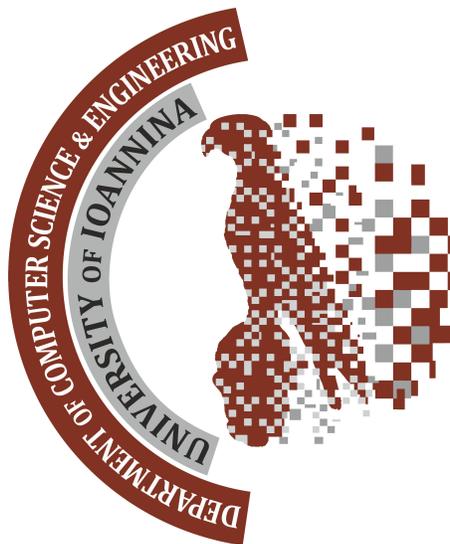


Design and Implementation of a single core RISC-V Central Processing Unit

ISA : RV32I

by **Nikos Deligiannis**

A thesis presented for the diploma of
Computer Science and Engineering



June 2019

Dedication

To my family and in the loving memory of my dear grandfather,
Nikos Deligiannis...

Acknowledgments

I would like to thank my thesis advisor, Aristides Eftymiou for all his help and support. His door was always open whenever I needed some assistance or some clarification on a certain matter.

Also, I deeply thank professor Chrysovalantis Kavousianos for providing me with all the study material and his useful opinions on my work when they were needed.

Last but not least, I would like to thank my family and friends for their endless patience and support they've given me those past few months.

Contents

1	Introduction	2
1.1	Motivation	2
1.2	Development	2
1.3	Report Structure	3
2	Background & Technical Information	4
2.1	Base ISA Overview	4
2.2	ISA Extensions	5
2.3	The RV32I	5
2.3.1	Instruction Length Encoding	5
2.3.2	Base ISA Model	5
2.3.3	Instruction and Immediate Formats	6
2.3.4	Assembler Mnemonics for Registers	7
2.3.5	The Instructions	8
2.3.5.1	Integer Computational Instructions	8
2.3.5.2	Control Transfer Instructions	9
2.3.5.3	Load and Store Instructions	10
3	Design of the RV32I Machine	11
3.1	Instruction Fetch - IF	12
3.1.1	Module's I/O	12
3.2	Instruction Decode - ID	13
3.2.1	Module's I/O	14
3.2.2	Multiplexer Network	14
3.2.2.1	Multiplexer Input	17
3.2.3	Register File	20
3.2.4	Immediate Generator	20
3.2.5	Adder	21
3.2.6	Stall & Forward Control	22
3.2.6.1	Scenario A	22
3.2.6.2	Scenario B	23
3.2.6.3	Scenario C	23
3.2.6.4	Stall Generation	23
3.3	Execute Stage - EXE	25
3.3.1	Module's I/O	25
3.3.2	Adder / Subtractor	26
3.3.2.1	Commands that Use The Module	28

3.3.3	Barrel Shifter	28
3.3.3.1	Commands that Use The Module	30
3.3.4	Logic Module	30
3.3.4.1	Commands that Use The Module	30
3.3.5	Branch Resolver	31
3.3.6	SLT Module	31
3.3.7	ALU Output	32
3.4	Memory Stage - MEM	33
3.4.1	Module's I/O	33
3.4.2	Byte Enable Module	34
3.4.3	Load Masking Module	35
3.5	Write Back - WB	37
3.5.1	Forward Scenario D	37
3.6	The Complete Pipeline	39
3.6.1	ALU Input Selector	41
3.6.2	Control Word Regroup Module	42
3.6.3	WB Selector	42
3.6.4	The M4K Block Issue	43
4	Evaluation of the RV32I Machine	44
4.1	Exploring The Official Tests	44
4.2	Testing Procedure and Final Words	45
4.2.1	A Test Example	45
4.2.1.1	About the Figure 4.1	47
4.2.1.2	About the Figure 4.2	47
4.2.2	Problems Found and Solved by Testing	48
4.2.2.1	The Branch and the Forwards	48
4.2.2.2	The Forward Path C	48
5	Conclusion and Future Work	49
5.1	Future Work	49

List of Figures

2.1	RV32I Instruction Length Encoding	5
2.2	Instruction Types	6
2.3	Immediate Types	6
3.1	Instruction Fetch schematic	12
3.2	Instruction Decode schematic	13
3.3	Control Word Format	17
3.4	Forwarding Paths	22
3.5	Execute Stage schematic	25
3.6	Overflow example	26
3.7	Overflow prohibit example	27
3.8	Adder/Subtractor module	27
3.9	Barrel Shifter module.	29
3.10	Branch Predictor Module	31
3.11	Memory Stage schematic	33
3.12	Byte Offsets and ALU_RES segments.	34
3.13	Write Back schematic	37
3.14	Forward Path D schematic,	38
3.15	RV32I schematic	40
3.16	ALU Input Selector schematic	41
3.17	Design problem due to the M4K blocks.	43
4.1	ADDI test #2.	45
4.2	ModelSim simulation of the "ADDI" test #2.	46

Chapter 1

Introduction

- 1.1 Motivation**
 - 1.2 Development**
 - 1.3 Report Structure**
-

RISC-V is an open-source hardware instruction set architecture (ISA), based on established reduced instruction set computer (RISC) principles. The project began in 2010 at the University of California, Berkeley. Nowadays, well known enterprises in the hardware sector like NVIDIA and Western Digital have announced a plan to start using RISC-V processors in their future products.

1.1 Motivation

In the 21st century, the use of processors in smartphones, tablet computers and Android/iOS devices in general, is following the RISC style architecture. This is why we began looking into RISC and not other architectures (e.g. CISC) to begin with. Being fascinated by the juvenileness of this specific ISA, we were thrilled to do some research and start working with it. Also, we would like to see how *VHDL* would stand up to the challenge since most architectures we could find (e.g. *BOOM*) were developed using CHISEL, a Scala variant hardware description language.

1.2 Development

The whole project was developed hierarchically using *VHDL* and every part was designed, so that it could be synthesizable. In fact, most of the modules we created for the needs of the project were also tested on Altera's Cyclone II - DE 2 FPGA Board. Concerning the evaluation methods used, we ran various simulations using ModelSim and Quartus-II's embedded simulator. Finally, the complete design was successfully tested, using the official tests designated for this ISA (*RV32I*).

1.3 Report Structure

What follows is a brief technical introduction to the “world” of RISC-V and the capabilities that come with it. Then, there will be an analytical presentation of our design and implementation, which is (as mentioned before) the single-core RISC-V 32I (Base ISA). Finally, there will be a chapter dedicated to how the full design was tested, how some issues found while testing were resolved and then, there will be a few words about future work and some improvements that can be done.

Chapter 2

Background & Technical Information

-
- 2.1 Base ISA Overview
 - 2.2 ISA Extensions
 - 2.3 The RV32I
-

2.1 Base ISA Overview

The 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 ISA 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.

The base integer ISA is named "I" (prefixed by RV32 or RV64 depending on integer register width) and contains integer computational instructions, integer loads, integer stores and control-flow instructions. It is mandatory for all RISC-V implementations.

2.2 ISA Extensions

There are various extensions to the Base ISA ("I") such as:

Extension Symbol	Extension Contents
"M"	Integer Multiplication & Division
"A"	Atomic Instructions
"F"	Single-Precision Floating Point
"D"	Double-Precision Floating Point
"Q"	Quad-Precision Floating Point
"L"	Decimal Floating Point
"C"	Compressed Instructions
"B"	Bit Manipulation
"J"	Dynamically Translated Languages
"T"	Transactional Memory
"P"	Packed SIMD Instructions
"V"	Vector Operations
"N"	User-Level Interrupts

Table 2.1: "ISA Extensions"

Therefore, various architectures can come up as a mix of some of those extensions. For example, an integer base ISA plus the extensions "M", "A", "F", "D" is given the abbreviation "G" and provides a general-purpose scalar instruction set. The *RV32G* and *RV64G* (64 simply means 64-bit registers) are the considered to be the default architectures.

2.3 The RV32I

2.3.1 Instruction Length Encoding

The base RISC-V ISA (RV32I) has fixed-length 32-bit instructions that must be naturally aligned on 32-bit boundaries. All the 32-bit instructions in this ISA have their lowest two bits set to 11.

```
xxxxxxxxxxxxxxxx | xxxxxxxxxxxxYYY11  YYY != 111
```

Figure 2.1: RV32I Instruction Length Encoding

2.3.2 Base ISA Model

There are 31 general-purpose registers x1-x32, which hold integer values. Register x0 is hardwired to the constant 0. There is one additional user-visible register: the program counter **pc** holds the address of the current instruction.

2.3.3 Instruction and Immediate Formats

There are four core instruction formats (R/I/S/U). All are fixed 32 bits in length and must be aligned on a four-byte boundary in memory. There are a further two variants of the instruction formats (B/J) based on the handling of immediates.

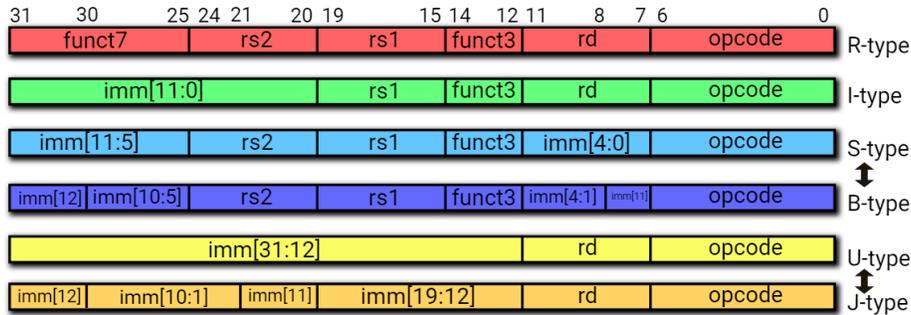


Figure 2.2: Instruction Types

The only difference between the S and B formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware, as is conventionally done, the middle bits ($\text{imm}[10:1]$) and sign bit stay in fixed positions, while the lowest bit in S format ($\text{inst}[7]$) encodes a high-order bit in B format.

Similarly, the only difference between the U and J formats is that the 20-bit immediate is shifted left by 12 bits to form U immediates and by 1 bit to form J immediates. The location of instruction bits in the U and J format immediates is chosen to maximize overlap with the other formats and with each other.

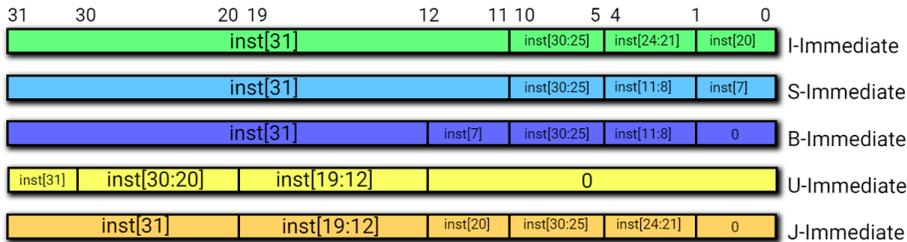


Figure 2.3: Immediate Types

The fields of Figure 2.3 are labeled with the instruction bits ($\text{instr}[y]$) used to construct their value. Sign extension always uses $\text{inst}[31]$.

Labeled as rd is the destination register, meaning the register at which the result of the operation will be stored and as $rs1, rs2$ (if any) are the registers (operands) that will be used. The funct7 , funct3 and opcode fields will be used for the decoding of the command and for the generation of the signals needed for its processing, while the $\text{imm}[x : y]$ fields will be used for the assembly of the command's immediate.

2.3.4 Assembler Mnemonics for Registers

In this subsection, we will list the assembler mnemonics for the x registers and their role in the standard calling convention. Every register in $RV32I$ is 32-bit wide. This listing is taken from the RISC-V User-Level ISA.

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

Table 2.2: Assembler mnemonics for RISC-V integer registers

2.3.5 The Instructions

2.3.5.1 Integer Computational Instructions

Most integer computational instructions operate on $XLEN$ ($= 32$) 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.

Note that there is no special instruction set support for overflow checks on integer arithmetic operations in the RV32I ISA. Overflow checks can be cheaply implemented using branches.

The commands on the tables below follow the color-code of Figure 2.2 and Figure 2.3

[A] Integer Register-Immediate Instructions

Command	Operation	Syntax
ADDI	Addition between imm and $rs1$	$addi\ rd, rs1, imm$
SLTI	If $rs1 < imm$ then $rd \leftarrow 1$ else $rd \leftarrow 0$	$slti\ rd, rs1, imm$
SLTIU	Same as SLTI, but UNSIGNED	$sltiu\ rd, rs1, imm$
ANDI	Bitwise AND between $rs1$ and imm	$andi\ rd, rs1, imm$
ORI	Bitwise OR between $rs1$ and imm	$ori\ rd, rs1, imm$
XORI	Bitwise XOR between $rs1$ and imm	$xori\ rd, rs1, imm$
LLI	Shift left logical of $rs1$ by given shift amount (imm)	$slli\ rd, rs1, imm$
SRLI	Shift right logical of $rs1$ by given shift amount (imm)	$srlr\ rd, rs1, imm$
SRAI	Shift right arithmetic of $rs1$ by given shift amount (imm)	$srai\ rd, rs1, imm$
LUI	$rd \leftarrow imm$	$lui\ rd, imm$
AUIPC	$rd \leftarrow pc + imm$	$auipc\ rd, imm$

Notes:

\Rightarrow The arithmetic operations that include addition ignore overflow scenarios as mentioned before. The immediates used in those commands are I-type Immediates and they are generated as shown at Figure 2.3. Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in $rs1$, and the shift amount is encoded in the lower 5 bits of the I-immediate field.

\Rightarrow Those two instructions are of the U-type and use the respective Immediate types as well. LUI stands for "Load Upper Immediate" while AUIPC stands for "Add Upper Immediate To PC". Again one can recur at Figure 2.3 for further details on the subject of Immediate generation.

Table 2.3: Register-Immediate Instructions

[B] Integer Register-Register Instructions

Command	Operation	Syntax
ADD	Addition between $rs1$ and $rs2$	$add\ rd, rs1, rs2$
SUB	Subtraction between $rs1$ and $rs2$	$sub\ rd, rs1, rs2$
SLT	If $rs1 < rs2$ then $rd \leftarrow 1$ else $rd \leftarrow 0$	$slt\ rd, rs1, rs2$
SLTU	Same as SLT, but UNSIGNED	$sltu\ rd, rs1, rs2$
AND	Bitwise AND between $rs1$ and $rs2$	$and\ rd, rs1, rs2$
OR	Bitwise OR between $rs1$ and $rs2$	$or\ rd, rs1, rs2$
XOR	Bitwise XOR between $rs1$ and $rs2$	$xor\ rd, rs1, rs2$
SLL	Shift left logical of $rs1$ by given shift amount ($rs2$)	$sll\ rd, rs1, rs2$
SRL	Shift right logical of $rs1$ by given shift amount ($rs2$)	$srl\ rd, rs1, rs2$
SRA	Shift right arithmetic of $rs1$ by given shift amount ($rs2$)	$sra\ rd, rs1, rs2$

Notes:

The shift amount is now held in the lower 5 bits of register $rs2$.

Table 2.4: Register-Register Instructions

[C] NOP Instruction

The NOP instruction does not change any user-visible state, except for advancing the **pc**. NOP is encoded as $ADDI\ x0, x0, 0$.

2.3.5.2 Control Transfer Instructions

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

[A] Unconditional Jumps

Command	Operation	Syntax
JAL	$GOTO\ pc + imm$ and $rd \leftarrow pc + 4$	$j\ al\ rd, imm$
JALR	$GOTO\ (rs1 + imm)[31 : 1] \& 0$ and $rd \leftarrow pc + 4$	$j\ alr\ rd, rs1, imm$

Notes:

\Rightarrow JAL (Jump and Link) is a J-type instruction. J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the pc, to form the jump target address. Jumps can therefore target a $\pm 1MiB$ range. JAL stores the address of the instruction following the jump (pc+4) into register rd.

\Rightarrow JALR (Jump and Link Register) uses the I-type encoding. The jump target address is obtained by adding the 12-bit signed 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.

Table 2.5: Unconditional Jumps

[B] Conditional Branches

All branch instructions use the B-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2 and is added to the current pc to give the target address. The conditional branch range is $\pm 4KiB$.

Command	Operation	Syntax
BEQ	<i>if</i> $rs1 = rs2$ then <i>JUMP</i>	<i>beq</i> $rs1, rs2, imm$
BNE	<i>if</i> $rs1 \neq rs2$ then <i>JUMP</i>	<i>bne</i> $rs1, rs2, imm$
BLT	<i>if</i> $rs1 < rs2$ then <i>JUMP</i>	<i>blt</i> $rs1, rs2, imm$
BLTU	Same as BLT, but UNSIGNED	<i>bltu</i> $rs1, rs2, imm$
BGE	<i>if</i> $rs1 \geq rs2$ then <i>JUMP</i>	<i>bge</i> $rs1, rs2, imm$
BGEU	Same as BGE, but UNSIGNED	<i>bgeu</i> $rs1, rs2, imm$

Notes:

The commands BGT, BGTU, BLE and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE and BGEU respectively.

Table 2.6: Conditional Branches

2.3.5.3 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 user address space that is byte-addressed and little-endian. The effective address in both cases is obtained by adding register $rs1$ to the sign-extended 12-bit offset.

Command	Operation	Syntax
LW	$rd \leftarrow MEM[31 : 0]$	<i>lw</i> $rd, imm(rs1)$
LH	$rd \leftarrow MEM[31 : 16]$ or $MEM[15 : 0]$	<i>lh</i> $rd, imm(rs1)$
LHU	Same as LH, but UNSIGNED	<i>lhu</i> $rd, imm(rs1)$
LB	$rd \leftarrow MEM[31 : 24]$ or $MEM[23 : 16]$ or $MEM[15 : 8]$ or $MEM[7 : 0]$	<i>lb</i> $rd, imm(rs1)$
LBU	Same as LB, but UNSIGNED	<i>lbu</i> $rd, imm(rs1)$
SW	$MEM \leftarrow rs2$	<i>sw</i> $rs2, imm(rs1)$
SH	$MEM[31 : 16]$ or $MEM[15 : 0] \leftarrow rs2[15 : 0]$	<i>sh</i> $rs2, imm(rs1)$
SB	$MEM[31 : 24]$ or $MEM[23 : 16]$ or $MEM[15 : 8]$ or $MEM[7 : 0] \leftarrow rs2[7 : 0]$	<i>sb</i> $rs2, imm(rs1)$

Notes:

The choosing of the byte that will be written or loaded in any case is done by the 2 least-significant bits of the calculated effective address.

→ In case of LH/LB operations, the loaded value will be sign-extended up to 32-bits while in case of LHU/LBU, the value will be zero-filled up to 32-bits.

→ Always the least-significant bits of the register $rs2$ are being stored at SH and SB operations.

MEM[X:Y] means to read/write a specific D\$ cell. X and Y point to the bits that should be read/written.

Table 2.7: Loads and Stores

Chapter 3

Design of the RV32I Machine

-
- 3.1 Instruction Fetch - IF**
 - 3.2 Instruction Decode - ID**
 - 3.3 Execute Stage - EXE**
 - 3.4 Memory Stage - MEM**
 - 3.5 Write Back - WB**
 - 3.6 The Complete Pipeline**
-

In this chapter, we will present one by one the CPU's pipeline stages and also the final design. Every part and every module that was created (via *VHDL*) was thoroughly tested before moving to the next one. The whole design consists of five pipeline stages, which are the Instruction Fetch, the Instruction Decode, Execute Stage, Memory Stage and Write Back Stage.

3.1 Instruction Fetch - IF

The Instruction Fetch (*IF*) module consists of a simple memory block (M4K - block)¹ that is used to emulate the Instruction Cache (I\$) in our system. We chose for the I\$ to have 1024 slots, which are 32-bit wide (since we are implementing a 32-bit architecture). So the total capacity of the memory is 4096 B.

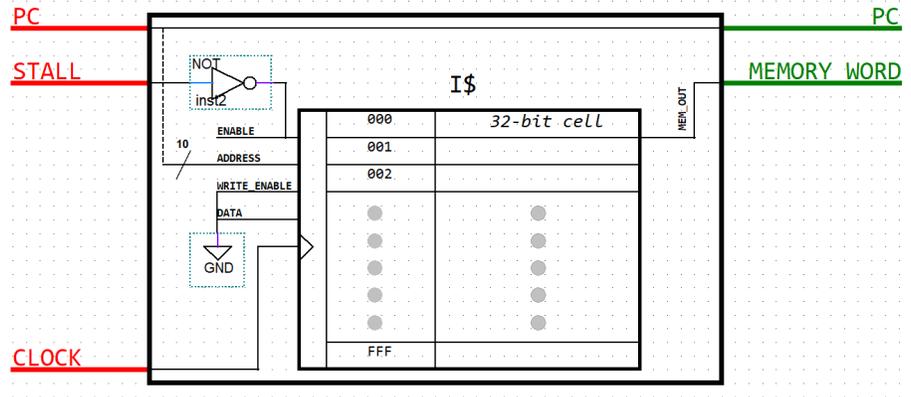


Figure 3.1: Instruction Fetch schematic

IF is responsible for fetching the proper instruction (*MEMORY_WORD*) from I\$. This is done by isolating some bits of the program counter (*PC*) and using them as address for the I\$. Since the memory has 1024 slots, we need $\log_2(1024) = 10$ bits to iterate through all of them and so, we use the *PC*[11..2] bits for this work. Last but not least, the M4K Memory used was automatically generated by Quartus's II Mega-Wizard Plug-In Manager.

3.1.1 Module's I/O

- Inputs:
 1. *CLOCK* : System clock.
 2. *STALL* : Pipeline control signal (1-bit). Its complement is used as Enable signal for I\$.
 3. *PC* : Program counter (32-bit).
- Outputs:
 1. *MEMORY_WORD* : Word fetched from I\$ (32-bit).
 2. *PC* : Program counter (32-bit).

¹The use of M4K was mandatory in our case, since we wanted the design to be synthesizable. Having that in mind, the only embedded memory available on our FPGA board was the M4K RAM memory type

3.2 Instruction Decode - ID

This is one of the most important and resourceful modules of this design. The Instruction Decode (*ID*) module is responsible for one thing among others; to “recognize” the command that was fetched on the previous cycle and activate all the signals that are needed for the command to be successfully processed.

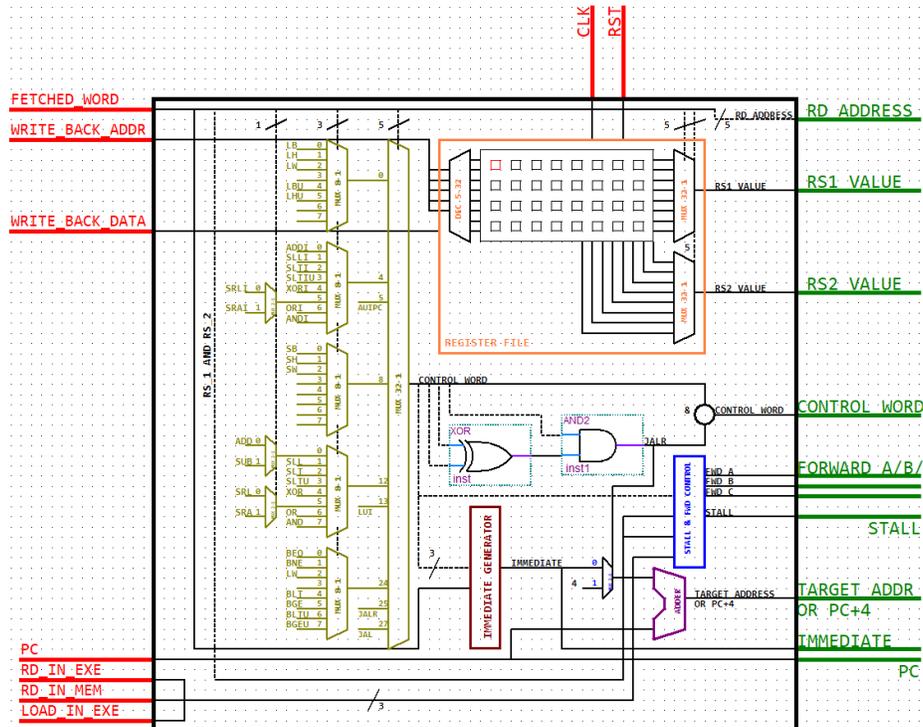


Figure 3.2: Instruction Decode schematic

Designing the *ID* module was an iterative process, due to the nature of the task that is asserted to it. While being the second part of the pipeline, its development suggests a deep knowledge of what will be done in one, two, and three clock cycles later for every(!) command that is currently being decoded. In this stage, we also detect true dependencies (*RAWs*) and handle them accordingly to maximize the performance, when it is needed. Also, *ID* is being equipped with a simple *Adder*, so that it can calculate either the return address of a *JALR* ($PC + 4$), or the target address of a *JAL/BRANCH* ($PC + IMMEDIATE$) according to the control-transfer command that is currently being decoded.

Once again, we will follow the color-code of Figure 3.2, so one can navigate through the design and the description.

3.2.1 Module's I/O

- Inputs:
 1. *CLK* : System clock.
 2. *RST* : System reset signal (1-bit).
 3. *FETCHED_WORD*: Feed from *IF*; word read from I\$ (32-bits).¹
 4. *WRITE_BACK_ADDR* : Feed from *Write Back* stage; *rd* register address (5-bits).
 5. *WRITE_BACK_DATA* : Feed from *Write Back* stage; value (result) that must be written to *rd* (32-bit).
 6. *PC* : Feed from *IF* stage; Program Counter (32-bits).¹
 7. *RD_IN_EXE* : Feed from *Execute* stage; *rd* register address of the previous command (5-bit).
 8. *RD_IN_MEM*: Feed from *Memory* stage; *rd* register address of the command before the previous one (5-bits).
 9. *LOAD_IN_EXE*: Control signal from *Execute* stage; Alerts the *ID* module if there is a *LOAD* command in *Execute* (1-bit).
- Outputs:
 1. *RD_ADDRESS* : The address of the *rd* register (5-bits).
 2. *RS1_VALUE* : Value of the *rs1* register (32-bits).
 3. *RS2_VALUE* : Value of the *rs2* register (32-bits).
 4. *CONTROL_WORD* : Control-Signals for the following stages. (18-bits).
 5. *FORWARD_A* : Control signal for Forward Path A (2-bits).
 6. *FORWARD_B* : Control signal for Forward Path B (2-bits).
 7. *FORWARD_C* : Control signal for Forward Path C (1-bit).
 8. *TARGET_ADDR_OR_PC + 4* : (32-bits).²
 9. *IMMEDIATE* : Value of the Command's Immediate (32-bits).
 10. *PC* : Program Counter (32-bits).

3.2.2 Multiplexer Network

Being tasked with the work of decoding the previously fetched instruction (**word**), we have to figure out a way to detect which one of the possible instructions it is. After the successful decode of the word, we have to provide all(!) the mandatory **control signals** for the next pipeline stages, to make sure that all the necessary actions for the process of the now decoded instruction will be done.

¹These signals come from a Pipeline Register.

²See Section 3.2.5 for more information about this signal

At Section 2.3.3, we listed all the instructions that belong to our *RV32I* implementation. Also, Figure 2.2 shows that all the instruction types have their 7 LSBs¹ dedicated for the *opcode* field. Also, in combination with Figure 2.1, we conclude that there is no point to process the two LSBs of any word that has to be decoded.

The first step for the process is to understand how the commands are being encoded by the Assembler; and to do so we used the official RISC-V Instruction Set Manual. The encoding of every command is displayed below at Table 3.1. Marked with bold style is all the information the Multiplexer Network uses to determine the identity of the command.

Observing the encoding information displayed on Table 3.1, we came to the following conclusions:

- According to their *opcode*² bits, the commands are either stand-alone or belong in a group with other commands of similar type or functionality.
- The groups are the following five:
 - Loads
 - I-type arithmetics
 - Stores
 - R-type commands
 - Branches
- Most of the commands that belong to a group, have a “unique” **funct3** 3-bit code.
- Some of the grouped commands have the same *funct3* code, but they have a different **funct7[5]** bit.

In conclusion the Multiplexer Network is responsible of selecting the correct control signal to pass to the following pipeline stages. Each multiplexer’s input is a static control signal, which dictates what operations must be done in every stage for the successful process of the command.

¹Least Significant Bits

²Except the two LSBs

imm[31:12]				rd	0110111	“LUI”	
imm[31:12]				rd	0010111	“AUIPC”	
imm[20&10:1&11&19:12]				rd	1101111	“JAL”	
imm[11:0]			rs1	000	rd	1100111	“JALR”
imm[12&10:5]	rs2	rs1	000	imm[4:1&11]	1100011	“BEQ”	
imm[12&10:5]	rs2	rs1	001	imm[4:1&11]	1100011	“BNE”	
imm[12&10:5]	rs2	rs1	100	imm[4:1&11]	1100011	“BLT”	
imm[12&10:5]	rs2	rs1	101	imm[4:1&11]	1100011	“BGE”	
imm[12&10:5]	rs2	rs1	110	imm[4:1&11]	1100011	“BLTU”	
imm[12&10:5]	rs2	rs1	111	imm[4:1&11]	1100011	“BGEU”	
imm[11:0]			rs1	000	rd	0000011	“LB”
imm[11:0]			rs1	001	rd	0000011	“LW”
imm[11:0]			rs1	010	rd	0000011	“LW”
imm[11:0]			rs1	100	rd	0000011	“LBU”
imm[11:0]			rs1	101	rd	0000011	“LHU”
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	“SB”	
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	“SH”	
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	“SW”	
imm[11:0]			rs1	000	rd	0010011	“ADDI”
imm[11:0]			rs1	010	rd	0010011	“SLTI”
imm[11:0]			rs1	011	rd	0010011	“SLTIU”
imm[11:0]			rs1	100	rd	0010011	“XORI”
imm[11:0]			rs1	110	rd	0010011	“ORI”
imm[11:0]			rs1	111	rd	0010011	“ANDI”
0000000	shamt	rs1	001	rd	0010011	“SLLI”	
0000000	shamt	rs1	101	rd	0010011	“SRLI”	
0100000	shamt	rs1	101	rd	0010011	“SRAI”	
0000000	rs2	rs1	000	rd	0110011	“ADD”	
0100000	rs2	rs1	000	rd	0110011	“SUB”	
0000000	rs2	rs1	001	rd	0110011	“SLL”	
0000000	rs2	rs1	010	rd	0110011	“SLT”	
0000000	rs2	rs1	011	rd	0110011	“SLTU”	
0000000	rs2	rs1	100	rd	0110011	“XOR”	
0000000	rs2	rs1	101	rd	0110011	“SRL”	
0100000	rs2	rs1	101	rd	0110011	“SRA”	
0000000	rs2	rs1	110	rd	0110011	“OR”	
0000000	rs2	rs1	111	rd	0110011	“AND”	

Notes:

”&” is the concatenation operator.

Table 3.1: RV32I Command Encoding

Starting from right to left, there is a $32 \rightarrow 1$ multiplexer. This multiplexer is used to select the correct group of the command (if the command belongs to a group) or the stand-alone command itself (e.g. “AUIPC”). This is done by using the *opcode*[6..2] bits of the word as selector. Since the *opcode*[6..2] bits alter from command to command in a non-sequential manner, we use all five of them and end up with a $2^5 = 32 \rightarrow 1$ multiplexer.

Previous to that, there are five $8 \rightarrow 1$ multiplexers, which use the three *funct3* bits (if any) of the word to select a specific command inside a group. Note that since we have five different command groups, we use five multiplexers in this layer.

Some of the commands, belong to the same group and also have the same *funct3* code (e.g. “ADD” - “SUB”). To separate them, we utilize the *funct7*[5] bit of the word which changes in that case. So using this bit as selector, we attach to the network three $2 \rightarrow 1$ multiplexers and so, we cover all the possible scenarios of decoding.

3.2.2.1 Multiplexer Input

As mentioned above, the input of every multiplexer is a static, hard-typed control signal of 20 bits of the following format.

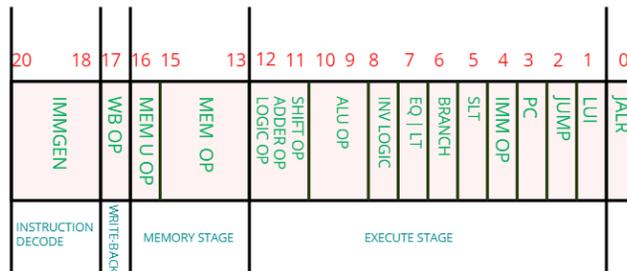


Figure 3.3: Control Word Format

Depicted with blue color at Figure 3.3 there are four groups of bits. These are the bits that concern each pipeline stage. The bit 0 is used also at ID stage along with bits 20..18, but it is appended later as Figure 3.2 shows and is not hard-typed in every input of the multiplexers like the rest of the control signal bits. It is also used for pipeline control.

- **IMMGEN** : Immediate Generation
 - 000 : I-type Immediate,
 - 001 : S-type Immediate.
 - 010 : B-type Immediate.
 - 011 : U-type Immediate.
 - 100 : J-type Immediate.
- **WB OP** : Write Back operation. 0 if the command doesn't require a Write Back action, 1 if it does.
- **MEM U OP** : Memory Unsigned Operation. 0 → *NO*, 1 → *YES*.
- **MEM OP** : Memory Operation.
 - 000 : LB
 - 001 : LH
 - 010 : LW
 - 100 : SB
 - 101 : SH
 - 110 : SW
 - 111 : MEM-Free operation.
- **ALU OP**: ALU Operation
 - 00 : Addition.
 - 01 : Subtraction.
 - 10 : Logic Operation.
 - 11 : Shift Operation.
- **SHIFT/ADDER/LOGIC OP**: Since bits [9..8] determine which ALU module will be used there is no problem using the same bits [11..10] to represent different operations in different ALU modules.
 - Shift Module:
 - * 00 : Shift Right Logical.
 - * 01 : Shift Left Logical.
 - * 10 : Shift Right Arithmetic.
 - Adder Module:
 - * 0X : Signed Addition or Subtraction.
 - * 1X : Unsigned Addition or Subtraction.
 - Logic Module:
 - * 00 : And.
 - * 01 : Or.
 - * 10 : Xor.
- **INV & EQ/LT**: Used for resolving a branch case. ¹
- **BRANCH**: 0 → the command is not a Branch, 1 → the command is a Branch.
- **SLT**: 0 → the command is not SLT, 1 → the command is SLT.
- **PC**: 0 → PC is not required for calculations. 1 → PC is required for calculations.
- **JUMP**: 0 → the command is not a JUMP. 1 → command is a JUMP.
- **LUI**: 0 → the command is not a LUI. 1 → the command is a LUI.
- **JALR**: 0 → the command is not a JALR. 1 → the command is a JALR.

¹Refer to subsection 3.3.5 for further analysis of those two bits

Command	Control Word
LB	000-1-0-000-0X-00-X-X-0-0-1-0-0-0-0
LH	000-1-0-001-0X-00-X-X-0-0-1-0-0-0-0
LW	000-1-0-010-0X-00-X-X-0-0-1-0-0-0-0
LBU	000-1-1-000-0X-00-X-X-0-0-1-0-0-0-0
LHU	000-1-1-001-00-00-X-X-0-0-1-0-0-0-0
ADDI	000-1-X-111-0X-00-X-X-0-0-1-0-0-0-0
SLLI	000-1-X-111-01-11-X-X-0-0-1-0-0-0-0
SLTI	000-1-X-111-0X-01-X-X-0-1-1-0-0-0-0
SLTIU	000-1-X-111-1X-01-X-X-0-1-1-0-0-0-0
XORI	000-1-X-111-10-10-X-X-0-0-1-0-0-0-0
SRLI	000-1-X-111-00-11-X-X-0-0-1-0-0-0-0
SRAI	000-1-X-111-10-11-X-X-0-0-1-0-0-0-0
ORI	000-1-X-111-01-10-X-X-0-0-1-0-0-0-0
ANDI	000-1-X-111-00-10-X-X-0-0-1-0-0-0-0
SB	001-0-0-100-0X-00-X-X-0-0-1-0-0-0-0
SH	001-0-0-101-0X-00-X-X-0-0-1-0-0-0-0
SW	001-0-0-110-0X-00-X-X-0-0-1-0-0-0-0
ADD	XXX-1-X-111-0X-00-0-0-0-0-0-0-0-0-0
SUB	XXX-1-X-111-0X-01-0-0-0-0-0-0-0-0-0
SLL	XXX-1-X-111-01-11-0-0-0-0-0-0-0-0-0
SLT	XXX-1-X-111-0X-01-0-0-0-0-1-0-0-0-0
SLTU	XXX-1-X-111-1X-01-0-0-0-0-1-0-0-0-0
XOR	XXX-1-X-111-10-10-0-0-0-0-0-0-0-0-0
SRL	XXX-1-X-111-00-11-0-0-0-0-0-0-0-0-0
SRA	XXX-1-X-111-10-11-0-0-0-0-0-0-0-0-0
OR	XXX-1-X-111-01-10-0-0-0-0-0-0-0-0-0
AND	XXX-1-X-111-00-10-0-0-0-0-0-0-0-0-0
BEQ	010-0-X-111-0X-01-0-1-1-0-0-0-0-0-0
BNE	010-0-X-111-0X-01-1-1-1-0-0-0-0-0-0
BLT	010-0-X-111-0X-01-0-0-1-0-0-0-0-0-0
BGE	010-0-X-111-0X-01-1-0-1-0-0-0-0-0-0
BLTU	010-0-X-111-1X-01-0-0-1-0-0-0-0-0-0
BGEU	010-0-X-111-1X-01-1-0-1-0-0-0-0-0-0
AUIPC	011-1-X-111-0X-00-X-X-0-0-1-1-0-0-0
LUI	011-1-X-111-XX-00-X-X-0-0-1-1-0-1-0
JALR	000-1-X-111-01-00-X-X-0-0-1-0-1-0-1
JAL	100-1-X-111-0X-00-X-X-0-0-1-1-1-0-0

Notes:

“X” stands for “Don’t Care”. It could be either 1 or 0.

Table 3.2: Commands and their Control-Words

This encoding is the result of many iterations, due to the fact that while being in the second pipeline stage (*ID*), we had to think about what will be needed to be done in the next pipeline stages. Finally, with respect to the encoding, we present the control-words (multiplexer inputs) for every instruction in our ISA.

3.2.3 Register File

The register file consists of 32 registers which are 32-bit wide as our architecture dictates. They all have the function of **read** and **write**(parallel load) except for one, the register 0 which has the value 0 hardwired inside it. This value cannot be altered, meaning the register cannot do a parallel load operation. All other registers have also a *RESET* and a *LOAD* control signal, which are activated only when a global¹ reset happens.

As shown in Figure 3.2, the register file is between a $5 \rightarrow 32$ decoder and two $32 \rightarrow 1$ multiplexers. The decoder is used for the *WRITE* operations and the multiplexers are used for the *READ* operations. Almost every command in our ISA has 5 bits ($[11..7]$)² dedicated for the register *rd*, which is the destination register, meaning the register in which the result of the command will be written into. In our pipeline architecture, this happens at the fifth (Write Back) stage. So, when a command reaches the Write Back stage and if it is a command that has to write a result into a register, then the Write Back provides the *WRITE_BACK_ADDR* and *WRITE_BACK_DATA* signals back to the register file. The *rd*'s address is connected to the decoder and then the proper *LOAD* signal is activated, so that the register that translates to *rd*'s address will be ready for a *WRITE* operation.

The two multiplexers are used to provide the operands' *rs1* and *rs2* (if any) values for the Execute Stage. The first multiplexer provides the *rs1* value, by using the instruction word's bits $[19..15]$ ² as selector, while the second multiplexer provides the *rs2* value, by using the instruction word's bit $[24..20]$ ² as selector. Every multiplexer input is paired with every register's output. For example, $Reg_0 \rightarrow I_0$, $Reg_1 \rightarrow I_1$, $Reg_2 \rightarrow I_2$ etc.

Of course, not every command requires a *rs1* or *rs2* value. In this case the register file will provide two values that will be random and that will not be needed. Later, we add further logic components to resolve this issue.

3.2.4 Immediate Generator

The Immediate Generator module is responsible for providing the immediate that is required, according to the instruction that is being decoded. It uses the three bits $[20..18]$ of the control word (Figure 3.3) and according to them, formats the proper immediate, by rearranging and manipulating the bits of the fetched word, as Figure 2.3 shows. When this procedure is over, the bits $[20..18]$ of the control word are no longer needed and thus they are removed of the control-word, before it leaves the *ID* stage. The immediate generator module implements the following algorithm:

¹When the CPU starts running, we assume that there will be a short reset to initialize all the pipeline components

²You can refer to Figure 2.2 for any clarifications needed.

Algorithm 1: Immediate Generator

```

input : CONTROL_WORD[20..18] and FETCHED_WORD
output: IMMEDIATE
1 IMM_TYPE ← CONTROL_WORD[20..18];
2 if IMM_TYPE == "000" then
3   | IMMEDIATE = I-type Immediate, f(FETCHED_WORD) ;
4 else if IMM_TYPE == "001" then
5   | IMMEDIATE = S-type Immediate, f(FETCHED_WORD) ;
6 else if IMM_TYPE == "010" then
7   | IMMEDIATE = B-type Immediate, f(FETCHED_WORD) ;
8 else if IMM_TYPE == "011" then
9   | IMMEDIATE = U-type Immediate, f(FETCHED_WORD) ;
10 else if IMM_TYPE == "100" then
11  | IMMEDIATE = J-type Immediate, f(FETCHED_WORD) ;
12 else
13  | IMMEDIATE = XXX..X ;
14 end

```

3.2.5 Adder

In the the case of Unconditional Jumps, two calculations are required. The calculation of the target address and the storing of the next instruction's address, the return address ($PC + 4$), to register rd. This translates into two addition operations that must be done in the event of those commands. All the computational force in a pipeline usually is entrusted to the pipeline's ALU¹. In our pipeline ALU lays in the third stage, the *Execute* stage. So we would have to equip the ALU module with two Adders since when a jump is imminent, two additions should be done. To relieve some workload of the ALU's design, we decided to equip the *ID* module with one of those two adders, an adder that will be responsible only for the calculation of the target address of the unconditional jumps and branches.

In theory, this would work fine, since all the necessary operands are available at *ID* stage. After further investigating the *JALR* command though, we see that the target address is calculated in a different way. Instead of $PC + IMM$, *JALR* dictates that the target address is calculated as $PC + RS1$. So this means that first, we have to access the register file to acquire the one of the two operands and then do the addition operation, since the *PC* value is already available (from the *IF* stage). In practice, we found that due to some propagation delays, we should handle the *JALR* case differently.

¹Arithmetic and Logic Unit

What we did to resolve this, is to treat the *JAL* and *JALR* commands in *ID* like this:

- If the command is *JAL* or *BRANCH* then:
 - Calculate the *TARGET_ADDRESS* as $PC + IMMEDIATE$.
- If the command is *JALR* then:
 - Calculate the *RETURN_ADDRESS* as $PC + 4$.

This is achieved by adding a $2 \rightarrow 1$ multiplexer that selects by *CONTROL_WORD*[0] (*JALR*) either the *IMMEDIATE* or the constant $+4$. So for *JAL*, we leave the calculation of the jump's return address to the *Execute* stage and for *JALR*, we leave the calculation of the jump's target address to the *Execute* stage.

3.2.6 Stall & Forward Control

This is a module that was added later on *ID* and it is responsible for detecting whether a stall or a forwarding is required. Forwarding (or Bypassing) is the counter-measure of the true dependency/hazard RAW¹, which is the only threat in our system, since we do not support OOO².

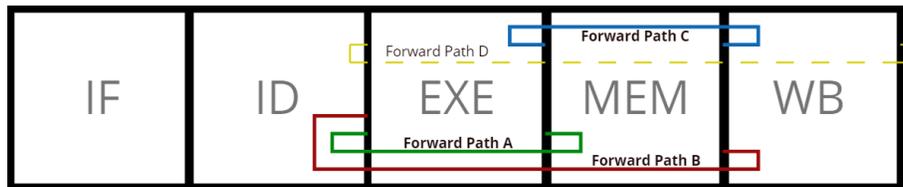


Figure 3.4: Forwarding Paths

The module is responsible for handling Forwards A,B, and C, which are shown at Figure 3.4, while forward D³ is being handled by some external logic. So, there are three main RAW scenarios that require resolution.

3.2.6.1 Scenario A

1	OP_A	Reg_X, Reg_A, Reg_B
2	OP_B	Reg_Y, Reg_X, Reg_C

Listing 3.1: Forward Path A Example

This scenario concerns the *ID* and *Execute* stages. *OP_A* writes its result to *Reg_X*. Then *OP_B* requires *Reg_X*'s value as *rs1* operand. Without the forwarding path, we would have to stall for at least two clock cycles and wait for *OP_A* to reach the *Write Back* stage. We know that the value which will be written at *Reg_X* will be available at the end of *Execute* stage. So, instead of stalling for two clock cycles, we activate the Forward Path A and transfer the value needed to *OP_B*.

¹Read After Write

²Out Of Order execution

³We will analyze the Forward D later.

3.2.6.2 Scenario B

1	OP_A	Reg_X , Reg_A , Reg_B
2
3	OP_B	Reg_Y , Reg_X , Reg_C

Listing 3.2: Forward Path B Example

This scenario concerns the *ID* and *Memory* stages. Once again, OP_A writes its result to Reg_X and then, after one command that is in-between them, OP_B requires this value as an operand. If not for the Forward Path B, we would have to stall again for one clock cycle and wait for OP_A to reach the *Write Back* stage.

3.2.6.3 Scenario C

1	LOAD	Reg_X , IMM(Reg_A)
2	STORE	Reg_X , IMM(Reg_B)

Listing 3.3: Forward Path C Example

This scenario concerns only the *Memory* stage. It only occurs when there is a *STORE* command after a *LOAD* command, of whom the later wishes to write in the memory the register value, that the first has loaded. We would normally have to stall for one clock cycle again, but with the Forward Path C we can overcome this issue.

3.2.6.4 Stall Generation

1	LOAD	Reg_X , IMM(Reg_A)
2	OP_A	Reg_A , Reg_X , Reg_B

Listing 3.4: Stall Scenario

In this case, we have to stall the OP_A and those who follow, because the *LOAD* command has to reach the *Memory* stage and then, activate the Forward Path B so that it can provide the needed value. This happens due to the nature of the *LOAD* commands, which have their result ready at the end of *Memory* stage and not at the *Execute* stage like the other computational commands.

Algorithm 2: Stall and Forward Control

```

input :  $RS1[4..0], RS2[4..0], RD\_IN\_EXE[4..0], RD\_IN\_MEMORY[4..0],$ 
          $LOAD\_IN\_EXE, CONTROL\_WORD[20..18]$ 
output:  $FWDA[2], FWDB[2], FWDC, STALL$ 

1 /* Initialization */
2  $FWDA[2] = \{0,0\};$ 
3  $FWDB[2] = \{0,0\};$ 
4  $FWDC = 0;$ 
5  $STALL = 0;$ 

6 /* Identify the Type of the Command in ID */
7  $COMMAND\_IN\_ID \leftarrow f(CONTROL\_WORD[20..18]) = (R/S/U/B/J);$ 

8 /* Equality Check between RS1,RS2 and RD in EXE and MEM */
9 if  $RS1 == RD\_IN\_EXE$  AND  $RS1 \neq REG_0$  then
10 |  $FWDA\_A\_RS1 \leftarrow 1;$ 
11 end
12 if  $RS2 == RD\_IN\_EXE$  AND  $RS2 \neq REG_0$  then
13 |  $FWDA\_A\_RS2 \leftarrow 1;$ 
14 end
15 if  $RS1 == RD\_IN\_MEM$  AND  $RS1 \neq REG_0$  then
16 |  $FWDA\_B\_RS1 \leftarrow 1;$ 
17 end
18 if  $RS2 == RD\_IN\_MEM$  AND  $RS2 \neq REG_0$  then
19 |  $FWDA\_B\_RS2 \leftarrow 1;$ 
20 end

21 /* Check if the command in ID has an RS1 , RS2 register on its type */
22 if  $COMMAND\_IN\_ID \neq U/J - type^a$  then
23 |  $FWDA[0] = FWDA\_A\_RS1;$ 
24 |  $FWDB[0] = FWDA\_B\_RS1;$ 
25 else
26 |  $FWDA[0] = 0;$ 
27 |  $FWDB[0] = 0;$ 
28 end
29 if  $COMMAND\_IN\_ID = S/B/R - type^b$  then
30 |  $FWDA[1] = FWDA\_A\_RS2;$ 
31 |  $FWDB[1] = FWDA\_B\_RS2;$ 
32 else
33 |  $FWDA[1] = 0;$ 
34 |  $FWDB[1] = 0;$ 
35 end
36 if  $LOAD\_IN\_EXE = 1$  AND  $COMMAND\_IN\_ID = S - type$  then
37 |  $FWDC = 1;$ 
38 end
39 if  $LOAD\_IN\_EXE = 1$  AND  $(FWDA\_A\_RS1 OR FWDA\_A\_RS2)$  then
40 |  $STALL = 1;$ 
41 end

```

^aIf the Command in *ID* is of U or J type then it does not have an *rs1* register, hence we must not activate a forward signal

^bOnly the Stores, Branches and R-type commands have an *rs2* register and so, a forward signal for *rs2* is required only in their case

3.3 Execute Stage - EXE

The *Execute* is the third stage in our pipeline. This is where all the computational force of our system is located. It consists of three main parts. The Adder/Subtractor, the Barrel Shifter and the Logical Module. Furthermore, this is the stage in which we decide whether to follow a Branch or not.

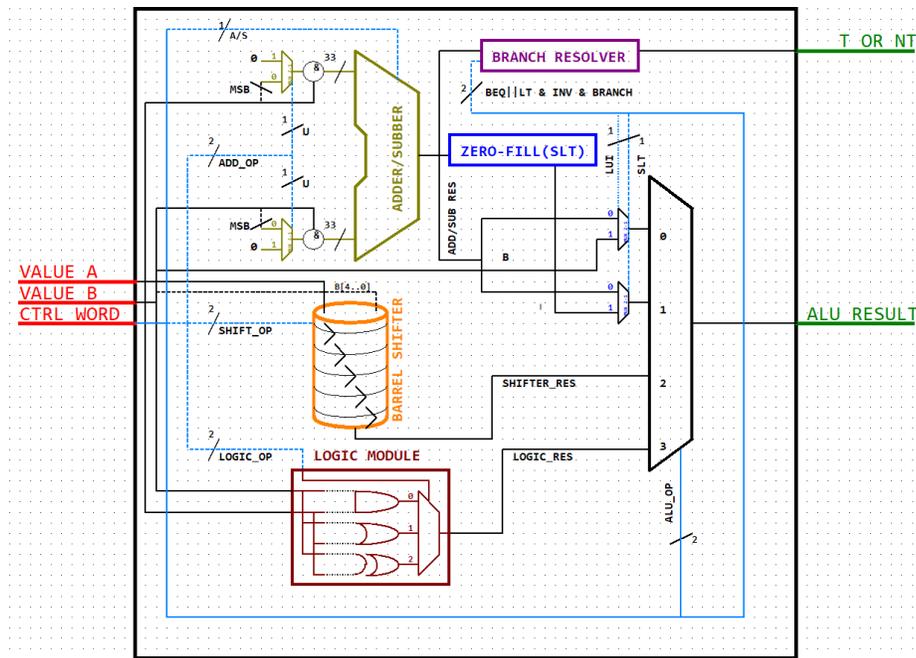


Figure 3.5: Execute Stage schematic

3.3.1 Module's I/O

- Inputs:
 1. *VALUE_A* : The first of two operands (32-bits).¹
 2. *VALUE_B* : The second operand (32-bits).¹
 3. *CTRL_WORD* : Control Signals from *ID* (9-bits¹).
 - bits[8..7]: ADD_OP/SHIFT_OP/LOGIC_OP.
 - bits[6..5]: ALU_OP.
 - bit[3]: EQ/LT.
 - bit[2]: BRANCH.
 - bit[1]: SLT.
 - bit[0]: LUI.
- Outputs:
 1. *T_OR_NT* : Branch condition result (1-bit).
 2. *ALU_RES* : (32-bit).

¹These signals come from the *ALU Input Selector* module (3.6.1)

Note that we only use 9 out of the total 18 control bits which were generated by the *ID* stage. This happens because we simply do not have any use for the other bits, hence we use a logic - circuit between the pipeline registers to separate them respectively, meaning the pipeline stage that they must be addressed to, and then in every clock cycle distribute them to the designated location.

3.3.2 Adder / Subtractor

This module is responsible for doing Signed and Unsigned addition and subtraction. It is in its core a ripple-carry architecture, in which we added a few customizations of our own. Besides the typical modification that the ripple-carry architecture requires so it can do a subtraction (2's complement on B operand), as one can notice in Figure 3.5 the module's inputs have a $2 \rightarrow 1$ multiplexer, which is responsible for the extra bit[33] (which we added), and is controlled by the select bit[8], which in case of addition or subtraction stands for Signed/Unsigned operation.

Normally, as mentioned before, we do not care for Overflow scenarios and we leave this to the programmer's hands. But there is a case that an overflow could be disastrous for us, an overflow in signed subtraction operations. If two numbers are subtracted and their signs are different, then overflow occurs if and only if the result has the same sign as the subtrahend¹. For example, lets say we have a module that subtracts 3-bit numbers. In this module we insert as minuend² the number 011(+3) and as subtrahend the number 101(-3).

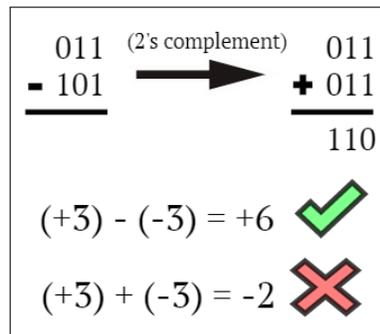


Figure 3.6: Overflow example

This problem should not bother us, since we leave this issue to the programmer. But, in some cases like the case of a *BGE* or *BLT* command, this fault could be catastrophic. In order to evaluate if we must take or not the jump, we do a subtraction and then check for the difference's sign to decide. The problem mentioned above would make us decide mistakenly the outcome of the branch command. The MSB of the 3-bit result is 1, which wrongly implies a negative number.

¹What is being subtracted.

²What is being subtracted from.

Since the sign is the most important information that we might lose with the overflow issue, we overcome this by adding one extra bit to our operands. Instead of subtracting 32-bit numbers, we sign-extend the values up to 33-bits and then continuing with the operation without any risks of sign alteration at the results. With the same numbers as operands, we attach this logic to the previous example.

$$\begin{array}{r}
 \begin{array}{r}
 0011 \text{ (2's complement)} \\
 - 1101 \\
 \hline
 \end{array}
 \longrightarrow
 \begin{array}{r}
 0011 \\
 + 0011 \\
 \hline
 0110
 \end{array} \\
 \\
 (+3) - (-3) = +6 \quad \checkmark \\
 (+3) + (+3) = +6 \quad \checkmark
 \end{array}$$

Figure 3.7: Overflow prohibit example

With respect to the logic that was analyzed above, we use a 33-bit Adder/Subtractor in our ALU, that does Signed operations by using sign-extension up to 33-bits and Unsigned operations by using zero-fill up to 33-bits. Since we add 1 extra bit to our module for signed actions, we have to utilize it properly for the unsigned ones. Hence, we use zero-fill instead for all the Unsigned operations, so that the result will not be corrupted. The two $2 \rightarrow 1$ multiplexers are responsible for the sign-extension/zero-fill and then the selected value is concatenated to the operands' MSB¹.

Since our architecture is 32-bit, we cannot simply continue with 33-bit values in our pipeline. The result of the Adder/Subtractor gets cropped down to 32-bits, while the 33rd bit is being used by other modules (e.g *BRANCH_RESOLVER*) for further computations.

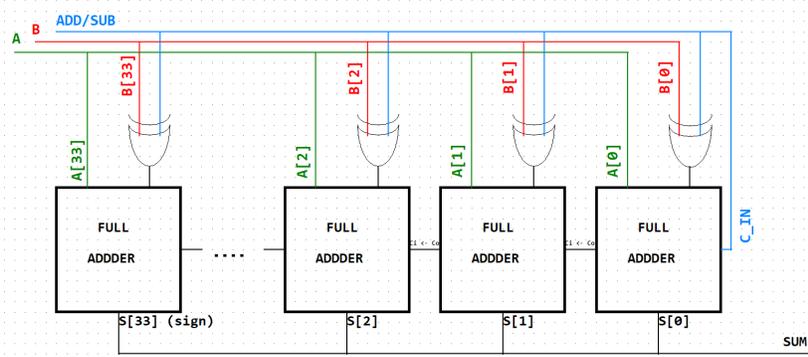


Figure 3.8: Adder/Subtractor module

¹Most Significant Bit.

3.3.2.1 Commands that Use The Module

With respect to the sequential order that the Table 3.1 declares, here is a list of the commands that utilize the Adder/Subtractor module.

Command	Reason	Comment
AUIPC	$IMMEDIATE + PC$	-
JAL	$PC + 4$	Next command's address
JALR	$PC + RS1$	Target address
BRANCHES	$RS1 - RS2$	Equality/Inequality check
LOADS	$RS1 + IMMEDIATE$	Effective address
STORES	$RS1 + IMMEDIATE$	Effective address
ADDI	$RS1 + IMMEDIATE$	-
SLTI[U]	$RS1 - IMMEDIATE$	The sign of the result will be the value for rd^1
ADD	$RS1 + RS2$	-
SUB	$RS1 - RS2$	-
SLT[U]	$RS1 - RS2$	Same as SLTI[U]

Table 3.3: Commands that use the Adder/Subtractor

3.3.3 Barrel Shifter

The barrel shifter is a digital circuit that can shift a data word by a specified number of bits ² in just 1 clock cycle without the use of any sequential logic, only pure combinatorial logic. The way we implemented it, 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**. Generally, the number of multiplexers required for an n -bit word is $n \log_2(n)$; in our case, we have 32-bit words, hence normally we would need $32 \log_2(32) = 160$ multiplexers in total.

Our system has 32-bit words, which can be shifted up to 32 slots left or right. So, to represent the shift amount (shamt) we need $\log_2(32) = 5$ bits. According to those five bits, we separate the shifting operation into stages:

- **Stage #1:** Shift by 16; controlled by shamt[4].
- **Stage #2:** Shift by 8 ; controlled by shamt[3].
- **Stage #3:** Shift by 4 ; controlled by shamt[2].
- **Stage #4:** Shift by 2 ; controlled by shamt[1].
- **Stage #5:** Shift by 1 ; controlled by shamt[0].

In every shifting stage, we need multiplexers to represent each bit of the data word which is being shifted, so every stage needs 32 multiplexers. This is how the number 160 comes up. But, our barrel shifter module must be able to do right arithmetic, right logical and left logical shifting operations. We need 160 multiplexers for just right or left shift. To be able to do all three kinds of shift, we have to use more circuitry; in fact, we used $[2 \times 32 \log_2(32)] + 1 = 321$ multiplexers:

¹If $RS1 - IMMEDIATE < 0$ then the result will have bit[33] (sign) = 1. So we can easily handle this command by only looking at the result's sign

²Shift amount (shamt).

- For each stage, we use two multiplexers instead of one ($2\times$).
- We have five stages for every bit of the shifting word ($32\log_2(32)$).
- We use one extra multiplexer for right arithmetic shifts ($+1$).

The barrel shifter requires 2 control bits to do one of the shifts. Note the MSB signifies the Arithmetic shift, while the LSB the left or right shift.

Shift Type	Control Word
Right	0-0
Left	0-1
Right Arithmetic	1-0
ERROR	1-1

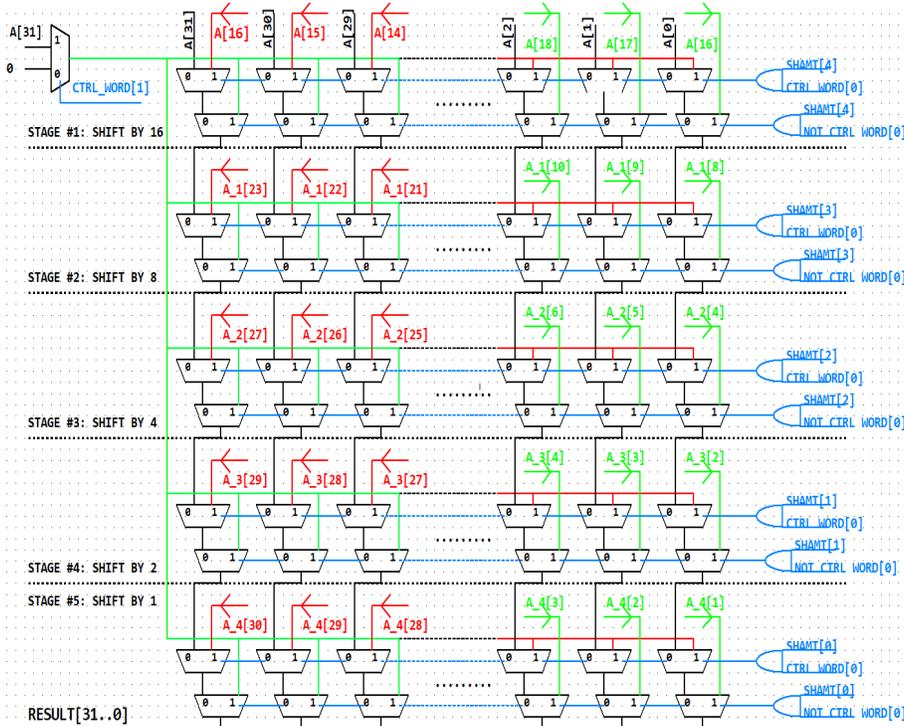


Figure 3.9: Barrel Shifter module.

All the theory mentioned above is now aggregated into Figure 3.9. There are five shifting stages and for every bit we use two multiplexers, where the top one is responsible for handling the left shifts and the bottom one the right and the right arithmetic shifts.

3.3.3.1 Commands that Use The Module

The commands that utilize the barrel shifter module are the following:

Command	Reason	Comment
SLLI	$RS1 \ll IMM$	-
SRLI	$RS1 \gg IMM$	-
SRAI	$RS1 \ggg IMM$	Vacant positions are filled with MSB
SLL	$RS1 \ll RS2[5..0]$	-
SRL	$RS1 \gg RS2[5..0]$	-
SRA	$RS1 \ggg RS2[5..0]$	Vacant positions are filled with MSB

Table 3.4: Commands that use the Barrel Shifter

3.3.4 Logic Module

The logic module is the simplest part in our ALU. It consists of three large logical gates; the AND, the OR and the XOR, which are 32-bit wide. The two inputs for each gate are the two operands of the ALU, A and B. The module's output is selected from a $4 \rightarrow 1$ multiplexer, which has two control bits as selector, the bits[12..11] of the control word (Figure 3.3).

3.3.4.1 Commands that Use The Module

The commands that utilize the logic module are the following:

Command	Reason	Comment
XORI	$RS1 \oplus IMM$	-
ORI	$RS1 + IMM$	-
AND	$RS1 \wedge IMM$	-
XOR	$RS1 \oplus RS2$	-
OR	$RS1 + RS2$	-
AND	$RS1 \wedge RS2$	-

Table 3.5: Commands that use the Logic Module

3.3.5 Branch Resolver

The Branch Resolver module is responsible for determining whether we should follow or not a *BRANCH* command. Note, that our design does not possess a branch prediction unit and so, we deal with branch cases in Execute stage. This means, that if the branch is finally *Taken* that we have to flush the previous two pipeline stages (*IF, ID*), because the commands that have been fetched are not valid anymore.

It takes as input three control bits and the result of the Adder/Subtractor module. It works according to the following truth table:

Command	BEQ	INVERT LOGIC	BRANCH
BEQ	1	0	1
BNE	1	1	1
BLT	0	0	1
BGE	0	1	1

Table 3.6: Branch Command Encoding

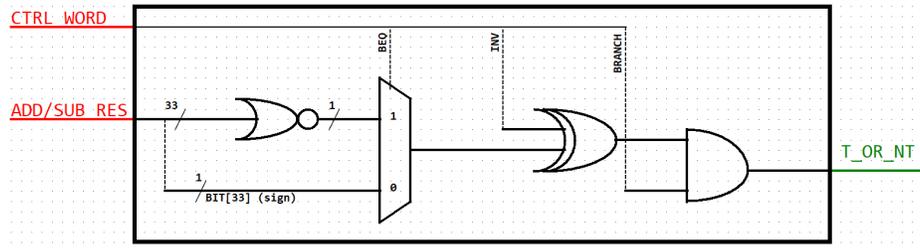


Figure 3.10: Branch Predictor Module

The general idea for the development of this module was the following:

$$\mathbf{A \ relop \ B} \iff \mathbf{A - B < 0}^1$$

We determine equalities and inequalities via subtraction and then, we extract the result (Adder/Subtractor feed) and decide, with the circuit of Figure 3.10, if we must Take or Not the branch. Reminder; the result from the *ADD/SUB_RES* is 33 bits and not 32, due to the overflow avoidance strategy we chose.

3.3.6 SLT Module

This module is dedicated purely to the Set Less Than (SLT) command. It simply takes the sign bit of the Adder/Subtractor module and zero-fills it up to 32-bits. So the output of the module will be either 0 or 1, which are the two options for the SLT command's result. For example:

- *SLT REG_X, 1, 2* :: $1 - 2 = -1$ (bit[33] = 1), so the result is 1.
- *SLT REG_X, 2, 1* :: $2 - 1 = +1$ (bit[33] = 0), so the result is 0.

¹We can easily check the result (sign - bit) of the subtraction's result

3.3.7 ALU Output

All the results of the stage's modules are accumulated to the $4 \rightarrow 1$ multiplexer (Figure 3.5). The multiplexers inputs are selected according to the control word bits[10..9]. Furthermore, there are two extra $2 \rightarrow 1$ multiplexers, which are responsible for the LUI and SLT commands.

The Load Upper Immediate (LUI) command does not require any computational effort from our ALU unit. It simply stores a U-Immediate to the *rd* register. So, we encoded it accordingly and used the same ALU OPCODE with the Additions. The first $2 \rightarrow 1$ multiplexer (with control bit[1] as selector) is responsible for providing this U-Immediate in the case of LUI commands.

In a similar way, we used another $2 \rightarrow 1$ multiplexer (with control bit[5] as selector) for the SLT command, but in this case, since the command uses the Adder/Subtractor module to perform a subtraction operation, we used the ALU OPCODE of the Subtraction operations.

3.4 Memory Stage - MEM

The fourth stage in our pipeline is the *Memory* stage. This stage is used exclusively by *LOAD* and *STORE* commands. Here lies another M4K memory block, which emulates the Data Cache (D\$) of our system. Usually, in a modern CPU, the D\$ is organized as a hierarchy of more cache levels (L1,L2, etc.), but we used a simple memory block for the needs of our project. Also, this is the place where global variables in a piece of code are being stored (.data field). For more information concerning the M4K memory blocks we used (I\$ and D\$), you can read [Altera's Internal Memory \(RAM and ROM\) User Guide](#) and [Intel's Embedded Memory User Guide](#).

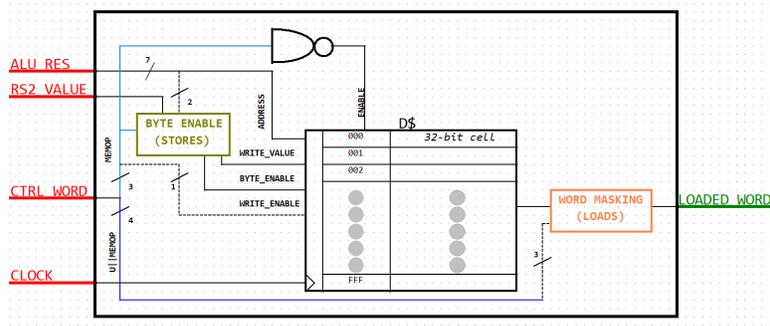


Figure 3.11: Memory Stage schematic

The D\$ we used has 128 cells, which are 32-bit wide. Hence, to iterate through all of them, we need $\log_2(128) = 7$ bits. Those 7 bits are acquired from the result that ALU provides to the pipeline. We isolate from those 32-bits of the result, the nine (9) LSBs, of whom the bits[1..0] are used as byte offset in case of a byte or a half load/store operation. Last but not least, the *MEMOP* control signal¹ is 111 for the commands that are neither *STORE* nor *LOAD* and so, we use a reductive NAND gate to generate an *ENABLE* signal for our D\$².

3.4.1 Module's I/O

- Inputs:

1. *CLOCK* : Global Clock.
2. *ALU_RES* : LSBs from the ALU to be used as address and offset (9-bits).³
3. *RS2_VALUE* : The value that will be written (32-bits).³
4. *CTRL_WORD* : Control Signals from *ID* (4-bits).³
 - bit[3]: Signed or Unsigned Load.
 - bits[2..0]: Memory Operation (see page 18 for details).

- Outputs:

1. *LOADED_WORD* : Word read from D\$ (32-bits).

¹Refer to Table 3.2 for more information about the command encoding.

²Also, MEMOP[2] is used as a Write Enable signal because its 0 for *LOADS*

³Those signals are actually split into two categories. Those who concern the *LOADS* and those who concern the *STORES*. The *STORE* signals are bypassing the Pipeline Register, while the *LOAD* signals do not.

3.4.2 Byte Enable Module

The M4K Memory block allows the user to write in specific bytes inside a memory cell (*BYTEEN*). In our case, since we have 32-bit cells, we can divide the memory cell into four bytes which can be written individually. Furthermore, we can modify two bytes inside a memory cell. This means, that we can write either on bits[31..16] or bits[15..0] of a memory cell with the SH (Store Half) command.

To achieve that, we must also have a number that will imply the byte or the bytes that will be written inside the memory cell. With respect to everything mentioned above, we use the two LSBs of the address value (*ALU_RES*) as an iterator for every memory cell. For referring to a byte half, we use the bit[1] of the *ALU_RES* and for bytes, we use the bits[1..0] as Figure 3.12 depicts.

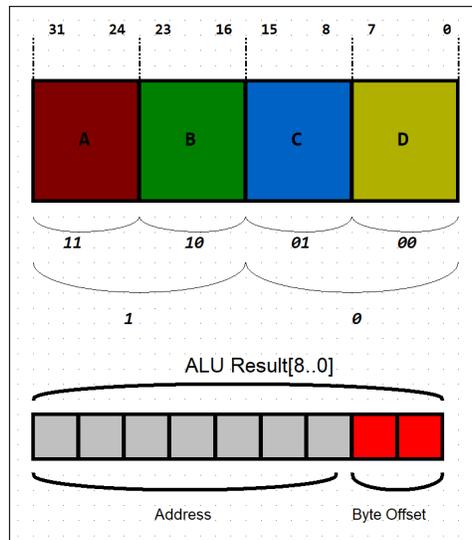


Figure 3.12: Byte Offsets and ALU_RES segments.

This introduction was mandatory to explain the functionality of the Byte Enable Module, which is dedicated to *STORE* commands. It takes as input the value of the register *RS2* and the two LSBs of the *ALU_RES*, along with the three *MEMOP* control bits, which represent the *STORE* command type. Then, according to the value of the *MEMOP* and the two LSBs, it generates the proper *BYTEEN* signal¹. After the signal is generated, the module reforms *RS2*'s value according to the *STORE* type. If the command is SB, then the reformed word is $rs2[7..0] \times 4$. If the command is SH, then the reformed word is $rs2[15..0] \times 2$. If the command is SW, then word remains the same. This means that we can only write either the lowest byte or lowest-half of the *RS2* value if not all of it. Algorithm 3 demonstrates the behavior of Byte Enable Module;

¹Refer to the [Altera's Internal Memory \(RAM and ROM\) User Guide](#) for further information about the Byte Enable option of the M4K Block

Algorithm 3: Byte Enable Module

```

input : CONTROL_WORD[2..0], RS2_VALUE[31..0] and ALU_RES[6..0]
output: BYTEEN[3..0] and REFORMED_RS2_VALUE[31..0]

1 MEMOP ← CONTROL_WORD[1..0];
2 PAD ← ALU_RES[1..0];
3 BYTE ← RS2_VALUE[7..0];
4 HALF ← RS2_VALUE[15..0];
5 /* Store Byte Case */
6 if MEMOP == "00" then
7   | if PAD == "00" then
8     | BYTEEN = "0001";
9   | else if PAD == "01" then
10    | BYTEEN = "0010";
11   | else if PAD == "10" then
12    | BYTEEN = "0100";
13   | else if PAD == "11" then
14    | BYTEEN = "1000";
15   | REFORMED_RS2_VALUE = BYTE & BYTE & BYTE & BYTE; a
16 /* Store Half Case */
17 else if MEMOP == "01" then
18   | if PAD[1] == "0" then
19     | BYTEEN = "0011";
20   | else if PAD[1] == "1" then
21     | BYTEEN = "1100";
22   | REFORMED_RS2_VALUE = HALF & HALF;
23 /* Store Word Case */
24 else if MEMOP == "10" then
25   | BYTEEN = "1111";
26   | REFORMED_RS2_VALUE = RS2_VALUE; // No Modification
27 else
28   | BYTEEN = "XXXX";

```

^a: "&" is the Concatenation Operator.

3.4.3 Load Masking Module

Besides the primary LW (Load Word) command, the RV32I ISA supports Signed and Unsigned Loads for Bytes and Halves, while the M4K memory block can only provide a word, which is read from one of its memory cells. So we must design a circuit that will be responsible for making all those Load commands viable. The Load Masking Module is responsible for taking as input the word, which has just been read from the D\$ and modifying it accordingly as the Load type command instructs. The module was designed behaviorally and here is the algorithm that represents it:

Algorithm 4: Load Masking Module

```

input : CONTROL_WORD[3..0], MEMORY_VALUE[31..0] and
         ALU_RES[1..0]
output: LOADED_WORD[31..0]

1 MEMOP ← CONTROL_WORD[1..0] ;
2 PAD ← ALU_RES ;
3 SIGNED/UNSIGNED ← CONTROL_WORD[3] ;
4 A ← MEMORY_VALUE[31..24];
5 B ← MEMORY_VALUE[23..16];
6 C ← MEMORY_VALUE[15..8];
7 D ← MEMORY_VALUE[7..0];
8 /* Load Byte Case */
9 if MEMOP == "00" then
10 | /* Signed */ if SIGNED/UNSIGNED == "0" then
11 | | if PAD == "00" then
12 | | | LOADED_WORD = SE(MEMORY_VALUE[7]) & D; a
13 | | else if PAD == "01" then
14 | | | LOADED_WORD = SE(MEMORY_VALUE[15]) & C;
15 | | else if PAD == "10" then
16 | | | LOADED_WORD = SE(MEMORY_VALUE[23]) & B;
17 | | else if PAD == "11" then
18 | | | LOADED_WORD = SE(MEMORY_VALUE[31]) & A;
19 | /* Unsigned */ else if SIGNED/UNSIGNED == "1" then
20 | | if PAD == "00" then
21 | | | LOADED_WORD = ZF(MEMORY_VALUE[31..8]) & D; b
22 | | else if PAD == "01" then
23 | | | LOADED_WORD = ZF(MEMORY_VALUE[31..8]) & C;
24 | | else if PAD == "10" then
25 | | | LOADED_WORD = ZF(MEMORY_VALUE[31..8]) & B;
26 | | else if PAD == "11" then
27 | | | LOADED_WORD = ZF(MEMORY_VALUE[31..8]) & A;
28 /* Load Half Case */
29 if MEMOP == "01" then
30 | /* Signed */
31 | if SIGNED/UNSIGNED == "0" then
32 | | if PAD[1] == "0" then
33 | | | LOADED_WORD = SE(MEMORY_VALUE[15]) & CD;
34 | | else if PAD[1] == "1" then
35 | | | LOADED_WORD = SE(MEMORY_VALUE[31]) & AB;
36 | /* Unsigned */
37 | else if SIGNED/UNSIGNED == "1" then
38 | | if PAD[1] == "0" then
39 | | | LOADED_WORD = ZF(MEMORY_VALUE[31..16]) & CD;
40 | | if PAD[1] == "1" then
41 | | | LOADED_WORD = ZF(MEMORY_VALUE[31..16]) & AB;
42 else
43 | /* Load Word Case (Default) */
44 | LOADED_WORD = MEMORY_VALUE;

```

^a: "SE" stands for Sign Extension.^b: "ZF" stands for Zero Fill.

3.5 Write Back - WB

The *Write Back* stage is the final and simplest stage of our pipeline. It is the stage, which signifies the completion of a command. It gathers the data that must be written to the command's *RD* register (if any), along with the *RD*'s address and sends this information back to the Register File, which is located at the *ID* stage.

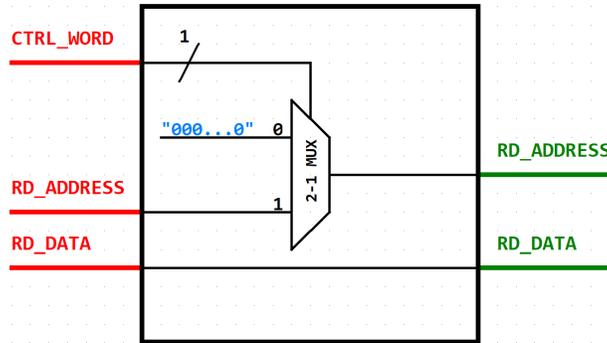


Figure 3.13: Write Back schematic

The write back data can either be the result of the ALU, if it is a computational command, or the result of the *MEM* stage, if it is a *LOAD* command. Some commands, for example, all the *STOREs* do not have a destination register so the *WriteBack* stage has no use for them. This is why we use a $2 \rightarrow 1$ multiplexer, so for every instruction that has no *rd* register, we provide the address of the *x0* register, which will do nothing, since, as we mentioned before, the register *x0* cannot be written and its hardwired to the constant 0. The proper input of the multiplexer is selected by the control bit *WB OP*, which was previously generated at *ID* stage.

3.5.1 Forward Scenario D

Previously, at page 22, we analyzed the Forwarding/Bypassing paths of our pipeline. We covered Forward of type A,B and C. But there is one extra forwarding that must happen, when we encounter the following Scenario:

1	OP_A	Reg_X ,	Reg_A ,	Reg_B
2
3
4	OP_B	Reg_Y ,	Reg_X ,	Reg_C

Listing 3.5: Forward Path D Example

This forward path is being handled by some external, pipeline logic that does not belong to the *Stall and Forwarding Predictor Module* of the *ID* stage. Normally, this would not be an issue, if we could write and then instantly read from the same register inside the Register File. But, when we tested the full design without the Forward Path D, we confronted some data corruption cases, since in some tests the Write and Read of the same register were successful and in others were not.

3.6 The Complete Pipeline

We presented with great detail the five basic stages of our system. In this section, we will analyze the complete pipeline of the RV32I ISA, and all the components that were previously mentioned as external/pipeline modules. Also, we will discuss about the peculiarity of the M4K memory blocks and how they were a “problem” to our initial design plans.

Lets begin by introducing the extra, yet mandatory components for a pipeline architecture; the pipeline registers. Shown in Figure 3.15, there are four registers (A/B/C/D), one for each data transaction between the CPU modules. They were implemented using behavioral architectures based on the standard D flip flop logic. All four registers have a reset RST ¹ signal (the same signal we mentioned for the previous modules), which is positive edge triggered and occurs when the CPU starts working (on clock cycle #0). All registers have as many inputs as the number of output signals the previous²CPU module provides, plus some signals that are transferred from one stage to the next one³. Registers A and B have also two extra control signals, the $FLUSH$ ⁴ and $STALL$ signals, which occur every time we encounter a RAW hazard ($STALL$) and whenever we have a conditional or unconditional jump case ($FLUSH$).

Furthermore, there is one extra register, the PC (Instruction Pointer), which is a simple 32-bit register that holds the memory address of the next instruction that would be fetched. After providing the address to IF module, its value gets updated accordingly:

- Either provides the address of the next instruction ($PC + 4$).
- Or in case of Jump/Branch provides the target address of the Jump.

Secondly, we will make it easy for one to navigate into Figure 3.15 since it has a lot of detail and it may seem confusing. So we will begin by pairing colors to the respective purposes and functionalities that they bespeak:

- The PC register and the next PC signal (NPC) are drawn with **purple** color.
- Figure’s 3.4 color code for forwarding paths (FWD **A/B/C/D**) is the same in Figure 3.15.
- The $STALL$ and $FLUSH$ control signals are depicted with **deep blue** color.
- The $CONTROL_WORD$ and all its forks are depicted with **orange** color.
- The $CLOCK$ and RST signals are depicted with **dark red** color.

Lastly, there are three extra pipeline modules that we created to make the signal distribution and the workload for every stage less and easier. These modules are the **ALU INPUT SELECTOR**, the **CONTROL WORD REGROUP** module and the **WB SELECTOR** respectively.

¹It is used to initialize the whole system.

²For example Pipeline Register A has two data inputs ($I\$$ WORD and PC VALUE).

³For example, in EXE stage there are two signals transferred directly to register C from register B (RD ADDRESS and RS2 VALUE).

⁴Empties all the data the register currently has.

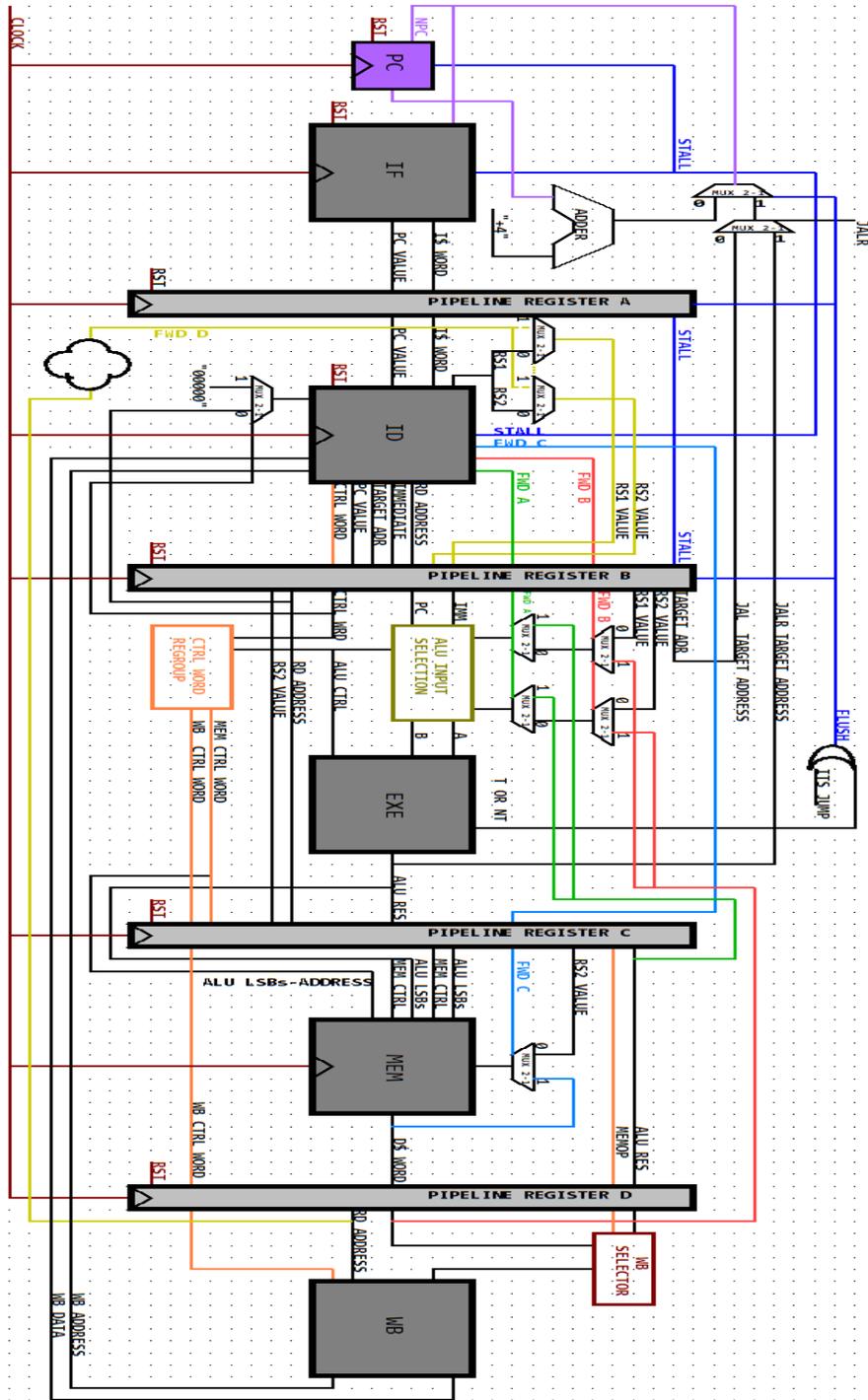


Figure 3.15: RV32I schematic

3.6.1 ALU Input Selector

The *EXE* stage, which has the Arithmetic and Logic Unit, simply requires two operands and one control word to operate. The problem here is that there are many different commands in our ISA, which require different operations and different operands, hence, we should either attach more circuitry to the *ID* module (which is already overloaded with functions) or design an outer-pipeline stage module dedicated for this work, which is what we finally chose.

The ALU Input Selector module takes as input the *RS1* and *RS2* values from the Register File, the *PC* and *IMMEDIATE* from the *ID* stage, along with some control bits, the *JALR*, *JUMP*, *PC* and *IMM*¹ and decides according to the following schematic which are going to be operands A and B that will finally go to the *EXE* stage.

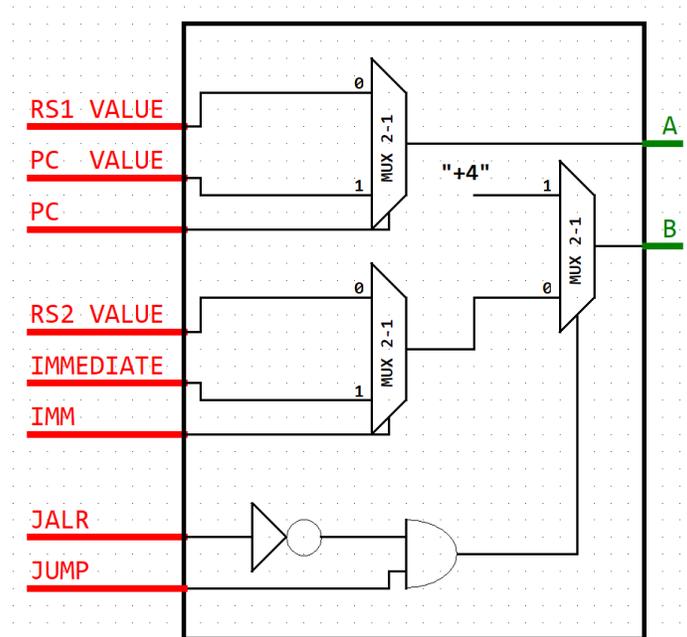


Figure 3.16: ALU Input Selector schematic

The first two $2 \rightarrow 1$ multiplexers are selecting either the *RS1/2* values or their alternatives, which for *RS1* is the *PC* value (the commands that do not have an *RS1* register in their type, e.g “*AUIPC*”) and for *RS2* is the *IMMEDIATE* that is generated at *ID* (the commands that do not have an *RS1* register in their type, e.g “*ADDI*”). The inputs are selected by the corresponding control bits. The third $2 \rightarrow 1$ multiplexer is used to implement the theory we developed at sub-section 3.2.5, in which we segregate the work of the *ID*’s Adder and *EXE*’s ALU that has to be done in *JALR* and *JAL/BRANCH* cases.

¹Redirect to page 18 for more information about the meaning of the control bits.

3.6.2 Control Word Regroup Module

As we indicated before, the *ID* stage generates a control-word, which is necessary for the following pipeline stages, since it dictates which operations they must undertake. This control-word though, is large and its not completely useful in every stage. To facilitate our design, we designed the Control Word Regroup Module, which given the control-word generated from *ID*, it divides it into smaller ones, which are addressed to the respective pipeline stages. It also provides the essential control bits to the ALU Input Selector module. The module's behavioral architecture is described by the following simple algorithm:

Algorithm 5: Control Word Regroup

```

input : CONTROL_WORD[17..0]
output: ALU_SELECTOR_WORD[3..0], ALU_CONTROL_WORD[8..0],
          MEM_AND_WB_CONTROL_WORD[4..0]

1 /* Bits for ALU Selector */
2 JALR ← CONTROL_WORD[0];
3 JUMP ← CONTROL_WORD[2];
4 PC ← CONTROL_WORD[3];
5 IMM ← CONTROL_WORD[4];

6 /* Bits for ALU */
7 SHIFT/ADDER/LOGIC OP ← CONTROL_WORD[12..11];
8 ALU OP ← CONTROL_WORD[10..9];
9 INV LOGIC ← CONTROL_WORD[8];
10 EQ||LT ← CONTROL_WORD[7];
11 BRANCH ← CONTROL_WORD[6];
12 SLT ← CONTROL_WORD[5];
13 LUI ← CONTROL_WORD[1];

14 /* Bits for MEM and WB */
15 MEM U OP ← CONTROL_WORD[16];
16 MEM OP ← CONTROL_WORD[15..13];
17 WB OP ← CONTROL_WORD[17];

18 ALU_SELECTOR_WORD ← JALR & JUMP & PC & IMM;
19 ALU_CONTROL_WORD ← SHIFT/ADDER/LOGIC OP & ALU OP &
    INV LOGIC & EQ||LT & BRANCH & SLT & LUI;
20 MEM_AND_WB_CONTROL_WORD ← MEM U OP & MEM OP & WB OP;

```

We do not separate at this point further the control-word for *MEM* and *WB* stages, because they have many bits in common. The separation will occur later inside the pipeline registers.

3.6.3 WB Selector

This module is actually a $2 \rightarrow 1$ multiplexer, which decides if the value that must be written back to the Register File's *RD* register should be the result from *EXE*'s ALU, or *MEM*'s D\$. This is done by and-reducing the control-bits of *MEMOP* and selecting with it either the ALU result (if the result is 1) or the D\$ read word (if the result is 0):

```

if (MEMOP[2] ∧ MEMOP[1] ∧ MEMOP[0])' == 1 → D$_WORD
if (MEMOP[2] ∧ MEMOP[1] ∧ MEMOP[0])' == 0 → ALU_RES

```

3.6.4 The M4K Block Issue

The memory type of our caches is the M4K block. We chose this type of memory, because we want our design to be fully synthesizable and since we have already worked with Altera's DE-2 FPGA board, this is the only memory we could use and actually test. But with this selection, a design problem appeared; these memory blocks have by default registered inputs and optionally registered outputs. This means for stages *IF* and *MEM*, who have the caches embedded into them, that for every instruction that passes through them, we would need two clock cycles and not one, as we initially had planned.

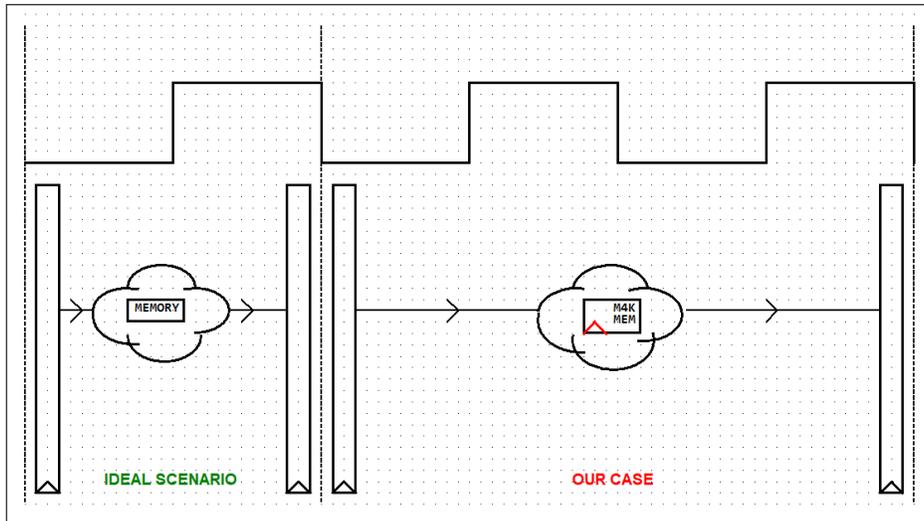


Figure 3.17: Design problem due to the M4K blocks.

So, for the *ID* and *MEM* pipeline stages, we had to figure out a solution to this problem; having 2 clock cycles for some pipeline stages and one clock cycle for the rest, was considered unacceptable by us. The plan is for every pipeline stage to be operative in 1 clock cycle. So, we simply ignored the *PC* register in the case of *IF* stage and we ignored the *PIPELINE REGISTER C* in the case of *MEM* stage. So every signal that is directed either towards the I\$ or the D\$ is coming directly from another module and not from a register.

- For *IF*, which has the I\$ the signal that skips the *PC* register is the Next Pc Value (*NPC*), which is used as address.
- For *MEM*, which has the D\$ the signals that skip the Pipeline Register C are:
 - The control-bits which concern the *STORE* commands.
 - The *RS2* register value which concerns the *STORE* commands.
 - The LSB's from the ALU, which are used as offset.
 - The *ALU_RES*[8..2] which is used as address.

Chapter 4

Evaluation of the RV32I Machine

4.1 Exploring The Official Tests

4.2 Testing Procedure and Final Words

4.1 Exploring The Official Tests

Since we completed the design of our system, it is now time to test it, so that we can be sure that our work is accurate. To do so, we searched at the official [RISC-V GitHub repository](#) and we found out that they actually provide a toolchain, in which various tests for every architecture can be found. So, after installing and configuring their software, we retrieved 37 assembly and machine code files, which were later broke down to *.hex* and *.dump* files. So, for every instruction in our ISA we have one *.dump* file and one *.hex* file.

The official RV32I tests have the following structure:

- Every test file (for every instruction) has many mini-tests inside it.
- Every mini-test:
 - Uses the instruction in which it is dedicated.
 - On success, it updates the *GP*¹ register and moves to the next one.
 - On failure, it jumps to an *ECALL* instruction which has a specific opcode and terminates the test ².

Judging by the way the tests were programmed, we conclude that they were designed so that they can guarantee that every possible hazardous scenario for every instruction has been accounted for and avoided. We will mention some cases, which we haven't initially considered, but we discovered later during testing the pipeline. Alternatively, we could try and use our own tests but, unfortunately, we could not find a stand-alone assembler for our cause. This means, that in order to test the system ourselves we would have to program our very own RV32I Assembler...

¹Global Pointer

²Since the *ECALL* command is not included to our current ISA, for the purpose of testing we added one extra signal to the *ID* module which is the *ECALL* signal. When it comes it simply the signal takes the value 1 and so we understand that the test is finished

4.2 Testing Procedure and Final Words

After understanding the testing logic, we used python (version 3) to create a script that takes the *.hex* file and generates a *.mif*¹, which represents the loaded I\$. For *LOADS* and *STORES* tests, which had some values stored in D\$, we manually initialized the data cache *.mif* files for every one of them. In conclusion, we end up with 37 different *.mif* files for I\$ and also 8 *.mif* files for D\$².

All that's left now is to run one timing simulation for every test that we have. For our initial tests, we used Quartus-II³ embedded simulator. Unfortunately, the first tests were not successful (as we expected), due to various logic errors in the final structural design and the debug process was not easy, because we could not probe the suspicious internal nodes and signals easily using this simulator. We had to modify every file and add extra signals, so that they could show up at the simulation. This means alternation to package files and all modules that use the circuit in which we appended a new probing signal to observe.

After many debug attempts, we switched to ModelSim where we could easily add and remove every node we wished in the simulation window, something that we made our debug process much easier than it was before. So after various errors we spotted during the testing process, we were able to finally find and fix every bug that we found. But for every change we made, we couldn't be sure that this change would not affect previous tests that were completed successfully. So, after the final/last changes we made we performed a regression testing, meaning that we run all the tests again. The results were satisfying, since every test was completed successfully.

4.2.1 A Test Example

We will now showcase a sample of our testing process. This test concerns the "ADDI" command. All we need is just the *.dump* file, so that we can understand and follow the simulation and also, we need the simulation window. The "ADDI" test file has 25 mini-tests, so we cannot go through all of them. We will present the first of those tests and its prosperous results.

```

00000000 <_start>:
  0:  00000093          li  ra,0
  4:  00008f13          mv  t5,ra
  8:  00000e93          li  t4,0
  c:  00200193          li  gp,2
 10:  27df1c63          bne t5,t4,288 <fail>

00000014 <test_3>:
 14:  00100093          li  ra,1

```

Figure 4.1: ADDI test #2.

¹Memory Initialization File

²There are 5 tests for all the *LOAD* instructions and 3 for all *STORE* instructions.

³Version 9.1sp Web Edition

4.2.1.1 About the Figure 4.1

The Load Immediate (*li*) command is a pseudo-operation, which translates to: *addi rd, x0 immediate* register. The Move *mv* command is also a pseudo-operation, which translates to *addi rd, rs1, 0*. So, all this test does, is to load to register \$ra the immediate 0, then copy its value to register \$t5. Furthermore, it loads to register \$t4 the zero immediate again and to register \$gp¹ it loads the id number of the test (immediate 2). Finally, it conducts a branch, which ends to the fail section, if it is Taken. The branch simply compares the value of \$t5 and \$t4 registers, which should have the same value if everything went according to plan. If the test succeeds, then the next command that we should see being fetched, it would be the 00100093 *li*.

4.2.1.2 About the Figure 4.2

We will pair the simulation waves shown in the Figure with a Pipeline diagram to make it easier and more lucid.

Clock Cycle	<i>IF</i>	<i>ID</i>	<i>EXE</i>	<i>MEM</i>	<i>WB</i>
0	li ra, 0	-	-	-	-
1	mv t5, ra	li ra, 0	-	-	-
2	li t4, 0	mv t5, ra	li ra, 0	-	-
3	li gp, 2	li t4, 0	mv t5, ra	li ra, 0	-
4	bne t5, t4, 288	li gp, 2	li t4, 0	mv t5, ra	li ra, 0
5	li ra, 1	bne t5, t4, 288	li gp, 2	li t4, 0	mv t5, ra
6	-	li ra, 1	bne t5, t4, 288	li gp, 2	li t4, 0
7	-	-	li ra, 1	-	li gp, 2
8	-	-	-	li ra, 1	-

Notes:

→ The command is completed at the next clock cycle.

Table 4.1: Pipeline Diagram of test #2

The legend of the Figure 4.2 monitors some signals that we have selected for display. Note that at the bottom row, there is a signal dedicated to the \$gp register that gets updated with the value 2 at clock cycle 9, as we expected to happen. But it would get updated, even if the branch was taken and the test failed. What makes sure that the test is successful is that, when the branch reaches its final stage (the *EXE* stage), does not generate a *TAKEN* and therefore there is no *FLUSH* signal to empty the previous pipeline registers.

¹Global Pointer, this register gets updated with the test's id number. So if the register is written with all the mini-test ids we can tell that all the tests were successful

4.2.2 Problems Found and Solved by Testing

We will present two notable problems that were found while testing the complete RV32I design:

4.2.2.1 The Branch and the Forwards

While running the tests for the *BRANCH* instructions, we noticed some unusual behavior in our system, when we had a *BRANCH* command being followed by an *I-type* command. The problem was that, when the *BRANCH* entered the *EXE* stage (meaning that the *I-type* command was at the *ID* stage) the *ID* generated a *FWD_A* signal. After reviewing once more the Instruction Types (Figure 2.2) and breaking down the commands, we found out that the bits[11..7] of the *BRANCH* command were the same as the *rs1* bits of the *I-type* command, hence, the *FWD_A* signal would be generated. We did not take into consideration that NOT every instruction has a bit-field dedicated to *rd* register. To overcome this problem, we added one extra $2 \rightarrow 1$ multiplexer that can be seen at Figure 3.15 right below the *ID* module. This multiplexer is responsible for providing the address of the *x0* register¹, whenever a command, that does not have an *rd* register, enters the *EXE* stage. In that way, we prevent the false forwarding signal generation at *ID* stage.

4.2.2.2 The Forward Path C

Back at Section 3.6.4, we presented the main problem of the M4K Memory Block usage. Based on this issue, we analyzed even more the Forward C scenario, which has to do with the *MEM* stage. The issue is that, whenever a scenario that requires the activation of the C forward path occurs, the path activation signal (which is generated at *ID* stage) would have to pass through:

- The Pipeline Register B.
- The Pipeline Register C.
- The Registered Inputs of the D\$.

Furthermore, the signal should reach the *MEM* stage the moment that the *LOAD* command (which precedes the *STORE*) occupies it. The reason is that, the *LOAD* will read the D\$ word (which is also required by the *STORE*) and will send it directly to the Pipeline Register D. Then, 1 clock cycle later the signal will arrive, but the value that must be forwarded will not be available anymore. So, when the Forward C signal is generated, it skips the Pipeline Register B and goes directly to the Pipeline Register C, along with the result of the ALU for the *LOAD* command.

¹The register which is hardwired to the constant 0 and cannot be written

Chapter 5

Conclusion and Future Work

5.1 Future Work

Our goal was to design a fully operative and synthesizable ISA for the RISC-V architecture, the RV32I using *VHDL*. Every single component in our hierarchical design, was fully and thoroughly tested with both timing and functional simulations. Furthermore, the final design was successfully tested, by running the official RV32I tests found from the Berkeley's RISC-V GitHub repository. Those tests were 37 in total, one for each command of the ISA. Finally, after many hours of exhaustive designing, improvements and testing we can be sure, that our design is fully evaluated and fits the RV32I single-core standards.

The FPGA device in which our system's design was based on, was the Altera Cyclone II - DE 2 Board. So, the last piece of our work, was to find the lowest (worst case scenario) frequency that our core can achieve. After running the Quartus TimeQuest Analyzer timing simulator and using the slow model for the simulation, the worst-case frequency found is **29.48MHz**.

5.1 Future Work

On first look, we could try and change the architecture of some modules of our design to achieve a better frequency. For example, at *EXE* stage (section 3.3), for the Adder/Subtractor module, we used a ripple-carry architecture. The ripple carry is the most basic adder, made just by joining adders with no exercise on speed or hardware. Instead of this implementation, we could use a carry look-ahead adder or a Kogge-Stone adder. Furthermore, we could add a branch prediction unit and change the way we handle all the *BRANCH* cases. Last but not least, we wish to append the "M" and "F" ISA extensions on our design, since at the RV32I ISA every component in our CPU is scalable. All of those mentioned above, is left for Future Work due to time limitations.

Bibliography

- [1] The RISC-V Instruction Set Manual. Volume I: User-Level Isa (Version 2.1).
- [2] The RISC-V Instruction Set Manual. Volume II: Privileged Architecture (Version 1.10).
- [3] Altera: Internal Memory (RAM and ROM) User Guide.
- [4] Altera: Cyclone II Memory Blocks
- [5] Intel: Embedded Memory (RAM: 1-PORT, RAM: 2-PORT, ROM: 1-PORT, and ROM: 2-PORT) User Guide.
- [6] Design alternatives for barrel shifters.
- [7] Performance, analysis and comparison of digital adders.