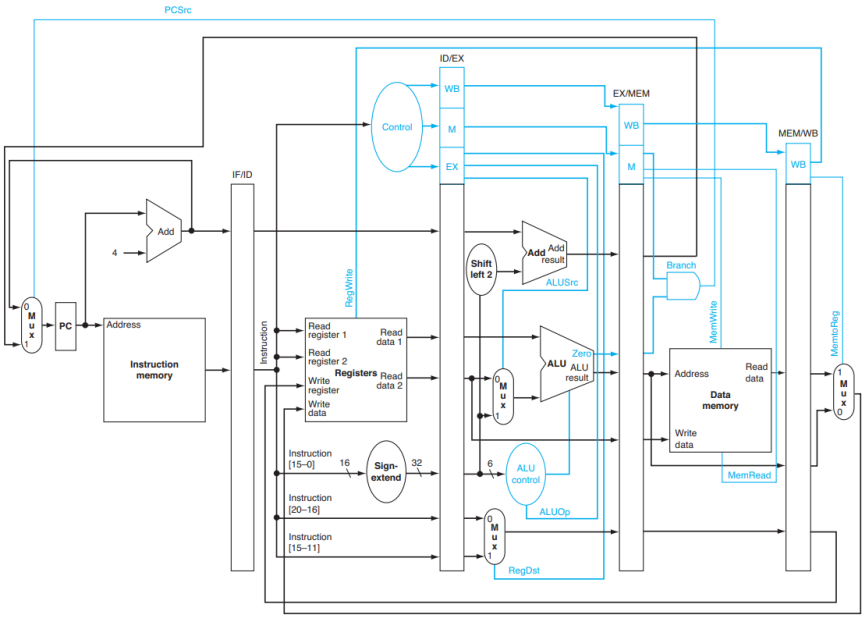
Pipelined Implementation of MIPS Architecture



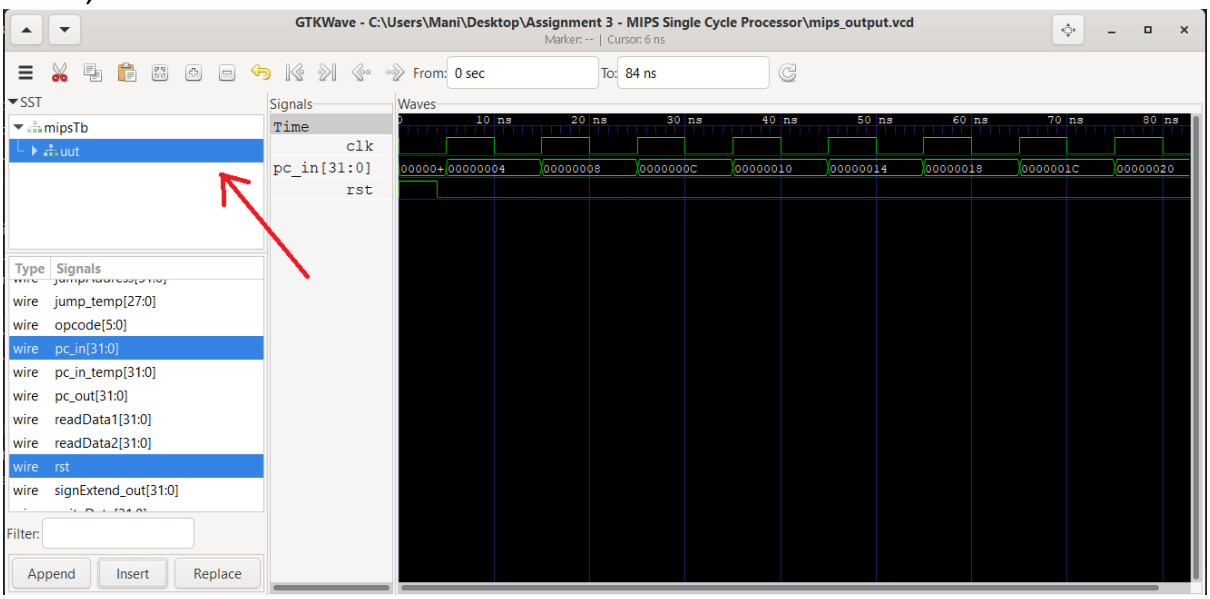
Manikandan Gunaseelan

Table of Contents

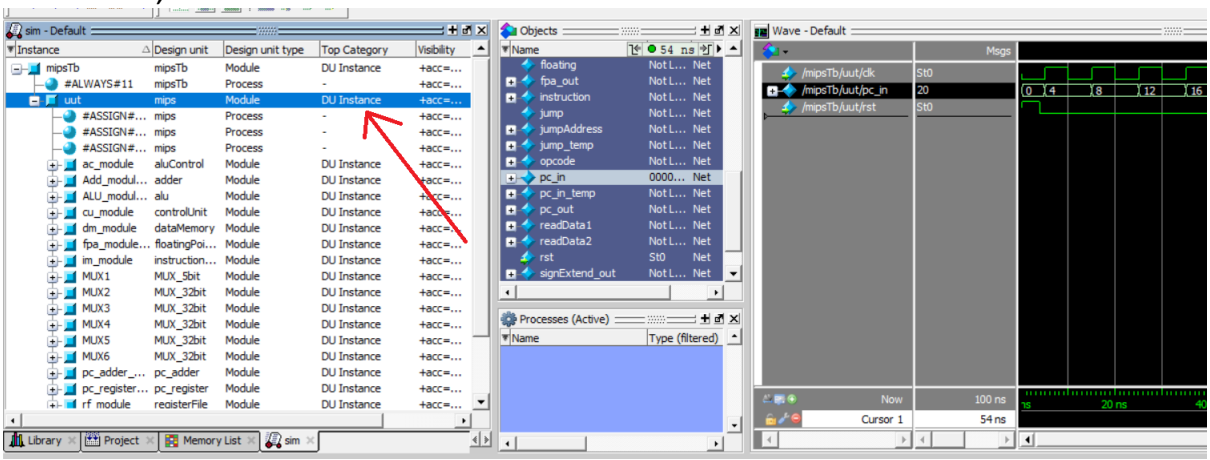
Simulation of the Design

To run the processor, one must simulate the “mipsTb” module. Do note that this module only has clk and rst as inputs as all of the other signals are internal. So to view the internal signals, we have to go to the “uut” section in the waveform window and then select the signals we wish to see.

On GTKWave, this can be done here -



And on ModelSim, it can be done here -



In this report, I will be showing the outputs on GTKWave as it is easier to run and view. However, the same outputs are also obtained on ModelSim.

Instructions Implemented

The following instructions have been implemented in the processor -

1. R - type instructions
    - add
    - sub
    - AND
    - OR
    - slt

2. addi instruction

3. lw instruction

4. sw instruction

5. beq instruction

Different programs have been written to demonstrate all of these instructions separately and I will now show the outputs of each of them, before moving on to the actual design of the processor.

A Forwarding unit and a hazard detection unit have been written to prevent data hazards that may arise during execution.

Control hazards have been taken care of using a control hazard unit. For branch instructions, it is assumed that branches, by default, will not be taken. If the branch is taken, three instructions will be flushed from the pipeline (by making control signals zero so they cannot change the state of the processor).

The various instructions and the handling of hazards has been shown in the demonstration.

Demonstration of the Processor

In this section, different outputs for various programs will be shown. For each of these, the program has to be written in the instruction memory (instructionMemory.v file). The programs demonstrated have been given as comments in the file.

As mentioned previously, I have used GTKWave to show the results in the report as it is easier to simulate but the same results can be seen on ModelSim too.

There are five programs that are simulated. Each of them show different aspects of the processor and the handling of various hazards. The programs are :

1. Instruction flow through the pipeline
2. Data hazard solved using forwarding
3. Load-use hazard solved by inserting bubble using hazard detection unit
4. Control hazard and the handling of branch instructions

The different control signals, inputs and outputs to the internal modules required for each instruction have been highlighted separately.

The signal names try to follow the notation given in the diagram of the Patterson and Hennessy book, the image of which is also shown in the cover page.

Note - We have to load the instruction memory from instructionMem[1] and not [0] as instructionMem[1] is the first instruction that the PC reads.

In the register file, $t0 starts from register number 8, $t0 is register number 9 and so on.

The programs shown are quite basic as the focus is on the processor and not the instructions and we want to see the ability of the processor to handle different hazards. Only the required signals are shown to get an idea of what the processor is doing without being bombarded by hundreds of different signals.

1. Instruction going through the pipeline

The program is as follows -

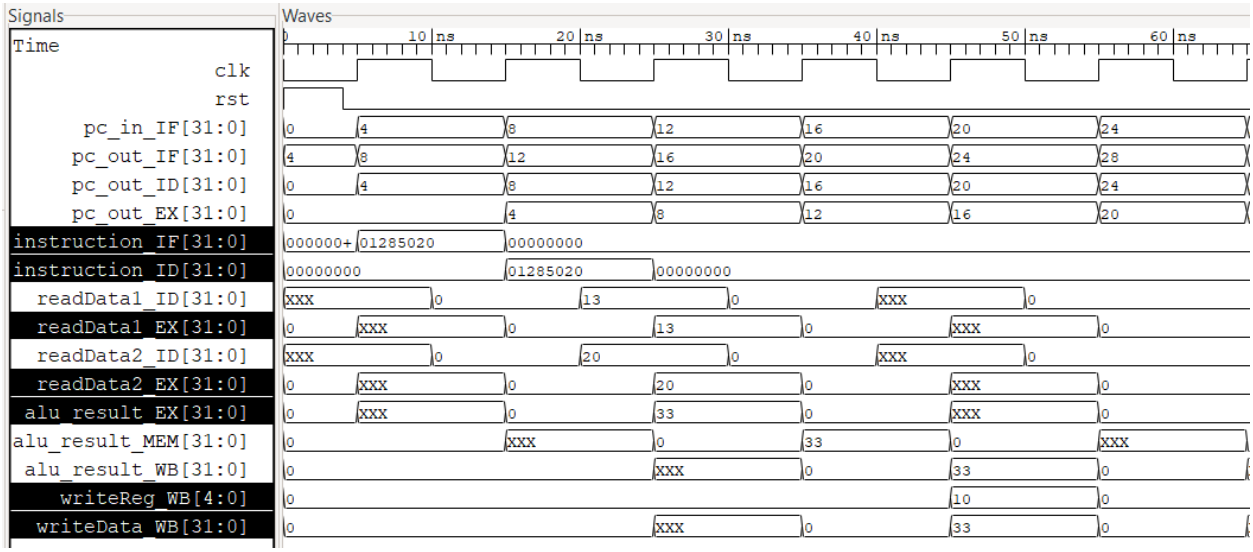
```
add $t2, $t1, $t0
```

The instruction memory is loaded

```
instructionMem[1] = {6'd0, 5'd9, 5'd8, 5'd10, 5'd0, 6'd32};      // add $t2, $t1, $t0
```

In the register file, $t0 and $t1 are pre-loaded

```
registerMem[8] = 20;      // $t0
registerMem[9] = 13;      // $t1
```



There is a lot going on here. First look at the four PC signals and how the pc_out value propagates through the IF ID and EX stages.

instruction_IF and instruction_ID show the propagation of the instructions from IF to ID.

Note that the register file is read in the second half of the clock cycle – this is to prevent hazards which we will cover in the next demonstration. When the read_data1 and 2 results reach the EX stage, the alu_result is calculated and propagated to the MEM and WB stage.

Finally, we see that the writeReg in the WB is 10 (reg $t2) and the Write data is 33, which is the correct answer.

We have, thus, seen the propagation of an instruction through the pipeline without any hazards.

2. Data Hazard solved using forwarding

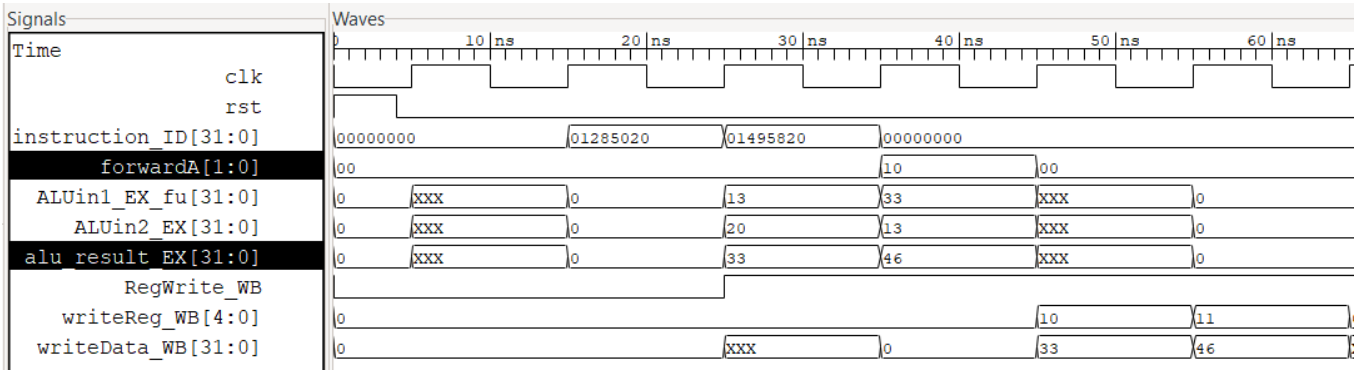
add $t2, $t1, $t0
add $t3, $t2, $t1

Here, the result of the first addition will be needed for the second instruction. Since $t2 will be written into only at the end of WB stage of the first instruction, we have to forward the data back while the second instruction is in its EX stage in order to maintain the pipeline.

Loading of instruction memory :
instructionMem[1] = {6'd0, 5'd9, 5'd8, 5'd10, 5'd0, 6'd32}; // add $t2, $t1, $t0
instructionMem[2] = {6'd0, 5'd10, 5'd9, 5'd11, 5'd0, 6'd32}; // add $t3, $t2, $t1

In the register file, $t0 and $t1 are pre-loaded
registerMem[8] = 20; // $t0
registerMem[9] = 13; // $t1

If, by the end of the second instruction, 33 + 13 = 46 is being written into $t3, we would confirm that the forwarding has happened correctly.



The two instructions in the ID stage correspond to both of the add instructions. Notice the forwardA signal being 10 in the third clock cycle, which shows that there is a data hazard and forwarding is being done for register $t2.

The ALUin1 and ALUin2 signals shown are after the forwarding MUXes. We can see that the result of the first instruction (33) is being used as an ALU input for the next instruction even before being written into the register.

Finally, we look at the WB signals. We can see 33 being written to reg 10 ($t2) and 46 being written to reg 11 ($t3), which is the expected outcome. Hence, we see that data forwarding works correctly. The processor can also handle 1b, 2a and 2b hazards.

3. Load use hazard

lw $t1, 4($t0)
add $t3, $t1, $t0

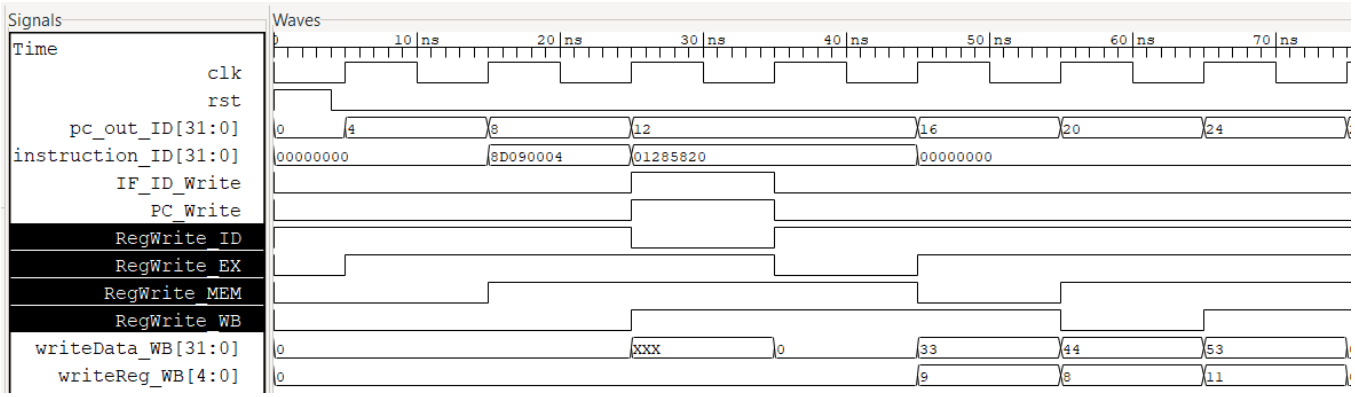
Here, only forwarding is not useful as the dependence between load and the next add instruction ($t1) actually goes back in time (as the result is available in the MEM/WB stage). The hazard detection unit is used to detect this hazard and insert a bubble (state of the processor won't be changed) in the pipeline. After the nop, the add instruction must work as expected with the correct value of $t1.

Instruction memory :
instructionMem[1] = {6'd35, 5'b01000, 5'b01001, 16'd4}; // lw $t1, 4($t0)
instructionMem[2] = {6'd0, 5'd9, 5'd8, 5'd11, 5'd0, 6'd32}; // add $t3, $t1, $t0

Register loading:
registerMem[8] = 20; // $t0

Data Memory:
dataMem[6] = 32'd33;

We will be able to confirm that the program handles the hazard if by the end of the second instruction, 20 + 33 = 53 is being written back to $t3



Here, IF_ID_Write and PC_Write are shifted to 1 when a stall is needed. This stops writing of the PC and IFID register. We can see that as PC and instruction are held constant for two cycles. The stall is also noticed as control signals are turned 0 for one cycle and allowed to propagate through the pipeline. This is shown as RegWrite is turned 0 and sent through the pipeline. Finally, we verify that data 33 is being written to reg 9 ($t1), followed by a stall and then data 53 is being written to reg 11 ($t3). Hence, the load-use hazard is handled correctly.

4. Control Hazard

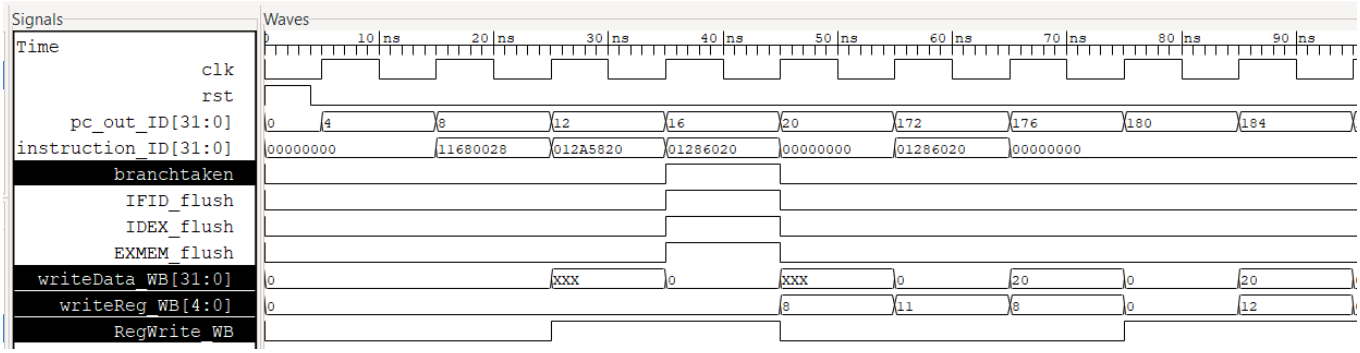
```
        beq $t3, $t0, brdest
        add $t3, $t1, $t2
        add $t4, $t1, $t0
        sub $t0, $t3, $t0
brdest : add $t4, $t1, $t0
```

Instruction memory contents are as follows

```
instructionMem[1] = {6'd4, 5'd11, 5'd8, 16'd40};              // beq $t3, $t0, +40
instructionMem[2] = {6'd0, 5'd9, 5'd10, 5'd11, 5'd0, 6'd32};    // add $t3, $t1, $t2
instructionMem[3] = {6'd0, 5'd9, 5'd8, 5'd12, 5'd0, 6'd32};    // add $t4, $t1, $t0
instructionMem[4] = {6'd0, 5'd8, 5'd11, 5'd8, 5'd0, 6'd34};    // sub $t0, $t3, $t0
instructionMem[42] = {6'd0, 5'd9, 5'd8, 5'd12, 5'd0, 6'd32};    // add $t4, $t1, $t0
```

In the register file, $t0 and $t3 are both loaded as 20, remaining registers are 0.

The branch condition will be satisfied. But by then, the pipeline will have fetched the following instructions. These results are invalid and hence must not be stored in the registers. Therefore, three flush signals are used so a nop is inserted in the pipeline and the instructions present inside are changed to bubbles (control signals are made zero and hence, these instructions will not change the state of the processor). The branch is taken and the PC value is changed to fetch the instruction at the branch destination.



When the branch decision is made (branchtaken -> 1), the flush signals are made active – IFIDflush makes the next instruction a nop (0 instruction can be seen in the next cycle) and IDEXflush and EXMEMflush make the control signals zero so no data is written. We can then see that sub $t0, $t3, $t0 tries to write 20 to reg 8 ($t0) but RegWrite is zero (because of flush) and hence it is not written. However, the instruction at the branch is able to write the 0 + 20 = 20 in register number 12 ($t4).

5. Miscellaneous Instructions

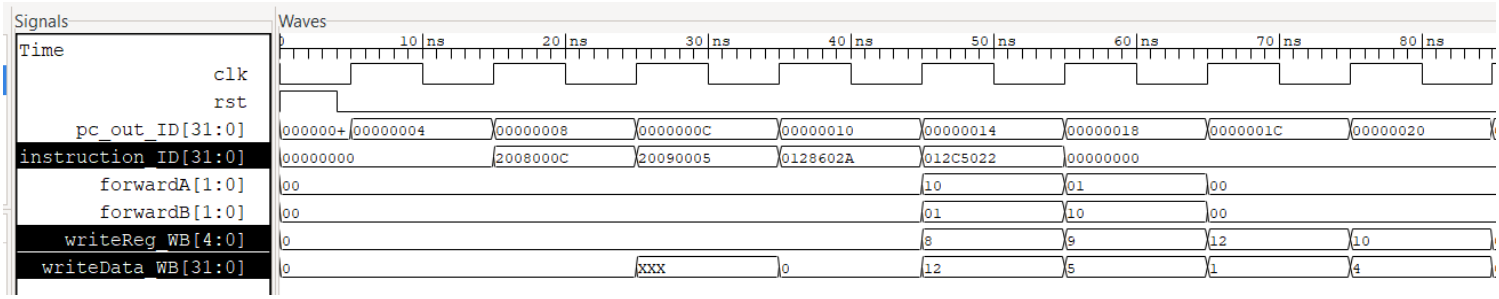
addi $t0, $zero, 12
addi $t1, $zero, 5
slt $t4, $t1, $t0
sub $t2, $t1, $t4

The instruction memory contents are

```
instructionMem[1] = {6'd8, 5'd0, 5'd8, 16'd12};              // addi $t0, $zero, 12
instructionMem[2] = {6'd8, 5'd0, 5'd9, 16'd5};              // addi $t1, $zero, 5
instructionMem[3] = {6'd0, 5'd9, 5'd8, 5'd12, 5'd0, 6'd42};    // slt $t4, $t1, $t0
instructionMem[4] = {6'd0, 5'd9, 5'd12, 5'd10, 5'd0, 6'd34};    // sub $t2, $t1, $t4
```

All registers are zero before execution.

This program shows the addi, slt and sub instructions which we had not seen till now. There is an EX data hazard between instruction 2 and 3, and there is a MEM data hazard between instructions 1 and 3 as their results are not written into their destination registers by the time it is required by the following instructions. There is also an EX data hazard between the 3rd and 4th instructions. slt $t4, $t1, $t0 will set $t4 as $t1 is less than $t0. sub $t2, $t1, $t4 will store the value 5 – 1 = 4 in $t2.



We can see that at the end, 4 is being written into reg 10 ($t2), so our program has ran correctly. The forward signals have also been shown that tackle the data hazards that we discussed previously.

Design of the Processor

Note : The underlying circuit of the processor is based on my previous single cycle processor and a lot of the files are common between the two implementations. Hence, the description of the common files are also the same.

In this section, a brief overview of all the modules implemented will be provided. The entire code has been divided into 17 different files for easy organization and readability of the code. I will now describe the actions of each module one by one.

For a detailed description, do go through the code as it is self-explanatory and has plenty of comments.

1. **mips.v**

   This file has the module **mips**, which contains all the connections of the circuit and instantiates all of the internal modules. The top module only has two inputs – clk and rst, as all of the other signals are internal. This is the module that is simulated in the testbench and so, to view the waveforms we need to delve into uut signals as mentioned before.

2. **mipsTb.v**

   This file has the testbench module **mipsTb**, which is the module needed to be run for simulation. It makes the simulation time 104 ns, which is enough for all the programs demonstrated earlier, however for writing your own programs, do feel free to increase this time as needed. Also, these two lines of code (line 14 and 15)

   ```
   $dumpfile("mips_output.vcd");
   $dumpvars(0, mipsTb);
   ```

   Are used for saving the waveform for viewing in GTKWave and can be removed if the code is being simulated using ModelSim.

   You may also want to remove the $finish command if simulating on ModelSim.

3. **pc.v**

   The file pc.v has two modules. **pc_register** is the PC register which is set to 0 when rst is enabled and otherwise works like a normal register during a clock edge. This is done to make sure the PC is 0 at the beginning of the program. This module, however, was used for

the single cycle processor. In the pipelined implementation, a separate register model file has all the registers in the circuit.

pc_adder increments the pc_in by 4 to make pc_out whenever pc_in changes, that is, at the clock edge or rst. This value can be used to make pc_in later depending on branch and jump conditions.

4. instructionMemory.v

The module **instructionMemory** stores the instructions of the program. It is made up of 128 memory locations each of 32 bits. PC is given as the input to the instructionMemory module, which returns the 32 bit instruction pointed to by the PC.

All of the sample programs discussed before have been written as comments in the instruction memory. Do note that you should leave the first location (which is instructionMem[0]) blank and start giving your instructions from instructionMem[1].

For running your own programs, feel free to add your instructions in this module.

5. controlUnit.v

The **controlUnit** module has two inputs, which is the 6 bit opcode of the instruction and the control_mux_select signal. If the control_mux_select signal is active, all the control signals are made 0. This is done to insert bubbles in the pipeline.

The opcode is given by instruction[31:26] and it generates the different control signals based on the opcode. The instructions implemented in the processor have been summarized before but I will list those again here with their opcode –

R-type instructions (opcode 0)
addi instruction (opcode 8)
lw instruction (opcode 35)
sw instruction (opcode 43)
beq instruction (opcode 4)

The different control signals generated by the control unit are –
RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch and ALUOp (the signals are based on the Patterson and Hennessy book).

These control signals are derived from the opcode using a case statement.

6. MUX.v

MUX.v contains three modules – **MUX_5bit**, which is a 5 bit MUX ,**MUX_32bit**, which is a 32 bit MUX and **MUX_3_1**, which is a 3:1 MUX used for forwarding. All the MUXes in the processor are implemented using these modules.

7. registerFile.v

This has the module **registerFile**, which stores, updates, and writes to the 32 registers in the processor. If the input control signal regWrite is 1, it will write the writeData on to writeReg. Otherwise, outputs readData1 and readData2 will get the values present in readReg1 and readReg2, which are provided from the appropriate bits of the 32 bit instruction.

Writing of the registers is done in the first half of the clock cycle and reading of the registers is done in the second half to prevent hazards by allowing the writing and reading (of the intended data) in the same clock cycle.

In the demonstration programs, we have used $zero, which is the first register of the file at registerMem[0]. Temporary registers start from registerMem[8] and $s0 register is at registerMem[16], as given in the Patterson and Hennessy book.

8. signExtend.v

The **singExtend** module returns a 32 bit sign extend version of a 16 bit number. It does this by repeating the sign bit of the 16 bit number 16 more times.

9. aluControl.v

The **aluControl** module generates the aluControlSignal, which tells the ALU what instruction has to be performed. The inputs to the aluControl module are the function field of the instruction and ALUOp.

If ALUOp is 0 (LW or SW or addi instruction), aluControlSignal is 2 for addition. If ALUOp is 1 (beq instruction), aluControlSignal is 6 for subtraction. Else, for R-type instructions, aluControlSignal depends on the function field of the operation.

10. alu.v

The **alu** module has two 32 bit inputs, a 4 bit control signal and two outputs – aluZero, to show when the result is zero, and aluOut, the 32 bit output of the computation. A case statement is written to calculate the output depending on the control signal.

One of the 32 bit inputs is directly from the register file and the other is taken, using a MUX, either from the register file or from the sign extended version of instruction[15:0] depending on the ALUSrc control signal.

In the Patterson and Hennessy book, this output is used directly as the alu result. However, I have also implemented floating point arithmetic so that must be taken into account while deciding on the alu result.

11. dataMemory.v

The **dataMemory** module has 128 memory locations of 32 bits each. Using the control signals MemRead and MemWrite, data can be read from or written to the data memory. For the sample program of load-use hazard, a memory location has already been loaded with a value.

12. shiftLeft2.v

This file has two modules, one for each shiftLeft2 block in the processor circuit. The **shiftLeft2_26bit** module takes a 26 bit value and shifts it left 2 bits to make a 28 bit value. This is used for calculating the jump address. The **shiftLeft2_32bit** has a 32 bit input and a 32 bit output which is the input shifted left by 2 bits. It is used in the branch operation.

13. adder.v

The **adder** module is used to calculate the new PC value in case of branch instructions. The PC + 4 value is added to the branch offset and fed to a MUX, where a select signal determines whether the branch will be taken. This is demonstrated in the sample program regarding the beq instruction.

14. registerModel.v

The **registerModel** contains eight models for registers, to account for different input signals and lengths of inputs.

The pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB) and the PC register are all implemented using these modules. It is important to note that the pipeline registers are not a single register, but each signal they forward has been implemented using different registers. Reading the **mips** module will make this point clear. Some of the pipeline registers need a separate flush signal and Write signal so separate modules have been written for those.

15. **forwardingUnit.v**

The forwarding unit is designed based on the Patterson and Hennessy book. Four different kinds of data hazards are considered : 1a and 1b (EX hazards) and 2a and 2b (MEM hazards). The output signals generated are ForwardA and ForwardB which are used as select signals for two forwarding MUXes in the EX stage. The use of this module has been shown in the sample programs

16. **hazardDetectionUnit.v**

The **hazardDetectionUnit** is also based on the design in the Patterson and Hennessy book. Three different registers are compared to check for load-use hazards and three output signals are generated to insert a bubble in the pipeline in case there is a load-use hazard. IFIDWrite, when asserted, prevents data being written to the IF/ID register. PCWrite is used to prevent the PC from being incremented and controlMuxSelect is used as an input to the control unit to make all control signals zero, so that the state of the processor cannot be changed. The sample program for the load-use hazard goes more into depth on this module.

17. **controlHazardUnit.v**

The **controlHazardUnit** checks for hazards that arise due to branch instructions. It is assumed that branches are never taken and hence, the following instructions continue to be fetched. If the branch does turn out to be taken, three instructions in the pipeline are flushed and a nop is inserted. This is done using IFID_flush, IDEX_flush and EXMEM_flush signals in the **mips** module. This module is present in the MEM stage as that is when the branching decision can be made.