register.
- Branch instructions (BNE and BEQ) now uses PC as reference point instead of PC+4.

2.4.2 Memory layout modifications

- The memory layout now features two entirely separate external memories. One for instructions and one for data, instead of a partially shared region as described in the original specification.

2.4.3 System environment modifications

- The I/O unit has been left out since the lab system does not have the required hardware to implement the data communication needed.

3.0 Project Goals

This project aims at creating a pipe-lined implementation of the Concert'02 architecture. The implementation must be synthesisable for the Xilinx Virtex XCV300 devices. The goal is to achieve the best possible cost/performance ratio for a given set of benchmarking programmes.

The purpose of the project is to gain experience from:

1. The construction of a pipe-lined microprocessor.
2. Computer based design tools.
3. Implementation of a complex digial system.

4.0 Overview of the Concert'02 Implementation

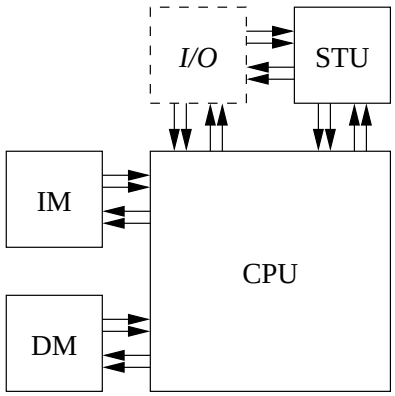
As the specification for Concert'02 is intended for the lab system for this project we have divided the design into two parts: the CPU core and the CPU system environment.

This chapter first discusses the CPU system environment and is followed by a description of the CPU core implementation. The final section of this chapter deals with the issues that we faced during the implementation process. It covers the testing, design choices and the final performance.

4.1 The CPU System Environment

The CPU system environment defines the surroundings to the CPU core. As seen in figure 5 the environment consists of the units I/O, STU, DM and IM. The memories, DM and IM, are supplied by the lab system and are not defined in the environment. As the lab system lacks the hardware to build the I/O unit, this unit is left out as well. The STU, however, is defined in the environment together with the logic required to generate a RESET signal from the lab system keyboard.

FIGURE 5. System Overview



4.1.1 The Timer

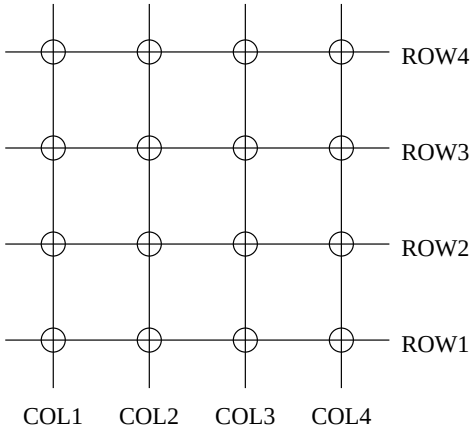
The timer is implemented as specified in chapter 2.1.1. When the timer recieves a RESET signal it sets the counter to 1 (instead of to zero). This allows the CPU core to initialize and setup a valid PSW before the counter reaches zero and retrieves new settings from the PSW.

4.1.2 The RESET Generation Logic

The keyboard is implemented as a 4 by 4 matrix of switches as shown in figure 6 (the switches are shown as circles). These switches are connected between a column line and a row line. When the key is pressed the switch is closed and the column and row lines are connected.

To generate a RESET signal we must hold a column high and the rest low. We listen to one of the row lines that we connect to the matrix in a pull-down configuration. When the appropriate switch is closed the row line that we listen to will be forced high. This signal is used as a RESET signal to the CPU core and the STU.

FIGURE 6. The Keyboard

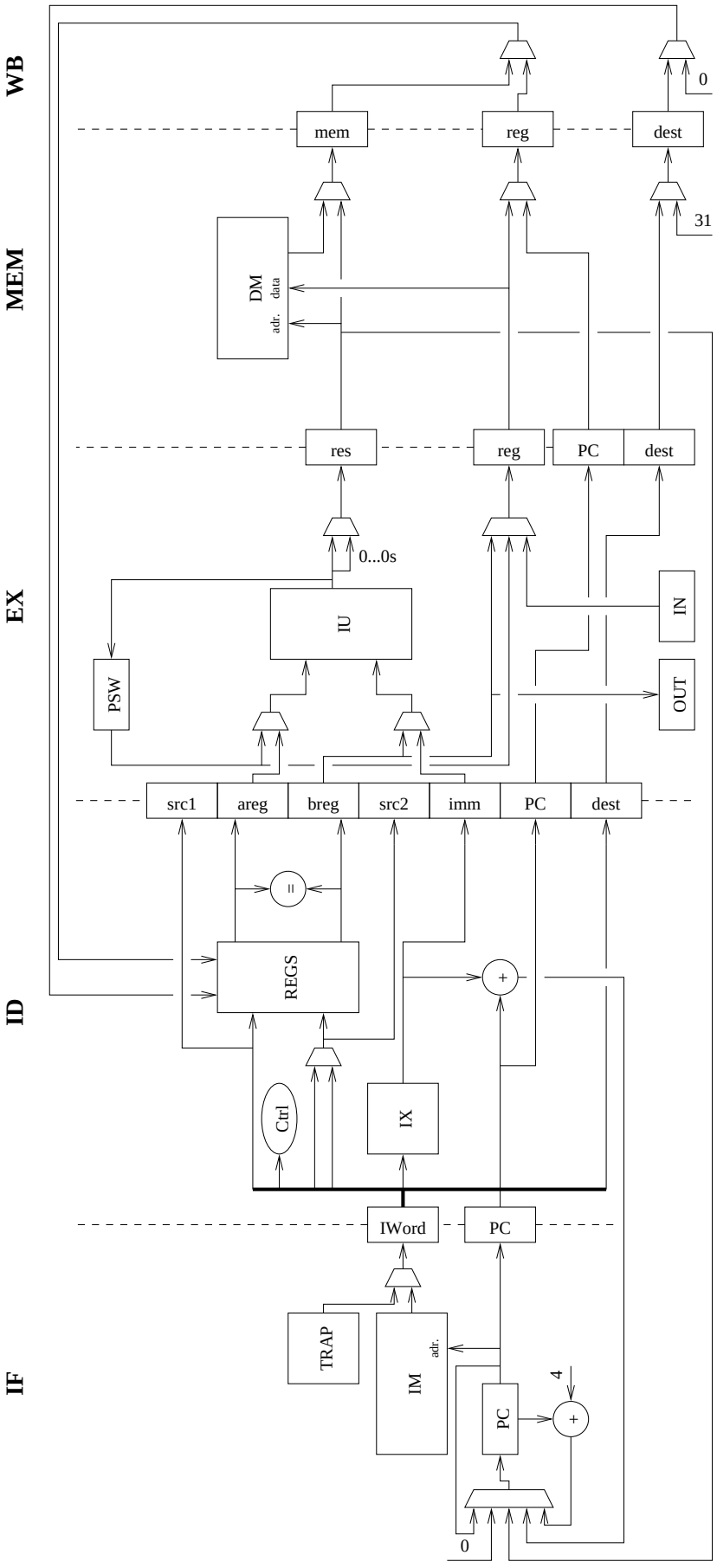


4.2 Overview of the CPU Core

We have choosen to implement the Concert'02 architecture as a five pipe-line stages RISC processor. The processor core is refered to as JAM CPU and has been designed using VHDL.

This chapter begins with an introduction to the basic design of the pipe-line and the design choices that we have made followed by a more in-depth discussion of the pipe-line. The pipe-line stages are: Instruction Fetch, Instruction Decode, EXecute, MEMory and Write Back as shown in figure 7. Each pipe-line stage will be covered here, followed by a discussion of the forwarding and hazard checking logic. The last section is devoted to traps and interrupts.

FIGURE 7. Overview of the Pipe-line Datapath



4.2.1 The CPU Core Pipe-line

Instead of basing our design around the interpretation algorithm and basic operations methodology as suggested in [1], we have choosen to derive our design from the DLX implementation described in [3]. The implications from this design choice is that we have a five stage pipe-line with one branch delay-slot. The data-path of the pipe-line can be seen in figure 7.

The datapath consists of a set of logic units connected with the pipe-line registers through a set of multiplexers. These multiplexers are controlled from the control signals generated by the control unit in the ID stage. In normal operation all units can be considered to be combinational with the exception of the instructions MUL, MULH and SW. These instructions stall the pipe-line from the EX stage (MUL, MULH) or MEM stage (SW).

The other reason to stall the pipe-line is that a combination of instructions that prohibits the forwarding logic to avoid a stall. An example of this is an LW instruction directly followed by an BNE instruction that uses the result from the LW instruction. As the LW result is made available in the MEM stage and is needed in the ID stage one clock cycle before it exists, such a forwarding path is not possible. These conditions are covered in detail in chapter 4.2.7.

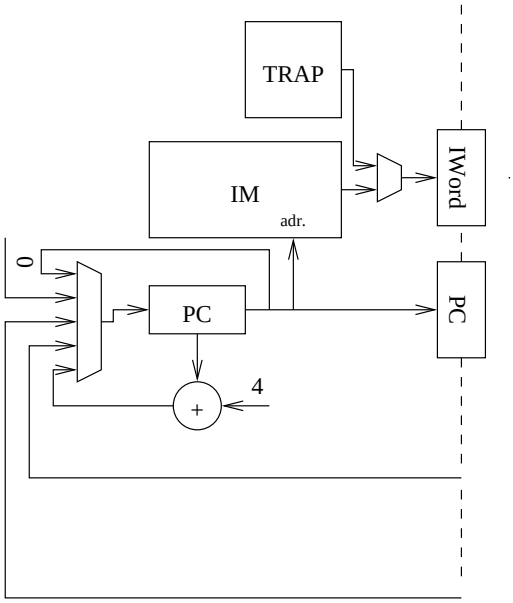
Branch instructions are evaluated in the ID stage. This leaves us with a branch delay-slot which we allow the user to fill. Putting a JUMP, TRAP, SUBV, BNE, BEQ, ADDV or an illegal operation in the delay-slot will result in unpredictable behavour. The other program flow control instructions, JUMP and TRAP, are evaluated in the MEM stage. Any instructions in the IF, ID and EX stages are flushed from the pipe-line when a JUMP or TRAP reaches the MEM stage.

Interrupts from the outside world are inserted into the pipe-line as TRAP instructions. To avoid having these instructions flushed, an external trap causes the IF stage to feed the pipe-line with the right TRAP instruction until one TRAP instruction has reached the MEM stage. This is covered in detail in chapter 4.2.8.

4.2.2 IF

The instruction fetch pipe-line stage feeds the pipe-line with instructions. It can be seen in figure 8. It can get each instruction from on of two sources: the instruction memory, IM, or the interrupt injector.

FIGURE 8. The IF Stage



To fetch the instructions from the IM a memory access unit, MAU, is used. This unit is an exact copy of the unit used in the MEM stage, but wired to only perform read operations. Please refer to section 4.2.5.1 for details concerning the MAU.

The IM contains the actual program code to run. Please notice that the instruction memory is completely separated from the data memory, i.e. you cannot write self-modifying code.

The interrupt injector is used when a hardware interrupt has occured. The SYNCTRAP signal notifies the CPU core that an interrupt has occured. When this happens, the CPU Core checks if it is fetching an instruction from a branch delay-slot. In this case, the interrupt is postponed one clock cycle. When not fetching such an instruction, a predefined instruction word containing a TRAP instruction is introduced to the pipeline. This instruction is inserted into the pipe-line each clock cycle until it reaches the MEM stage and is executed. This is to avoid the loss of an external interrupt if, for example, a JUMP instruction flushes the pipe-line. To avoid mixing up a software initiated TRAP instruction from an externaly initiated TRAP an extra control bit is submitted along the instruction word. This is needed to make sure that two TRAPs are executed when a software TRAP is in the pipe-line and a hardware interrupt occurs.

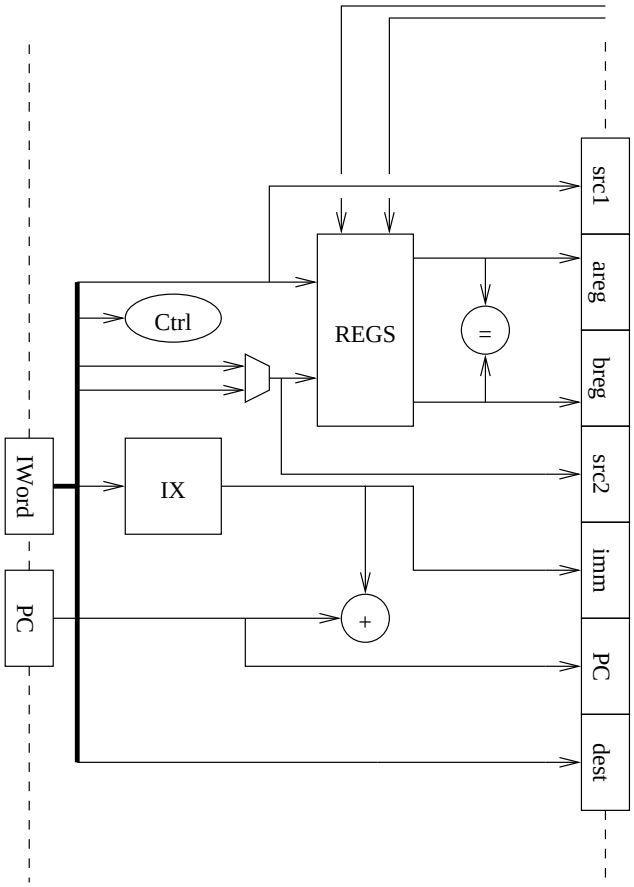
4.2.3 ID

The instruction decode pipe-line stage is responsible for interpreting the instructions into data and control signals. It can be seen in figure 9. This is the last stage dealing with the actual instruction word, from here on the instruction will progress through the pipe-line as a set of control signals.

The data is retrieved from the register bank and produced from the immediate extender unit, IX, from the immediate field. Some instructions treat the destination field as a source, this is managed by a multiplexer just before the register file.

The control signals are generated in the control unit. The control unit is implemented as a ROM containing all control signals for each of the 64 available instruction words. All illegal instructions are converted into the appropriate TRAP instruction.

FIGURE 9. The ID Stage



In the instruction decode stage we also evaluate the branches early. This reduces the number of branch delay-slots to one. The logic needed to do this consists of a compare between the two register file outputs and two signals from the control unit indicating either BNE or BEQ.

4.2.4 EX

In the EX stage all arithmetic and logical operations are evaluated. Both address calculations for SW, LW and JUMP and results from data processing operations such as ADD, XOR, etc. is taken care of here.

The stage can be seen in figure 10. The IU has the capability to stall the pipeline. In figure 10 the logic used to implement the stalling is shown. This solution has to be used as the IU is implemented as a state-machine and would need one clock cycle to emit a stall signal. As shown, the control signal to trigger the multi-cycle instruction is connected to the pipe-line registers enable signal. When the state machine is ready, it tells

the pipe-line registers to stop stalling by setting a signal low during one cycle. The integer unit, IU, is described in detail in the next section.

The special register PSW is handled in the EX stage. This is handled by an enable signal generated by the control unit which causes the register to load the the IU result.

FIGURE 10. The EX Stage

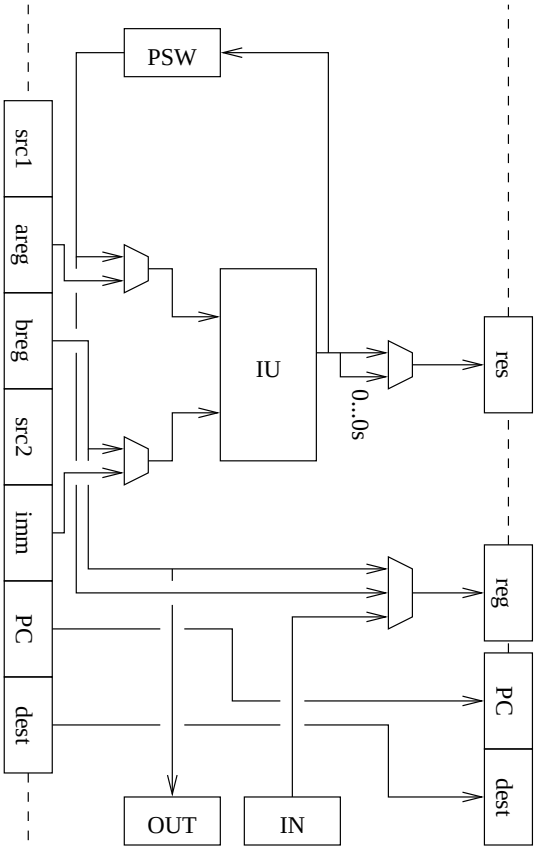
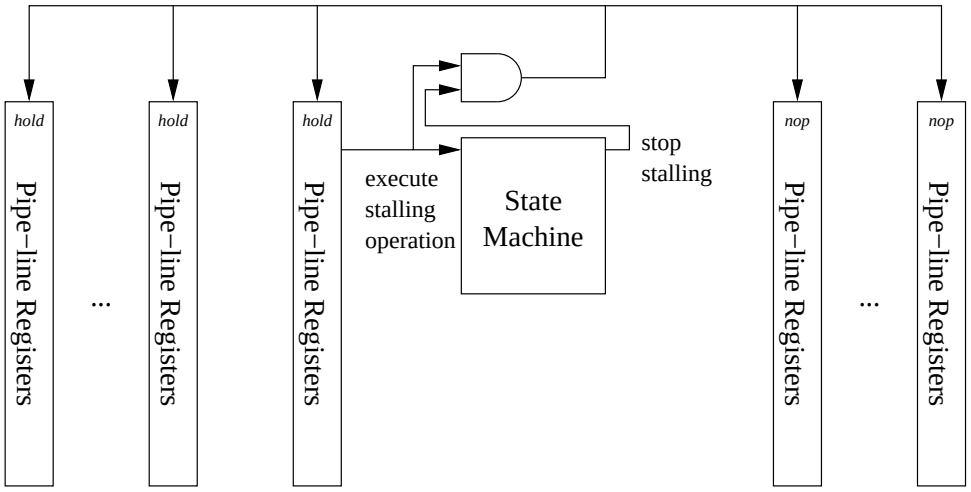


FIGURE 11. The Generic Stalling Logic



4.2.4.1 Integer Unit

The integer unit in the JAM CPU core can handle the following integer operations:

- Multiplication
- Addition
- Subtraction
- Bitwise OR
- Bitwise AND
- Bitwise XOR
- Shift and fill with zero
- Shift and fill with sign

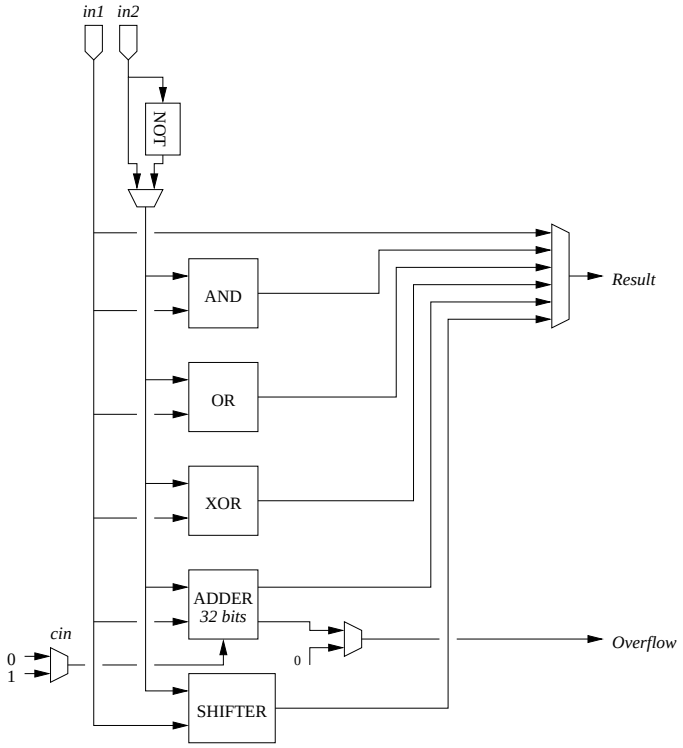
All operations beside multiplication take one cycle, i.e. the whole integer unit acts like combinational logic. Multiplication takes 33 cycles (one setup cycle and 32 cycles for the multiplication) and the integer unit is completely self-controlling during this time.

The integer unit contains control logic, a shift register used for multiplication and an ALU that handles addition, subtraction and the logical operations.

4.2.4.2 ALU

The ALU used in this integer unit has a very simple design and only handles the most basic operations. Figure 12 shows the basic design.

FIGURE 12. The ALU



The inverter on in2 is controlled from the outside for OR, AND and XOR operations but is also used on subtraction (see below).

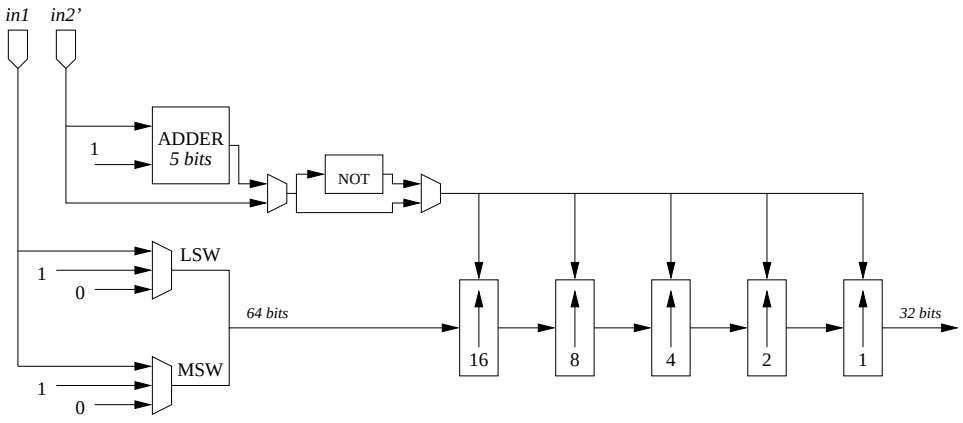
The 32 bits adder is used on both addition and subtraction, the only difference being that in2 is inverted and cin is set to 1 on subtraction. Overflow is detected by comparing MSB on indata and the result.

The compare operation is implemented as a subtraction, but we extract only the sign and put it in LSB of a word that consists of zeroes.

4.2.4.2.1 Shifting

The shift logic can shift a 32 bits value up to 32 steps in any direction. This is done by shifting in five stages as shown in figure 13. The first stage shifts 16 or zero steps to the right, the second stage shifts 8 or zero steps to the right and so on. This makes it possible to shift an optional amount (zero up to 32) of steps to the right. To handle both left and right shifts we operate on a 64 bits value. The value to be shifted is placed in the upper part for left shift and in the lower part for right shift, and the rest of the bits are filled with zeros or ones depending on the sign (or always zero if we want normal shift). To shift to the left we now simply invert the control signals to the five stages (controlling shift on/off). The result can be found in the lower part of the 64 bits value when finished.

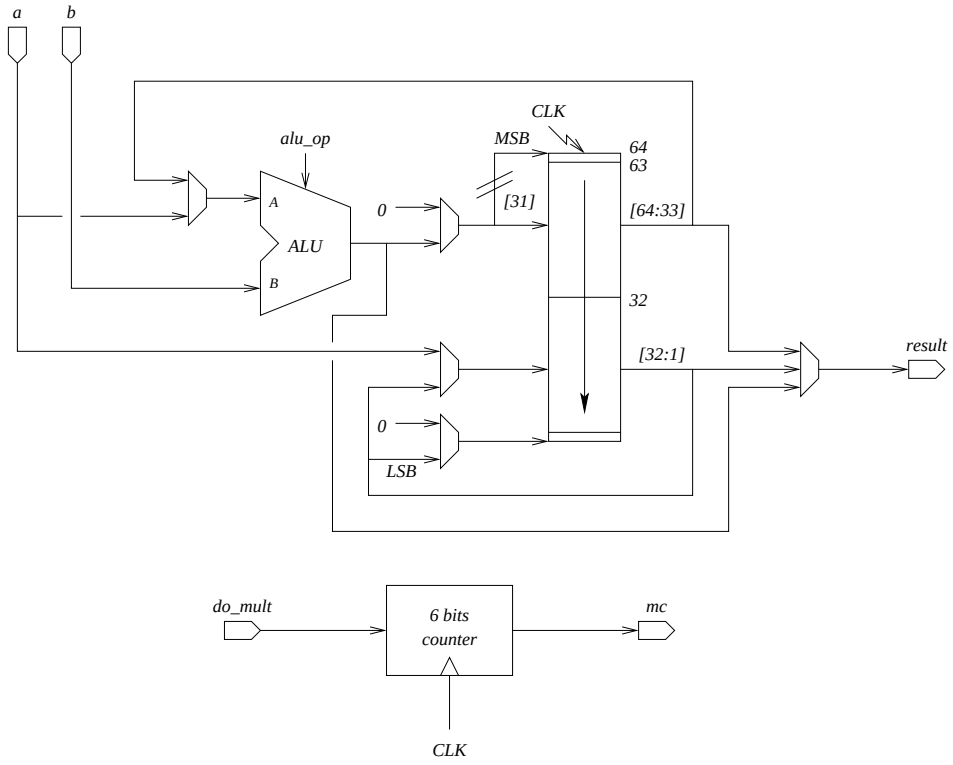
FIGURE 13. The Shifter



4.2.4.2.2 Multiplication

Booth's algorithm [2] is used to calculate the product in multiplication operations. As seen in figure 14 we use the ALU, a 65 bits shift register, a 6 bits counter and the nesecary control logic.

FIGURE 14. The Multiplication Logic



The A input to the ALU is set to bit 65 down to 34 from the shift register and the B input to the multiplicand (b) during the whole operation. On the setup cycle, the shift register is reset to [65..34 <= 0, 33..2 <= a, 1 <= 0]. On each following cycle the shift register will read the result from the ALU and put it in bits 64 down to 34 (bit 65 is set to the sign of the result) and the rest will be shifted one step to the right. The ALU performs an addition, subtraction or pass A on each cycle depending on the lowest two bits from the shift register ("01" means addition, "10" means subtraction and every thing else means pass A).

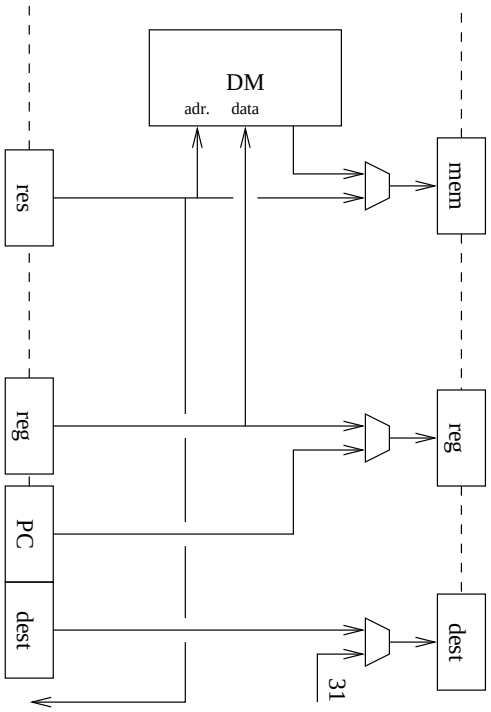
A counter tracks the number of cycles and gives a signal when finished. The result is then taken from the lower or upper part of the register depending on the type of multiplication operation.

Its also possible to design single cycle multiplication hardware, but the area needed for this will be relatively large. If the space is available and it does not create a critical path then its quite simple to do this. The simplest way is to just do "a*b" in VHDL but most vendors of FPGA chips also have freely available reference code that is optimal for their hardware.

4.2.5 MEM

The memory pipe-line stage has three distinct jobs: memory accesses, jump and traps (including interrupts). It can be seen in figure 15.

FIGURE 15. The MEM Stage



The memory stage will stall for on cycle at memory store instructions (SW), but otherwise execute all operations in one clock cycle.

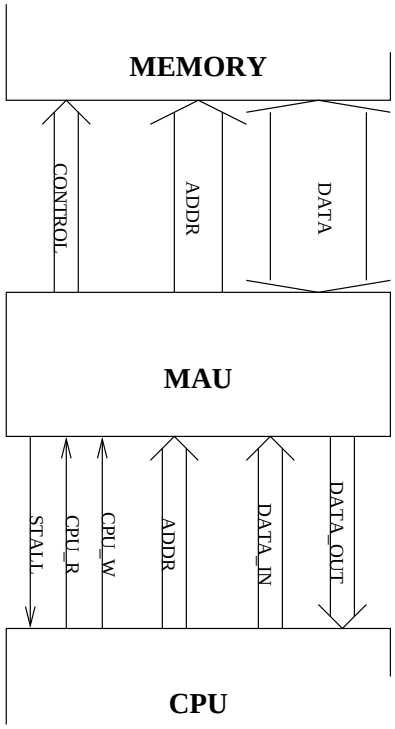
4.2.5.1 MAU

All communication with DM and IM is managed through the two Memory Access Units, MAU:s. By isolating the memory access in a separate unit we simplify any future modifications to the memory system and increase the portability of the microprocessor to different hardware platforms.

The interface between the MAU and the CPU core are simply an address bus, an indata bus and an outdata bus, all 32 bits wide. There are also three control signals controlling the actions of the MAU: read, write and reset. Finally it has a stall signal to signal the CPU to stall when needed (i.e. for a write).

The interface between the MAU and the memory is an address buss of 19 bits, a data buss of 64 bits and 9 bits wide controle buss. Seven bits in the control buss is CS and the other two is OE and WE.

FIGURE 16. The Memory - MAU - CPU Interfaces



Bits 19..1 of the CPU address are always forwarded directly to the memory, due to the limitation in lab system memory size (512k) not all 32 bits are used. The 0:th bit is used for selecting the high or low 32 bit word (since the lab system memory width is 64 bits).

When write is not set by the cpu, data is set to high-Z. Otherwise input data is forwarded to both high and low word of the memory data bus. The CS signals are also controled by the write signal.

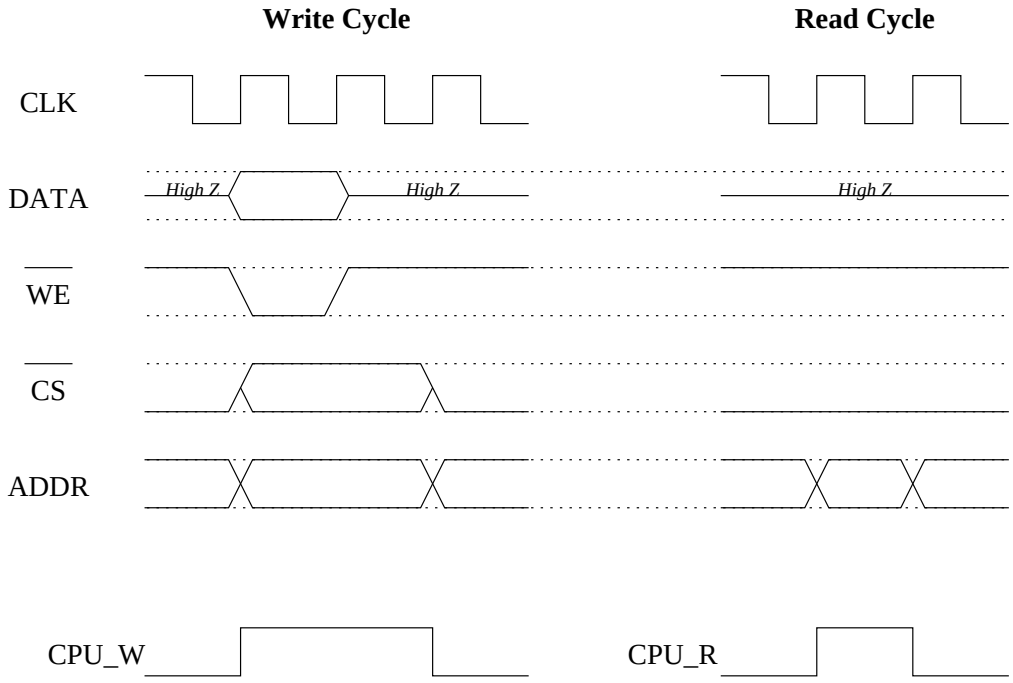
A write cycle is initiated by the write signal. Stall signal to the cpu is then set and write enable to the mem is also set. By doing this the memory puts high-Z on the data buss so that we do not get a buss collision. The address is forwarded (as always). See figure 17. Correct word is selected using the CS signals, so that we only write half of the 64 bits (63..32 or 31..0).

In the next cycle the same address is kept since pipe-line registers do not change during the stall. Now write enable is zeroed and data is put to high-Z (by MAU). The stall signal is also zeroed. This second cycle is needed to avoid malcious writes to wrong address[1].

In a similar way as the write, a read cycle is initiated with a read signal from the CPU. Data word is selected with zeroth address bit and delivered from memory to the CPU. All mem ICs is always selected unless it is a writecycle. See figure 17.

1. A single cycle write could be implemented by gating the clock, but the performance gain was too small to make it worth implementing within the dead-line for this project.

FIGURE 17. The Read and Write Cycles



4.2.5.2 Jumps and Traps

If any of the HW-interrupt, illegal op, SW-trap or overflow control bits are set when the instruction reaches the memstage a jump to address 0 (trap vector) is taken. If the instruction is in the branch delay slot this jump is NOT taken.

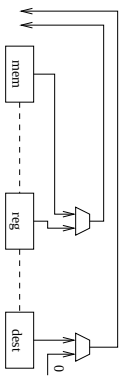
JUMP instruction will also be executed in this stage.

In both cases the pipeline will be flushed and the PC loaded with the relevant address.

4.2.6 WB

The write back stage is used to update the register file and is shown in figure 18. As we do not know if an instruction is supposed to update the registers when we start executing it we have the ability to disable the destination, and always address register zero (which is read-only). We also choose which data to write to the register. It can be either from the memory or ALU, or from a register such as IN, PSW and PC.

FIGURE 18. The WB Stage



4.2.7 Forwarding and Hazard Checking

To avoid unnecessary pipe-line stalls the CPU core uses forwarding throughout the design. All multi-cycle units are responsible for buffering any data values that they might need in later cycles as data is only guaranteed to be valid in the first cycle. This design simplifies the design of the forwardning tremendesly.

The multi-cycle units are the integer unit (in the EX stage) and the memory access unit (in the MEM stage). A complete overview of the datapath including the forwarding logic is found in figure 19.

If the instruction following a load instruction uses the result the pipe-line will be stalled (this is called a LW hazard).

If a branch instruction enters into the ID stage before all parameters are available the pipe-line will be stalled (this is called a stall for branch logic hazard).

On stalls we will also try to finish instructions that preceeds the stalling instruction by inserting no-ops (resetting pipe-line registers). Table 4 shows the complete logic for this. Table 5 describes when the different signals refered to in table 4 are set.

TABLE 4. Pipe-line Control Conditions

Pipe-line Register	Action	Criteria
IFID	Hold	id_stalling == 1 || ex_stalling == 1|| mem_stalling == 1
	Reset	mem_clear == 1
IDEX	Hold	ex_stalling == 1 || mem_stalling == 1
	Reset	(id_stalling == 1 && idex_hold == 0) || mem_clear == 1
EXMEM	Hold	mem_stalling == 1
	Reset	mem_clear == 1 ||
		(ex_stalling == 1 && mem_stalling == 0)
MEMWB	Hold	mem_stalling == 1
	Reset	n/a

TABLE 5. Stalling Causes

Signal	Cause
id_stalling	Stall for branch logic when needed data is in EX.
ex_stalling	Stall for multiplication or LW hazards (i.e. we need data from MEM in EX).
mem_stalling	Stall from MAU (for SW).
mem_clear	Flushes pipe at JUMP or TRAP.

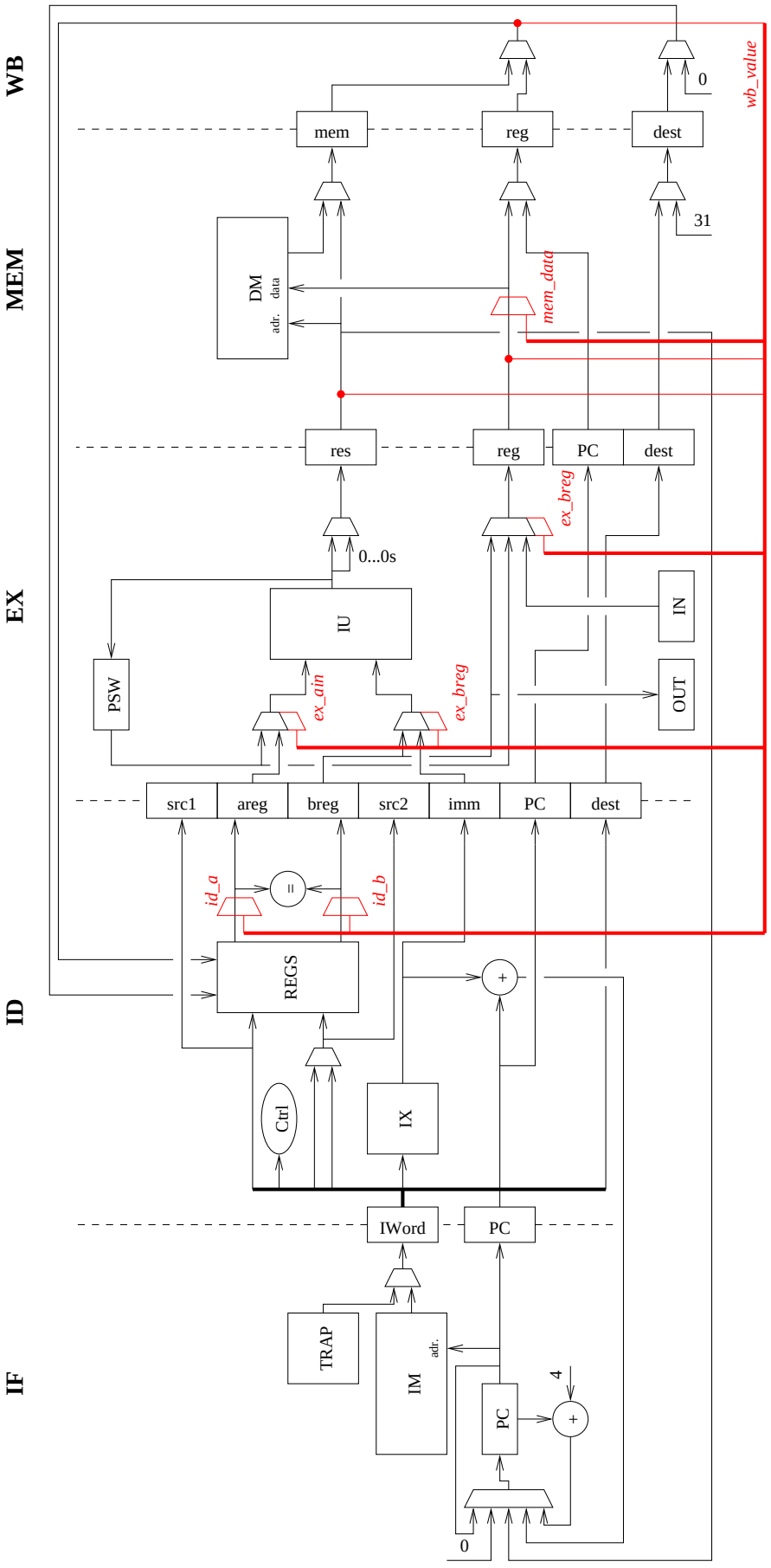
Table 6 shows all forwarding paths and conditions.

TABLE 6. Forwarding

Forward to	Forward from	Condition
id_a/b	exmem_reg.res	scr1/src2 == exmem_reg.dest && res_valid
	exmem_reg.reg	src1/src2 == exmem_reg.dest && reg_valid
ex_ain/ex_breg	exmem_reg.res	src1/src2 == exmem_reg.dest && res_valid
	exmem_reg.reg	src1/src2 == exmem_reg.dest && reg_valid
	wb_value	src1/src2 == memwb_reg.dest && wb_valid
mem_data	wb_value	dest == memwb_reg.dest && wb_valid && write

The valid flag is set by the control unit in the ID stage if forwarding data is valid. If the destination is R0 the forwarding data is always considered invalid and thus ignored by the forwarding logic.

FIGURE 19. Overview of the Pipe-line Datapath with Forwarding



4.2.8 Traps and Interrupts

The CPU core only knows of the instruction TRAP, and three special control bits associated with that instruction. The bits causes either the overflow or the illegal operation bit to be set in PSW, or stops the interrupt injector in the EX stage from injecting the pipe-line with more TRAP instructions.

When an illegal operation reaches the ID stage it is translated into a TRAP instruction. The special illegal operation control bit is also set. This bit causes the illegal operation bit in PSW to be set when the instruction reaches the MEM stage.

When an overflow checking operation (either ADDV or SUBV) causes an overflow, the special overflow control bit is set in the EX stage. This causes the MEM stage to treat the instruction as a TRAP instruction and set the overflow bit in PSW.

When an external interrupt occurs it triggers the interrupt injector. If the instruction being fetched at the time is located in a branch delay-slot, the trap injector will postpone the interrupt one clock cycle. When the injector activates it will start feeding the pipe-line with TRAP instructions with an additional injector cancelation control bit. This bit is used to detect that the hardware interrupt initiated TRAP instruction actually reaches the MEM stage, and is not flushed by an earlier instruction (such as JUMP or TRAP). When the bit reaches the MEM stage, it stops the trap injector from introducing more traps to the pipe-line until a new external interrupt is detected.

In the MEM stage all TRAP instructions are treated the same way. They simply cause the PC to reset to zero, and all earlier pipe-line stages to flush. The special control bits are treated separately from this logic to ensure function.

4.3 Design Choices and Performance

This chapter deals with the process of verifying the design. The verification results are used not only to remove errors, but also to optimise the performance of the CPU.

4.3.1 Testing

In order to verify the function of the pipe-line we have simulated it at several different levels.

At the lowest level, each component has been simulated separately. This is to ensure that we do not hide any errors in the structure of the pipe-line. Parts of the components are very rarely used in the pipe-line, but they must still work properly.

At the next level, we assembled a simple pipe-line without any hazard checking or forwarding. It was at this level that we started to introduce instructions into the pipe-line. We successfully ran the test program P0 using this simple pipe-line.

The next thing we introduced was the forwarding logic. With this pipe-line most instruction sequences could be tested as long as no-ops were introduced to avoid impossible forwarding situations.

After this we introduced the hazard checking and started testing multi-cycle instructions, and instructions with “impossible” dependecies. To solve these dependencies the hazard checking logic has to stall parts of the pipe-line.

The last testing was done by programming the target FPGA with the design. Thanks to the thorough testing it worked properly right from the start.

4.3.2 Design Choices

In this section we will discuss three design choices that we have made during the implementation of the CPU core. We have faced more choices during the work but feel that these three represent all categories of trade-offs that we have made.

4.3.2.1 Too Aggressive Forwarding

Suppose that the CPU is feed with an instruction sequence as the one shown here:

```
ADDI    R1, R1, 1
BEQ     R1, R2, Label
```

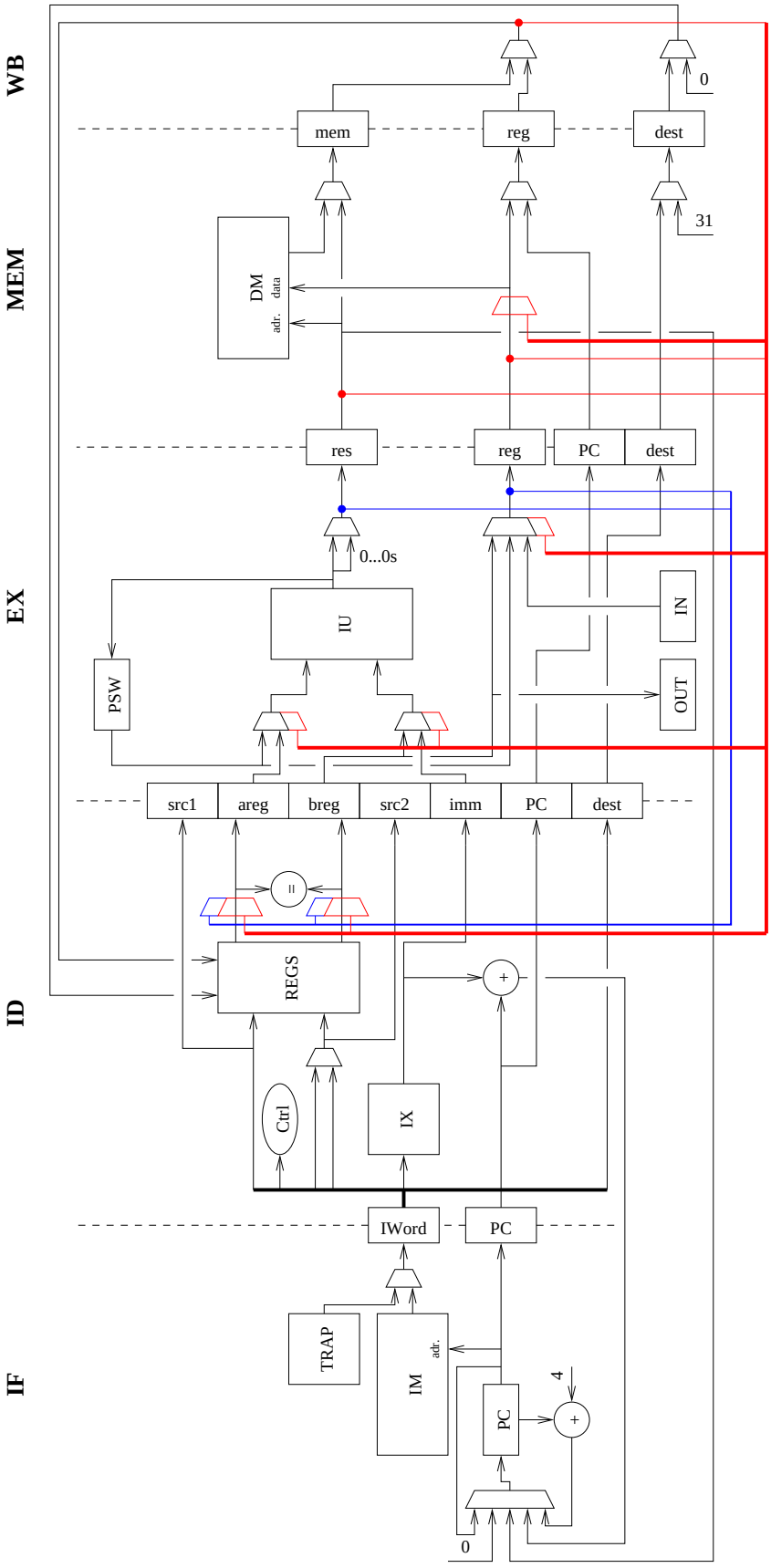
As the result calculated for the ADDI instruction in the EX stage is needed for the branch evaluation in ID stage we need to forward from the end of the EX stage to the middle of the ID stage to avoid a stall.

When implementing this stall free solution we achieved a critical path through the entire EX stage continuing through the branch evaluation logic (blue forwardning in figure 20). This limited the clock frequency to approximately 15 MHz.

When we found that this was critical path of the entire CPU core we decided to sacrifice this solution and to stall for one cycle in this situation. This increased the possible clock frequency to approximately 24 MHz.

As the stall can be avoided through instruction re-ordering by the compiler it does not increase the CPI number of the CPU, nor does it affect the instruction count. Thus we achieved a speed-up of $n = T_{original}/T_{improved} = F_{improved}/F_{original} = 24/15 = 1.6$, i.e. a 60% gain. The only concideration is to have at least one instuction between the loop counter update instruction and the loop branch evaluation to take advantage of this speed-up.

FIGURE 20. Overview of the Pipe-line with Additional Forwarding[1]



1. Please notice that this is not the pipe-line used in the final implementation.

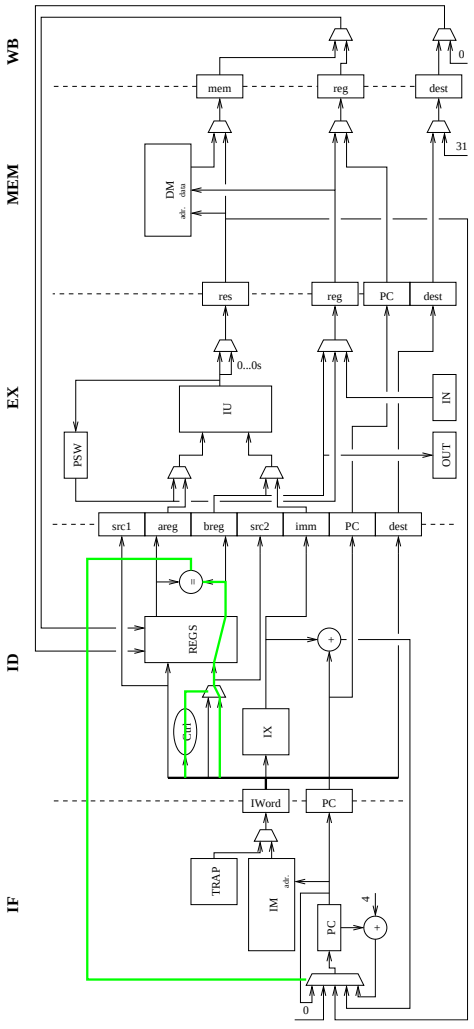
4.3.2.2 Early Branch Evaluation

In the early stages of design we made the decision to evaluate our branches early and to let the user use the branch delay slot freely. The design choice was made because of the following reasons:

- The usage of branch delay slots simplify the design and improve performance. Both these gains come from the removed need to flush parts of the pipe-line after a miss prediction (we did not concider stalling at each branch an option).
- It is hard to utilize many delay slots. This means that if we evaluate the branch earlier in the pipe-line, we get a higher percentage of useful delay slots and thus higher performance.

The cost of this solution proved to be the critical path. The longest path in the entire CPU core is shown in green in figure 21. The path starts through the control unit that controls the multiplexer selecting which part of the instruction word to use as the second register index. It continues through the branch evaluation logic to control the multiplexer selecting the PC for the next cycle. This is the path that limits our design to 24 MHz.

FIGURE 21. The Critical Path



4.3.2.3 Shorter Critical Path

As we saw in the previous section, the critical path wanders through the control unit into the multiplexer before the register bank. To shorten the critical path we attempted to control the multiplexer directly from the op-code part of the instruction word.

There are three instructions that needs the second register index to be taken from the destination field instead of the actual second register field: BNE, BEQ and SW. These three instructions were directly decoded into a control signal for the multiplexer outside the control unit.

This design shortened the critical path as was thus successful. The clock frequency improvement was 3 MHz, from 22 MHz to 25 MHz. The achieved speed-up was 1.14.

4.3.3 Performance

According to the design tools the CPU can be run at a clock frequency of 25 MHz. As the host system only supplies a clock of 80 MHz we choose to divide this clock twice and thus run the CPU at a clock frequency of 20 MHz.

In this section we begin by presenting the entire clock cycles per instruction table and average CPI. This is followed by the analysis of the test programmes P0, P1 and P2.

4.3.3.1 CPI

The cycles required for each instruction and the average use of each instruction is listed in table 7. Please notice that all instructions include all possible versions of that instruction, for example, ADD includes ADDI, ADDX and ADDD.

When we execute code without any hazards that the forwarding logic cannot handle we get a weighted CPI of 1.71 clock cycles per instruction. As we use a clock frequency of 20 MHz we will get 11.7 MIPS. This is the theoretical maximum performance of the CPU.

TABLE 7. Clock Cycles per Instruction and Instruction Frequency

Instruction	Cycles	Frequency (%)
ADD	1	16
ADDV	1	0
SUB	1	1
SUBV	1	0
CMP	1	6
MUL	33	2
MULH	33	0
AND	1	3
OR	1	2
XOR	1	1
RESET	1	1
SET	1	2
SHZ	1	12
SHS	1	1
LW	1	15
SW	2	7
GET	1	6
PUT	1	7
BEQ	1	8
BNE	1	8
JUMP	1	2
TRAP	1	0

If we would run the CPU at 25 MHz we would get 14.6 MIPS, which is slightly better. If we were going to make the improvement discussed in chapter 4.2.4.2.2, and introduce a one cycle multiplier we could however improve the CPI and MIPS dramatically. The average CPI would then be 1.07, and under the assuption that we still can run the CPU at 20 MHz we would get 18.7 MIPS. The second CPI increasing instruction, SW, can also be made a one cycle operation without much work, as noted in 4.2.5.1. This would give a weighted CPI of one, which is the optimum for a scalar design.

4.3.3.2 P0 Performance

The test program P0 consists of an instruction sequence free of hazards. It requires no forwarding, and no contact with external units except IM. Notice that we end the program with an infinite loop as this is not the case in the original P0.

```
-- Test program P0
ADDI    R1, R0, 56
ADDI    R2, R0, -218
RESET   R3, R0, R0
ADDX    R1, R1, -4
ADDX    R2, R2, 1
SHZI    R3, R3, 8
AND     R4, R1, R0
CMP     R5, R1, R2
BEQ     R0, R0, 0
ADD     R0, R0, R0
```

As the code is free from any hazards the average CPI is one cycle per instruction. Notice that all potential dependencies through R0 are ignored, so the last two instructions do not have a dependency even though the branch uses the result of the addition as a source. Please refer to the enclosed signal sheet for the simulation results of P0.

The execution time for P0 is eight cycles (400 ns at 20 MHz), plus four additional start-up cycles (200 ns at 20 MHz), and an infinite number of cycles in the final loop.

4.3.3.3 P1 Performance

The test program P1 uses DM and IM. It contains hazards, multi-cycle instructions and JUMP instructions. Please notice that the version below has not been optimized, all branch delay slots simply contains no-ops.

```
-- Test program P1
-- Call sub routine INIT
JUMP    R30, R0, INIT
-- Initialize R1 and R4
ADDI    R1, R0, 12
ADDI    R4, R0, R0
L1:
-- Process the data
LW      R2, R1, 1024
LW      R3, R1, 1040
MUL     R5, R2, R3
ADDI    R1, R1, -4
ADD     R4, R4, R5
CMP     R5, R0, R1
BNE     R5, R0, L1
NOP
-- Store the result
SW      R4, R0, 1056
-- Inifinite loop
END:
BEQ     R0, R0, 0
NOP
-- Init routine, fills out data
INIT:
ADDI    R1, R0, 32
L2:
SW      R1, R1, 1020
ADDI    R1, R1, -4
BNE     R1, R0, L2
NOP
JUMP    R0, R30, 4
```

The first JUMP instruction will use 4 cycles to reach the initialization routine. The initialization routine initializes for one cycle, and loops (L2) for 48 cycles (8 loops, 6 cycles each). The returning JUMP uses 4 cycles to reach the first ADDI. The initialization of the main loop (L1) needs 2 cycles. The main loop (L1) uses 168 cycles (4 loops, 42 cycles each). The final SW uses 2 cycles, and then an infinite loop follows. As always we also use four start up cycles to fill the pipe-line.

P1 will use 234 cycles in total (11.7 ms at 20 MHz), plus four cycles to start-up (200 ns at 20 MHz). During this time 71 instructions are executed, thus our CPI is 3.3. This calculation does not include the final loop. Additional performance can be achieved by moving the branch condition calculation away from the branch instruction and using the branch delay slots. As P1 is too long to fit on a reasonable number of papers we do not supply any waveforms of a simulation.

4.3.3.4 P2 Performance

Test program P2 uses the I/O capabilities of the CPU. As the hardware does not support this, we do not test this in actual hardware. When calculating the CPI we assume that 2 bytes are available each time. Of the input, one byte is negative, the other is positive or zero.

```
-- Test program P2
-- Init PSW, counter and masks
SETX    R1, R0, 1
ADDI    R2, R0, 0
ADDI    R3, R0, 0x0001
ADDI    R4, R0, 0x0010
-- Check all IO ports
L1:
-- Check if anything is on the port
AND     R5, R1, R3
BEQ     R5, R0, L3
NOP
-- Data available
GET     R6, R0, R3
-- Zero if less than zero
CMP     R5, R0, R6
BEQ     R5, R0, L2
NOP
ADD     R6, R0, R0
L2:
PUT     R6, R0, R4
L3:
SHZI    R3, R3, 2
SHZI    R4, R4, 2
ADDI    R2, R2, 1
CMPI    R5, R2, 4
BNE     R5, R0, L1
NOP
-- Enable timer interrupt and wait
RESETI  R0, R0, Ox8000
BEQ     R0, R0, 0
NOP
```

The initialisation and cleaning up each run uses 5 cycles (not counting the infinite loop). The outer loop (L1) contains 9 single cycle instructions, and causes one stall, i.e. 10 cycles per loop, all four loops. The data acquiering, sign checking and outputting code holds five single cycle instructions that will be used in two iterations. Addition to zero out negative input is run one time and will use one cycle.

In total P2 uses 56 cycles (280 ns at 20 MHz) each run while running 52 instructions, yielding an average CPI of 1.1. The stall is due to a branch condition calculation taking place just before the actual branch instruction. By reordering the instructions properly a CPI count of one would be possible. If the branch delay slots were to be utilized properly ten more cycles could be saved.

5.0 Synthesis of the JAM CPU Core

We have synthesised the JAM CPU core for several FPGA models. In this chapter we will first discuss the synthesis for the Xilinx Virtex XCV300 FPGA in detail. In the later sections we will briefly discuss the synthesis for other FPGAs and report the area and timing parts of the synthesis reports.

5.1 Xilinx Virtex XCV300

We have used Synplify Pro to synthesize our construction. Below we show the relevant data from the performance and resource usage reports. The circuit documented below-has successfully been tested in an FPGA.

5.1.1 Performance Summary

```
Worst slack in design: -29.174
Requested frequency: 80.0 MHz
Estimated frequency: 24.0 MHz
```

5.1.2 Resource Usage Report

```
BUFGs + BUFGPs: 2 of 4 (50%)
Total  LUTs: 3163 (51%)
```

5.2 Xilinx Virtex2 XC2V3000

The Virtex2 FPGAs are the evolution of the Virtex FPGAs. We can see that the FPGA technology has evolved and we get a big performance increase.

5.2.1 Performance Summary

```
Worst slack in design: -7.033
Requested frequency: 80.0 MHz
Estimated frequency: 51.2 MHz
```

5.2.2 Resource Usage Report

```
BUFGs + BUFGPs: 2 of 8 (25%)
Total  LUTs: 3162 (11%)
```

5.3 Altera MERCURY EP1M350

To be able to compare different FPGA manufacturers we have synthesized the design for an FPGA from Altera.

5.3.1 Performance Summary

```
Worst slack in design: -11.084
Requested frequency: 80.0 MHz
Estimated frequency: 42.4 MHz
```

5.3.2 Resource Usage Report

```
Logic resources:  4789 ATOMs of 14400 (33%)
```

6.0 Conclusions

We have come to the following conclusions during this project:

- Always, always think in terms of hardware. VHDL is not a software language. This makes the design easier to synthesize.
- Do not try to do clever optimisations or hacks in your VHDL code. It will only confuse the synthesis tool, and besides, it probably can optimise better than you.
- Try to extract as much parallellalism in you VHDL. Hardware is truly parallell and this can give huge performance gains. The synthesis tool will still extract common resources and optimise both for speed and area better than you can.

7.0 References

[1] Vasell, Jonas, *Datorarkitekturimplementering i VLSI*, Institutionen för datorteknik, Chalmers tekniska högskola, Göteborg, Andra upplagan, 1998

[2] D. Patterson and J. Hennessy, *Computer Organization & Design 2nd Edition*, Morgan Kaufmann 1998, page 259.

[3] D. Patterson and J. Hennessy, *Computer Architecture*, Morgan Kaufmann 1996