Princess Sumaya University for Technology

King Abdullah II Faculty of Engineering

Computer Engineering Department

COMPUTER ARCHITECTURE II (EE: 22540)

VERILOG PROJECT

GROUP NO. 6

Authors:

Ahmad Arrabi	20170534
Khaled Sa’deh	20160779

Supervisors:

Dr. Amjed Al Mousa

Eng. Remah Bani Younisse

Jan 10, 2021

Abstract

A processor pipeline is the process of accumulating instructions from the processor through a pipeline. It allows storing and executing instructions in an orderly process, it is also attempts to keep every part of the processor busy by dividing instructions into a set of sequential steps called stages.

In this project we aimed to design a limited version MIPS processor, with a set of chosen instructions, some of which were custom made for the sake of the project, the design includes the pipeline of the processor, in addition to the Verilog code to get the processor to run, tested by some test benches.

TABLE OF CONTENTS

1. Introduction

The processor is an integrated circuit that performs a set of calculations that run a computer. It performs arithmetical, logical, input/output and other basic instructions that are passed from the operating system.
Pipelining increases the CPU instruction throughput, the number of instructions completed per unit of time, however; it does not reduce the execution time of an individual instruction.
In this project we will be aiming to design a simple processor with its pipeline, using Verilog language to code each block of the pipeline, after each block is coded individually, it will be connected to the rest of the blocks using a “Top” module, in which the whole image of the pipeline will show.
The module is going to be tested using a set of test benches provided with the project requirements, and some blocks will also be tested individually with separate test benches.

1.1 Objectives

The main objective of this project is to design a simple MIPS processor that works on a set of instructions, the design will include the pipeline stages, in addition to the code that runs this pipeline.

The processor should receive an instruction, decode it, execute it properly and saves the data calculated in its proper register, it should also allow forwarding and flush stages depending on the hazard detection unit.

2. Processor Design

2.1 IF Stage

IF stands for instruction fetch, and the main blocks of this stage are the Instruction Memory & the PC (Program Counter).

The instruction Memory will take the output the of PC as it’s input, then save it in the IF/ID register in order to send it to the next stage, The PC’s input will depend on a set of 2-to-1 Multiplexers, those multiplexers will choose their output depending on the Control Unit, which will send signals depending on the instruction decoded, those signals will decide whether the PC will be added by 4 (to go to the next instruction), or branch/jump to a whole new address if the instruction was branch or jump.

The IF/ID register will be separating the bits of the instruction received and sending them to the next stage.

2.2 ID Stage

The ID (Instruction decoding) stage is responsible for decoding the instruction received from the previous stage, putting the bits of the instruction received into their proper place, then passing it to the following stage.

The main blocks in this stage are the Register file and the Control Unit; the Register file contains the registers used in the processor, and it will receive its input from the IF/ID register, it will be responsible for assigning the registers that will be reading the data, and the registers that we will be writing the data onto (destination registers) using a set of 2-to-1 multiplexers to choose which register will take which role. Then sending the output to the ID/EXE register to pass the data onto the next stage.
The Control Unit will be taking its data as bits that were separated in the IF/ID register, the input

received will be deciding the operation of the instruction, the operation of the instruction depends mainly on the OP code and the Function ID of the instruction (or only the OP code in the I-type instructions).
The Control unit will then separate those bits even more and sending them to each assigned block to distinguish which wire will be used in this particular instruction, and which will not be needed, it'll also pass some of those bits to the ID/EXE register, which will pass them into the next stage.This stage also contains the Hazard detection unit, which will decide whether we are going to be stalling/flushing the previous stage, depending on the input received to it.

2.3 EXE Stage

The EXE (Execution) stage is responsible for doing the arithmetic operations of the data, whether it is to calculate a mathematical operation, choose which register is the destination register, or to calculate a branch instruction address, before passing it in the next stage to be saved in the memory. The main block in this stage is the ALU (Arithmetic Logic Unit).

In the ALU, registers are operated on using mainly their OP code and/or their Function ID, the ALU receives these data as bits from the Control Unit, to decide which instruction is going to be executed, the inputs of the ALU are registers decided from two 3-to-1 multiplexers, as well as two inputs that will act as most significant bits in case of a 64-bit operation. Giving two outputs, one is 32-bits, and the other is 64-bits. It will then pass the output to the EXE/MEM register.

This stage also contains an adder; this adder is used to calculate the branch address in case of a branch instruction.

The Forwarding Unit is responsible for detecting dependencies in the instructions, and then forwarding the needed data to a said instruction in order for it to be executed properly.

2.4 MEM Stage

The Memory stage is responsible for storing the data and the addresses while the program is being executed.

The Data Memory is the main block in this stage, it takes it's inputs from the EXE/MEM register as following: an input for the address, and two for the data being written, separated into 32-bits and 64-bits, we control which of those two inputs to take based on signals sent from the Control Unit. The output of the Data Memory is then passed to the Write Back stage using the MEM/WB register.

Not all data passing through this stage needs to be stored in the memory. Some data, like the destination register, is passed immediately to the Write Back stage.

2.5 WB Stage

During the WB (Write Back) stage, the instructions write their results into the register file in the ID stage and stored in the destination register.

The data being written back is passed into this stage using the MEM/WB register; the data is chosen using two 2-to-1 multiplexers, where one multiplexer is responsible for the 32-bits results while the other is responsible for the 64-bits result controlled by signals coming from the Control Unit.

2.6 Control Unit

The Control Unit is where each block receives its signals from, to decide which instruction is being executed, depending mainly on the OP code and the Function ID of the instruction.

The Control Unit gets its input from the IF/ID register, then, depending on the inputs, separate signals will be sent to multiplexers, the Register File, the ALU and the Data Memory to select the proper wires that should be used for the instruction being executed.

It is very crucial for the Control Unit to send all the signals correctly, as one wrong signal, even the 1-bit signals, can cause the whole pipeline to fail and the instruction to not work.

2.7 Hazard Detection Unit

The Hazard Detection Unit is the part of the processor responsible for detecting any hazards, like data dependencies, that may occur in the pipeline.

It takes several inputs from the Control Unit and the IF/ID register and will determine the output depending on the value of these input signals.

If the appropriate input signals are sent (their values are 1); the Hazard Detection Unit will send output signals that are going to stall the previous stage (IF), and a signal to stall the ID stage. This is done to ensure the integrity of the data in the case of a load instruction.

2.8 Forwarding Unit

The forwarding unit enables two kinds of forwarding, ALU-ALU forwarding and MEM-ALU forwarding. It compares accessed register indexes from different stages, if dependencies occur then it forwards the register values to get correct results.

In ALU-ALU forwarding, the forwarding unit compares the destination register in the memory stage with any of the two source registers (registers being read) in the execution stage. If they are equal, then ALU-ALU forwarding shall occur. The forwarding is done by sending a control signal, 1, to the multiplexers that determine what inputs the ALU.

In MEM-ALU forwarding, the forwarding unit compares the destination register in the write-back stage with any of the two source registers (registers being read) in the execution stage. If they are equal, then MEM-ALU forwarding shall occur. The forwarding is done by sending a control signal, 2, to the multiplexers that determine what inputs the ALU.

3. Design Analysis

3.1 Instruction Fetch

Our design was positive edge triggered. So, at each positive edge the PC sends a new value to the instruction memory to fetch a new instruction. This PC value is either the current value but added with 4 or a completely new value, in the case of a branch or a jump.

We designed our PC to have a control signal that determines if the pipeline was to be stalled. This signal, if activated, prevents the PC from fetching new data from its input, but recycles the old data that was from the previous stage. That way we insert a bubble into the pipeline (to fully include a bubble further analysis should be done in next stages). The control signal is received from the hazard detection unit in the decoding stage.

A problem we faced was to determine how to input the initial PC value. Notice that the PC's input depends on the previous stage. In this case, a previous stage does not exist! To solve this issue, we added a 2 to 1 multiplexer at the input of the PC. The first input of the MUX is the normal PC value that we mentioned earlier, PC+4 or a new PC value. The second input of the MUX is the initial PC value. The MUX is controlled by a control signal that is initialized with 1 and stays at this state for a whole clock cycle (100 time units). Then, switches to 0 forever.

3.2 Instruction Decoding

After separating the bits of the instruction in the IF/ID register, we input these assembled bits into different modules to implement the instruction. Op code, Function, Ft, and Fmt are sent to the control unit to generate the control signals.

To implement jump instructions, we calculated the jump address in the decoding stage. The first 26-bits of the instruction are shifted left 2 times. Then, concatenated with the last 4 bits of the PC value (PC value in the decoding stage refers to the next instruction, after adding 4).

Some instructions use sign or zero extended immediate values. These values are computed in simple modules that sign and zero extend an input of 16-bits to transform it to 32-bits.

Moreover, in this stage we pass the values of the following signals: Rs, Rt, and Rd indexes, Shamt, and the PC to the next stage, ID/EXE register.

A critical part of the decoding stage is reading from the register file. Multiple 2-1 MUXs were added to support the variety of options to read from in the register file. Floating point instructions read from the floating-point registers, so they do not use Rs as index but Fs. Consequently, we added a MUX to choose between the two, the control signal of the MUX is received from the control unit. Rt and Ft are composed of the same bits of the instruction, so they basically hold the same index, we do not need a MUX in this case. Another MUX we added is at the output of the register file, we read from registers Hi, Lo, and Rs. But we practically only need one value from them, this value is determined by the control signal HILO_read_control from the control unit.

There are many instructions that write in the register file. The first MUX we added regarding the writing process of the register file is to choose between the PC value, in the case of jump and link, or the write-back data. The output of this MUX is an input to another MUX to select between it and the Hi/Lo values, in the case of move from Hi/Lo. Regarding the index of the register that we will write on, we added a MUX to choose between Fd and Rt or Rd.

The hazard detection unit detects if there is a load-use hazard. If detected, it triggers control signals that flushes both the IF\ID register and the ID\EXE register. Also, the PC stall signal.

The ID stall signal is sent to a 3-input OR gate. The other two inputs of the gate are a signal indicating a branch and another one indicating a jump. If one of the three is triggered, then the MUX that chooses the control signal will select zeros instead of the output of the control unit, resulting in a bubble.

3.3 Control Unit

The Control Unit sends signals of various bits to block in the pipeline to be executed properly, depending on the instruction.

This unit has 4 inputs, all of which are coming from the IF/ID register as follows:

-Two 6-bit values; the OP_code and the Funct_ID. These two values come from the instruction format itself, as the OP_code represents the last 6 bits of the instruction, while the Funct_ID represents the first 6 bits of the instruction (in the case of R/FR type instructions), I/J/FI type instructions don't use Funct_ID.

- Two 5-bit values; Ft and Fmt, Fmt represents the 21st to the 25th bit of the instruction format in the FR/FI type instructions, while Ft represents the 16th to the 20th bits in the instruction format in the FR/FI type instructions.

The output of the Control Unit depends entirely on the input, as each of the outputs will have different values depending on what instruction is being executed. Each of the outputs has different purpose as follows:

- Store_FP is a 1-bit signal that sends 1 to the Register file in the case of a FP store instruction.
- ID_Flush & IF_Flush are two 1-bit signals that are sent in the case of a Jump/Jump and link execution, to flush the ID and IF stages.
- Store_Byte_control is a 1-bit signal that sends 1 to the Register file in case of a Store Byte instruction.
- float_control_read is a 1-bit signal that sends 1 to the Register file in case we need to read from a float register.
- HILO_read_control is a 2-bit signal that is sent to the Register File in case we need to read from HI/LO registers in the case of instructions that use them; such as move from Hi instruction.
- control_signal is a 36-bit signal that gets sent to the ID/EXE register; this one big signal was designed to make the code and overall pipeline more neat, instead of using plenty of smaller signals that are going to be used in the next stages and not the ID stage, this signal gets separated to it's proper smaller signals in each stage where those signals are going to be used in, where 23-bits are going to be used in the EXE stage, 3-bits are going to be used in the MEM stage, and 10-bits are going to used in the WB stage.

3.4 Register File

The register file in our design is negative edge triggered. It writes data on the negative edge of the clock and reads data in a combinational manner. Inside the register file there are two types of 32-bit registers, floating-point and integer. Each type has 32 registers. In addition to the Hi and Lo registers. The inputs of the register file are 4 indexes, 3 represent the registers we want to read and one for the write register. Also, the data that we want to write, a port for 32-bit data and another for 64-bit data. As well as control signals to determine how we will read and write in the register file.

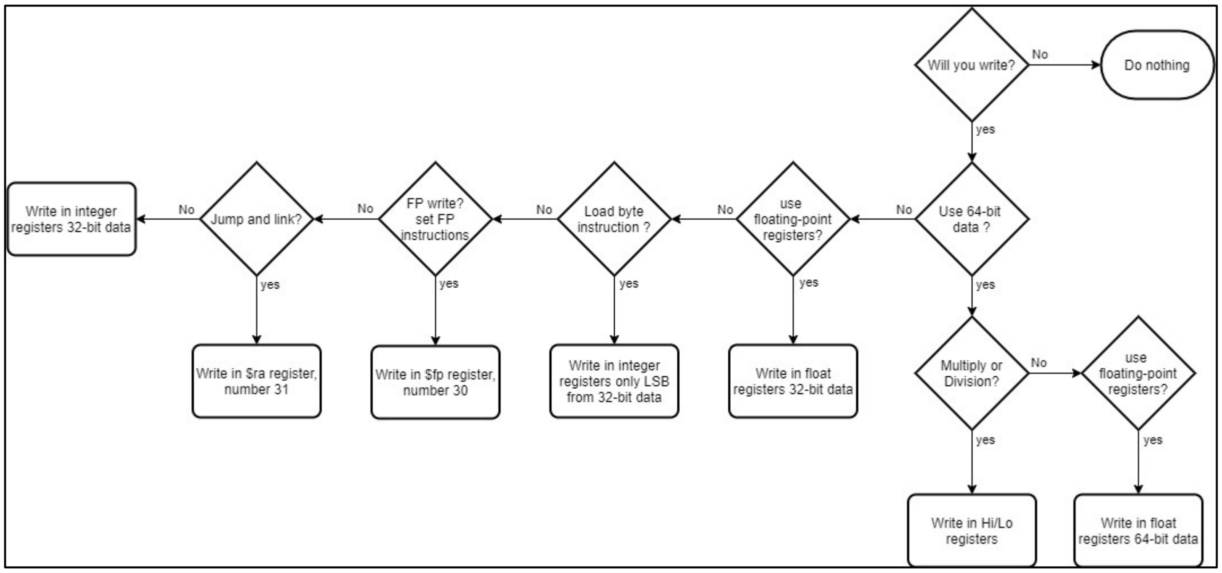
When writing in the register file we followed the flow chart illustrated in Fig. 1. Each decision is represented as a control signal from the control unit. Table 1 represents each decision and its corresponding control signal in the design.

An important point we have to stress on is the fact that the floating-point registers may represent double-precision values. We are dealing with big endian architecture, so values of the most significant words are saved in the smaller register index. If a 32-bit data is stored in a floating-point register, then the previous register, even register, is filled with zeros.

Decision	**Control signal**
Write?	RegWrite
Use 64-bit data?	Write32_64

Multiply or Division?	MulDiv_control
Floating-point registers?	float_control_write
Load byte?	Load_Byte_control
FP write?	FPwrite_control
Jump and link?	Jal_control

-Table1: control signals in the register file-


Will you write?
No
Do nothing
yes
Use 64-bit data ?
No
use floating-point registers?
No
Load byte instruction ?
No
FP write? set FP instructions
No
Jump and link?
No
Write in integer registers 32-bit data
yes
Write in $ra register, number 31
Write in $fp register, number 30
Write in integer registers only LSB from 32-bit data
Write in float registers 32-bit data
Multiply or Division?
use floating-point registers?
Write in Hi/Lo registers
Write in float registers 64-bit data

-Fig.1: Flow chart of the writing process in the register file-

The register file reads 5 register values, two of them are the Hi and Lo registers, and a flag that represents the frame pointer. It first outputs the Rd register value, some instructions use Rd in the execution stage. In the case of Store floating-point instruction, the register file outputs the Rs data from the integer registers while outputs Rt data from the floating-point registers.

Another special case we covered is the store byte instruction. The output Rt data is only the LSB of the integer register, the remaining bits are zeros.

If an instruction is a floating-point instruction, then we output both Rs and Rt values from the floating-point registers. In all other core instructions, the outputs are 32-bit data read from the integer registers.

Notice that there are extra outputs that represent the most significant words in both Rs and Rt. This was due to double-precision instructions that need 64-bits values in the execution stage.

3.5 Execution Stage

In the EXE stage, various control signals are going to be sent through the ID/EXE register as follows:

- The WB signal is a 10-bit signal that is going to be sent through the stage to the next register (EXE/MEM).
- The MEM signals is a 3-bit signal that is going to be sent through this stage to the next register (EXE/MEM) to be separated in the next stage.

– The EXE signal is an 23-bit signal that is going to be separated into smaller signals and sent to its proper block to execute the instruction. Where 11-bits will go to the ALU to determine the operation of the ALU, 1-bit signal will be sent to a multiplexer in the IF stage to determine the input to the Program counter (PC), and two 1-bit control signals will be sent to AND gates in the stage, the second input to these AND stages is the zero signal coming from the ALU, the result of these AND gates will be put through an OR gate, where if any of the gates produce a (1); we have a branch instruction, and thus the branch signal is sent to a multiplexer in the IF stage to determine the input of the PC, there's also another 1-bit signal that will send 1 to a multiplexer in the IF stage if the instruction happens to be jump.

We also have two 1-bit signals for the floating point conditioned branch, branch on FP true or branch on FP false; these signals will be sent to two AND gates; the other inputs of those AND gates is a signal coming from the Register File to determine whether we are dealing with floating point instructions, both of the outcomes from the AND gates will be put through an OR gate; where if one of them is true then we have a FP branch instruction, where we will send a branch signal to a multiplexer in the IF stage to determine the input of the PC.

Moreover, one 1-bit signal will be sent to a multiplexer in the EXE stage to determine whether the destination register is going to be Rd or Rt, this is done because some instructions, like the Add immediate, has the Rt register as the destination register rather than the Rd register.

The rest of the signals are related to the input of the ALU, they will be discussed in the ALU section below.

The EXE stage is also responsible for calculating the branch address to be sent to the PC, this is done by using an Adder that adds the PC address by an immediate number that will be shifted to the left by 2 to make the addition execute properly.

Overall, the EXE stage is where we calculate the branch address, send signals that determine if we have a branch/jump instruction, and do the mathematical and logical operations in the ALU.

3.6 ALU

The ALU performs simple addition, subtraction, multiplication, division, and logic operations, such as OR and AND.

In our design, the ALU takes 4 inputs, all of which are 32-bits, but two of these inputs are responsible for giving 64-bits output, these two inputs (IN_ALU_MSG1 & IN_ALU_MSG2) are only used for instructions that are 64-bits; like the FP add double & FP compare double.

The other two 32-bits inputs (IN_ALU_1 & IN_ALU_2) take their values from two 3-to-1 multiplexers; the first multiplexer is responsible for the value of IN_ALU_1, depending on the instruction; the output of the multiplexer is going to be either Rt/ Rs (chosen by a multiplexer controlled by Rs_Rt_control control signals), the 32-bits output of the ALU (taken from the EXE/MEM register), or the 32-bits data written back from the WB stage.

The second multiplexer is responsible for the value of IN_ALU_2, depending on the instruction; the output of the multiplexer is going to be either the output of a big 4-to-1 multiplexer that gives either Rt/Rd (chosen by a multiplexer controlled by the Rt_Rd_control control signal), the immediate value, the immediate value but zero extended, or the shift amount value. The other two possible outputs of the multiplexer are either the 32-bits output of the ALU (taken from the EXE/MEM register), or the 32-bits data written back from the WB stage.

After the input to the ALU is decided; an operation is performed on them depending on the instruction; the ALU knows which instruction to perform from the ALU_control control signal that is sent to it from the EXE control signal, this is a 12-bits signal that contains the OP code and the Function ID of the instruction, in the R and FR- type instructions we need the whole 12-bits of the signal (both the OP code and the Function ID) to determine which operation to perform. While the I, J and FI-type instructions need only the OP code (most significant 6-bits of the signal).

After the ALU takes the signal and performs the operation, it produces three outputs; one is 32-bits (OUT_ALU32), one is 64-bits (OUT_ALU64), and one is a 1-bit zero flag signal, ZF_ALU. If the output is 32-bits; the 64-bits output's value is going to be 0, if the output is 64-bits; the 32-bits output value is going to be 0. And if both the outputs are zero; the zero flag (ZF_ALU)'s value produces 1.

After the outputs are produced; the 32-bits and 64-bits outputs are sent to the EXE/MEM register to be passed into the next stage; while the zero flag value is sent to AND gates in the stage to determine the branch address in the case of branch equal / branch not equal instructions.

3.7 Data Memory

MIPS architecture is big-endian. A big-endian system stores the most significant byte at the smallest memory address and the least significant byte at the largest. E.g., storing the word ABCD in address 0 will result as in table 2 shown below (each letter is a byte). Reading from address 4 will result in data = XYZW

Address	Data
0	A
1	B
2	C
3	D
4	X
5	Y
6	Z
7	W
8	.
.	.
.	.
.	.
1023	.

-Table 2: Memory example-

The data memory has 3 control signals. One to indicate reading and one for writing, the third one is for writing 64-bit data. When reading from the memory, we input an address and two outputs are generated. The first output is the data in the memory address, while the second output is the data in the next address. This was implemented to read double values.

When writing to the memory, we only need the base address that we want to right on, regardless of the instruction. All store instructions are implemented the same way. We only need to identify if we write 32-bit or 64-bit data.

3.8 Write-Back Stage

The WB stage places the results back into the register file in the middle of the pipeline in the ID stage.

This stage contains two multiplexers; one for the 32-bit results and the other for the 64-bit results. Those 2-to-1 multiplexers pick between two inputs, one of the inputs comes directly from the ALU in the EXE stage, while the other comes from the Data Memory in the MEM stage. The multiplexers are controlled via control signals in the MEM/WB register, those control signals were originally one 10-bit signal, but were separated in the register, two of those signals (MemToReg64 & MemToReg) are responsible for controlling the output choice of the multiplexers. If the output is 64-bit it will go straight to the register file, if it was 32-bit then it will go to the register file, as well as to a multiplexer in the EXE stage where it might be picked as an input to the ALU.

One of the control signals (RegWrite_MEM_WB) will be sent to the Forwarding Unit, as well as the register file. This isn't the only wire in this stage that gets sent to the Forwarding Unit, as the (RegWr_WB) gets sent there, as well as to the Register File, to determine the address of the write register.

The rest of the control signals get sent to the ID stage, each of them has a particular purpose as follows:

- Jal_control: sends 1 to a multiplexer in the ID stage in the case of a jump and link instruction.
- HILO_write_control: sends 1 to a multiplexer in the ID stage in case the instruction needs to the use the HI/LO registers in the Register File.
- Write32_64: determines if we need to use a 32-bit register or an extra 32-bit register next to it in the case of instructions that have 64-bits outputs.
- float_control_write: sends 1 to the Register File in the case of a float instruction.
- MulDiv_control: sends 1 to the Register file in the case of a multiply/division instruction.
- FPwrite_control: sends 1 to the Register file in case we need to write in a float register.

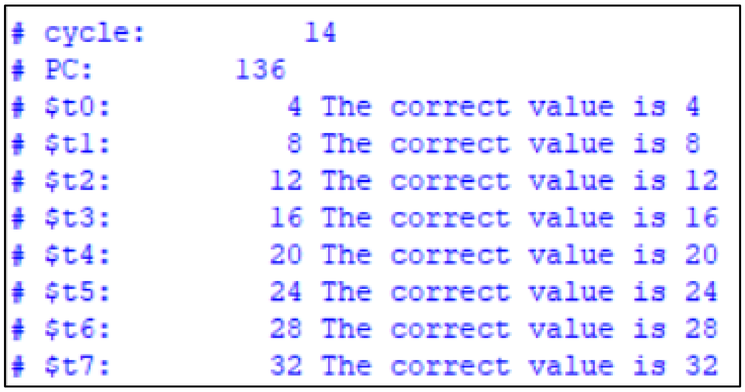
4. Tests runs and discussion

4.1 Test Run #1

The first testbench focused on load instructions. Loaded values were previously added to the data memory. The values were multiples of 4.

```
lw $t0,0($0)

lw $t1,4($0)

lw $t2,8($0)

lw $t3,12($0)

lw $t4,16($0)

lw $t5,20($0)

lw $t6,24($0)
```

lw $t7,28($0)



```
# cycle:         14
# PC:        136
# $t0:           4 The correct value is 4
# $t1:           8 The correct value is 8
# $t2:          12 The correct value is 12
# $t3:          16 The correct value is 16
# $t4:          20 The correct value is 20
# $t5:          24 The correct value is 24
# $t6:          28 The correct value is 28
# $t7:          32 The correct value is 32
```

-Fig 2: Test bench #1 output-

4.2 Test Run #2

The second testbench focused on arithmetic operations, without any forwarding.

addi $s1,$0,5

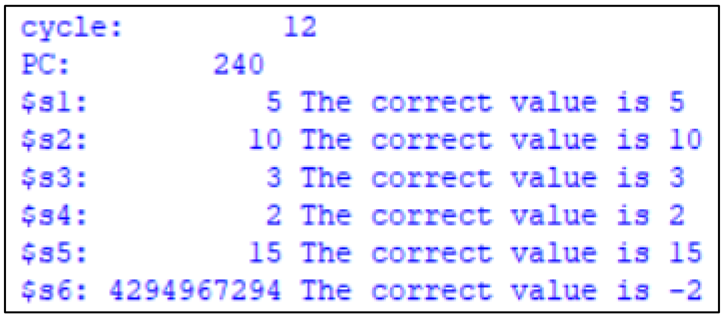
addi $s2,$0,10

addi $s3,$0,3

addi $s4,$0,2

add $s5,$s1,$s2

sub $s6,$s1,$s3



```
cycle:          12
PC:        240
$s1:           5 The correct value is 5
$s2:          10 The correct value is 10
$s3:           3 The correct value is 3
$s4:           2 The correct value is 2
$s5:          15 The correct value is 15
$s6: 4294967294 The correct value is -2
```

-Fig 3: Test bench #2 output-

4.3 Test Run #3

The third testbench focused on logical operations, without any forwarding.

addi $s1,$0,15

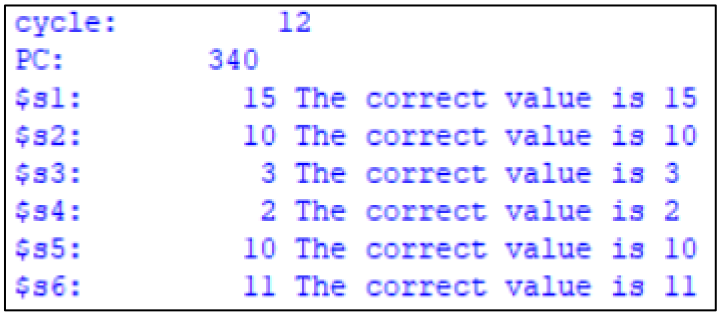
addi $s2,$0,10

addi $s3,$0,3

addi $s4,$0,2

and $s5,$s1,$s2

or $s6,$s2,$s3



```
cycle:          12
PC:         340
$s1:          15 The correct value is 15
$s2:          10 The correct value is 10
$s3:           3 The correct value is 3
$s4:           2 The correct value is 2
$s5:          10 The correct value is 10
$s6:          11 The correct value is 11
```

-Fig. 4: Test bench #3 output-

4.4 Test Run #4

The fourth test bench focused on store operations.

addi $s1,$0,15

addi $s2,$0,10

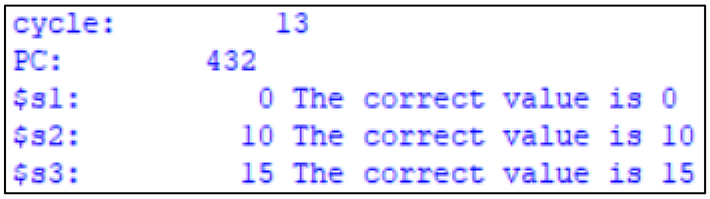
addi $s3,$0,3

sw $s1,100($0)

addi $s1,$0,0

addi $s2,$0,10

lw $s3,100($0)



```
cycle:          13
PC:         432
$s1:           0 The correct value is 0
$s2:          10 The correct value is 10
$s3:          15 The correct value is 15
```

-Fig. 5: Test bench #4 output-

4.5 Test Run #5

The fifth test bench focused on branch instructions, where branch was not taken.

addi $s1,$0,15

addi $s2,$0,10

addi $s3,$0,3

addi $s4,$0,2

beq $s1,$s2,-5

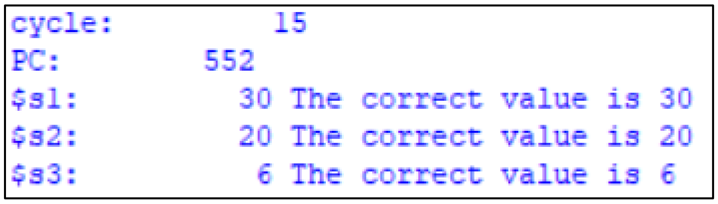
addi $s1,$0,30

addi $s2,$0,20

addi $s3,$0,6

addi $s3,$0,6

addi $s3,$0,6



```
cycle:          15
PC:         552
$s1:          30 The correct value is 30
$s2:          20 The correct value is 20
$s3:           6 The correct value is 6
```

-Fig. 6: Test bench #5 output-

4.6 Test Run #6

The sixth test bench implemented ALU-ALU forwarding.

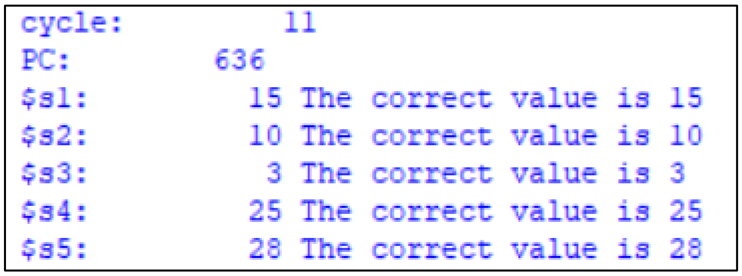
addi $s1,$0,15

addi $s2,$0,10

addi $s3,$0,3

add $s4,$s1,$s2

add $s5,$s4,$s3



```
cycle:          11
PC:         636
$s1:          15 The correct value is 15
$s2:          10 The correct value is 10
$s3:           3 The correct value is 3
$s4:          25 The correct value is 25
$s5:          28 The correct value is 28
```

-Fig. 7: Test bench #6 output-

4.6 Additional Test runs

4.6.1 ALU-ALU forwarding & MEM-ALU forwarding.

This test bench tested the ability to choose the most recent, updated, value of the destination register. In this case, both MEM-ALU and ALU-ALU forwarding occurred at the same cycle.

addi $s1,$0,15

addi $s2,$0,10

addi $s3,$0,3

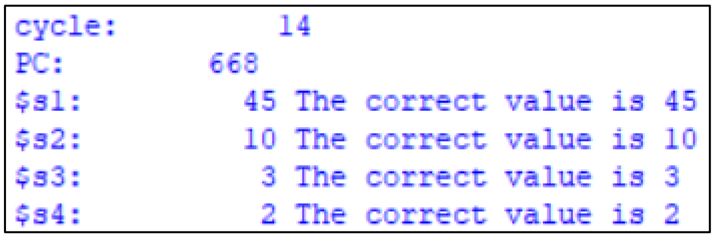
addi $s4,$0,2

add $s1,$s1,$s1

add $s1,$s1,$s2

add $s1,$s1,$s3

add $s1,$s1,$s4



```
cycle:          14
PC:         668
$s1:          45 The correct value is 45
$s2:          10 The correct value is 10
$s3:           3 The correct value is 3
$s4:           2 The correct value is 2
```

-Fig. 8: Test bench #7 output-

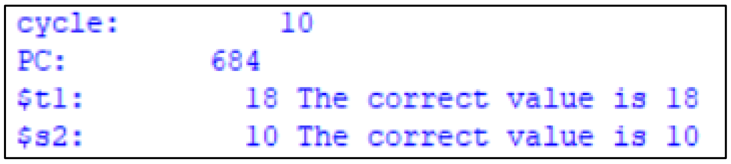
4.6.1 Load-use hazard

This test bench investigated the implementation of a bubble in the pipeline. Load-use data hazard occurrence led to the creation of the bubble.

addi $s2,$0,10

lw $t1,4($0)

add $t1,$t1,$s2



```
cycle:          10
PC:         684
$t1:          18 The correct value is 18
$s2:          10 The correct value is 10
```

-Fig. 9: Test bench #8 output-

5. Conclusion

The test results show that we have successfully designed a MIPS pipelined processor that runs the given set of instructions.
Even with the new instructions (not included in the original ISA) we were able to adjust the pipeline accordingly in order to execute those instructions properly. However, our design did not give much attention to resource utilization that improve the performance of the processor, our design was merely focused on the practicality and the proper execution of the instructions rather than speed and efficiency. E.g., the branch address could have been calculated in the decoding stage rather than the execution stage.

References

D. A. Patterson, J. L. Hennessy. Computer Organization and Design: The Hardware Software Interface, 5th Edition. (2013, Elsevier).