# Design and Implementation of RISC-V Microprocessor


Diploma Thesis

Ketoglou Theocharis
Department of Computer Science & Engineering, University of Ioannina

2020-2021

# Dedication

To my family Athanasios,Ifigeneia,Georgia,Anastasios,Anthoula my grandmother Maria and my grandfather George.

# Acknowledgments

I would like to thank my supervisor professor mr.Euthimiou for his excellent work making me understand the basics of how a CPU works through his lessons.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Prologue

In the 21st century the technology is progressing faster than any other century. Every day new things are invented and the most of them are in the computer industry. With quantum computers at our door step the need for fast and cheap processors is more necessary than ever. Today two main companies are responsible for the future of processors. Intel and AMD control the market of most desktop and notebook computers but with the rise of the mobile computing we can see new companies come to the foreground and try to get a share in the market. Most of these new companies are basing their processors on ARM, another big company in the processor industry. As a result of all these, the processor industry is controlled from these three companies with a few exceptions, so sooner or later the need of independence from them will become more realistic than ever. But why do these companies control the industry? Basically is because of their ISA (which will be explained in another section) which another company must license so it can build a processor. In this diploma thesis a microprocessor is created based in a new open source ISA named RISC-V which is believed to be the future in the computer industry. This microprocessor is a single core with a basic implementation so it can be adapted to a project like this diploma thesis.

## 1.2 Motivation

Many articles support our belief which is that the RISC-V processors will be the future. Except of our passion for engagement with the hardware and anything new we can learn from it, the motivation for this project was that the RISC-V will be the future and we must know how to deal with this

architecture. At the beginning of this project we had to choose between CISC and RISC architectures, we chose the second because most of the electronic devices make use of a microprocessor or a microcontroller which most of them are based at RISC architecture and because of its simplicity. In a second phase we chose RISC-V among other RISC architectures because it is open source, it is a future ISA and offers a lot of simplicity to the hardware design compared with other architectures.

## 1.3 What is an instruction set architecture (ISA)?

An instruction set architecture (ISA) it is a section in the computer architecture, more or less like a model, which describes how the programming is connected with the data types of the machine, the instructions and the registers. It also contains the addressing modes, the memory architecture, the interrupts-exceptions handling and the external handling of Input/Output. The most critical part of an ISA is the set of opcodes of the machine which is basically the commands that the microprocessor can execute. When a manufacturer begins to build a microprocessor the first thing that it does is the selection of the ISA that will be implemented. An ISA is categorized as a reduced instruction set computer (RISC) or as a complex instruction set computer (CISC), the RISC category contains simple commands that execute one instruction at the time and the CISC category contains complex commands that execute more instructions at the time. CISC may sound better but in some cases it is not due to the difficulty of the implementation. These two main categories RISC and CISC are not the only categories that exist but they are the main ones, another categories are VLIW, EPIC and MISC. Some of the most known ISAs are the x86, x64, 8080, ARM/A32, AVR, MIPS, PowerPC, SPARC and RISC-V. In conclusion, an ISA is a model that describes what commands a microprocessor can execute and how the architecture will be in general.

## 1.4 How ISA and microprocessor connect?

Let's say that a manufacturer A is implementing a microprocessor that will be available in the market. The microprocessor can be a result from the hard work from the engineers of A, so it is closed source which means the implementation of it is not open to the public. The manufacturer A is releasing only the ISA documentation at public so a programmer which reads this ISA

can understand which commands this microprocessor can execute, how it handles memory, interrupts etc. As a result the programmer can write and execute programs to that microprocessor and to other microprocessors with the same ISA.

## 1.5 Why RISC-V?

RISC-V is a completely open ISA that is freely available to academia and industry. It is believed that the most microprocessors will implement it because it is a future and free ISA. It is collective work from universities and companies thus the errors are reduced or zeroed. It has a variety of extensions and ready to use privileged or/and unprivileged levels. All of these make new rules to the microprocessor manufacturing industry and new companies can join which in the past could not because they had to pay to use an ISA. A lot of people say that RISC-V is the Linux of hardware, so with the support and work from communities around the world the RISC-V ISA can become a standard ISA that will be implemented in the majority of computers.

## 1.6 Report Structure

The structure of this report is divided in five chapters. The first chapter is an introduction to this diploma thesis, it contains a prologue, basic concepts of a microprocessor and motivations. The second chapter has fundamentals of the RISC-V ISA, basic instructions and some registers needed for interrupt handler are explained. The third chapter refers to the implementation of the microprocessor, it contains detailed information about every module that is part of the core. At fourth chapter the results of some evaluation tests are presented, these tests have been simulated for this microprocessor. Final chapter contains some conclusions and ideas for future work.

# Chapter 2

# Basics of RISC-V ISA

RISC-V ISA is defined as a base integer ISA, which must be present in any implementation, plus optional extensions to the base ISA. The base integer ISAs is very similar to that of the early RISC processors except with no branch delay slots and with support for optional variable-length instruction encodings. RISC-V has been designed to support extensive customization and specialization. The RISC-V ISA has a lot of extensions as the following table shows.

| Subset | Name | Implies |
|---|---|---|
| Base ISA | | |
| Integer | I | |
| Reduced Integer | E | |
| Standard Unprivileged Extensions | | |
| Integer Multiplication and Division | M | |
| Atomics | A | |
| Single-Precision Floating-Point | F | Zicsr |
| Double-Precision Floating-Point | D | F |
| General | G | IMADZifencei |
| Quad-Precision Floating-Point | Q | D |
| Decimal Floating-Point | L | |
| 16-bit Compressed Instructions | C | |
| Bit Manipulation | B | |
| Dynamic Languages | J | |
| Transactional Memory | T | |
| Packed-SIMD Extensions | P | |
| Vector Extensions | V | |
| User-Level Interrupts | N | |
| Control and Status Register Access | Zicsr | |
| Instruction-Fetch Fence | Zifencei | |
| Misaligned Atomics | Zam | A |
| Total Store Ordering | Ztso | |

Table 2.1: ISA Extensions

The base integer ISA is named “I” and contains integer computational instructions, integer loads, integer stores, and control-flow instructions. In this diploma thesis ”I” and ”Zicsr” extensions are implemented with some registers from the machine level. The registers that used from RV32I are shown in Figure 2.1.

| Register | ABI Name | Description |
|---|---|---|
| x0 | zero | Hard-wired zero |
| x1 | ra | Return address |
| x2 | sp | Stack pointer |
| x3 | gp | Global pointer |
| x4 | tp | Thread pointer |
| x5 | t0 | Temporary/alternate link register |
| x6–7 | t1–2 | Temporaries |
| x8 | s0/fp | Saved register/frame pointer |
| x9 | s1 | Saved register |
| x10–11 | a0–1 | Function arguments/return values |
| x12–17 | a2–7 | Function arguments |
| x18–27 | s2–11 | Saved registers |
| x28–31 | t3–6 | Temporaries |

Figure 2.1: Registers

RV32E is a reduced version of RV32I designed for embedded systems. The only change is to reduce the number of integer registers to 16, RV32E uses the same instruction-set encoding as RV32I, except that only registers x0–x15 are provided where x0 is a dedicated zero register. The format of the instructions are shown in Figure 2.2.

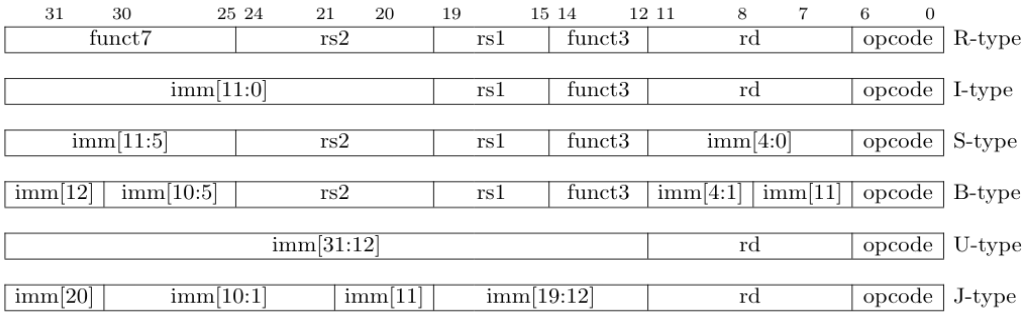



Figure 2.2: Instructions Format

## 2.1 RV32I Instructions

In the base RV32I ISA, there are four core instruction formats (R/I/S/U). All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction-address-misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. This exception is reported on the branch or jump instruction, not on the target instruction. No instruction-address-misaligned exception is generated for a conditional branch that is not taken. The RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions(Zicr), immediates are always sign-extended, and are generally packed towards the left most available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.

### 2.1.1 Integer Computational Instructions

Most integer computational instructions operate on 32 bits of values held in the integer register file. Integer computational instructions are either encoded as register-immediate operations using the I-type format or as register-register operations using the R-type format. The destination is register rd for both register-immediate and register-register instructions. No integer computational instructions cause arithmetic exceptions.

| 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| I-immediate[11:0] | src | ADDI/SLTI[U] | dest | OP-IMM |
| I-immediate[11:0] | src | ANDI/ORI/XORI | dest | OP-IMM |

Figure 2.3: ADDI,SLTI,ANDI,ORI,XORI Instructions

ADDI adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored and the result is simply the low 32 bits of the result. ADDI rd, rs1, 0 is used to implement the MV rd, rs1 assembler pseudoinstruction. SLTI (set less than immediate) places the value 1 in register rd if register rs1 is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to rd. SLTIU is similar but compares the values as unsigned numbers (i.e., the immediate is first sign-extended to 32 bits then treated as an unsigned number). Note, SLTIU rd, rs1,1 sets rd

to 1 if rs1 equals zero, otherwise sets rd to 0 (assembler pseudo-instruction SEQZ rd, rs). ANDI, ORI, XORI are logical operations that perform bitwise AND, OR, and XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd. Note, XORI rd, rs1, -1 performs a bitwise logical inversion of register rs1(assembler pseudoinstruction NOT rd, rs).

| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| imm[11:5] | imm[4:0] | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| 0000000 | shamt[4:0] | src | SLLI | dest | OP-IMM |
| 0000000 | shamt[4:0] | src | SRLI | dest | OP-IMM |
| 0100000 | shamt[4:0] | src | SRAI | dest | OP-IMM |

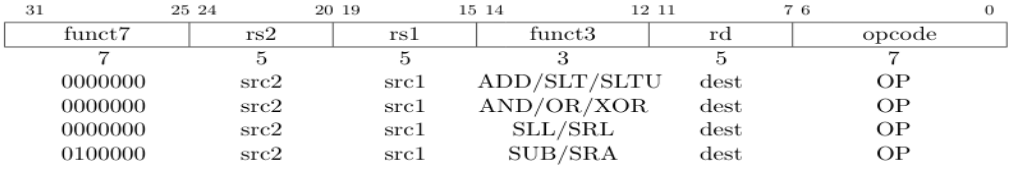
Figure 2.4: SLLI,SRLI,SRAI Instructions

Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shiftedi s in rs1, and the shift amount is encoded in the lower 5 bits of the I-immediate field. The right shift type is encoded in bit 30. SLLI is a logical left shift (zeros are shifted into the lower bits), SRLI is a logical right shift (zeros are shifted into the upper bits), and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).

| 31 12 | 11 7 | 6 0 |
|---|---|---|
| imm[31:12] | rd | opcode |
| 20 | 5 | 7 |
| U-immediate[31:12] | dest | LUI |
| U-immediate[31:12] | dest | AUIPC |

Figure 2.5: LUI,AUIPC Instructions

LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest12 bits with zeros. AUIPC (add upper immediate to pc) is used to build pc-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the address of the AUIPC instruction, then places the result in register rd.



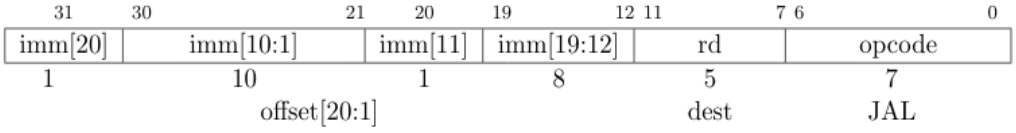
| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| 0000000 | src2 | src1 | ADD/SLT/SLTU | dest | OP |
| 0000000 | src2 | src1 | AND/OR/XOR | dest | OP |
| 0000000 | src2 | src1 | SLL/SRL | dest | OP |
| 0100000 | src2 | src1 | SUB/SRA | dest | OP |

Figure 2.6: ADD,SUB,SLT,SLTU,AND,OR,XOR,SLL,SRL,SRA Instructions

ADD performs the addition of rs1and rs2. SUB performs the subtraction of rs2 from rs1. Overflows are ignored and the low 32 bits of results are written to the destination rd. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to rd if rs1< rs2, 0 otherwise. Note, SLTU rd,x0,rs2 sets rd to 1 if rs2 is not equal to zero, otherwise sets rd to zero (assembler pseudoinstruction SNEZ rd, rs). AND, OR, and XOR perform bitwise logical operations. SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

### 2.1.2 Control Transfer Instructions

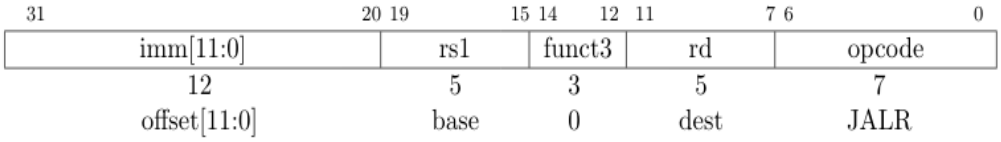
RV32I provides two types of control transfer instructions: unconditional jumps and conditional branches. Control transfer instructions in RV32I do not have architecturally visible delay slots.



| 31 | 30 21 | 20 | 19 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode |
| 1 | 10 | 1 | 8 | 5 | 7 |
| | offset[20:1] | | | dest | JAL |

Figure 2.7: JAL Instruction

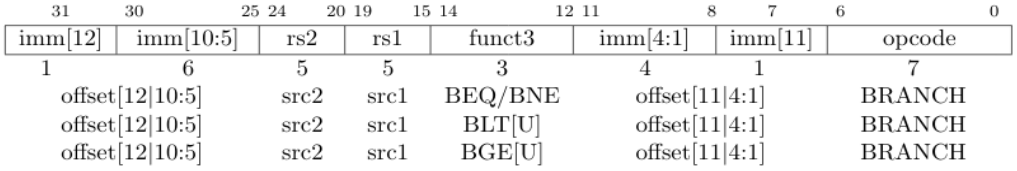
The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes assigned offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Jumps can therefore target a ±1 MiB range. JAL stores the address of the instruction following the jump (pc+4) into register rd. The standard software calling convention uses x1 as the return address register and x5 as an alternate link register.



| 31 – 20 | 19 – 15 | 14 – 12 | 11 – 7 | 6 – 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| offset[11:0] | base | 0 | dest | JALR |

Figure 2.8: JALR Instruction

The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register rs1, then setting the least-significant bit of the result to zero. The address of the instruction following the jump(pc+4) is written to register rd. Register x0 can be used as the destination if the result is not required.



| 31 | 30 – 25 | 24 – 20 | 19 – 15 | 14 – 12 | 11 – 8 | 7 | 6 – 0 |
|---|---|---|---|---|---|---|---|
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode |
| 1 | 6 | 5 | 5 | 3 | 4 | 1 | 7 |
| offset[12\|10:5] | | src2 | src1 | BEQ/BNE | offset[11\|4:1] | | BRANCH |
| offset[12\|10:5] | | src2 | src1 | BLT[U] | offset[11\|4:1] | | BRANCH |
| offset[12\|10:5] | | src2 | src1 | BGE[U] | offset[11\|4:1] | | BRANCH |

Figure 2.9: BEQ,BNE,BLT,BLTU,BGE,BGEU Instructions

Branch instructions compare two registers. BEQ and BNE take the branch if registers rs1 and rs2 are equal or unequal respectively. BLT and BLTU take the branch if rs1 is less than rs2, using signed and unsigned comparison respectively. BGE and BGEU take the branch if rs1is greater than or equal to rs2, using signed and unsigned comparison respectively.

### 2.1.3 Load and Store Instructions

RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers. RV32I provides a 32-bit address space that is byte-addressed.

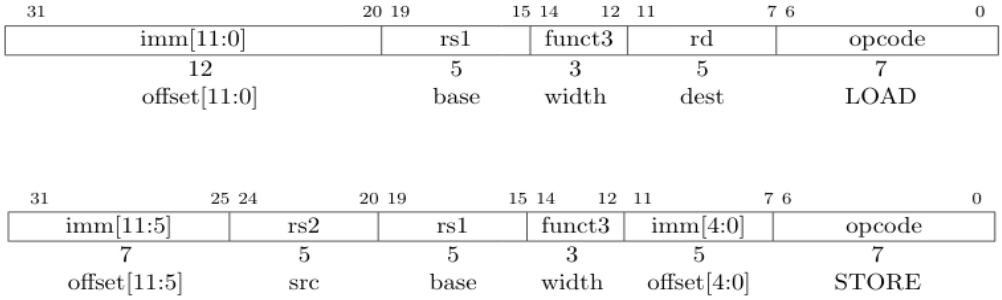



Figure 2.10: LOAD,STORE Instructions

The LW instruction loads a 32-bit value from memory into rd. LH loads a 16-bit value from memory,then sign-extends to 32-bits before storing in rd. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in rd. LB and LBU are defined analogously for 8-bit values. The SW, SH, and SB instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register rs2 to memory.

## 2.2 Zicr Instructions

Control and Status Register(CSR) Instructions. All CSR instructions atomically read-modify-write a single CSR, whose CSR specifier is encoded in the 12-bit csr field of the instruction held in bits 31–20. The immediate forms use a 5-bit zero-extended immediate encoded in the rs1 field.

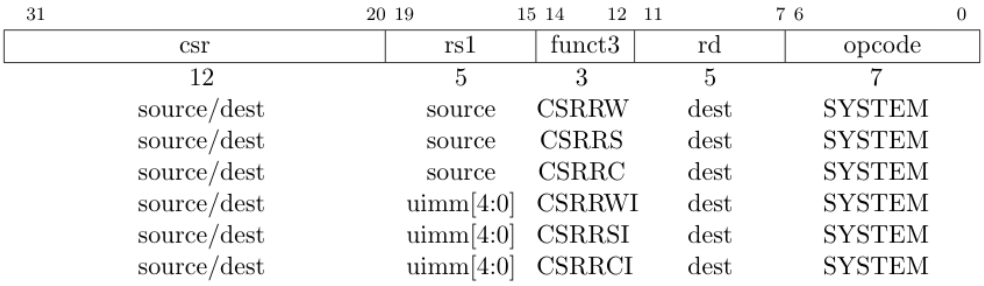



Figure 2.11: CSR Instructions

The CSRRW (Atomic Read/Write CSR) instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to 32 bits,then writes it to integer register rd. The initial value in rs1 is written to the CSR. If rd=x0, then the instruction

shall not read the CSR and shall not cause any of the side effects that might occur on a CSR read. The CSRRS (Atomic Read and Set Bits in CSR) instruction reads the value of the CSR, zero-extends the value to 32 bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in rs1will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written). The CSRRC (Atomic Read and Clear Bits in CSR) instruction reads the value of the CSR, zero-extends the value to 32 bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected. The CSRRWI, CSRRSI, and CSRRCI variants are similar to CSRRW, CSRRS, and CSRRC respectively, except they update the CSR using an 32-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field instead of a value from an integer register. For CSRRSI and CSRRCI, if the uimm[4:0] field is zero, then these instructions will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write.

## 2.3 Machine Level Registers

RISC-V has privileged levels, in simple words that means that has separate sets of registers for machine level, user level and supervisor level. Because this microprocessor was not designed to be used with operating system only the machine level register set is enough. Not all registers from the machine level will be used in order to simplify the processor. To write/read register Zicsr extension was implemented. These registers must be saved by the program on every exception-interrupt that occurs and discarded after the exception-interrupt code has been executed, the reason is that another exception-interrupt can occur when the program services the previous one. Below are the registers that are needed.

### 2.3.1 mstatus

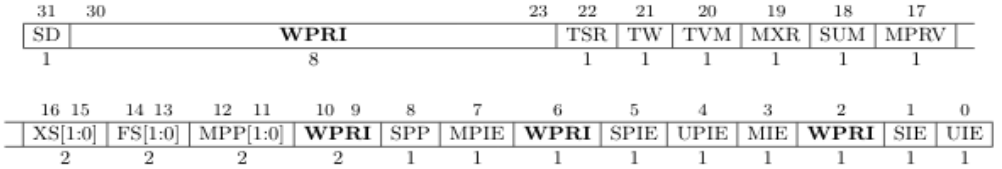



Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

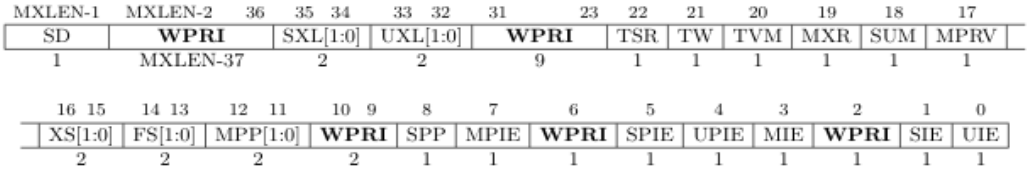



Figure 2.12: mstatus Register

The mstatus register is an 32-bit read/write register formatted as shown in Figure 2.12. The mstatus register keeps track of and controls the hardware thread's current operating state. Because no hardware threads exist in our processor except the main one, only the bit MIE(Machine Interrupt Enable) will be used from this register for general interrupt enable.

### 2.3.2 mtvec


Figure 2.13: mtvec Register

The mtvec register is an 32-bit read/write register that holds the trap vector configuration,consisting of a vector base address (BASE) and a vector mode (MODE). When MODE=Direct(0), all traps into machine mode cause the pc to be set to the address in the BASE field. When MODE=Vectored(1),all synchronous exceptions into machine mode cause the pc to be set to the address in the BASE field, whereas interrupts cause the pc to be set to the address in the BASE field plus four times the interrupt cause number.

### 2.3.3 mepc


Figure 2.14: mepc Register

Machine Exception Program Counter (mepc) . When a trap is taken into machine mode,mepc is written with the virtual address of the instruction that was interrupted or that encountered the exception. Otherwise,mepc is never written by the implementation, though it may be explicitly written by software.

### 2.3.4 mcause


Figure 2.15: mcause Register

The mcause register is an 32-bit read-write register formatted as shown in Figure 2.15. When a trap is taken into machine mode, mcause is written with a code indicating the event that caused the trap. Otherwise,mcause is never written by the implementation, though it may be explicitly written by software. Below Table 2.2 shows the values that the mcause register can take when a trap occur and describe in what every value corresponds.

| Interrupt | Exception Code | Description |
|---|---|---|
| 1 | 0 | User software interrupt |
| 1 | 1 | Supervisor software interrupt |
| 1 | 2 | *Reserved for future standard use* |
| 1 | 3 | Machine software interrupt |
| 1 | 4 | User timer interrupt |
| 1 | 5 | Supervisor timer interrupt |
| 1 | 6 | *Reserved for future standard use* |
| 1 | 7 | Machine timer interrupt |
| 1 | 8 | User external interrupt |
| 1 | 9 | Supervisor external interrupt |
| 1 | 10 | *Reserved for future standard use* |
| 1 | 11 | Machine external interrupt |
| 1 | 12–15 | *Reserved for future standard use* |
| 1 | ≥16 | *Reserved for platform use* |
| 0 | 0 | Instruction address misaligned |
| 0 | 1 | Instruction access fault |
| 0 | 2 | Illegal instruction |
| 0 | 3 | Breakpoint |
| 0 | 4 | Load address misaligned |
| 0 | 5 | Load access fault |
| 0 | 6 | Store/AMO address misaligned |
| 0 | 7 | Store/AMO access fault |
| 0 | 8 | Environment call from U-mode |
| 0 | 9 | Environment call from S-mode |
| 0 | 10 | *Reserved* |
| 0 | 11 | Environment call from M-mode |
| 0 | 12 | Instruction page fault |
| 0 | 13 | Load page fault |
| 0 | 14 | *Reserved for future standard use* |
| 0 | 15 | Store/AMO page fault |
| 0 | 16–23 | *Reserved for future standard use* |
| 0 | 24–31 | *Reserved for custom use* |
| 0 | 32–47 | *Reserved for future standard use* |
| 0 | 48–63 | *Reserved for custom use* |
| 0 | ≥64 | *Reserved for future standard use* |

Table 2.2: mcause values after trap

### 2.3.5 mtval


Figure 2.16: mtval Register

The mtval register is an 32-bit read-write register formatted as shown in Figure 2.16. When a trap is taken into machine mode, mtval is either set

to zero or written with exception-specific information to assist software in handling the trap. Otherwise, mtval is never written by the implementation,though it may be explicitly written by software. The hardware platform will specify which exceptions must set mtval informatively and which may unconditionally set it to zero.

### 2.3.6 mscratch


Figure 2.17: mscratch Register

The mscratch register is an 32-bit read/write register dedicated for use by machine mode. Typically, it is used to hold a pointer to a machine-mode hart-local context space and swapped with a user register upon entry to an machine mode trap handler.

### 2.3.7 Trap Return Instruction

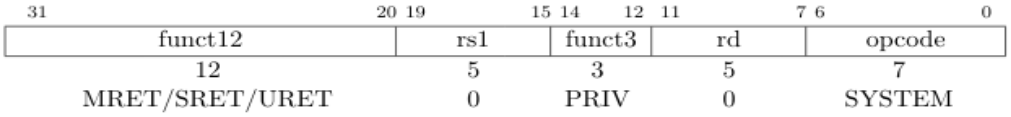


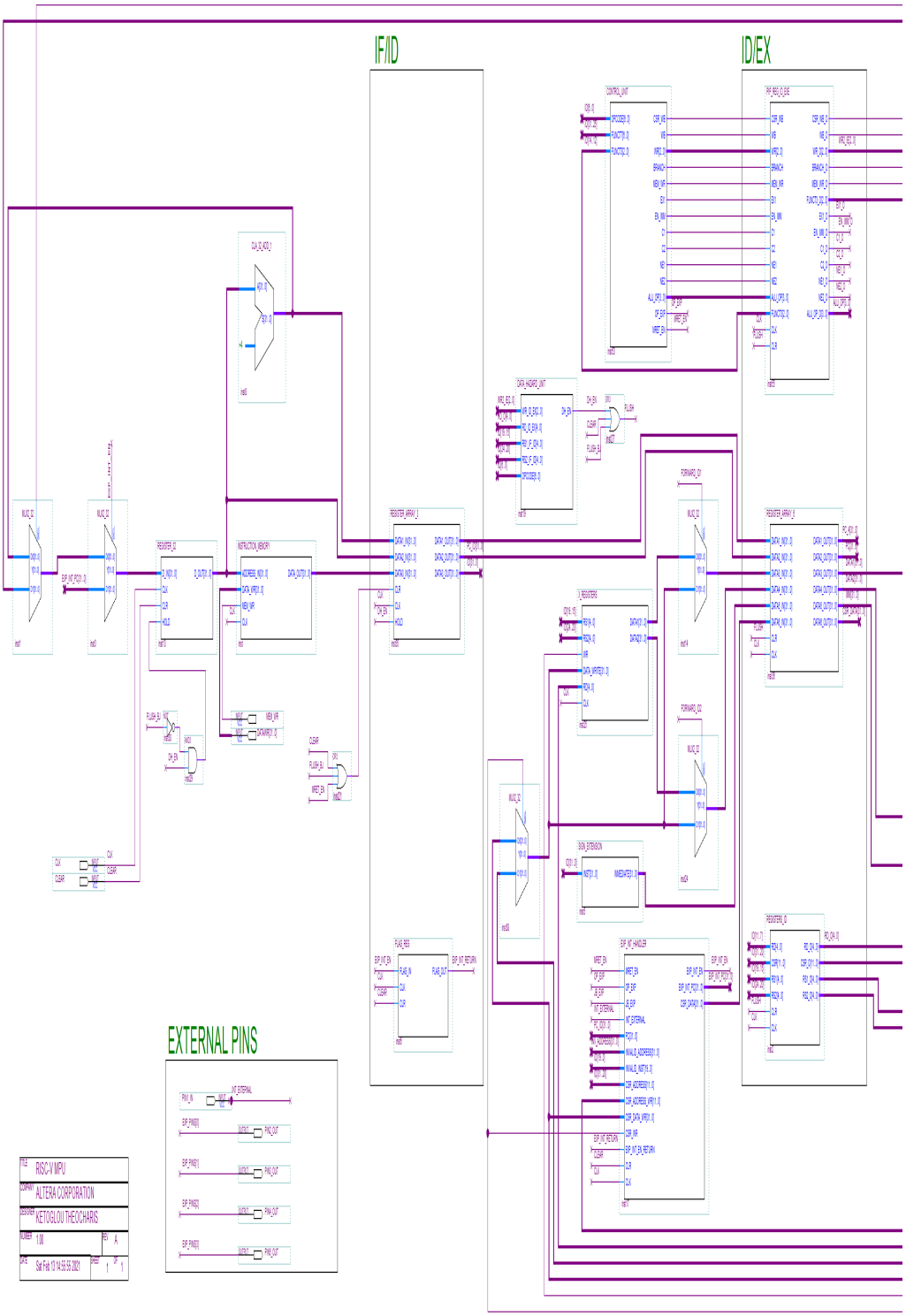

Figure 2.18: MRET Instruction

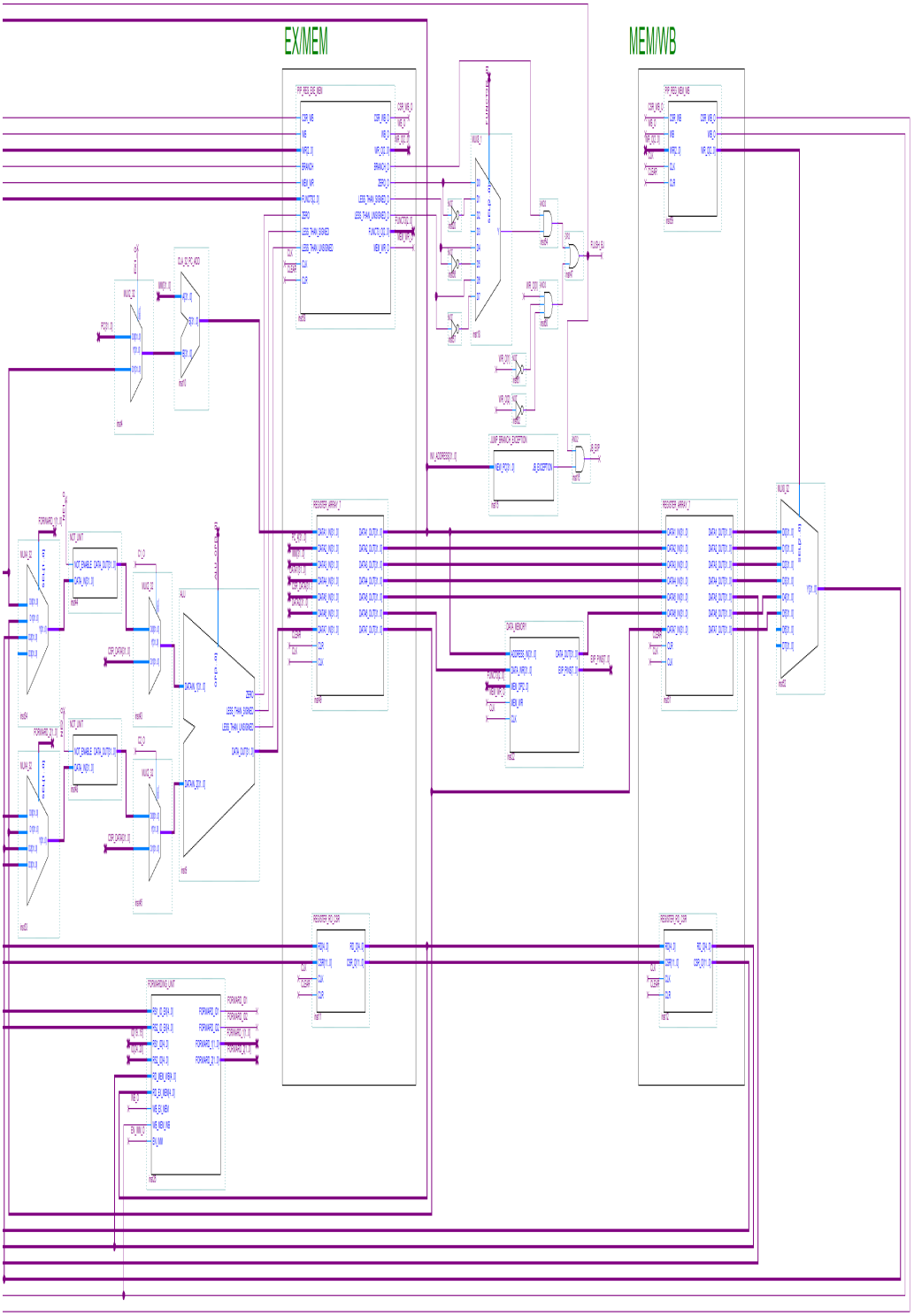
To return after handling a trap, there are separate trap return instructions per privilege level:MRET, SRET, and URET. MRET is always provided. SRET and URET are not implemented because no privileged levels exist on this microprocessor. When a MRET instruction is executed then the hardware transfer the control back to the main code.

# Chapter 3

# Implementation of the microprocessor

In this diploma thesis a RISC-V based microprocessor was created in VHDL. VHDL is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits.
The microprocessor is pipeline based and the full RV32E ISA was implemented except instructions ECALL, EBREAK because this microprocessor is not intended to work with operating systems and FENCE instruction because is single core processor with no hardware threads. Also registers mstatus(only MIE bit), mtvec, mepc, mcase, mtval, mscratch and instruction MRET from the Machine level ISA were implemented so exceptions and interrupts can work.
Due to the above the Zicsr extension was implemented so the registers can be written/read.
The implemented exceptions are wrong opcode and address out of bounds for jump-branch instructions. Arithmetic exceptions do not occur. Exceptions for loads,stores instructions for out of bounds addresses does not occur because is hardwired to AND mask. Lastly an external interrupt pin was created.
In the next Figure full schematic of the microprocessor is presented and in the following sections all of the components are presented one by one.


IF/ID
ID/EX
EXTERNAL PINS
TITLE: RISC-V MPU
COMPANY: ALTERA CORPORATION
DESIGNER: KETOGLOU THEOCHARIS
DATE: Sat Feb 13 14:55:59 2021


EX/MEM
MEM/WB

## 3.1 Instruction Fetch

The first stage of the microprocessor is the Instruction Fetch (IF) stage. At this stage the appropriate command is selected so it can be forwarded to the next stage. At the Figure 3.1 we can see all the components that from this stage.

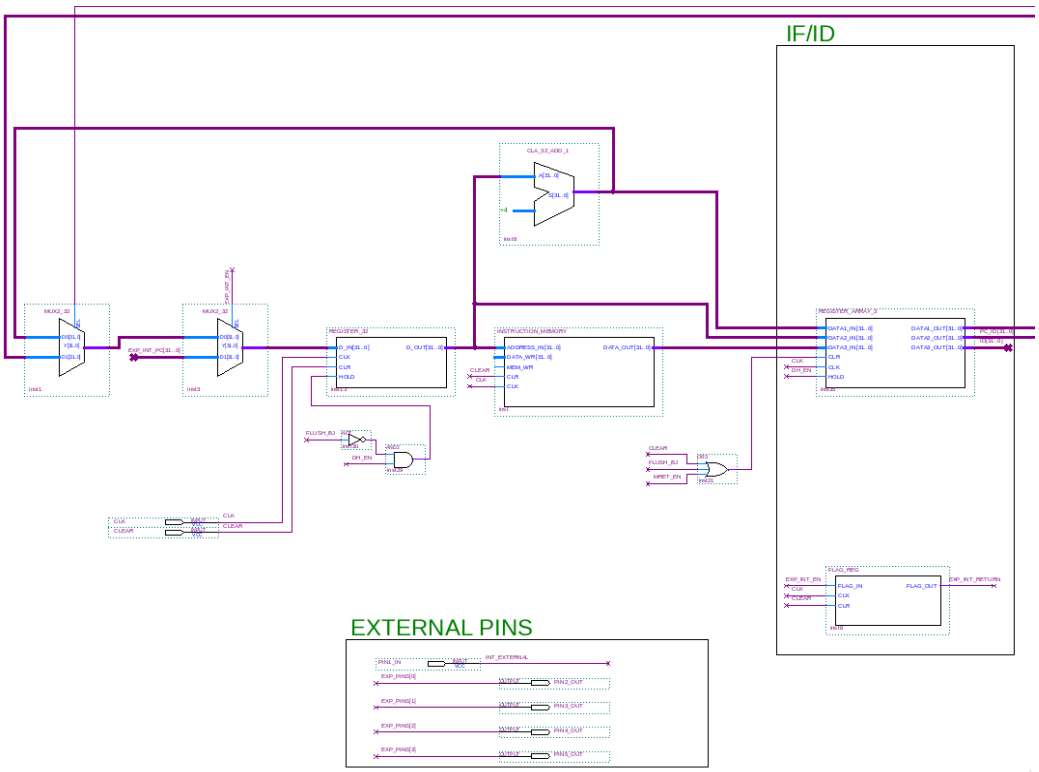



Figure 3.1: Instruction Fetch (IF) stage

### 3.1.1 Instruction Memory

The **Instruction Memory** contains all the instructions (program) that the microprocessor will execute. The **ADDRESS_IN** of this component accepts the address of the next instruction and output the instruction at outputs **DATA_OUT**. When **MEM_WR** is set, at the next clock cycle the data at **DATA_WR** will be written to the memory in address defined by **ADDRESS_IN**. To clear this memory **CLR** must be set and one clock cycle must pass. The clock is provided from **CLK**.

### 3.1.2 Multiplexers and Fixed CLA Adder

In this stage there are two multiplexers that together play the role of the selection of the next address for the Instruction Memory and one fixed **carry-lookahead adder** (CLA,inst8) that adds +4 to the present instruction address and produce the next instruction address. First multiplexer (inst1) selects between the address produced from fixed CLA and from an address that comes from MEM stage that represent an address that is produced from a conditional or jump instruction. Second multiplexer (inst3) is used only when an exception-interrupt occur and can choose the address that coming from the first multiplexer or from the interrupt address that comes from Exception-Interrupt handler(more in ID section).

### 3.1.3 Registers and Signals

Three are the main registers at this stage, the register labeled as inst13 is the **PC (Program Counter) register**, that means that it holds the next instruction address. The remaining two registers are part of the pipeline architecture, so register inst35 holds the PC, PC+4 and the instruction while inst6 register holds the exception-interrupt signal. Register inst35 except from **CLEAR** signal it can be cleared from **FLUSH_BJ** (when branch or jump is taken) or by **MRET_EN** (MRET instruction executed). The PC register (inst13) holds its data when data hazard unit is enabled (**DH_EN**) and no branch or jump (FLUSH_BJ) has been taken.

### 3.1.4 I/O Pins

The Figure 3.1 shows the external pins of this microprocessor, four output pins marked as **PINx_OUT** and three inputs marked as **CLK**, **CLEAR** and **PIN1_IN**. The four PINx_OUT pins are output pins and they are connected to the Data Memory of MEM stage and show whatever is written in a specific address of the Data Memory(more in Data Memory section). The PIN1_IN is an input pin and used as external interrupt. The pin CLK used as main clock for the microprocessor while the CLEAR pin used to clear all the critical components.

## 3.2 Instruction Decode

The second stage of the microprocessor is the Instruction Decode (ID) stage. At this stage the instruction is decoded so it can be forwarded to the next stage, also Control Unit, Data Hazard Unit and Exception-Interrupt Handler

will be explained in this section although they are independent. At the Figure 3.2 we can see all the components that form this stage.

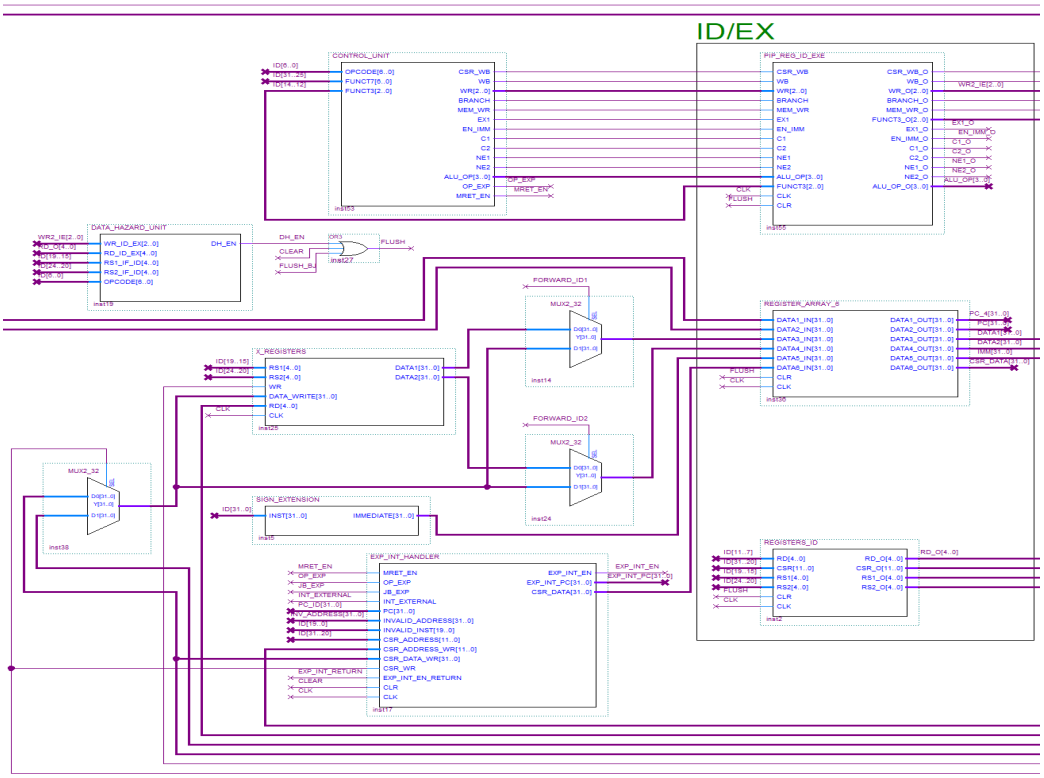



Figure 3.2: Instruction Decode (ID) stage

### 3.2.1 X Registers and Sign Extension

The **X Registers** module contains a set of 16 registers as defined by RV32E. Except register x0(address 0x00000000) which is hardwired to ground all the other registers can be written. Every time an instruction is at the ID stage the X registers module decodes automatically from the instruction the **RS1** and **RS2** address and output their data to **DATA1** and **DATA2** respectively regardless if their data needed by the instruction. When **WR** is set then the data at **DATA__WRITE** is written to the register with address that is taken from **RD**. The module is synchronous with clock input from **CLK**.
The **Sign Extension** module takes the whole instruction from **INST** and decodes it fast to find its type (I,S,U,J,B,CSR). The fast decoding is based on the bits of the opcode part of the instruction, as we can see at Table 3.1 the bits 6,5,3,2 are critical for that operation and in Figure 3.3 we can see the logic that decodes the instructions. Depending on what instruction type

it finds the module produces the correct extended immediate and outputs it to **IMMEDIATE**.

| Type | Bits(6,5) | Bits(3,2) |
|---|---|---|
| **I** | 11 | 01 |
| **I** | 00 | 00 |
| **S** | 01 | 00 |
| **B** | 11 | 00 |
| **U** | 01 | 01 |
| **U** | 00 | 01 |
| **J** | 11 | 11 |
| **CSR** | 11 | 00 |

Table 3.1: Sign Extension Fast Decode Truth Table

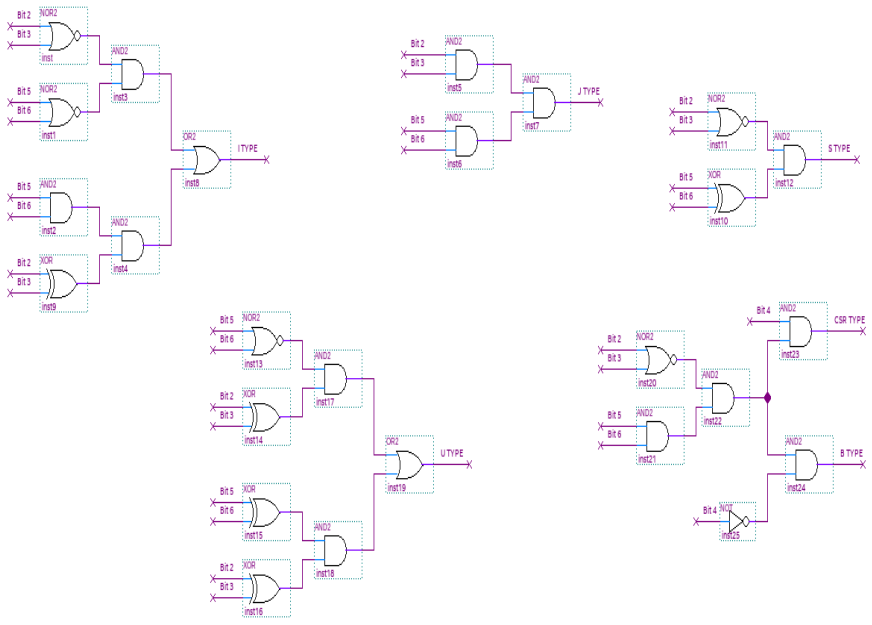


Figure 3.3: Sign Extension Fast Decode Logic

### 3.2.2 Multiplexer, Registers and Signals

The only multiplexer (inst38) at this stage is present to select which data will be written at the X registers module (RD). The selection is coming from the Write Back stage and it is selected between CSR data or other data that will be explained in Write Back stage. The remaining registers are part of the pipeline architecture, so register inst55 holds all the signals from the Control Unit, register inst36 holds the data from PC, PC+4, RS1, RS2 and Sign Extended Immediate, last the register inst2 holds RD, CSR addresses to forward them until they reach the Write Back stage and RS1, RS2 addresses to be used for comparison in Forwarding Unit (more at Instruction Execute section). All of these registers in this stage are cleared from **FLUSH** signal, the FLUSH signal can be produced either from Data Hazard Unit, Branch-Jump Flush or General CLEAR (OR gate inst27).

### 3.2.3 Control Unit

The Control Unit uses as inputs the instruction **OPCODE**, **FUNCT7** and **FUNCT3** to decode the instruction. The function of this module is to produce the right signals that the present command needs and forward them to the next stage. The output signals of this module can be found in Table 3.2.

| Output Signal | Operation |
|---|---|
| **CSR_WB** (CSR Write Back) | This signal is used for the selection of CSR data to be written in RD from multiplexer inst38 and data from Write Back stage to be written in the selected CSR. Instructions that set this signal: MRET, CSRRW, CSRRWI, CSRRS, CSRRSI, CSRRC, CSRRCI. |
| **WB** (Write Back) | This is a basic signal used in Write Back stage to select which data will be written to X registers module. Instructions that set this signal: All except BEQ, BNE, BLT, BGE, BLTU, BGEU, SB, SH, SW. |
| **WR[2..0]** (Write) | This is a basic signal which selects from the multiplexer at Write Back stage which are the data that will be written back. Also it is used in a logic circuit at Memory stage to find out if we have a Branch-Jump instruction. Instructions that set this signal: All except AUIPC, BEQ, BNE, BLT, BGE, BLTU, BGEU, SB, SH, SW. |
| **BRANCH** | This signal it is used in a logic circuit at Memory stage to enable Branch and if the Branch is taken then the FLUSH_BJ signal is set. Instructions that set this signal: BEQ, BNE, BLT, BGE, BLTU, BGEU. |
| **MEM_WR** (Memory Write) | This is a basic signal which selects if the Memory at Memory stage will be written. Instructions that set this signal: SB, SH, SW. |
| **EX1** (Enable Multiplexer 1) | This signal selects from the multiplexer inst4 at Execution stage between PC or RS1 data. Instructions that set this signal: JALR, LB, LH, LW, LBU, LHU, SB, SH, SW. |
| **EN_IMM** (Enable Immediate) | This signal is used from the Forwarding Unit so it can know that the immediate must forwarded to ALU and select appropriately the signal FORWARD_2. Instructions that set this signal: ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI, CSRRRSI, CSRRRCI. |
| **C1** (CSR 1 DATA) | This signal is used to select from multiplexer inst43 between CSR data and other data. Instructions that set this signal: CSRRRSI, CSRRRCI. |
| **C2** (CSR 2 DATA) | This signal is used to select from multiplexer inst45 between CSR data and other data. Instructions that set this signal: CSRRS, CSRRC. |
| **NE1** (Not Enable 1) | This signal applies NOT logic to the data that comes from multiplexer inst34. Instructions that set this signal: CSRRC. |
| **NE2** (Not Enable 2) | This signal applies NOT logic to the data that comes from multiplexer inst33. Instructions that set this signal: CSRRRCI. |
| **ALU_OP** (ALU Operation) | This is a basic signal and selects what operation will the ALU execute next with the data. Instructions that set this signal: All except LUI, AUIPC, JAL, JALR, LB, LH, LW, LBU, LHU, SB, SH, SW, MRET, CSRRW. |
| **OP_EXP** (Opcode Exception) | This signal is used to notify Exception-Interrupt Handler that the instruction opcode does not exist in order that the Handler will generate an exception. |
| **MRET_EN** (MRET Enable) | This signal is used to notify Exception-Interrupt Handler that the present instruction is the MRET. |

Table 3.2: Control Unit Signals Operation

### 3.2.4 Data Hazard Unit

The Data Hazard Unit function is to find data which are needed from data memory but have not been saved to an architectural register (X) yet. For example, when an instruction wants to save data to a register (e.g. x1) and the

same register (x1) is needed from the next instruction then a hazard occurs because this instruction will get the old data from the register (x1). Data Hazard Unit has five inputs the **WR_ID_EX[2..0]** signal from Execution stage that will inform the Unit if the previous instruction will write to a register, the **RD_ID_EX[4..0]** from Execution stage and **RS1_IF_ID[4..0]**, **RS2_IF_ID[4..0]** from Decode stage, these three signals will be compared and the Unit will find out if a match occurs. If a match occurs then if **OPCODE[6..0]** signal is from an instruction that will write to the RD register, then we have a hazard. In that case **DH_EN** signal is set and the instruction will be hold for one or two clock cycles depending on the hazard.

### 3.2.5 Exception-Interrupt Handler

The Exception-Interrupt Handler is responsible for processing all exceptions and interrupts. It also contains the CSRs registers described at Machine Level Registers section. The Exception-Interrupt Handler has as outputs the **EXP_INT_EN** signal, **EXP_INT_PC[31..0]** address and **CSR_DATA[31..0]**. The function of this module is to output CSR data when needed and force the PC to jump to another address when an exception or interrupt occurs. The input signals of this module can be found at the Table 3.3.

| Input Signal | Operation |
|---|---|
| **MRET_EN** (MRET Enable) | When this signal is set then a MRET instruction is present. The Handler Enable the global exceptions/interrupts so new exceptions/interrupts can occur and output to EXP_INT_PC the address of the instruction that has been interrupted. Also it sets EXP_INT_EN to make this address the next one in IF stage. |
| **OP_EXP** (Opcode Exception) | This input signal (exception) is set when the Opcode of an instruction does not exist. |
| **JB_EXP** (Jump/Branch Exception) | This input signal (exception) is set when the address from a Jump/Branch instruction is out of bounds. |
| **INT_EXTERNAL** (External Interrupt) | This input signal (interrupt) is set when the external PIN1_IN pin is set. |
| **PC[31..0]** (Program Counter) | The current program counter to be saved and used later. |
| **INVALID_ADDRESS** (Invalid Address) | The out of bounds address to be saved when JB_EXP signal is set. |
| **INVALID_INST** (Invalid Instruction) | The invalid instruction to be saved when OP_EXP signal is set. |
| **CSR_ADDRESS** | Address of the CSR to be read. |
| **CSR_ADDRESS_WR** (CSR Address Write) | Address of the CSR to be written. |
| **CSR_DATA_WR** (CSR Data Write) | Data that will written to the CSR. |
| **CSR_WR** (CSR Write) | This signal enables CSR write. |
| **EXP_INT_EN_RETURN** (Exception Interrupt Enable Return) | When EXP_INT_EN signal returns from IF stage then clears it so normal operation can continue. |
| **CLR** (Clear) | This signal is used to reset the Handler. |
| **CLK** (Clock) | The module clock. |

Table 3.3: Exception-Interrupt Handler Signals Operation

## 3.3 Instruction Execute

The third stage of the microprocessor is the Instruction Execute (EXE) stage. At this stage the instruction data are processed by ALU or by the external CLA adder. Forwarding Unit is in this stage and the function it has is to forward data when they are needed by an instruction but are not written from a previous one. At the Figure 3.4 we can see all the components that from this stage.

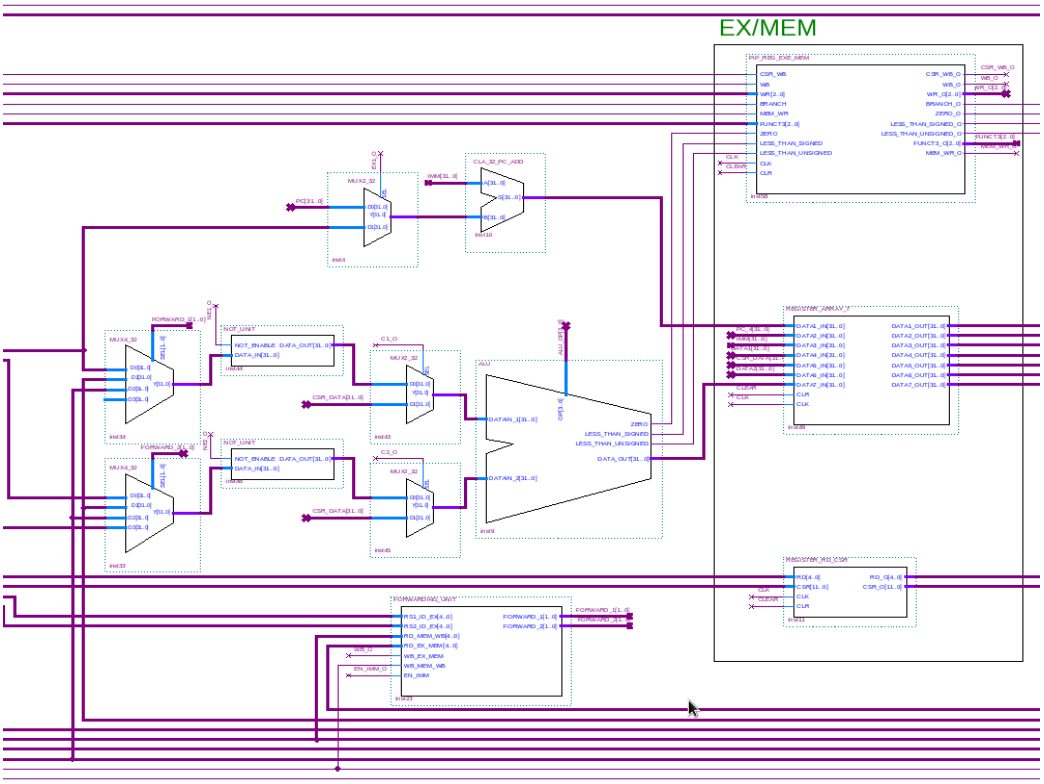



Figure 3.4: Instruction Execute (EXE) stage

### 3.3.1 Multiplexer and CLA Adder

These two components are used to compute the target address for branch and unconditional jump instructions that need the ALU to make another computation. The first multiplexer (inst4) that will be explained in this stage is the one that drives the data to CLA Adder. This multiplexer is controlled from EX1 signal that is generated from Control Unit in the previous stage, EX1 chooses between PC and RS1 data depending on the instruction and forwards the data to CLA Adder. Next the CLA Adder adds the data from the multiplexer with the immediate data and produces the right data. Regardless if the data is needed or if the data is correct the addition in the CLA Adder will be done every time and will be ignored if it is not needed. Branch instructions (BEQ, BNE, BLT, BLTU, BGE, BGEU) and unconditional jump instructions (JAL, JALR) make use of these components, branch instructions need the ALU to make the comparison so it cannot compute the target address at the same time and unconditional jump instructions need ALU to compute the PC + 4 result that will be saved in rd.

### 3.3.2 Multiplexers and Not Units

Four multiplexers are responsible for the data that the ALU will receive. The first two multiplexers (inst33, inst34) are controlled from Forwarding Unit via the signals FORWARD_1[1..0] and FORWARD_2[1..0] respectively. The multiplexer with inst34 label chooses between RS1 data, data coming from MEM stage which have not yet been saved or data coming from WB stage which have not yet been saved. The Forwarding Unit is responsible for the right choice of data. Similarly the multiplexer with inst33 label receives the same data inputs except for the RS1 data which is replaced by the RS2 data, this multiplexer gets another one input which is the sign extended immediate. The data from the two multiplexers ends up at two Not Units, the Not Units are used only from specific CSR instructions and invert the data, generally the data will proceed without inversion. Lastly the data will end up in another two multiplexer which are controlled from C1_O and C2_O signals respectively. These two multiplexers will make the final choice between the data forwarded from Not Units or the CSR data which are selected from specific CSR instructions, finally the data will arrive in ALU and will proceeded accordingly.

### 3.3.3 Arithmetic Logical Unit (ALU)

It can be said that Arithmetic Logical Unit is the most basic part of every processor and this can not be doubted. ALU in short is a critical module of the processor because it is the unit that calculates the result of the data. ALU can make arithmetic operations such addition, subtraction and logical operations like AND, OR, XOR. The ALU module is a set of other modules that will be explained in order in that section. For addition and subtraction a **Carry-Lookahead Adder** module is used, for left/right logical/arithmetic shift a **Barrel Shifter** module is used, **AND, OR, XOR** are calculated every time from the appropriate gates and finally the **ZERO**, **LESS_THAN_SIGNED** and **LESS_THAN_UNSIGNED** singals that they are used for branches. At Figure 3.5 the ALU module is presented.

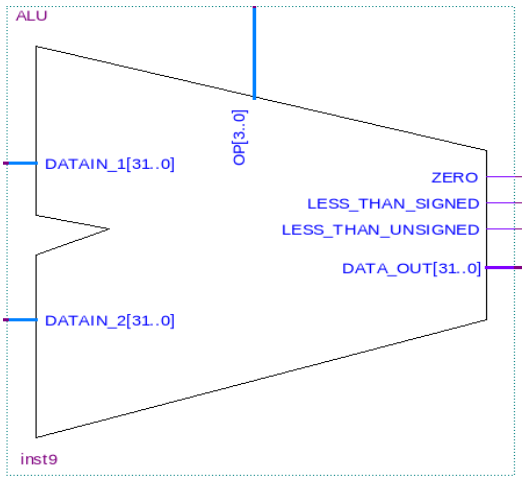



Figure 3.5: Arithmetic Logical Unit (ALU)

### Carry-Lookahead Adder (CLA)

A Carry-Lookahead Adder improves speed by reducing the amount of time required to determine carry bits. A common adder contains a chain of full adders that the carry out of one adder connects to the carry in of the next adder, this increases the time that the circuit need to calculate the result because the last full adder must wait all the previous ones to calculate their carries. A Carry-Lookahead Adder computes the carry in of every full adder so every full adder can work independently. A 4-bit CLA is presented in the Figure 3.6 as an example. The logic of CLA uses the concepts of generating and propagating carries. In the case of binary addition, $A + B$ generates if and only if both $A$ and $B$ are 1. If we write $G(A, B)$ to represent the binary predicate that is true if and only if $A+B$ generates, we have $G(A, B) = A \cdot B$ where $A \cdot B$ is logical $AND$ between $A$ and $B$. In the other hand $A + B$ propagates if and only if at least one of $A$ or $B$ is 1. If $P(A, B)$ is written to represent the binary predicate that is true if and only if $A + B$ propagates, one has $P(A, B) = A \oplus B$ where $A \oplus B$ is logical $XOR$ between $A$ and $B$. Given these concepts of generate and propagate, a bit of addition carries precisely when either the addition generates or the next less significant bit carries and the addition propagates. Written in boolean algebra, with $C_i$ the carry bit of bit i, and $P_i$ and $G_i$ the propagate and generate bits of bit i respectively, $C_{i+1} = G_i + (P_i \cdot C_i)$. For addition $C(0) = 0$ is initialized and for subtraction two's complement logic is used so $C(0) = 1$ is initialized and the second binary number is inverted $B \Rightarrow \neg B$. Two's complement overflow detection can be achieved by XOR the carry-in and the carry-out bits of the leftmost full adder.

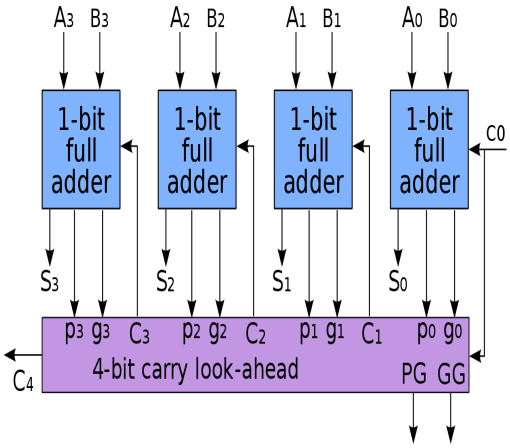



Figure 3.6: 4-bit Carry-Lookahead Adder (CLA)

### Barrel Shifter

A Barrel Shifter is a digital circuit that can shift a data word by a specified number of bits without the use of any sequential logic, only pure combinational logic. The way to implement a barrel shifter is as a sequence of multiplexers where the output of one multiplexer is connected to the input of the next multiplexer in a way that depends on the shift distance. The number of multiplexers required for an n-bit word is $nlog_2n$, so for this 32-bit ALU $32 \times log_2 32 = 32 \times 5 = 160$ multiplexers needed. The Table 3.4 represents the operation table of the Barrel Shifter

| Inputs | Operation |
|---|---|
| 00 | Shift Right Logical |
| 01 | Shift Left Logical |
| 10 | Shift Right Arithmetic |
| 11 | Shift Left Arithmetic |

Table 3.4: Barrel Shifter Truth Table

Barrel Shifter schematic is presented in Figure 3.7. Because of the 32-bit architecture the words can be shifted 32 slots left or right, according to that the Barrel Shifter needs to know the shift amount (Shamt = $S_4S_3S_2S_1S_0$) which is 5 bits ($2^5 = 32$) .The operation is selected from $OP[1..0]$ according to Table 3.4. In the Figure 3.7 are distinguished 5 stages, every stage shift by an amount of slots the word and is controlled for the appropriate Shamt

bit, the Table 3.5 presents every stage and the shift amount that this stage can make and also the control bit of this stage.

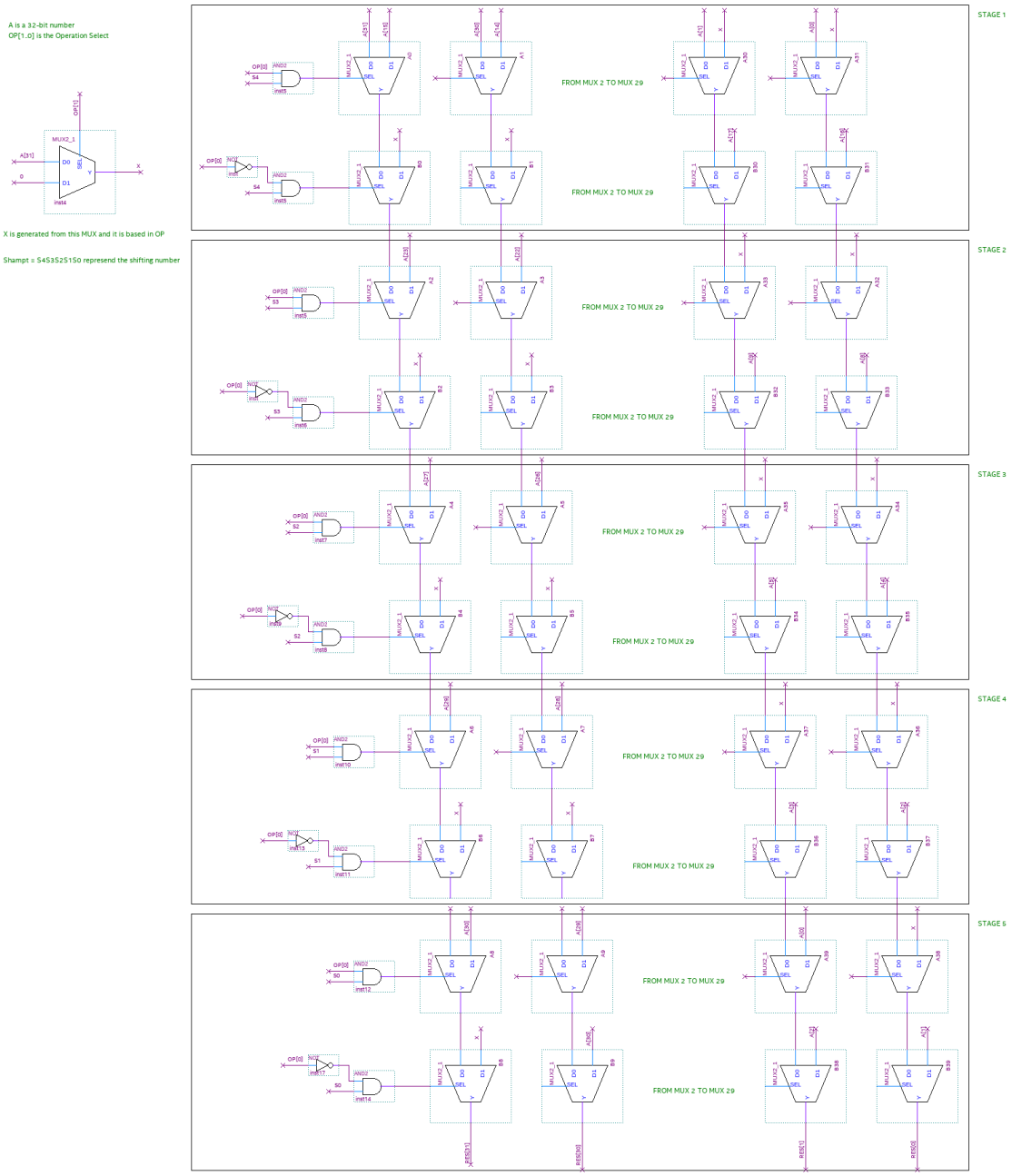



Figure 3.7: 32-bit Barrel Shifter

| Stage | Shift Amount | Control Bit |
|---|---|---|
| 1 | 16 | S4 |
| 2 | 8 | S3 |
| 3 | 4 | S2 |
| 4 | 2 | S1 |
| 5 | 1 | S0 |

Table 3.5: Barrel Shifter Stages

### ZERO,LESS_THAN_SIGNED and LESS_THAN_UNSIGNED signals

The signals ZERO, LESS_THAN_SIGNED and LESS_THAN_UNSIGNED are used for branch purposes mainly. A branch is basically a comparison between two numbers, so a subtraction between these two numbers is selected to computed from the ALU. When these numbers are equal the ZERO signal is set otherwise it is cleared.
The Figure 3.8 shows the logic for LESS_THAN_SIGNED signal if the branch is between signed numbers. From that Figure it is understood that if the numbers A, B have different signs then $R[31]$ is selected in the MUX which is the sign of the result, so if $R[31] = 0$ it means that $A > B$ and LESS_THAN_SIGNED signal is cleared otherwise if $R[31] = 1$ then $A < B$ and LESS_THAN_SIGNED signal is set. If the numbers A, B have same signs then $A[31]$ is selected in the MUX which means that if $A[31] = 0$ then $B[31] = 1$ so $A > B$ and LESS_THAN_SIGNED signal is cleared otherwise if $A[31] = 1$ then $B[31] = 0$ so $A < B$ and LESS_THAN_SIGNED signal is set.
If the branch is between unsigned numbers then LESS_THAN_UNSIGNED signal is used. To determine the value of that signal we take into account the Carry Out bit from the subtraction that has been made in CLA. If $C_{out} = 0$ then $(A - B) < 0 \Rightarrow A < B$ and the LESS_THAN_UNSIGNED signal is set otherwise if $C_{out} = 1$ then $(A - B) > 0 \Rightarrow A > B$ and the LESS_THAN_UNSIGNED signal is cleared.

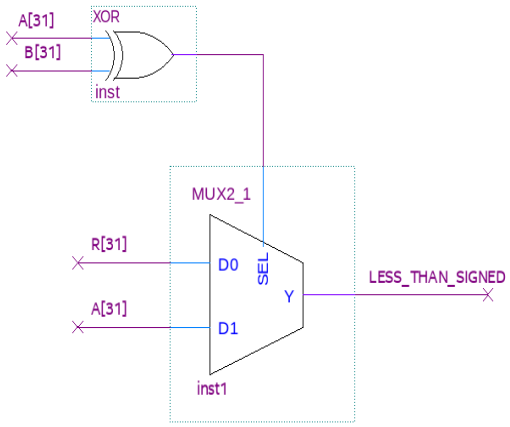



Figure 3.8: Less Than Signed Module

### 3.3.4 Forwarding Unit and Registers

The Forwarding Unit is responsible for the right selection of data for the first two multiplexers that drive data to ALU. The function of this Unit is to check if the address of RD from MEM stage is the same with the addresses of RS1, RS2 from EXE stage or if the address of RD from WB stage is the same with addresses of RS1, RS2 from EXE stage. If a match occurs and the RS1 or RS2 data are needed then it forwards the data from the MEM stage and/or from the WB stage. If RD has the same address in MEM and WB stage then priority is given to the first. If RD address is zero then no forwarding is needed because the zero register is hardwired to ground. If a register in ID stage must be read but the new data from WB stage will be written to the same register at the next clock pulse then in that case the Forwarding Unit selects the data for the EXE stage from the multiplexers in the ID stage.
At the end of this stage we find the pipeline registers. The register with label inst58 holds all the signals that need to be forwarded to the next stages, register with label inst49 holds the various data that need to be forwarded to the next stages and the register with label inst11 hold RD and CSR addresses to forward them at the next stages until (and if) they needed for the data to be written back to the appropriate register.

## 3.4 Data Memory

The fourth stage of the microprocessor is the Data Memory (MEM) stage. At this stage the program data can be written to the memory and read after. A branch-jump resolver unit can be found at this stage to determine if branch is taken or if jump instruction is present alongside with a branch-jump exception generator in case the address is out of bounds. At the Figure 3.9 we can see all the components that from this stage.

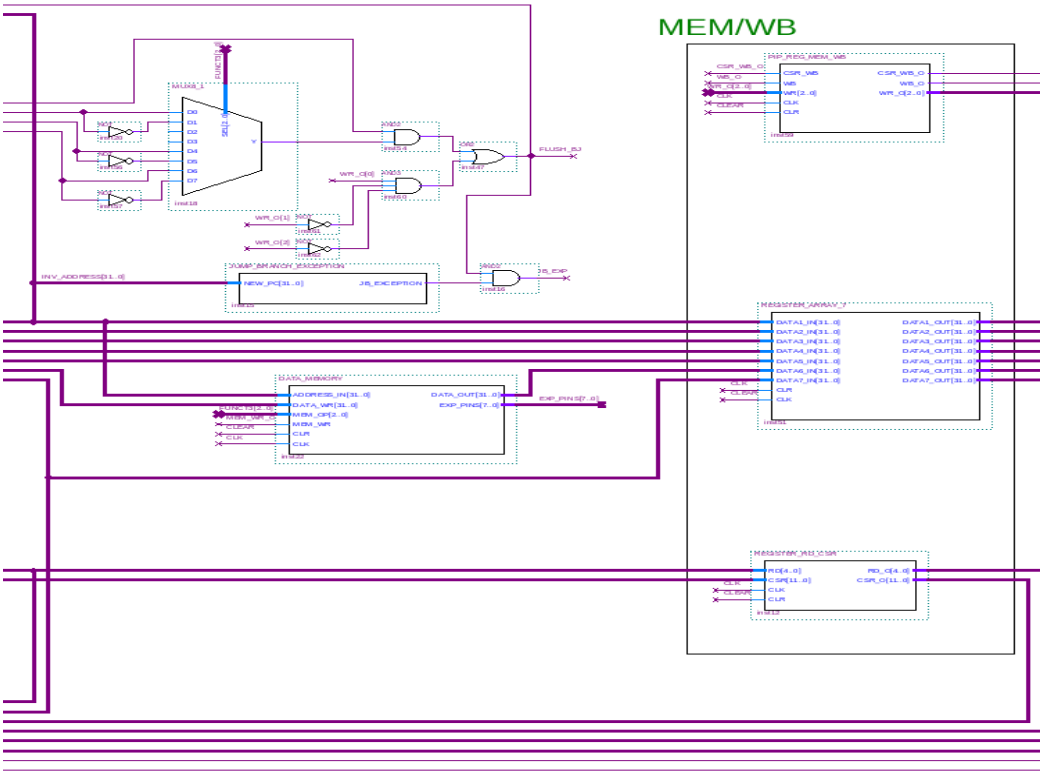



Figure 3.9: Data Memory (MEM) stage

### 3.4.1 Branch-Jump Signal/Exception

The Branch-Jump Signal/Exception is a logic unit that resolves if a branch is taken when a branch instruction is present and if a jump instruction is present. The Figure 3.10 shows this unit. First the branch part of the logic will be explained.
If a branch instruction fetched then the Control Unit set the BRANCH signal and output the FUCNT3[2..0] signal which determines what kind of branch the instruction is. In Table 3.6 is presented how the MUX output the final

result for the branch instruction. If MUX output 1 then the branch is taken and because BRANCH signal is set the AND gate (inst54) output 1, next the OR gate (inst47) output 1 as well so **FLUSH_BJ** signal is set. FLUSH_BJ signal forces the pipeline registers in IF and ID stages to be cleared so the next instruction will be the branched one.
For a jump instruction (JAL,JALR) the Control Unit generate $WR = 001$ signal, in such case the AND gate (inst60) output 1, next the OR gate (inst47) output 1 as well so **FLUSH_BJ** signal is set. Like branch part FLUSH_BJ signal forces the pipeline registers in IF and ID stages to be cleared so the next instruction will be the jump target.
In case of out of bounds address for a jump/branch instruction the **JUMP_BRANCH_EXCEPTION** module generates a signal and if the FLUSH_BJ is set to (branch taken/jump occur) then the AND gate (inst16) is output 1. In such occasion the **JB_EXP** signal is set and an exception occurs that makes EXP_INT_HANDLER from ID stage to stop the data forwarding and begin the appropriate actions.

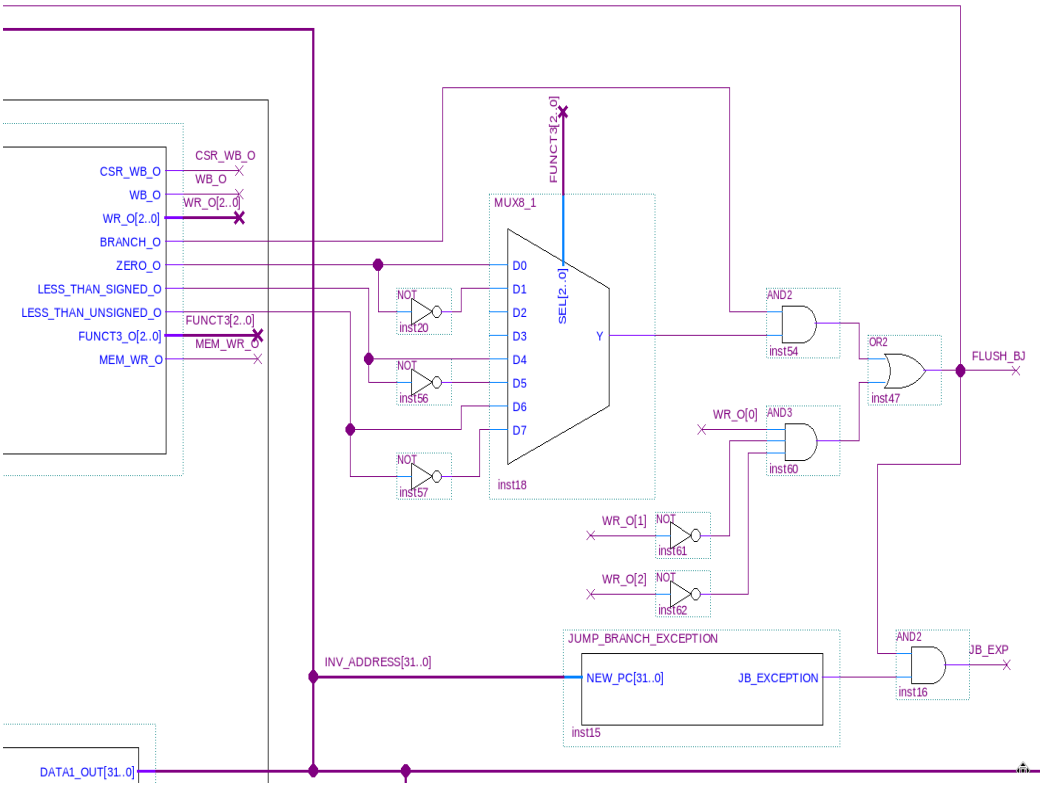



Figure 3.10: Branch-Jump Signal/Exception

| MUX Input | Signal (from ALU) | Explanation |
|---|---|---|
| 000 | Zero | For BEQ instruction, if Zero is set then the two numbers are equal. |
| 001 | Inverted Zero | For BNE instruction, if Zero is not set then the two numbers are not equal. |
| 010 | - | - |
| 011 | - | - |
| 100 | Less Than Signed | For BLT instruction, when LESS_THAN_SIGNED is set then the first number is less than the second. |
| 101 | Inverted Less Than Signed | For BGE instruction, when LESS_THAN_SIGNED is not set then the first number is greater than the second. |
| 110 | Less Than Unsigned | For BLTU instruction, when LESS_THAN_UNSIGNED is set then the first number is less than the second. |
| 111 | Inverted Less Than Unsigned | For BGEU instruction, when LESS_THAN_UNSIGNED is not set then the first number is greater than the second. |

Table 3.6: MUX Inputs for Branches

### 3.4.2 Data Memory and Registers

Data Memory, as its name suggests, holds the data of the program. These data can be written or read at anytime. Input **ADDRESS_IN[31..0]** takes the address in memory that must be read or write and no exception is generated if ADDRESS_IN is outside of bounds, but only the 6 LSB bits of ADDRESS_IN are being held due to an AND mask inside the memory. **DATA_WR** takes the data that will be written and the **MEM_WR** signal if it set it enables the writing process otherwise if MEM_WR is cleared no writing occurs but a reading process begins from the address declared from ADDRESS_IN. **MEM_OP** defines how bytes will be written/read in from the memory, in Table 3.7 is presented how bytes are being written in memory based on MEM_OP bits while in Table 3.8 is present how bytes are being read from the memory based on MEM_OP bits. The **CLK** signal is the clock for the module while the **CLR** signal clear the memory and clear

all its slots.
The last registers in the pipeline are the register with label inst59 which holds the signals from the Control Unit, the register with label inst51 which holds all the data which one of them will be selected (if it is needed) to the WB stage to be written back and the register with label inst12 which holds RD and CSR addresses.

| MEM_OP | Bytes Write |
|---|---|
| 000 | 1 (8 bits) |
| 001 | 2 (16 bits) |
| 010 | 4 (32 bits) |

Table 3.7: Data Memory Write Operations

| MEM_OP | Bytes Read | Sign Extension |
|---|---|---|
| 000 | 1 (8 bits) | SIGN |
| 001 | 2 (16 bits) | SIGN |
| 010 | 4 (32 bits) | NO |
| 100 | 1(8 bits) | ZERO |
| 101 | 2 (16 bits) | ZERO |
| 110,111 | 4 (32 bits) | ZERO |

Table 3.8: Data Memory Read Operations

## 3.5 Write Back

Fifth and final stage of the microprocessor. At this stage only one multiplexer exists that chooses which data will be written back. The signal **WR** which is forwarded from Control Unit at ID stage selects between data coming from CLA adder in EXE stage, PC+4 data, immediate data, RS1 data, CSR data, ALU data or from Data Memory. Figure 3.11 shows the multiplexer.

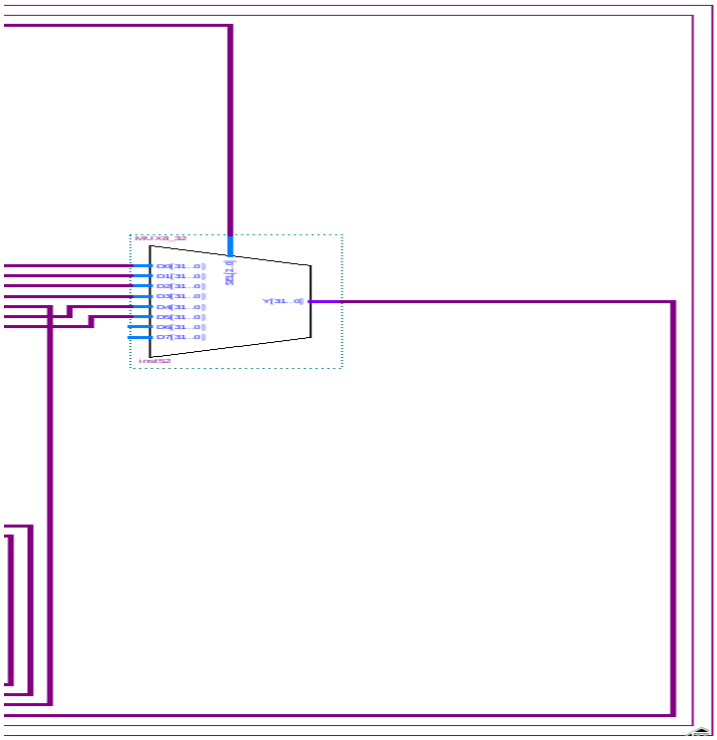


Figure 3.11: Write Back (WB) stage

# Chapter 4

# Evaluation of the Microprocessor

After the description of the RISC-V core it is time to test the design and evaluate the data. For evaluation purposes the Quartus Modelsim program was used which is a multi-language environment for simulation of hardware description languages such as VHDL. Modelsim produces waveform of the inputs and outputs of every component in the design, so a script is written which in the beginning is writes the data in the Instruction memory and after that the instructions that has been written are being executed.
For the evaluation of the CPU a set of tests was selected from the official RISC-V GitHub repository. This set of tests includes a test for every single instruction and every test contains mini tests that execute one function. Judging from the design of these tests the conclusion was that they where designed in such a way that they can guarantee that every possible hazardous scenario for every instruction has been accounted for and avoided. The instruction ECALL was removed from the tests because it was not implemented in the design. Most of these mini tests are loaded and evaluated in the design and the results where satisfactory. In this chapter a set of two test are presented by their waveform, every test is commented extensively on its section.
The instruction memory has been designed in such a way that it can be connected with another module (not designed in this diploma thesis) that its role will be to communicate with an external programmer via a serial protocol (only two pins needed clock and data) and load the instruction memory with the instructions at parallel. So loading of the memory has been made with a script to simulate such a scenario, this script runs from Quartus Modelsim, it loads the memory and resets the processor.
Evaluation of the tests is made by the waveform that has been produced by

Quartus Modelsim, combination of inputs and outputs are compared with the expected ones.
For every test we need 1 clock cycle (CC) for reset at the beginning, some CCs for writing the instructions to the instruction memory and 1 CC to make a reset and begin the execution of the instructions. After that 4 CCs are needed for the first instruction to execute and +1 CC for every next one. In this way we can approximately calculate how many cycles we need to simulate.

## 4.1 Test 1

The following mini test evaluates **addi** instruction. At the left side we can see the instruction in hexadecimal form and in the right side we can see the instruction in human readable form. The li (load immediate) instruction is a pseudo-instruction that can be translated to *addi rd, x0, imm*. The first instruction load 1 to the ra register, the second instruction makes an addition of ra register with 1, third and fourth instructions make a load to t4 and gp registers with 2 and 3 respectively and in fifth instruction a branch if not equal is executed between t5 and t4 registers.

Listing 4.1: Test 1 Instructions

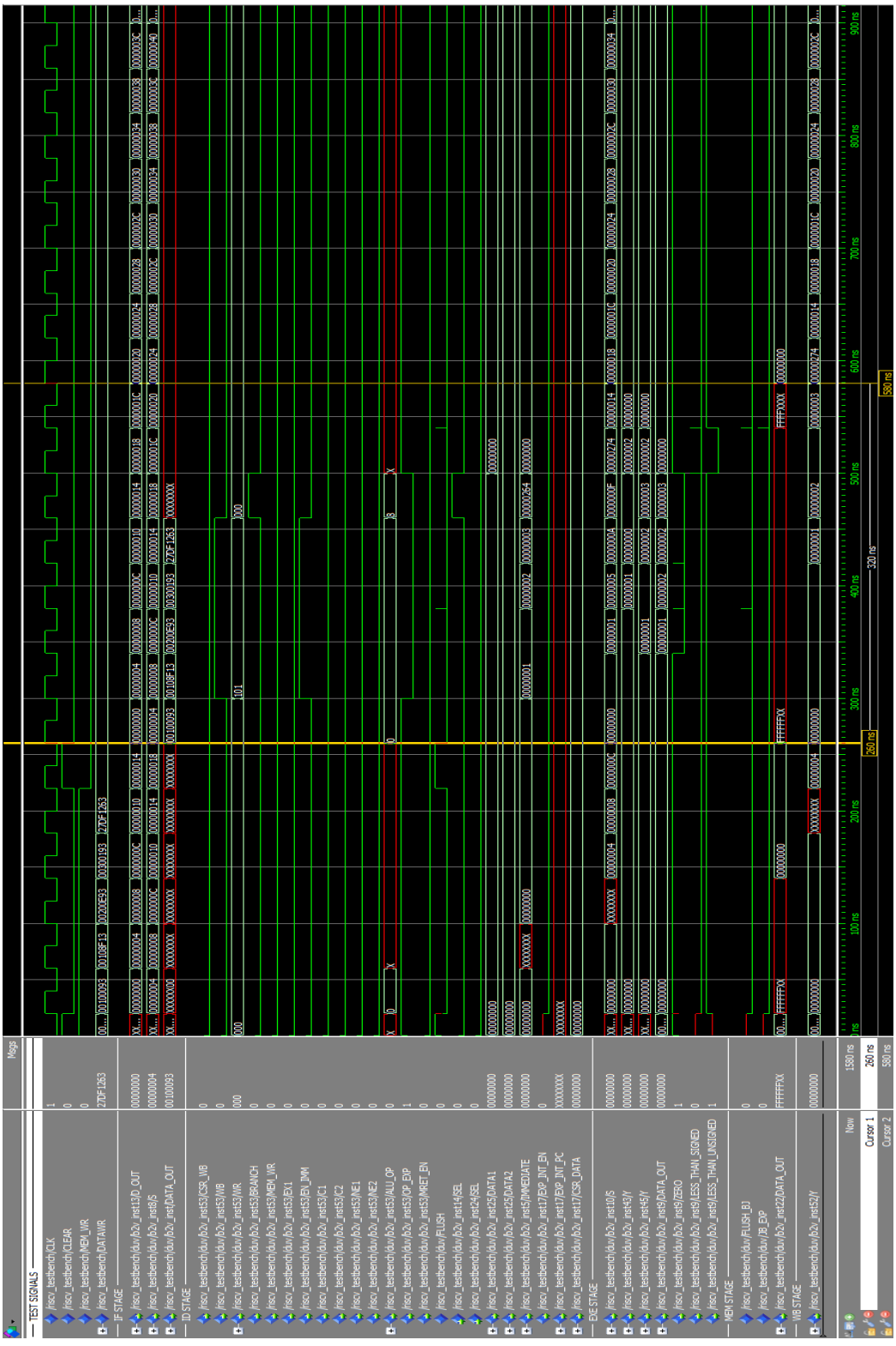
```
00100093        li      ra,1
00108f13        addi    t5,ra,1
00200e93        li      t4,2
00300193        li      gp,3
27df1263        bne     t5,t4,288 <fail>
```

The following Figure is the simulation waveform of this test. As we can see until the first yellow bar (260 ns) the mini test instructions are loaded at the Instruction memory, this is done by set the CLEAR pin at the first clock pulse so the program counter begin to count from the 0x00000000 address and set the MEM_WR pin of Instruction memory so new data can be written in by DATAWR pins. So from second pulse and after we can see the program counter (PC) to be increased by 4 every clock pulse and new data to be written for the next 5 clock pulses. After the data are written successfully the CLEAR pin is set again so the PC can count again from the first address (0x00000000) and the mini test can be executed. In the Table 4.1 we can see the signals we discussed above and their expected and real outputs. As we can see at the Figure all the signals from all the stages are the expected

ones, because we will need a lot of text to explain all the signal in every single clock we focus in some basics signals for this particular test. Specifically the first four instructions use the addi instruction so we will focus at the ALU output and for the fifth instruction we will focus at ZERO signal.

| **Clock Pulse** | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|
| **ALU Expected Output** | 1 | 2 | 2 | 3 | 0 |
| **ALU Real Output** | 1 | 2 | 2 | 3 | 0 |
| **ZERO Expected Output** | X | X | X | X | 1 |
| **ZERO Real Output** | X | X | X | X | 1 |

Table 4.1: Test 1 Signals



The first instruction is the **li ra,1** (addi ra,x0,1), as we can see from the Table 4.1 and from the Figure above we expect the x0(0) + 1 = 1 from the ALU output and the result to be saved to ra register. The second instruction is the **addi t5,ra,1**, from this instruction we expect ra(1) + 1 = 2 as a result to be saved to t5 register, the right use of ra means that the Forwarding Unit functions right. The third **li t4,2** and four **li gp,3** instructions are like the first, so we expect 2 and 3 as outputs from ALU. Finally the fifth instruction **bne t5,t4,288** is a branch if not equal, the address is out of bound for this design but we expect bne to fail, that happens because ALU makes a subtraction between t5 and t4 and the output is zero(0) so the ZERO signal is set and no branch is taken.

## 4.2 Test 2

The following mini test evaluates **beq** instruction. At the left side we can see the instruction in hexadecimal form and in the right side we can see the instruction in human readable form. The li (load immediate) instruction is a pseudo-instruction that can be translated to *addi rd, x0, imm*. The first instruction load 3 to the gp register, the second instruction load 1 to the ra register, the third instruction load 1 to the sp register,fourth instruction is a branch if equal and is executed between ra and sp registers, fifth instruction is a branch if not equal and is executed between zero and gp registers, sixth instruction is a branch if not equal and is executed between zero and gp registers, seventh instruction is a branch if equal and is executed between ra and sp registers and finally eighth instruction is a branch if not equal and is executed between zero and gp registers

Listing 4.2: Test 2 Instructions

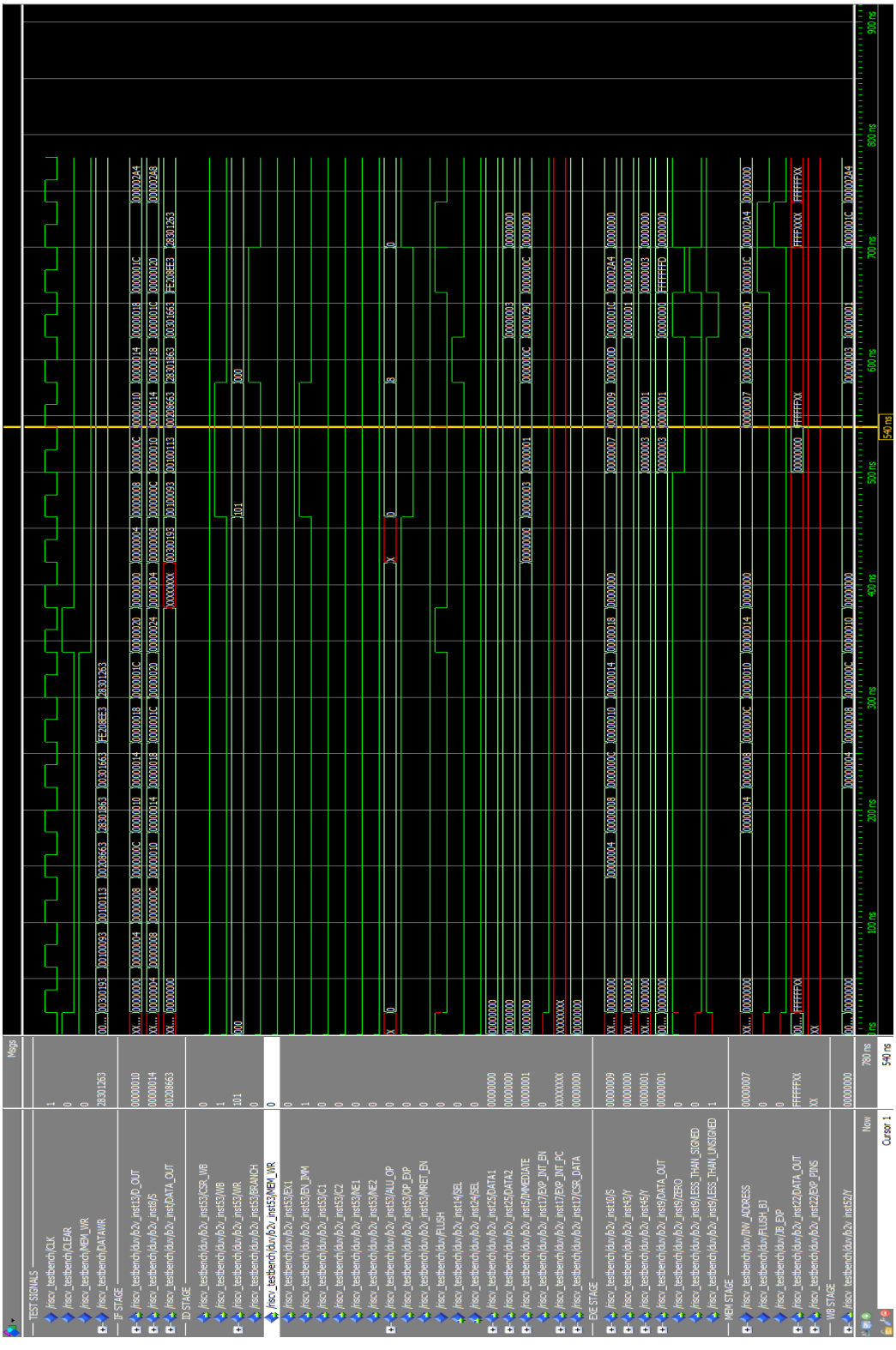
```
00300193          li      gp,3
00100093          li      ra,1
00100113          li      sp,1
00208663          beq     ra,sp,38 <test_3+0x18>
28301863          bne     zero,gp,2c0 <fail>
00301663          bne     zero,gp,40 <test_4>
fe208ee3           beq    ra,sp,34 <test_3+0x14>
28301263          bne     zero,gp,2c0 <fail>
```

The following Figure is the simulation waveform of this test. As we can see at the first 360 ns the mini test instructions are loaded at the Instruction memory, this is done by set the CLEAR pin at the first clock pulse so the

program counter begins to count from the 0x00000000 address and set the MEM_WR pin of Instruction memory so new data can be written in by DATAWR pins. So from second pulse and after we can see the program counter (PC) to be increased by 4 every clock pulse and new data to be written for the next 5 clock pulses. After the data are written successfully the CLEAR pin is set again so the PC can count again from the first address (0x00000000) and the mini test can be executed. In the Table 4.2 we can see the signals we discussed above and their expected and real outputs. As we can see at the Figure all the signals from all the stages are the expected ones, because a lot of explanation is needed for all the signals in every single clock we focus in some basics signals for this particular test. Specifically the first three instructions use the addi instruction so we will focus at the ALU output and for the rest instructions we will focus at ZERO signal.

| **Clock Pulse** | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|
| **ALU Expected Output** | 3 | 1 | 1 | 0 | X | -3 |
| **ALU Real Output** | 3 | 1 | 1 | 0 | X | -3 |
| **ZERO Expected Output** | X | X | X | 1 | X | 0 |
| **ZERO Real Output** | X | X | X | 1 | X | 0 |

Table 4.2: Test 2 Signals



The first instruction is the **li gp,3** (addi gp,x0,3), as we can see from the Table 4.2 and from the Figure above we expect the x0(0) + 1 = 1 from the ALU output and the result to be saved to gp register, second instruction is the **li ra,1** (addi ra,x0,1), from this instruction we expect the x0(0) + 1 = 1 from the ALU output and the result to be saved to ra register, third instruction is the **li sp,1** (addi sp,x0,1), from this instruction we expect the x0(0) + 1 = 1 from the ALU output and the result to be saved to sp register. From the fifth instruction **beq ra,sp,38** which is a branch if equal instruction, we expect to take the branch, that happens because ALU make a subtraction between ra and sp and the output is zero(0) so the ZERO signal is set and the branch is taken. In the Figure of this Test we can see that after the instruction **beq ra,sp,38** the instruction **bne zero,gp,2c0** follows but is a junk instruction because the branch is taken, so the pipeline is cleared and goes to the eighth instruction which is the final instruction **bne zero,gp,2c0**. The final instruction which is branch if not equal is executed and takes the branch but generates an error at the simulator because 0x2C0 address is out of bound for this memory.

## 4.3 Compilation Information

The compilation of the full design was successful in Quartus environment, the tool suggests to use a Cyclone 10 GX FPGA and specifically the 10CX220YF780I5G model which has a lot of LUTs to support such a project. At this design a **total of 11122 logic elements** were used with **10946** of them being **combinational ALUTs** with average LUT depth being 7.63 and max LUT depth being 46. In addition total of **3766 registers** were used and **1024 memory bits** for the memories. Finally the CPU makes use of 40 pins for different reasons, these pins can be found to the schematic.

## 4.4 Worst Case Scenarios

Last but not least part of this project is to find the worst case path in the design. Information about the worst case path can be useful to calculate the maximum frequency of the design, it is common to subtract 10% - 20% from the clock frequency just to be sure that no errors will occur during the execution. So after we ran the Quartus TimeQuest Analyzer simulator tool and make use of the slow model in the simulation the worst case path as can be seen in the Figure 4.1 is **6.544 ns**, this means that the maximum frequency

that the design can run without errors is **152.8 MHz** and by subtracting a safe limit of 20% then the maximum safe frequency of the design is **122.24 MHz**.
The worst case path as Quartus TimeQuest Analyzer suggests, begins from **REGISTER_RD_CSR** (inst12) RD output and continue to **FORWARDING_UNIT** (inst26) which outputs forwarding signals for the multiplexers before **ALU** inputs. The data from the multiplexers are being forwarded to ALU which output **ZERO** signal to the **PIP_REG_EXE_MEM** (inst58) register which is at the end of the path.

| | Slack | From Node | To Node | Launch Clock | Latch Clock | Relationship | Clock Skew | Data Delay | Worst-Case Operating Conditions |
|---|---|---|---|---|---|---|---|---|---|
| 1 | -5.278 | b2v_inst12\|REG_5[0] | b2v_inst58\|ZERO_S | CLK | CLK | 1.000 | -0.050 | 6.544 | Slow 900mV 100C Model |
| 2 | -5.260 | b2v_inst59\|WB_S | b2v_inst58\|ZERO_S | CLK | CLK | 1.000 | -0.050 | 6.526 | Slow 900mV 100C Model |
| 3 | -5.213 | b2v_inst12\|REG_5[0] | b2v_inst58\|ZERO_S | CLK | CLK | 1.000 | -0.050 | 6.479 | Slow 900mV 100C Model |
| 4 | -5.197 | b2v_inst12\|REG_5[3] | b2v_inst58\|ZERO_S | CLK | CLK | 1.000 | -0.050 | 6.463 | Slow 900mV 100C Model |
| 5 | -5.195 | b2v_inst59\|WB_S | b2v_inst58\|ZERO_S | CLK | CLK | 1.000 | -0.050 | 6.461 | Slow 900mV 100C Model |

Figure 4.1: Quartus TimeQuest Analyzer Results

# Chapter 5

# Conclusions and Future Work

## 5.1 Conclusions

The goal of this diploma thesis was to design a working RISC-V based pipeline CPU. The implementation was in VHDL at Intel Quartus environment. Every component was designed in VHDL except the pipeline which connects the components together, the pipeline was schematically designed and converted to VHDL so it can be simulated. For the simulation Modelsim was used so we can see the waveform of every signal of the design and compare it with the expected one. For the simulation we take a collection of tests from the official RISC-V GitHub repository and run some of them for every instruction, the results where the expected ones for all of these tests. While we were testing, we found some errors to some of the components but not serious enough to discuss it. Unfortunately the cost of a real FPGA is high enough so we can't afford one to test the design. Finally we can tell that the design was what we were thinking from the beginning and worked at least at the simulator.

## 5.2 Future Work

The Exceptions-Interrupt part of this design was extra and wasn't tested. To test the component we need a mini test that makes a jump in out of bound address or to generate from external pin, which can not be done because we do not have a real FPGA. When an Interrupt or Exception is generated we must run instructions (Zicr) to load the appropriate registers of the Interrupt/Exception part. So for future work a test of this design will be a priority. Second priority will be to run all of the tests of the official RISC-V GitHub repository. Last but not least, will be the design of a branch

resolver component which function will be to predict the next address if a branch instruction is executed and load it.

# Bibliography


[1] David A. Patterson, and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers inc, 2010.

[2] Peter J. Ashenden. *Digital Design: An Embedded Systems Approach Using VHDL*. Morgan Kaufmann (September 14, 2007).

[3] Carry-lookahead adder
*https://en.wikipedia.org/wiki/Carry-lookahead_adder*
*Last access date : 10/01/2021*

[4] The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, Document Version 20191213
*https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf*
*Last access date : 10/01/2021*

[5] The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified
*https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf*
*Last access date : 10/01/2021*

[6] Michael J.Schulte and E.George Walters III. Design alternatives for barrel shifters
*https://www.princeton.edu/ rblee/ELE572Papers/Fall04Readings/Shifter_Schulte.pdf*
*Last access date : 10/01/2021*

[7] RISC-V Official GitHub Repository
*https://github.com/riscv?page=1*
*Last access date : 10/01/2021*

[8] RISC-V Tests GitHub Repository
*https://github.com/riscv/riscv-tests/tree/master/isa*
*Last access date : 10/01/2021*