Project Report

(Github link)

MIPS PIPELINE SIMULATOR WITH HAZARDS CHECK

April 26, 2022

Dhyeykumar Thummar

dhyeykumar.thummar@iitgn.ac.in

Haikoo Ashok Khandor

haikoo.ashok@iitgn.ac.in

Meet Vankar

vankarmeet@iitgn.ac.in

Shruhrid Banthia

shruhrid.banthia@iitgn.ac.in

Indian Institute of Technology, Gandhinagar

1 Abstract

Pipelining is the process of ordering the instructions corresponding to different stages of work performed on the instruction. It is an implementation technique where multiple instructions are executed parallelly. Today it has become an essential way of designing fast CPU's. Although there exists a seven stage and as high as twenty stage pipeline design, five stage pipeline is commonly used to teach pipelining. But there exist many data, structural and control dependencies that come with pipelining. These dependencies may result in hazards which cause problems and need to be tackled. In this project, we implemented the five stage MIPS pipeline design and introduced stalls wherever it was necessary. We observed the way the pipeline handled these hazards.

2 Introduction

- **The goal of this project:**

  Through this project we aim to design a five-stage pipeline MIPS simulator. The pipeline should be able to detect hazards like data and control hazards. Our simulator shall not track the data hazards and use forwarding to avoid them wherever possible. Other hazards will invite penalties in the form of stalls in the pipeline. The user will input a number of instructions and the program shall output the instructions that were run after introducing stalls in the pipeline. The program outputs the clock cycles for each of the instructions showcasing the stages IF, ID, EX, MEM and WB and stalls if any. The simulator will support various R-type, I-type and J-type instructions.

- **The rationale of the project**

  The introduction of pipelining has greatly increased the processor clock speed rates by increasing the instructions throughput rate. If it is a k stage pipeline, k different instructions are able to run at the same time i.e., parallelly which increases the efficiency of the processor. However, this might result in different instructions using the registers, be it reading data from memory, storing data to memory, writing data to registers, reading it from the registers at the same time. Unless the register has multiple ports for read and write of the data, the execution of the processor modelled in the form of a pipeline simulator will fail since it will not be able to execute all the instructions. Hence, we need to add stalls into the simulator but this reduces the instruction throughput rate and the processor speed. Thus, it becomes necessary to detect these hazards, minimise the number of stalls and implement an optimised five-stage MIPS pipelined simulator.

- **The importance of the project**

  Through the MIPS pipeline simulator, we are able to compute and execute instructions in a parallel way in less time. Using the pipeline simulator, we can learn how it might respond to different hazards posed by instructions where they occur individually or together. The ability of the simulator to deal with these hazards tells us about its effectiveness and compatibility.

3 Literature Review

MARS Simulator

In this research paper, the authors have presented and implemented a Microprocessor without Interlocked Pipeline Stages (MIPS) pipelined simulator built on top of the MIPS Assembler and Runtime Simulator (MARS) as a plug-in.The tool also provides additional functionality as such Branch Prediction for better understanding. Many available MIPS simulators do not support branch prediction which serves as an important concept. A branch predictor can be a great visualiser to learn about important concepts.

It supports pipeline instruction execution with the detection of hazards and eliminating them. It supports branch prediction technique which is a key differentiator from other softwares/simulators. Two types of branch prediction have been implemented namely static and dynamic branch prediction. A separate Pipeline simulator and branch explainer is another feature which contains the five stage pipeline simulator and the branch predictor.

There are many key features implemented in the MARS simulator. Our solution is a subset to this and in order to better improve our solution, we can take advantage of/inspiration from this simulator for branch predictors, etc.

Citation: 3. D. X. Lim and K. G. Smitha, "Pipelined MIPS Simulation: A plug-in to MARS simulator for supporting pipeline simulation and branch prediction," *2019 IEEE International Conference on Engineering, Technology and Education (TALE)*, 2019, pp. 1-7, doi: 10.1109/TALE48000.2019.9225934.

EduMIPS64 Simulator:

It was developed as a tool to facilitate easy understanding of the MIPS 5 stage pipeline design. It is a free simulator built by the University of Catania, Italy. The problems raised by them was that the available ISS(educational material) focused heavily on specific concepts and not the whole concept and understanding. Many of these resources focus on the individual aspects of the system which disables the student from viewing it as a system level view. It was quite difficult to teach the entire course of Computer Architecture from this point of view using the software.

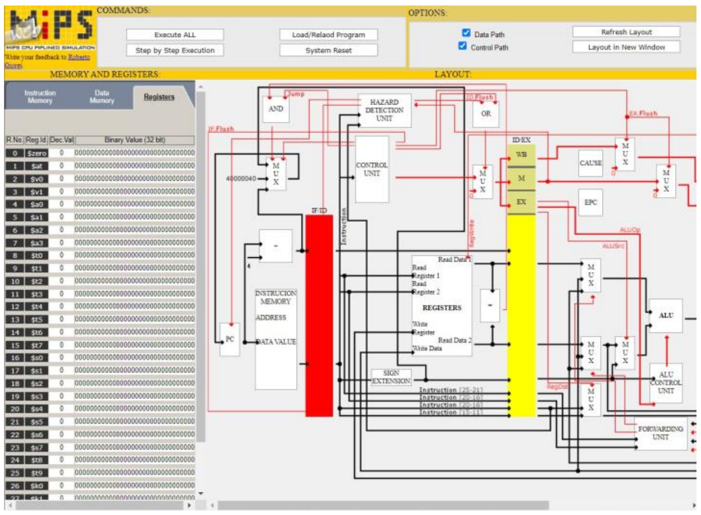
It is platform independent which makes it portable and easy to use on all types of platforms and operating systems. It is an open source program. After each cycle, each register value is updated which can be seen by the user and understood accordingly. Instruction forwarding is supported. Cache simulation can also be checked by the simulator without the user having to exit the simulator. Interrupts and system calls have also been implemented.

While on one hand a number of features have been implemented that make it easy for understanding and user-friendliness, on the other hand it is difficult to visualise branch control instructions since branch prediction is not implemented in the simulator. This is an implementation of the 64 bit MIPS implementation and not the more common 32 bit version. Our solution is the implementation of the 32 bit version of MIPS. The output of the algorithm visualised shows which stage of the pipeline runs in which stage of the clock and how many clock cycles are required to execute all the instructions. Our solution improves upon this solution but also does not have cache simulation. Hence it is a similar version with different support for certain features.

Citation: 2. D. Patti, A. Spadaccini, M. Palesi, F. Fazzino and V. Catania, "Supporting Undergraduate Computer Architecture Students Using a Visual MIPS64 CPU Simulator," in IEEE Transactions on Education, vol. 55, no. 3, pp. 406-411, Aug. 2012, doi: 10.1109/TE.2011.2180530.

WebMIPS Simulator:

WebMips is a MIPS simulation environment that is accessible from the web. It allows for access to the simulator without any overhead installations and configuration of the system, since it runs on the web itself. We observed that despite the varied nature of the user interface provided by the simulator, there could certainly be some improvements in the way the information has been displayed. They could give the state of the simulator at each clock cycle in a more user friendly manner. The figure below shows the display from the simulator:



✓ Data Path
✓ Control Path

R.No.	Reg.Id.	Dec.Val	Binary Value (32 bit)
0	$zero	0	00000000000000000000000000000000
1	$at	0	00000000000000000000000000000000
2	$v0	0	00000000000000000000000000000000
3	$v1	0	00000000000000000000000000000000
4	$a0	0	00000000000000000000000000000000
5	$a1	0	00000000000000000000000000000000
6	$a2	0	00000000000000000000000000000000
7	$a3	0	00000000000000000000000000000000
8	$t0	0	00000000000000000000000000000000
9	$t1	0	00000000000000000000000000000000
10	$t2	0	00000000000000000000000000000000
11	$t3	0	00000000000000000000000000000000
12	$t4	0	00000000000000000000000000000000
13	$t5	0	00000000000000000000000000000000
14	$t6	0	00000000000000000000000000000000
15	$t7	0	00000000000000000000000000000000
16	$s0	0	00000000000000000000000000000000
17	$s1	0	00000000000000000000000000000000
18	$s2	0	00000000000000000000000000000000
19	$s3	0	00000000000000000000000000000000
20	$s4	0	00000000000000000000000000000000
21	$s5	0	00000000000000000000000000000000
22	$s6	0	00000000000000000000000000000000
23	$s7	0	00000000000000000000000000000000
24	$t8	0	00000000000000000000000000000000
25	$t9	0	00000000000000000000000000000000
26	$k0	0	00000000000000000000000000000000
27	$k1	0	00000000000000000000000000000000

Figure: The WebMIPS interface 5. (link)

The target users of WebMIPS are students, but the display turns out to be overwhelming for the naive students. We in our Simulator plan to build an interface that is much cleaner and simpler but provides the information of execution as required. Furthermore, this simulator does not support Branch Prediction and non-pipelined CPU simulation. We aim to accomplish these in our approach.

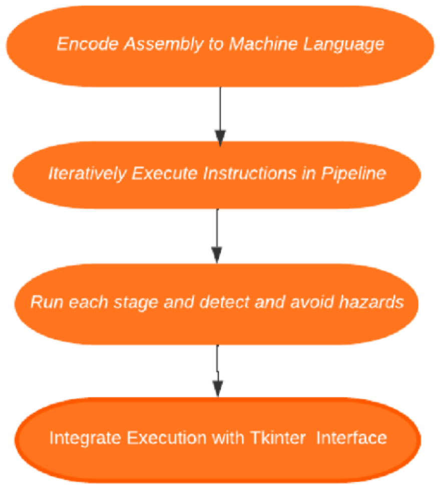
Citation:

1. Branović, Irina & Giorgi, Roberto & Martinelli, Enrico. (2004). WebMIPS: a new web-based MIPS simulation environment for computer architecture education. 10.1145/1275571.1275596.

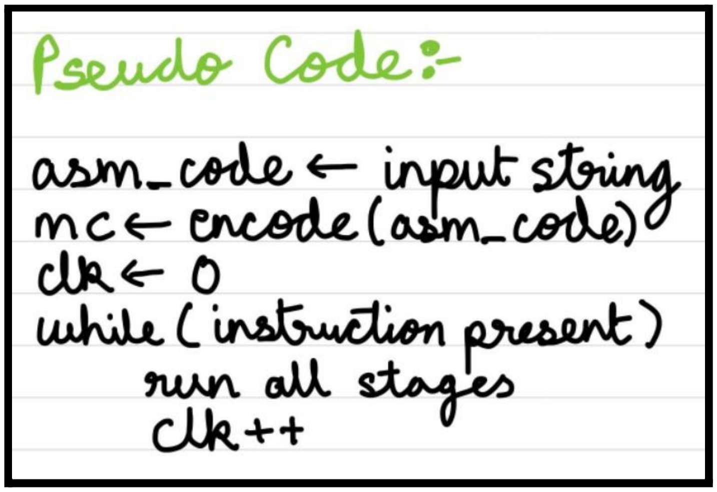
4 Our project idea

Our MIPS five stage pipeline simulator is a discrete event simulator. The simulator supports various R-type and I-type instructions like ADD, SUB, AND, OR, XOR, NOR, SLL, SRl, MULT, DIV, LW, SW and ADDI. We tried to cover control instructions like BEQ, J, JAL, JR and BNE but faced errors and problems in doing so.

In this section we would like to redefine our understanding of the MIPS 5-stage pipeline Simulator that we want to implement. We are determined to design the simulator that takes in instructions in assembly language as string, encodes them, executes the instructions, detects and avoids hazards,(our main focus will be on data and control hazards) Here is the Workflow of the entire process:


Encode Assembly to Machine Language
Iteratively Execute Instructions in Pipeline
Run each stage and detect and avoid hazards
Integrate Execution with Tkinter Interface

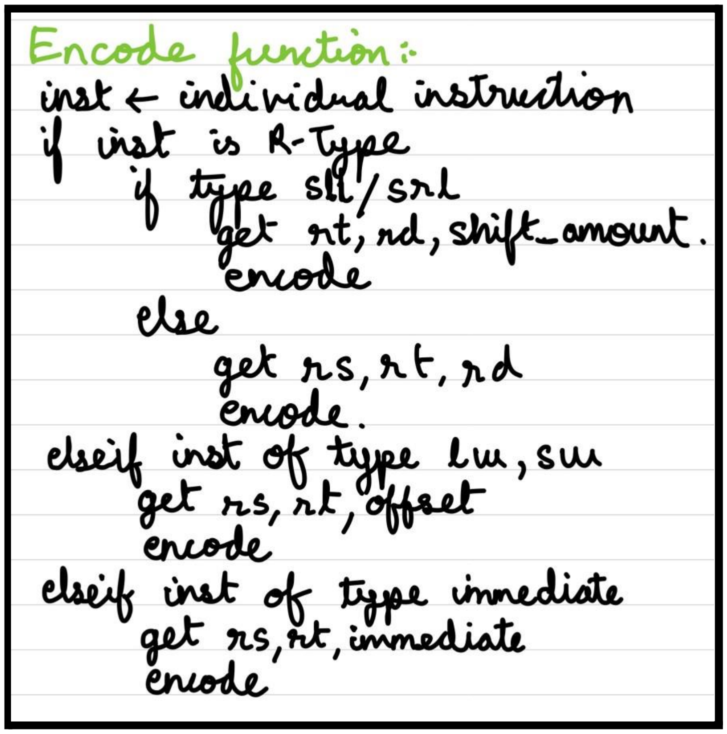
This encoding function is used by the main code. Upon encoding we execute each instruction until all instructions are executed. This can be done in the following manner :-



Pseudo Code:-

```
asm_code ← input string
mc ← encode(asm_code)
clk ← 0
while (instruction present)
    run all stages
    clk++
```

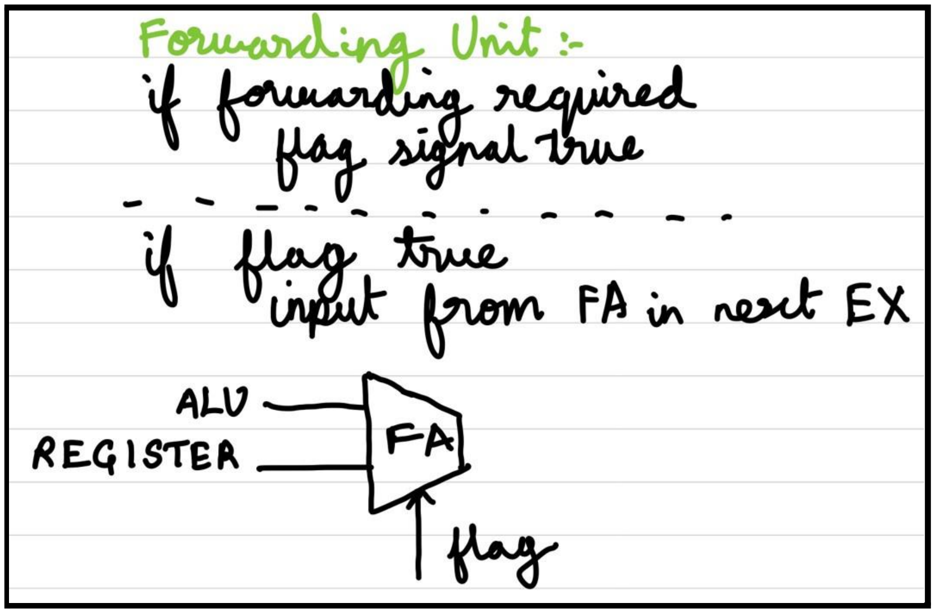
We then move one to encode the instructions one at a time. The pseudo code for encoding can be found below:



Encode function:

```
inst ← individual instruction
if inst is R-Type
    if type sll/srl
        get rt, rd, shift_amount.
        encode
    else
        get rs, rt, rd
        encode.
elseif inst of type lw, sw
    get rs, rt, offset
    encode
elseif inst of type immediate
    get rs, rt, immediate
    encode
```

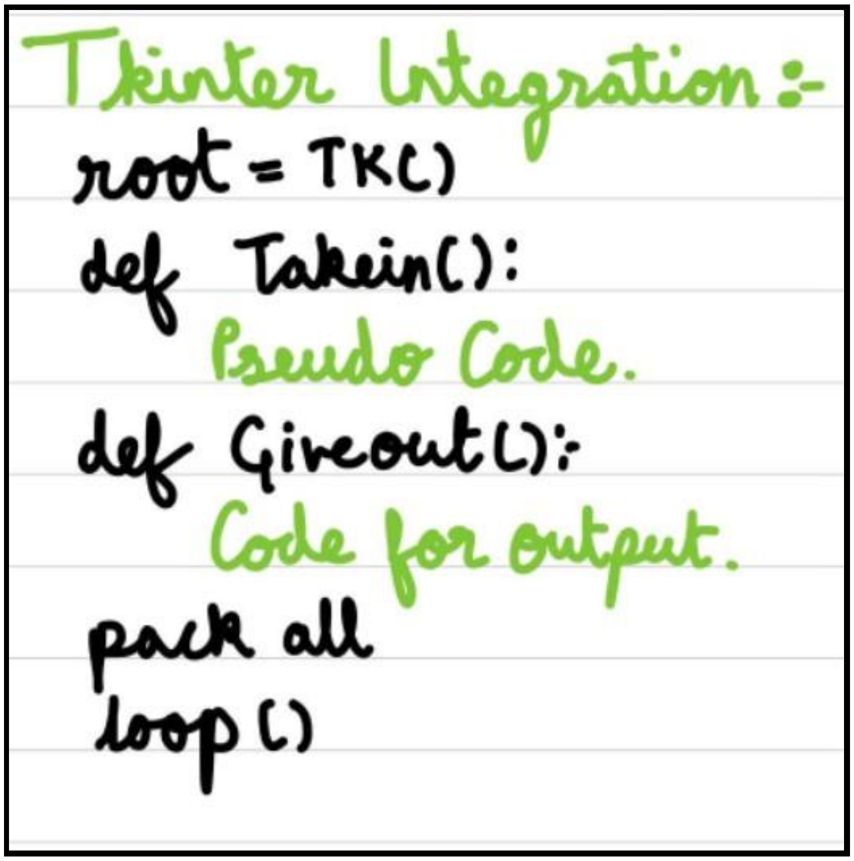
We plan to use forwarding to avoid data hazards that we would have encountered. The idea behind forwarding can be seen below:



Forwarding Unit :-

```
if forwarding required
    flag signal true
- - - - - - - - - - - -
if flag true
    input from FA in next EX
```

The pseudo code for wrapping up the above pseudo code within Tkinter can be seen below:



Tkinter Integration :-

```
root = TK()
def Takein():
    Pseudo Code.
def Giveout():
    Code for output.
pack all
loop()
```

5 Project Implementation

In this section, describe details of your project design and implementation, document challenges and obstacles and how you overcome them in design and implementation.

Working

We have implemented our MIPS pipeline simulator using three different files. We have implemented it using python. We are using a GUI interface to input the assembly code and show the results. First we would like to explain the inventory file mem_util.py and then the stage execution file sim_stages.py. Then we will move forward and explain sim_main.py to elaborate how the functions are called and what all it does.

mem_util.py

As the name suggests, util is an acronym for utilities. This is a type of pipeline inventory. We have implemented the following components, mainly hardware, in this file :-

- Instructions Mnemonics
- Their Corresponding Opcodes (R-type: 000000 ; lw: 100011 ; sw: 101011 ; addi: 001000)
- Register Names And Their Numbers/Addresses
- Types Of Errors And Their Flags
- PC=0, Instruction Memory Imem
- Data Memory Dmem
- Pipeline Registers/ Latch Registers/ Intermediate Registers Between The Stages
- Control Signals
- Forwarding Units

After that, we have defined an encoding function that encodes the assembly instruction into machine code. Then we are checking case by case. If instruction is R-type instruction, then we first check if it is shifting instruction or not. If it is shifting instruction, we need a record of shift amount 'shamt' and therefore we have distributed it. If it is other R-type instruction than shifting instruction, we consider and store rs, rt, rd, and a combined machine code of instruction. We also check for argument errors i.e. whether or not the correct number of arguments are provided or not. Also, we check for overflow and underflow errors.
If the instruction is load word or store word, we have a different way to separate the arguments in the instruction. We store rs, rt and offset. Rest of the things are the same. For I-type instruction, we use imm as the last argument to store immediate. Then we have defined a printing function that prints the processed file on the web browser.

Sim_stages.py

First, we are defining a control unit. It contains controls for all the signals and all possible formats of the instruction. Then, we have defined a forwarding unit as a function. There, we are checking all the conditions that apply to a specific stage and a particular type of instruction, and then executing appropriate control signals. We have defined two forwarding multiplexers that can forward the data when enabled. First forwarding unit takes data from the ID stage and forwards it to the ALU stage or WB as indicated by the signals. Second forwarding stage takes data from memory and forwards it to the registers.

Then we are defining hazard unit that checks for possible data and control hazards in the code. It checks the status of control signals and determines whether or not there is any hazard. It takes action appropriately.

After all of this, we are defining stages as different functions. IF function fetches instruction from memory. It first checks for a valid index for instruction and throws an exception accordingly which is handled by the try and except blocks. Then we are setting the simulator flags. It shows whether an instruction is running or not as well as whether or not the instant of execution is ideal. We do so to know whether or not to show the stage executed in the pipeline or not. Then we add values in the IF_ID pipeline register/ latch.

Then we have defined the ID stage. First we set the simulator flags. Then, if the pipeline is stalled, it makes all the control signals zero making all of them a bubble state. Else, it assigns the control signals their corresponding values, either high or low. Then we are storing the necessary data in the ID_EX register/latch.

Then we have defined the execution stage. It again sets simulator flags. We are first transferring controls from the previous latch to the next latch. We set ALU source operands based on the state of the control signal ALU_SRC. Then, we have set the "zero" signal coming out from the ALU. Then we are storing output. We first check the opcode and decide what kind of instruction it is. After that, we perform respective operations to store the result in the output of ALU. Finally, we store all of these remaining values in the next latch, EX_MEM.
Then we have a memory access stage. We set the simulator flags and pass the control signals to the next latch. Then we check for memory read signals. If it is asserted, we go and try to fetch data from memory. However, if the address is out of bounds, we throw an error and stop the execution. Then we check to write to the memory signal. If it is asserted, we try to write into the memory. If it is out of bounds, we throw an error message. Finally, we send the output of ALU and RD to the next latch.

At last, we have defined the write back stage. Setting the simulator flags, we check if the write to registers signal is asserted or not. If it is, we write to the corresponding register by using respective data, choosing from the ALU output or the loaded word from memory.

Sim_main.py

First, we are importing some modules. Then we are defining a global variable err that collects the errors occurring in a stage of the program so that we can see the error after the execution. Then we are creating the screen and applying the names and sizes. After that, we are creating a list that stores the clock’s history. Then we have defined a Takein function that takes the input as an assembly file and slits it in different lines. Then we are running a loop for the total number of instructions. Then we are encoding the instruction using the encoding function of sim_util. Then we are checking for errors at this state. If not, then we are appending the instruction in the instruction memory. If there is an error, we are identifying the actual error and concatenating it with the err variable. Then we print the error and simply stop the execution.

We create an empty clock history and set the clock at zero. Then we run a while loop with a condition such that it terminates only after all the stages of all the instructions are done. Then we are appending an empty list in the clock history that will be filled with data later on. Now, at this stage, we simulate the pipeline and transfer data from one stage to the next stage. We are doing it in the reverse manner. I.e. We are first passing the WB stage, then passing MEM to WB & so on. We are doing so because if we do IF to ID first then it will overwrite the data in the ID stage that is needed to pass to the EX stage. We are also checking for hazards. Then, we are storing the status of the stages in the appended empty list, in the form of tuples. Then we are incrementing the clock. After that we are appending this clock history of the current clock cycle to the global clock history.

We are defining Giveout function that outputs/prints the output of the program on the screen. We first check for the global error to be null. Then we are creating a history matrix that has all the data available of the previous state. We assign “ideal” condition if we want to show that stage on the output. After that we are printing the history board.

Now comes the GUI part. We have defined a function to exit if the exit button is pressed on the GUI window. Then, we are displaying buttons for input, output and exit. Then we are packing everything up and displaying it on the screen. Finally, we are initiating the main function when opening the file**.**

Challenges :-

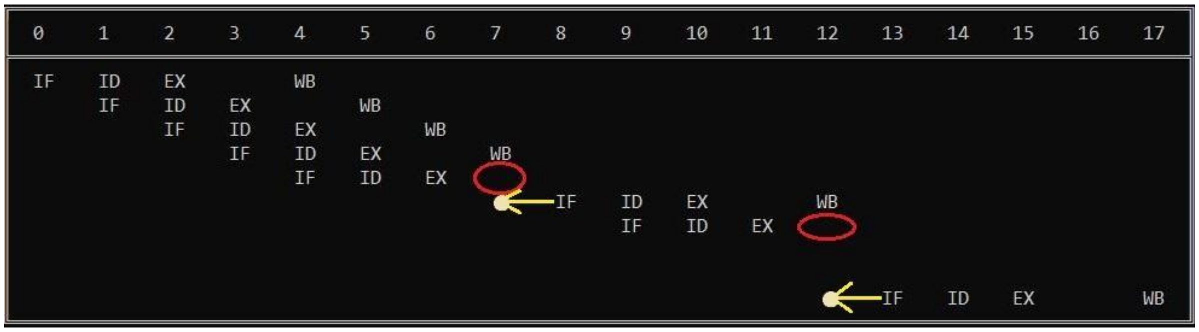
Beq and Bne errors :-

Problem :-

- We had tried to implement the branch instructions. We encountered an error as mentioned. We were trying to overcome the next IF by using stalls.
- The beq instruction was working perfectly except one thing, it added an extra stall to the pipeline. For example, for the following assembly code, it correctly executes both branch instructions, except that it adds an extra stall in 7th and 12th cycles. (Indicated the beginning of the stall by red oval. Yellow arrow shows the ideal position of the corresponding stage.)

- The same problem occurred with bne instructions.

- addi $s0, $zero, 2 addi $t0, $zero, 0 addi $t1, $zero, 5 addi $t2, $zero, 7 beq $t0, $t1, 3 add $t0, $s0, $t1 beq $t0, $t2, 2 addi $s1, $zero, 1 addi $s1, $zero, 2 addi $s7, $zero, 2



```
0    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17

IF   ID   EX        WB
     IF   ID   EX        WB
          IF   ID   EX        WB
               IF   ID   EX        WB
                    IF   ID   EX
                                        IF   ID   EX        WB
                                             IF   ID   EX

                                                                 IF   ID   EX        WB
```

Cause of the Problem :-

- This was happening because of the mentioned reason. We have implemented the code in such a way that it first shifts the WB block and completes it ; then it shifts MEM to WB and so on. This we did because if we were shifting IF to ID first, it over-written the data present in the ID stage that needed to pass on to the EX stage. Therefore, we needed the stage execution to work backwards.
- Ideally, all 5 stages should execute in parallel. We had separate instructions for passing the current stage to the next stage. Therefore, only after passing one stage, we can pass another stage because python programs execute one instruction at a time.
- That is why, after stalling during the ID stage, when it changed the PC address when branch was taken, it first passed ID to EX and then passed IF to ID, making it unable to detect that the opcode of the new instruction was changed. Therefore, it added one extra stall.

Proposed Solutions :-

- Make a parallel system where all the stages can be passed to the next stage at a unique instant simultaneously.
- Make a pseudo pipeline (in other words, a second pipeline) that keeps information of all stages in itself (making a copy of the same pipeline). Then pass the ith stage from pseudo pipeline to (i+1)th stage of original pipeline.

Attempted Solutions :-

- We introduced flags in ID and EX stage so that they can communicate with each other upon encountering branch instructions. But it did not work because they were called from different files.
- We introduced flags in the main.py program and made an ID to return values if a stall was installed due to specific instruction (beq, bne). If a branch was taken, we revert the execution order of ID and IF stage in a way such that IF occurs followed by ID. However, it also did not work due to mis-executing a wrong stage.
- We tried by stalling manually instead of by function but it did not work.

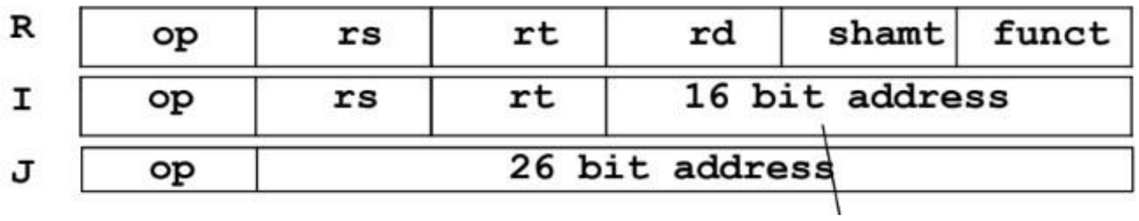
Conversion of Assembly code to Machine language:

Problem Encountered:

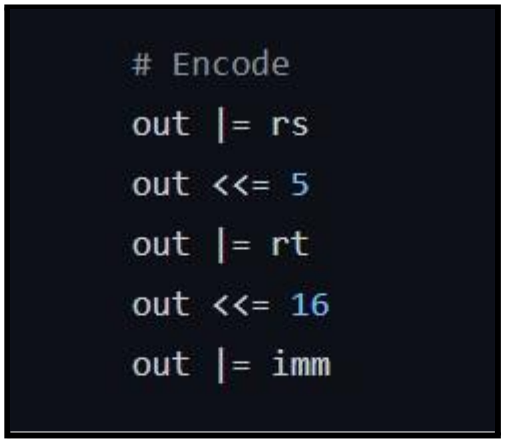
- The problem was faced when we were outlining the steps to be followed while writing the code for the simulator. How to convert the assembly code of an instruction," addi $t0, $t1, 1" to machine code was a challenge in itself.

Our solution:

- We decided to try to concatenate i.e. build the machine code as per the order of the registers in the MIPS machine code. Starting from rt, rd and shamt, we initialise each value by performing OR operation with a variable named, "out."
- The variable," out" is initialised to the value of 0. Performing OR operation assigns the value of the register to the variable. We then left shift the value by the number of bits it is designed to be in the MIPS implementation.
- Ex: It is 5 for rt,rd and shamt. Left shifting and performing the OR operation builds up the 32 bit machine code. The order in which the operands are placed depends upon the instruction type and this is taken care of by simple if conditions.
- At the end, we "OR" it with its opcode stored in the first index of the instructions array.



R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				



```
# Encode
out |= rs
out <<= 5
out |= rt
out <<= 16
out |= imm
```

Existing solution:

- There can be many ways of formulating the translation between the two languages.
- One must keep in mind about the specific nitigrities of assembly language related to MIPS so that the it's characteristics continue to remain valid. For example: easy to use, design, etc.

Our solution:

- Our solution offers an easy approach to build the machine code by scratch taking in account the relative order of the registers and building it in this way. It is just another solution to translate the code.

Integration of the Interface with the simulator:

Problem Encountered:

- A crucial idea behind the development of the simulator was to make it readily usable for simulation and visualisation of what was happening in the CPU, without going through the nitty gritty of command prompt.

Existing Solutions:

- Throughout the years quite a few attempts have been made for the visualisation of the execution of mips.
- The attempts have been made to strengthen the understanding of students of the mips execution procedure.
- The work done by researchers from Lahore University of Management Sciences named "A CONFIGURABLE, MULTI-CYCLE INTEGER AND FLOATING POINT MIPS PIPELINE SIMULATION TOOL FOR EDUCATIONAL PURPOSES" was very insightful.
- They have developed an interface to give the instructions in assembly language as shown in the figure:

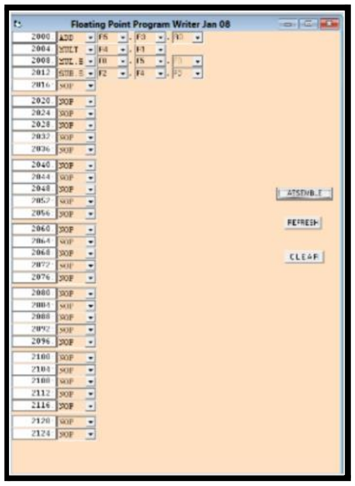


Figure : The Program Writing Interface

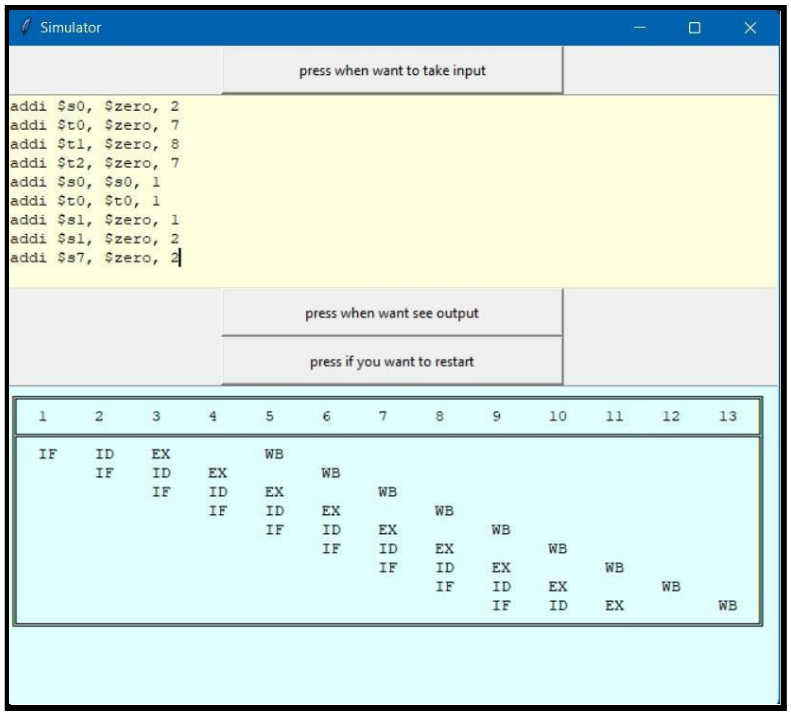
- They have also kept track of the registers and have presented it visually. Furthermore they have visually presented the datapath and control path in each clock step.

  4.[Farrukh, Anam & Ikram, Jahangir. (2014). A CONFIGURABLE, MULTI-CYCLE INTEGER AND FLOATING POINT MIPS PIPELINE SIMULATION TOOL FOR EDUCATIONAL PURPOSES: THE FIRST RELEASE.]

Our solution:

- We have developed an interface using the Tkinter library of Python.
- Our interface takes the program written as a string in assembly language as input.

- We internally do all the required simulation and give a table as shown below as output in the interface :-


Simulator
press when want to take input
press when want see output
press if you want to restart

```
addi $s0, $zero, 2
addi $t0, $zero, 7
addi $t1, $zero, 8
addi $t2, $zero, 7
addi $s0, $s0, 1
addi $t0, $t0, 1
addi $s1, $zero, 1
addi $s1, $zero, 2
addi $s7, $zero, 2
```

1	2	3	4	5	6	7	8	9	10	11	12	13
IF	ID	EX		WB								
	IF	ID	EX		WB							
		IF	ID	EX		WB						
			IF	ID	EX		WB					
				IF	ID	EX		WB				
					IF	ID	EX		WB			
						IF	ID	EX		WB		
							IF	ID	EX		WB	
								IF	ID	EX		WB

Data Hazards :-

Description

- We can not avoid data hazards in a pipeline processor. Data dependency occurs when a change in data of a register does not happen and another change in data or read is carried out from the same register. It leads us to a wrong execution of the program.
- There are 4 types of data hazards : RAW, RAR, WAR, WAW. WAR and WAW hazards do not occur in simple pipeline processors.
- We have to mainly deal with RAW and RAR hazards. RAW means read after write hazard. It occurs when we try to read from a register when an update to the value stored in the register is not completed.

Existing Solutions

- Implement a system block that detects the data hazards by checking the registers in the instructions. It checks dependencies between registers and throws signals that help handling such hazards.

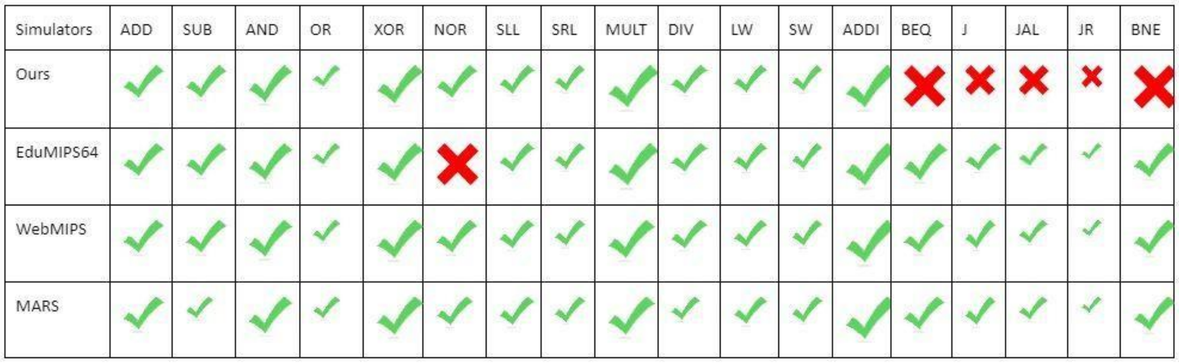
Implemented Solutions

- We are detecting data hazards by checking the registers being used in the instructions.
- When any data dependency has occurred, we use the forwarding/bypassing unit to transfer the data.

- By using this, we are avoiding the data hazards and smoothly continuing the pipeline simulation.

6 Data Analysis

Based on the data from the documentation of various pipeline simulators, we derived the following results. The multitude of instructions being supported is shown with a green tick mark if it is supported and a red cross if not.



Simulators	ADD	SUB	AND	OR	XOR	NOR	SLL	SRL	MULT	DIV	LW	SW	ADDI	BEQ	J	JAL	JR	BNE
Ours	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗
EduMIPS64	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
WebMIPS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MARS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Figure 3: Number of Instructions supported by various simulators

7 Conclusion

We implemented the five stage MIPS pipeline simulator and detected data and control hazards. Stalls were introduced in case the dependencies introduced hazards into the pipeline. We have also printed the number of clock cycles it takes for the instructions to run with the stalls.

Pipelining increases the efficiency of the processor. It is able to execute many instructions in parallel. Hazards are detected and worked on to improve the efficiency till the maximum possible. Our MIPS five stage pipeline simulator is a discrete event simulator. The simulator supports various R-type and I-type instructions like ADD, SUB, AND, OR, XOR, NOR, SLL, SRl, MULT, DIV, LW, SW and ADDI. We tried to cover control instructions like BEQ, J, JAL, JR and BNE but faced errors and problems in doing so. For hazard detection, we have used forwarding in the case of data hazards.

We could have added the support of more number and type of instructions to increase the compatibility and usability of the simulator. Input-output could have been handled in a better way i.e., it could have been designed better for the user's comfort. The GUI(Graphical User Interface) could have been designed better to make it attractive and user-friendly and easy to use.

Future work :-

1. In case of control hazards, the stalls added could have been avoided totally by the use of a branch predictor. In regard to this, better models could be implemented with increased accuracies and higher clock rate.
2. User Interface) could have been designed better to make it attractive and user-friendly and easy to use.

References

[1] Branović, Irina & Giorgi, Roberto & Martinelli, Enrico. (2004). WebMIPS: a new web-based MIPS simulation environment for computer architecture education. 10.1145/1275571.1275596.

[2] D. Patti, A. Spadaccini, M. Palesi, F. Fazzino and V. Catania, "Supporting Undergraduate Computer Architecture Students Using a Visual MIPS64 CPU Simulator," in IEEE Transactions on Education, vol. 55, no. 3, pp. 406-411, Aug. 2012, doi: 10.1109/TE.2011.2180530.

[3] D. X. Lim and K. G. Smitha, "Pipelined MIPS Simulation: A plug-in to MARS simulator for supporting pipeline simulation and branch prediction," *2019 IEEE International Conference on Engineering,TechnologyandEducation(TALE)*,2019,pp.1-7,doi:10.1109/TALE48000.2019.922593

[4] Farrukh, Anam & Ikram, Jahangir. (2014). A CONFIGURABLE, MULTI-CYCLE INTEGER AND FLOATING POINT MIPS PIPELINE SIMULATION TOOL FOR EDUCATIONAL PURPOSES: THE FIRST RELEASE.

[5] https://projects.ncsu.edu/wcae/ISCA2004/submissions/giorgi.pdf