

ES 215 Computer Organization and Architecture Project
Report



Indian Institute of Technology, Gandhinagar

PIPELINE SIMULATOR

(<https://github.com/pps-19012/COA>)

April 26, 2022

Daniel Giftson 20110051

daniel.giftson@iitgn.ac.in

Patel Vrajesh 20110134

patel.vrajesh@iitgn.ac.in

Pushpendra Pratap Singh 20110151

pushpendra.pratap@iitgn.ac.in

R Yeeshu Dhurandhar 20110152

r.yeeshu@iitgn.ac.in

1. Abstract

In this project, we have implemented a pipeline simulator using Python. In addition to the five stages of pipeline development, we have also considered data hazard detection and its prevention using data forwarding. We have elaborated our project ideas as well as explained in brief on how our project has been implemented. We have explained the core experimental design of our project and tested our code and explained each and every possible output case. Finally, we have also thrown light on the improvements and the ideas on future work of this project in this report.

2. Introduction

- The goal of this project is to build a five-stage pipelined Discrete event simulator with hazard check functionality embedded in it. We also aim to come up with solutions to prevent data hazards which include data forwarding, etc.
- The rationale of this project is to learn how pipelining affects the complexity and performance of the processor. We also wanted to learn the effects of data hazards so as to come up with solutions to prevent them.
- A five-stage pipeline processor is a very effective processor in terms of performance as it executes the instructions in comparatively less time thus increasing the performance of the CPU. But due to the occurrence of hazards in executing the instructions, we won't end up getting the desired output. So as an engineer, it is important for us to come up with solutions to get rid of the hazards in order to execute the instructions properly as well as to maintain the performance of the processor. That's the very reason which has motivated us to take up this project.
- We have coded a generic Pipeline simulator in Python language with data hazard detection functionalities and we have added/coded data forwarding units to resolve the data hazards. We will be running the script for a set of instructions and analyzing how it is getting executed in a 5-stage pipelined manner, detecting data hazards and analyzing how data hazards are resolved using data forwarding.

3. Literature Review

- This problem of simulating a 5-stage pipeline that is capable of displaying the status of hardware of the pipeline model and detecting and resolving hazards was raised by **Andrea Spadaccini** (with a group of students of the University of Catania (Italy) developed the EduMIPS64 simulator).
- There are a few proposals(simulators) made in order to resolve these issues. One of

such simulators is the EduMIPS64 [1].

- **EduMIPS64 [1]:** EduMIPS64 simulator simulates a 5-stage pipeline CPU with a simple datapath display and supports data hazard detection. However, it doesn't support branch prediction and also non-pipelined CPU simulation. More importantly, it doesn't resolve hazards rather it just detects it.
- Since our pipeline simulator deals not only with the detection of hazards as in the case of the EduMIPS simulator, but also resolves them by forwarding, we considered our solution to be improving the already existing solution.

4. Project idea

Pipelining is the process of accumulating instruction from the processor through a pipeline. It provides for the systematic storage and execution of instructions.

It is a method of executing multiple instructions at the same time. A pipeline is divided into stages, each of which is linked to the next to form a pipe-like structure. Instructions come in from one end and leave from the other.

In this project, we have tried to simulate a realistic imitation of the pipelining execution using a Pipeline Simulator. Pipeline Simulator is a powerful computer tool used to simulate actual pipelined systems(processors). This simulation can be used to study various factors related to pipelining and how it gets affected due to various hazards. Hence, we intend to build a five-stage Pipeline Simulator for MIPS architecture.

The simulator takes the input as a text file where each line contains the following parameters format:

Program counter(in hexadecimal), Machine code(in hexadecimal), MIPS code

Here, the user must use a decoding/encoding unity to convert the instructions of their choice in the following format.

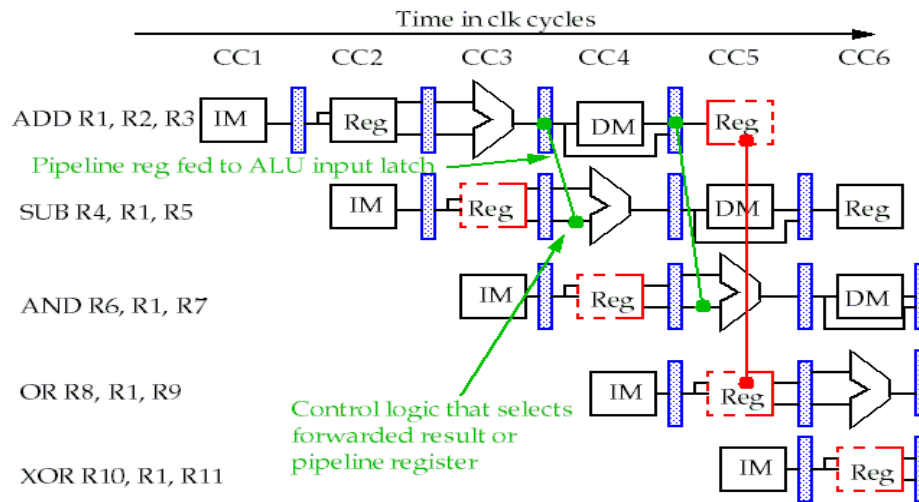


Figure : (Forwarding)

The simulator will consider potential hazards and use forwarding and other prevention techniques to avoid them. Hazard checks will be done at the Instruction fetch and

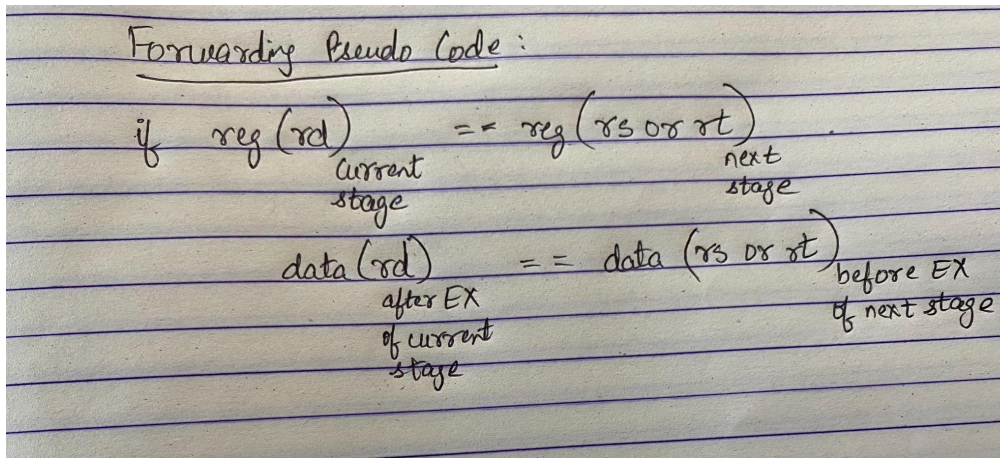


Figure : (Pseudo-code → Forwarding)

decode stages. However, the simulator will not keep track of data hazards from memory. Same as the specification used in 32 bit MIPS Architecture (i.e., regular instruction set, all instructions are 32-bit, three-operand arithmetical and logical instructions, 32 general-purpose registers of 32-bits each). The following are the instructions that we have covered in the project.

Instructions covered in the simulator
add
sub
addi
sll
and
or
slt
multu
mflo
beq
ori

Also note that to enable the branch instruction beq to work on our simulator, we have added two dummy instructions while giving input in the text file. When there is lw followed by add instruction (with RAW hazard) we have added one dummy instruction.

```
# beq $t1,$s1, M_done offset = 0x24
0x00000038 0x11310024 beq_$t1,$s1,M_done
0x0000003C 0x00000000 NOP
0x00000040 0x00000000 NOP
```

Figure : (Dummy Instructions as input)

5. Project Implementation.

Our approach to building the simulator was straightforward. We started with assigning and initializing the values of registers, latches, and control signals. For registers, we have used a 2D array to store the value and the serial number of the register. Similarly,

for latches and control signals we have used a 1D array for appropriate indices which correspond to the relevant stages in pipelining.

We have initialized the registers and the control signals based on the below datapath. For example, the control signal MemRead is initialized as a 1x3 vector since it controls the instruction decode(ID), execute(EX) and memory(MEM) stages.

Note:

- The control hazard can be tackled by giving two dummy inputs. However, the below explanation is for the data hazard case.
- The input should not contain any spaces in between them.
- There should be no line without # or instruction.
- The instructions should end with atleast 4 dummy lines for all the pipelining stages to complete properly.

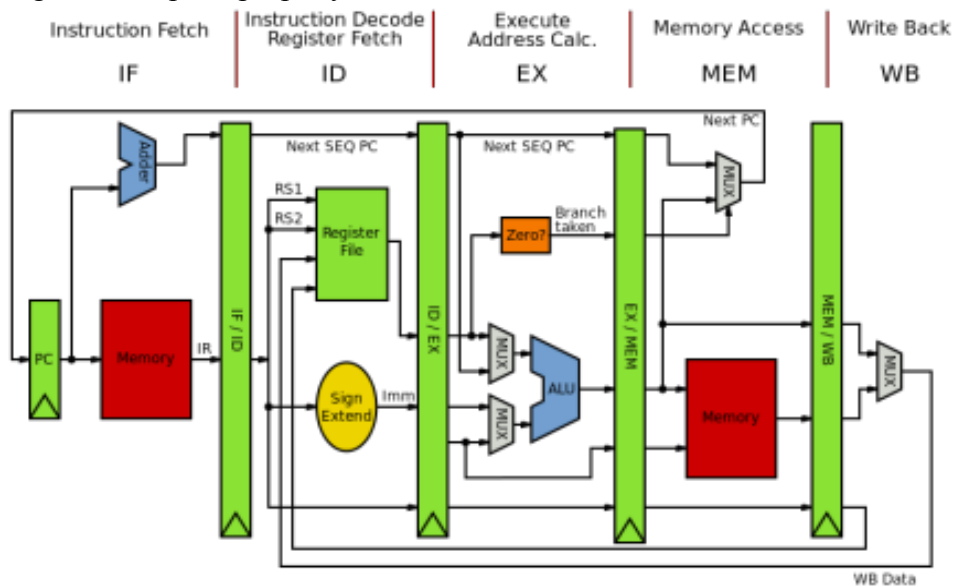


Figure : (Stages of Pipeline)

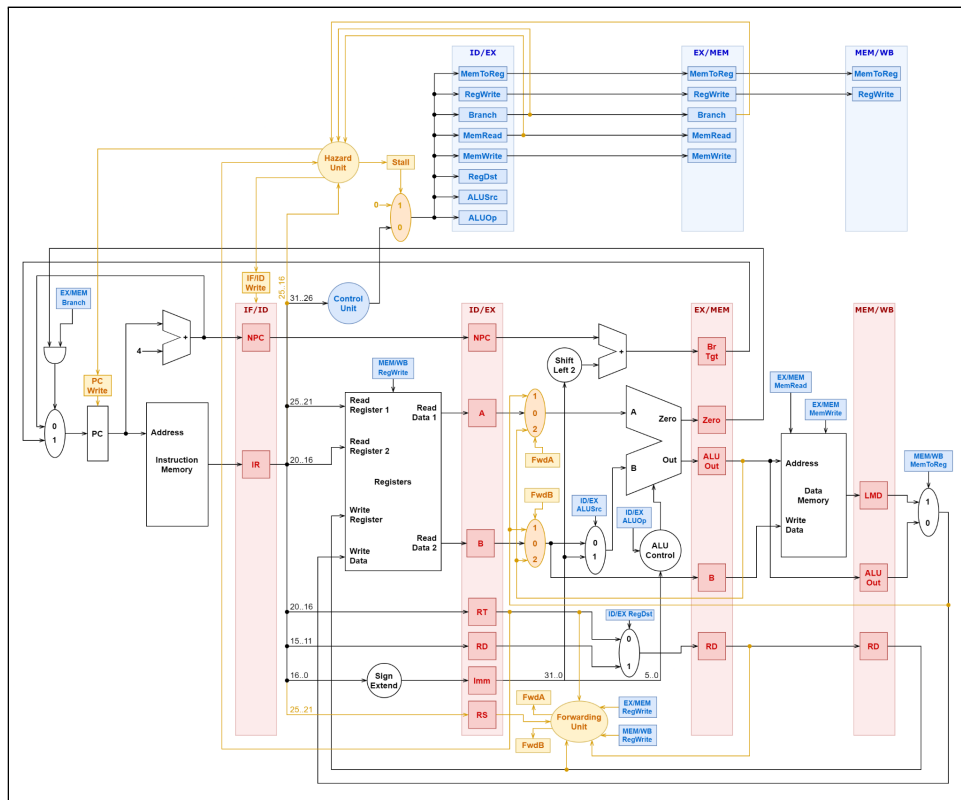


Image : (Datapath)

Next, we defined the basic instructions necessary for simulating the pipelining process, i.e. the read function, the print function, and the ALU control unit.

FETCH stage:

In the Fetch stage, the program records the location of the program counter, machine code, and MIPS instruction and prints the same.

DECODE stage:

Before decoding the instructions, it is essential to check for the hazards and first write-in register and then read. Depending upon the control signal (MemReg), it will either write-back memory data (in case of lw) or will write back the ALU result.

Next, the opcode values, rs, rt, rd, shamt, funct, etc are obtained by shifting the machine code instruction. Using the opcode, the program calculates the control signals and

updates accordingly. rs, rt, rd, and the control signals corresponding to the Decode stage are printed.

EXECUTE stage:

Using the control signal (ALUsrc) from the previous stage, the program reads accordingly if it is R-type or I-type instruction. Afterward, we check for the hazards present in the instruction. For hazard detection, we check

if RD of WB stage (RD[0]) == EX stage's Rs[2] or RT[2] then hazard is present

if RD of MEM stage (RD[1]) == EX stage's RS[2] or RT[2] then hazard is present

Thus, in the code/program we check the RD of both MEM and WB stages. Further, if the hazard is detected then we use forwarding unity to resolve the issue.

Case - (i) Checking RD of WB stage:

If the hazard is detected with RS, and if the detected instruction is I-type then we load the data from the ALU result, else we directly load the memory data. In this case, RS is forwarded from the WB stage. Similarly, if the hazard is detected with RT, then for I-type instruction register data is loaded else, memory data is loaded. In this case, RT is forwarded from the WB stage.

Case - (i) Checking RD of MEM stage:

If the hazard is detected with RS, then we load the data from the ALU result and RS is forwarded from the MEM stage. For the other case, RT is forwarded from the MEM stage.

Apart from hazard detection, the EX stage functions in a regular manner, calculating the required ALU operations or identifying the branch targets of instruction. In the end, the program prints values of relevant control signals, registers, and the result of branch operations (if present).

MEMORY stage:

The latch shifts and takes an appropriate value depending upon the control signals in this stage. Afterward, the results are latched into the relevant pipeline registers and it is printed.

WRITE BACK stage:

Here, the registers are updated by writing back for relevant instructions. At the end, all

the pipeline registers, control signals, and related latches are updated.

6. Testing and Experiments

For testing the code, the user has to input the instructions in the text file as seen in the below-mentioned images of input. Note that there cannot be an empty line in instruction since the line without '#' is considered a part of the code. Kindly note that the last 4 lines of the instruction have to be dummy lines for all the instructions to execute completely and keep PC within the index range.

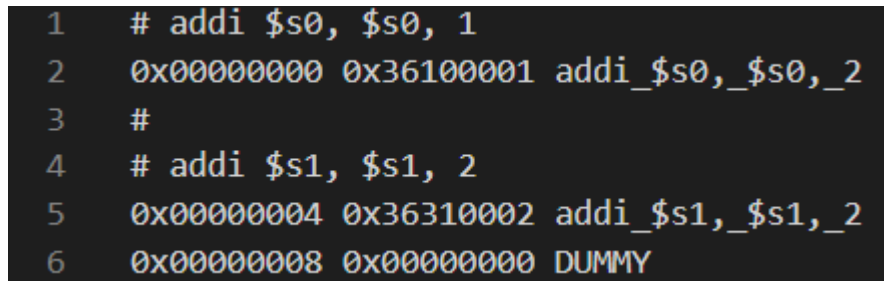
On experimenting, we got to know that on removing '_' from `addi_$s1,$s1,2`, and/or other instructions, the output does not represent the register used in the instruction.

In other words:

- If input instruction is of the form `addi_$s0,$s0,2`, the instruction after each stage is represented in the form Decoded: `addi_$s0,$s0,2`.
- If the input instruction is of the form `addi $s0 $s0 2`, the instruction after each stage is represented in the form Decoded: `addi`

There are two test cases possible:

- **Instructions with no hazard:**



```
1  # addi $s0, $s0, 1
2  0x00000000 0x36100001 addi_$s0,$s0,2
3  #
4  # addi $s1, $s1, 2
5  0x00000004 0x36310002 addi_$s1,$s1,2
6  0x00000008 0x00000000 DUMMY
```

Figure : (No Hazard Input)

```

/F\ Fetched = addi_$s0,$s0,_2 -----Clock cycle = 1
IF/ID ----- To decode = 0x00000000 fetched = 0x36100001
-----
/F\ Fetched = addi_$s1,$s1,_2 -----Clock cycle = 2
IF/ID ----- To decode = 0x36100001 fetched = 0x36310002
Decoded: addi_$s0,$s0,_2

ID/EX | for current execute= [0, 0, 0] result of current decode =[0, 0, 1]
RS = [0, 0, 0, 16]
RT = [0, 0, 0, 16]
RD = [0, 0, 0, 0]
MemRead = [0, 0, 0]
MemWrite = [0, 0, 0, 0]
MemToReg = [0, 0, 0, 0]
RegWrite = [0, 0, 0, 1]
regDst = [0, 0, 0, 0]
-----
/F\ Fetched = DUMMY -----Clock cycle = 3
IF/ID ----- To decode = 0x36310002 fetched = 0x00000000
Decoded: addi_$s1,$s1,_2

ID/EX | for current execute= [0, 0, 1] result of current decode =[0, 0, 2]
RS = [0, 0, 16, 17]
RT = [0, 0, 16, 17]
RD = [0, 0, 0, 0]
MemRead = [0, 0, 0]
MemWrite = [0, 0, 0, 0]
MemToReg = [0, 0, 0, 0]
RegWrite = [0, 0, 1, 1]
regDst = [0, 0, 0, 0]
AluSrc = [1, 1]
ALUOp = [3, 3]
funct = [1, 2]
shamt = [0, 0]
ALU SOURCES = 0 1
Branch Not Taken
Executed = addi_$s0,$s0,_2

EX/MEM | for current MEM= [0, 0] result of current execute = [1, 0]
Zero = 0 Next_PC = 12 Branch = 0
PC_MUX1= 12 PC+4 = 12 Branch_Target= 16

```

Figure : (Clock cycle 1,2,3 output)

```

/F\ Fetched = DUMMY -----Clock cycle = 4
IF/ID ----- To decode = 0x00000000 fetched = 0x00000000
Decoded: DUMMY

ID/EX | for current execute= [0, 0, 2] result of current decode =[0, 0, 0]
RS = [0, 16, 17, 0]
RT = [0, 16, 17, 0]
RD = [0, 0, 0, 0]
MemRead = [0, 0, 0]
MemWrite = [0, 0, 0, 0]
MemToReg = [0, 0, 0, 0]
RegWrite = [0, 1, 1, 1]
regDst = [0, 0, 0, 1]
AluSrc = [1, 0]
ALUOp = [3, 2]
funct = [2, 0]
shamt = [0, 0]
ALU SOURCES = 0 2
Branch Not Taken
Executed = addi_$s1, $s1, 2

EX/MEM | for current MEM= [1, 0] result of current execute = [2, 0]
Zero = 0 Next_PC = 16 Branch = 0
PC_MUX1= 16 PC+4 = 16 Branch_Target= 24
Memory = addi_$s0, $s0, 2

MEM/WB | for current WB= [0, 0] result of current MEM load = [0, 1]

```

Figure : (Clock cycle 4 output)

```

/F\ Fetched = DUMMY -----Clock cycle = 5
IF/ID ----- To decode = 0x00000000 fetched = 0x00000000
Decoded: DUMMY

ID/EX | for current execute= [0, 0, 0] result of current decode =[0, 0, 0]
RS = [16, 17, 0, 0]
RT = [16, 17, 0, 0]
RD = [0, 0, 0, 0]
MemRead = [0, 0, 0]
MemWrite = [0, 0, 0, 0]
MemToReg = [0, 0, 0, 0]
RegWrite = [1, 1, 1, 1]
regDst = [0, 0, 1, 1]
AluSrc = [0, 0]
ALUOp = [2, 2]
funct = [0, 0]
shamt = [0, 0]
ALU SOURCES = 0 0
SOURCES_UPDATED ALU SOURCES = 0 0
Branch Not Taken
Executed = DUMMY

EX/MEM | for current MEM= [2, 0] result of current execute = [0, 0]
Zero = 1 Next_PC = 20 Branch = 0
PC_MUX1= 20 PC+4 = 20 Branch_Target= 20
Memory = addi_$s1, $s1, 2

MEM/WB | for current WB= [0, 1] result of current MEM load = [0, 2]
WriteBack = addi_$s0, $s0, 2

$zero= 0x00000000 $at= 0x00000000
$vo= 0x00000000 $v1= 0x00000000
$a0= 0x00000000 $a1= 0x00000000
$a2= 0x00000000 $a3= 0x00000000
$t0= 0x00000000 $t1= 0x00000000
$t2= 0x00000000 $t3= 0x00000000
$t4= 0x00000000 $t5= 0x00000000
$t6= 0x00000000 $t7= 0x00000000
$s0= 0x00000001 $s1= 0x00000000
$s2= 0x00000000 $s3= 0x00000000
$s4= 0x00000000 $s5= 0x00000000
$s6= 0x00000000 $s7= 0x00000000
$t8= 0x00000000 $t9= 0x00000000
$ko= 0x00000000 $k1= 0x00000000
$gp= 0x00000000 $sp= 0x00000000
$fp= 0x00000000 $ra= 0x00000000

```

Figure : (Clock cycle 5 output)

```

/F\ Fetched = DUMMY -----Clock cycle = 6
IF/ID ----- To decode = 0x00000000 fetched = 0x00000000
Decoded: DUMMY

ID/EX | for current execute= [0, 0, 0] result of current decode =[0, 0, 0]
RS = [17, 0, 0, 0]
RT = [17, 0, 0, 0]
RD = [0, 0, 0, 0]
MemRead = [0, 0, 0]
MemWrite = [0, 0, 0, 0]
MemToReg = [0, 0, 0, 0]
RegWrite = [1, 1, 1, 1]
regDst = [0, 1, 1, 1]
RS forwarded from MEM stage. RS = 0
RT forwarded from MEM stage. RT = 0
AluSrc = [0, 0]
ALUOp = [2, 2]
funct = [0, 0]
shamt = [0, 0]
ALU SOURCES = 0 0
SOURCES UPDATED___ALU SOURCES = 0 0
Branch Not Taken
Executed = DUMMY

EX/MEM | for current MEM= [0, 0] result of current execute = [0, 0]
Zero = 1 Next_PC = 24 Branch = 0
PC_MUX1= 24 PC+4 = 24 Branch_Target= 24
Memory = DUMMY

MEM/WB | for current WB= [0, 2] result of current MEM load = [0, 0]
WriteBack = addi_$s1,$s1,_2

$zero= 0x00000000 $at= 0x00000000
$v0= 0x00000000 $v1= 0x00000000
$a0= 0x00000000 $a1= 0x00000000
$a2= 0x00000000 $a3= 0x00000000
$t0= 0x00000000 $t1= 0x00000000
$t2= 0x00000000 $t3= 0x00000000
$t4= 0x00000000 $t5= 0x00000000
$t6= 0x00000000 $t7= 0x00000000
$s0= 0x00000001 $s1= 0x00000002
$s2= 0x00000000 $s3= 0x00000000
$s4= 0x00000000 $s5= 0x00000000
$s6= 0x00000000 $s7= 0x00000000
$t8= 0x00000000 $t9= 0x00000000
$k0= 0x00000000 $k1= 0x00000000
$gp= 0x00000000 $sp= 0x00000000
$fp= 0x00000000 $ra= 0x00000000

```

Figure: (Clock cycle 6)

- Instructions with hazards:

```

1  # addi $t0, $zero, 2
2  0x00000000 0x01004024 and_$t0,$t0,_0
3  #
4  # and $t1, $t0, $zero
5  0x00000004 0x01004024 and_$t0,$t0,_0
6  0x00000003 0x00000000 Dummy

```

Figure : (Input with RAW Hazard)

```

/F\ Fetched = and_$t0,$t0,_0 -----Clock cycle = 1
IF/ID ----- To decode = 0x00000000 fetched = 0x01004024
-----
/F\ Fetched = and_$t0,$t0,_0 -----Clock cycle = 2
IF/ID ----- To decode = 0x01004024 fetched = 0x01004024
Decoded: and_$t0,$t0,_0

ID/EX | for current execute= [0, 0, 0] result of current decode =[0, 0, 16420]
RS = [0, 0, 0, 8]
RT = [0, 0, 0, 0]
RD = [0, 0, 0, 8]
MemRead = [0, 0, 0]
MemWrite = [0, 0, 0, 0]
MemToReg = [0, 0, 0, 0]
RegWrite = [0, 0, 0, 1]
regDst = [0, 0, 0, 1]
-----
/F\ Fetched = Dummy -----Clock cycle = 3
IF/ID ----- To decode = 0x01004024 fetched = 0x00000000
Decoded: and_$t0,$t0,_0

ID/EX | for current execute= [0, 0, 16420] result of current decode =[0, 0, 16420]
RS = [0, 0, 8, 8]
RT = [0, 0, 0, 0]
RD = [0, 0, 8, 8]
MemRead = [0, 0, 0]
MemWrite = [0, 0, 0, 0]
MemToReg = [0, 0, 0, 0]
RegWrite = [0, 0, 1, 1]
regDst = [0, 0, 1, 1]
AluSrc = [0, 0]
ALUOp = [2, 2]
funct = [36, 36]
shamt = [0, 0]
ALU SOURCES = 0 0
Branch Not Taken
Executed = and_$t0,$t0,_0

EX/MEM | for current MEM= [0, 0] result of current execute = [0, 0]
Zero = 1 Next_PC = 12 Branch = 0
PC_MUX1= 12 PC+4 = 12 Branch_Target= 65692

```

Figure : (Clock cycle 1,2,3 output)

```

/F\ Fetched = Dummy -----Clock cycle = 4
IF/ID ----- To decode = 0x00000000 fetched = 0x00000000
Decoded: Dummy

ID/EX | for current execute= [0, 0, 16420] result of current decode =[0, 0, 0]
RS = [0, 8, 8, 0]
RT = [0, 0, 0, 0]
RD = [0, 8, 8, 0]
MemRead = [0, 0, 0]
MemWrite = [0, 0, 0, 0]
MemToReg = [0, 0, 0, 0]
RegWrite = [0, 1, 1, 1]
regDst = [0, 1, 1, 1]
AluSrc = [0, 0]
ALUOp = [2, 2]
funct = [36, 0]
shamt = [0, 0]
ALU SOURCES = 0 0
Branch Not Taken
Executed = and_$t0,$t0,_0

EX/MEM | for current MEM= [0, 0] result of current execute = [0, 0]
Zero = 1 Next_PC = 16 Branch = 0
PC_MUX1= 16 PC+4 = 16 Branch_Target= 65696
Memory = and_$t0,$t0,_0

MEM/WB | for current WB= [0, 0] result of current MEM load = [0, 0]

```

Figure : (Clock cycle 4 output)

```

/F\ Fetched = Dummy -----Clock cycle = 5
IF/ID ----- To decode = 0x00000000 fetched = 0x00000000
Decoded: Dummy

ID/EX | for current execute= [0, 0, 0] result of current decode =[0, 0, 0]
RS = [8, 8, 0, 0]
RT = [0, 0, 0, 0]
RD = [8, 8, 0, 0]
MemRead = [0, 0, 0]
MemWrite = [0, 0, 0, 0]
MemToReg = [0, 0, 0, 0]
RegWrite = [1, 1, 1, 1]
regDst = [1, 1, 1, 1]
AluSrc = [0, 0]
ALUOp = [2, 2]
funct = [0, 0]
shamt = [0, 0]
ALU SOURCES = 0 0
SOURCES UPDATED___ALU SOURCES = 0 0
Branch Not Taken
Executed = Dummy

EX/MEM | for current MEM= [0, 0] result of current execute = [0, 0]
Zero = 1 Next_PC = 20 Branch = 0
PC_MUX1= 20 PC+4 = 20 Branch_Target= 20
Memory = and_$t0,$t0,_0

MEM/WB | for current WB= [0, 0] result of current MEM load = [0, 0]
WriteBack = and_$t0,$t0,_0

$zero= 0x00000000 $at= 0x00000000
$v0= 0x00000000 $v1= 0x00000000
$a0= 0x00000000 $a1= 0x00000000
$a2= 0x00000000 $a3= 0x00000000
$t0= 0x00000000 $t1= 0x00000000
$t2= 0x00000000 $t3= 0x00000000
$t4= 0x00000000 $t5= 0x00000000
$t6= 0x00000000 $t7= 0x00000000
$s0= 0x00000000 $s1= 0x00000000
$s2= 0x00000000 $s3= 0x00000000
$s4= 0x00000000 $s5= 0x00000000
$s6= 0x00000000 $s7= 0x00000000
$t8= 0x00000000 $t9= 0x00000000
$k0= 0x00000000 $k1= 0x00000000
$gp= 0x00000000 $sp= 0x00000000
$fp= 0x00000000 $ra= 0x00000000

```

Figure : (Clock cycle 5 output)

```

/F\ Fetched = Dummy -----Clock cycle = 6
IF/ID ----- To decode = 0x00000000 fetched = 0x00000000
Decoded: Dummy

ID/EX | for current execute= [0, 0, 0] result of current decode =[0, 0, 0]
RS = [8, 0, 0, 0]
RT = [0, 0, 0, 0]
RD = [8, 0, 0, 0]
MemRead = [0, 0, 0]
MemWrite = [0, 0, 0, 0]
MemToReg = [0, 0, 0, 0]
RegWrite = [1, 1, 1, 1]
regDst = [1, 1, 1, 1]
RS forwarded from MEM stage. RS = 0
RT forwarded from MEM stage. RT = 0
AluSrc = [0, 0]
ALUOp = [2, 2]
funct = [0, 0]
shamt = [0, 0]
ALU SOURCES = 0 0
SOURCES UPDATED___ALU SOURCES = 0 0
Branch Not Taken
Executed = Dummy

EX/MEM | for current MEM= [0, 0] result of current execute = [0, 0]
Zero = 1 Next_PC = 24 Branch = 0
PC_MUX1= 24 PC+4 = 24 Branch_Target= 24
Memory = Dummy

MEM/WB | for current WB= [0, 0] result of current MEM load = [0, 0]
WriteBack = and_$t0,_$t0,_0

$zero= 0x00000000 $at= 0x00000000
$v0= 0x00000000 $v1= 0x00000000
$a0= 0x00000000 $a1= 0x00000000
$a2= 0x00000000 $a3= 0x00000000
$t0= 0x00000000 $t1= 0x00000000
$t2= 0x00000000 $t3= 0x00000000
$t4= 0x00000000 $t5= 0x00000000
$t6= 0x00000000 $t7= 0x00000000
$s0= 0x00000000 $s1= 0x00000000
$s2= 0x00000000 $s3= 0x00000000
$s4= 0x00000000 $s5= 0x00000000
$s6= 0x00000000 $s7= 0x00000000
$t8= 0x00000000 $t9= 0x00000000
$k0= 0x00000000 $k1= 0x00000000
$gp= 0x00000000 $sp= 0x00000000
$fp= 0x00000000 $ra= 0x00000000

```

Figure : (Clock cycle 6 output)


```

-----
/F\ Fetched = Dummy -----Clock cycle = 7
IF/ID ----- To decode = 0x00000000 fetched = 0x00000000
Decoded: Dummy

ID/EX | for current execute= [0, 0, 0] result of current decode =[0, 0, 0]
RS = [0, 0, 0, 0]
RT = [0, 0, 0, 0]
RD = [0, 0, 0, 0]
MemRead = [0, 0, 0]
MemWrite = [0, 0, 0, 0]
MemToReg = [0, 0, 0, 0]
RegWrite = [1, 1, 1, 1]
regDst = [1, 1, 1, 1]
RS forwarded from WB stage. RS = 0
RT forwarded from WB stage. RT = 0
RS forwarded from MEM stage. RS = 0
RT forwarded from MEM stage. RT = 0
AluSrc = [0, 0]
ALUOp = [2, 2]
funct = [0, 0]
shamt = [0, 0]
ALU SOURCES = 0 0
SOURCES UPDATED___ALU SOURCES = 0 0
Branch Not Taken
Executed = Dummy

EX/MEM | for current MEM= [0, 0] result of current execute = [0, 0]
Zero = 1 Next_PC = 28 Branch = 0
PC_MUX1= 28 PC+4 = 28 Branch_Target= 28
Memory = Dummy

MEM/WB | for current WB= [0, 0] result of current MEM load = [0, 0]
WriteBack = Dummy

$zero= 0x00000000 $at= 0x00000000
$v0= 0x00000000 $v1= 0x00000000
$a0= 0x00000000 $a1= 0x00000000
$a2= 0x00000000 $a3= 0x00000000
$t0= 0x00000000 $t1= 0x00000000
$t2= 0x00000000 $t3= 0x00000000
$t4= 0x00000000 $t5= 0x00000000
$t6= 0x00000000 $t7= 0x00000000
$s0= 0x00000000 $s1= 0x00000000
$s2= 0x00000000 $s3= 0x00000000
$s4= 0x00000000 $s5= 0x00000000
$s6= 0x00000000 $s7= 0x00000000
$t8= 0x00000000 $t9= 0x00000000
$k0= 0x00000000 $k1= 0x00000000
$gp= 0x00000000 $sp= 0x00000000

```

Figure : (Clock cycle 7 output)

7. Conclusion

The Pipeline Simulator takes the input(instructions) as a text file in the format:

Program counter(in hexadecimal) Machine code(in hexadecimal) MIPS code.

The output it returns is the value of all the registers, control signals, etc. as shown in the image below.

The Simulator can detect the potential data Hazards and use Forwarding to counter the same. Our model was mainly based on the datapath shown in one of the images. The simulator tackles the data hazard with the use of bypassing (in case of RAW) hazard that does not include lw followed by add kind of an instruction where we need at least a stall to make it executable. To make the simulator working for branch instructions, one has to add two dummy instructions in the text file that contains input. In a similar way, to make lw followed by add instruction (one containing RAW hazard) executable, we have to add one dummy instruction in between them and use forwarding thereafter.

8. Improvements and Ideas for future work:

We were able to make a simulator that can detect the control hazard and also tells after execution whether it takes a branch or not. But we were not able to resolve this hazard. Theoretically speaking, we have to flush the two instructions that follow the branch instruction. But it seems too difficult for us to flush the instruction when it is written in the form of the code(hence we have added dummy instructions in the .txt file which contains instructions). While we are decoding the branch instruction, the next instruction is being fetched. Now, once it is being fetched, there are two possible causes. One, if the branch is taken then it won't create a problem but if the branch is not taken then we have to flush that instruction(which is a challenging task to implement). Also, one can implement a branch predictor for better performance of the pipeline. One more area wherein there requires some research work is a structural hazard. If one can implement a simulator that can operate based on the number of serial reads and write ports, then it would be of great utility in real-life applications.

If we would have done a little better time management, we would want to build a User Interface that shows the instructions in a pipelined format. The number of instructions can be increased to many more. However, branch instructions would have still created a problem.

References:

- [1] Andrea Spadaccini, *EduMIPS64 Documentation, Release 1.2.10*,
<https://readthedocs.org/projects/edumips64/downloads/pdf/latest/#:~:text=EduMIPS64%20is%20developed%20by%20a,differences%20between%20the%20two%20simulators.>
- [2] <https://github.com/KanegaeGabriel/mips-pipeline-simulator>
- [3] <https://github.com/GeorgeSaman>
- [4] <https://github.com/daniel4lee/MIPS-Pipeline-Simulator>