Lyles College of Engineering
Department of Electrical and Computer Engineering

Technical Report

Experiment Title: 5-Stage Pipelined MIPS Processor
Course Title: ECE 174 Advanced Computer Architecture
Instructor: Dr. Hayssam El-Razouk
Date Submitted: 18 May, 2023

Prepared By:	Sections Written:
Puya Fard	Section 1, 2, 3, 4, 5, 6
Carlos Lopez	

INSTRUCTOR SECTION

Comments:

Final Grade: Team Member 1: Puya Fard
Team Member 2: Carlos Lopez

TABLE OF CONTENTS

1. STATEMENT OF OBJECTIVES

The objective of this project is to integrate and prototype the datapath and the control units of the simple 32-bit MIPS processor with five pipeline stages. This processor should be written in Verilog hardware language and must be able to perform arithmetic/logic, data movement, and flow control instructions. Furthermore, students should design and architect the memory to support the following zones: code section and data section. Finally, having the processor operate on an Intel DE2-115 FPGA development board is the ultimate goal of this project. The learning outcomes of this project will result in students having hands- on experience on building a computer processor and implementing it on hardware.

2. THEORETICAL BACKGROUND

Modelsim - source: (“ModelSim HDL simulator | Siemens Software”)

ModelSim is a hardware simulation and debugging tool developed by Mentor Graphics, which is used for designing and testing digital circuits. It is one of the most widely used simulation tools in the industry and is often used in the development of electronic systems, ranging from simple digital circuits to complex integrated circuits (ICs) and system-on-chip (SoC) designs.

ModelSim supports both Verilog and VHDL hardware description languages (HDLs), which are used to describe the behavior of digital circuits. HDLs are programming languages that allow designers to describe the function and behavior of a digital circuit, including its inputs, outputs, and internal workings, in a way that can be simulated and tested.

ModelSim uses a powerful simulation engine that can simulate millions of logic gates and thousands of flip-flops in a matter of seconds. It allows designers to simulate and test their designs before they are physically implemented, which can help to catch errors and reduce the time and cost of development.

In addition to simulation, ModelSim also provides advanced debugging features, such as waveform viewing and tracing, which allow designers to visualize and analyze the behavior of their designs. It also includes support for advanced verification methodologies such as Universal Verification Methodology (UVM) and Open Verification Methodology (OVM).

MIPS Implementation - source: (https://www.d.umn.edu/~gshute/mips/mips-intro.xhtml)

Before looking at the diagram we need a little bit of context:

A very brief introduction to logic circuits.
A very high-level view of the processor's datapath.
A quick overview of some intermediate-level processor organizations. The single-cycle organization is one of these organizations.

After looking at the diagram we will:

Look at black-box descriptions of the components of the diagram.
Take a quick look at the functions of control signals in the diagram.

Logic Circuits: Logic circuits use two different values of a physical quantity, usually voltage, to represent the boolean values true (or 1) and false (or 0). Logic circuits can have inputs and they have one or more outputs that are, at least partially, dependent on their inputs. In logic circuit diagrams, connections from one circuit's output to another circuit's input are often shown with an arrowhead at the input end.

In terms of their behavior, logic circuits are much like programming language functions or methods. Their inputs are analogous to function parameters and their outputs are analogous to function returned values. However, a logic circuit can have multiple outputs.

There are two basic types of logic circuitry: combinational circuits and state circuits.

- Combinational circuit behaves like a simple function. The output of a combinational circuit depends only on the current values of its input.
- State circuitry behaves more like an object method. The output of state circuitry does not just depend on its inputs — it also depends on the past history of its inputs. In other words, the circuitry has memory.

This is much like an object method whose value is dependent on the object's state: its instance variables. These two types of circuitry work together to make up a processor datapath.

Processor Datapath:

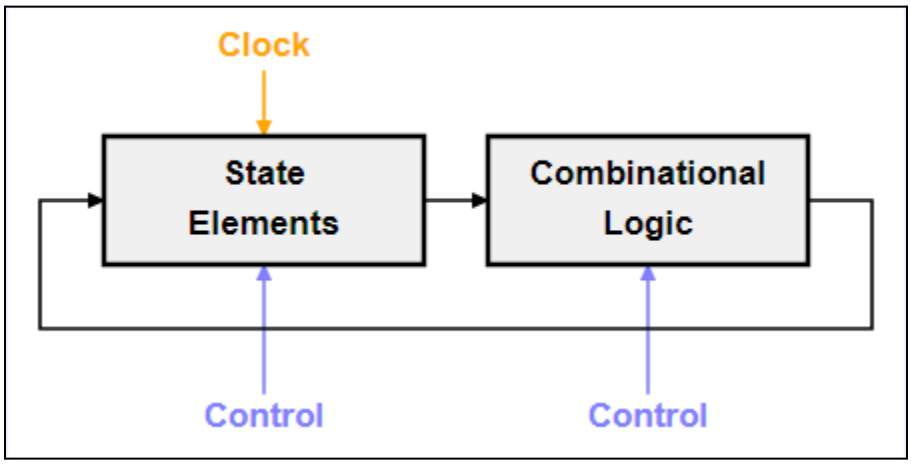


Figure 2.1: Processor datapath

A processor's datapath is conceptually organized into two parts:

- State elements hold information about the state of the processor during the current clock cycle. All registers are state elements.
- Combinational logic determines the state of the processor for the next clock cycle. The ALU is combinational logic.

This diagram, like most diagrams on this web site, adheres to the following conventions:

- Clock signals are colored orange.
- Control signals are colored blue.

There are four major processor organizations:

- Single-cycle organization: This is the organization described in this web presentation. It is characterized by the fact that each instruction is executed in a single clock cycle. It is not a realistic implementation — it requires two separate memories: one for data and one for instructions. Also, the clock cycle has to be made quite long in order for all of the signals generated in a cycle to reach stable values.

- Multicycle organization: This organization uses multiple clock cycles for executing each instruction. Each cycle does only a small part of the work required so the cycles are much shorter. Much of the circuitry is the same as the single-cycle implementation. However, more state components must be added to hold data that is generated in an early cycle but used in a later cycle.

- Pipelined organization: Like the multicycle organization, the pipelined organization uses multiple clock cycles for executing each instruction. By adding more state components for passing data and control signals between cycles, the pipelined organization turns the circuitry into an assembly line. After the first cycle of one instruction has completed you can start the execution of another instruction, while the first moves to its next cycle. Several instructions can be in different phases of execution at the same time.

- Register renaming organization: Register renaming is an extension of the pipelining idea. It deals with the data dependence problem for a pipeline — the fact that instructions in the pipeline produce data needed by other instructions in the pipeline.

Pipelining - source: (https://www.d.umn.edu/~gshute/mips/mips-intro.xhtml)

The basic concept of pipelining is to break up instruction execution activities into stages that can operate independently. Every instruction passes through the same stages much like an assembly line.

For example, we could set up the following stages for a MIPS pipeline.

- IF - instruction fetch and PC increment
- ID - source register fetch and instruction decode
- EX - ALU source selection, ALU operation, and branch target calculation
- MEM - data memory access
- WB - write back to destination register

With these pipeline stages, a sequence of instructions can be executed as shown below. Time progresses from left to right. Each horizontal division represents one clock period.

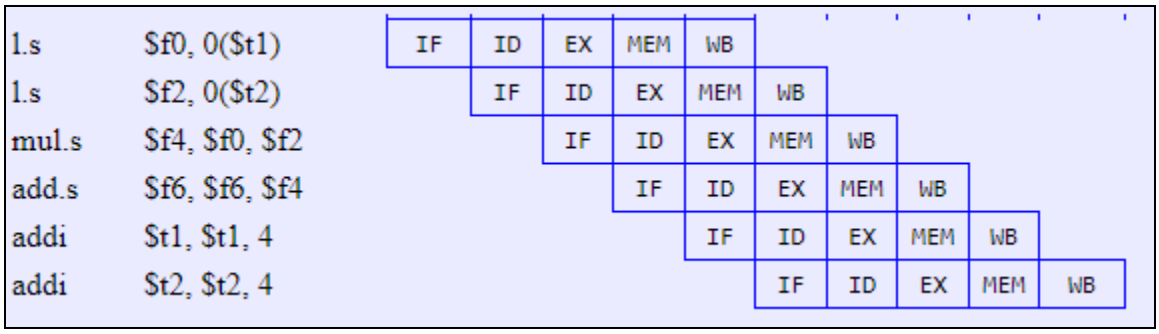


Figure 2.2: Pipeline stages

Benefit of Pipelining - source: (https://www.d.umn.edu/~gshute/mips/mips-intro.xhtml)

As you can see from the figures below, pipelining increases instruction throughput. Notice that after the 5th cycle, the unpipelined execution completes only one instruction every 5 cycles, while the idealized pipelined execution completes 5.

Ideally, instruction throughput is increased to 1 instruction per clock. In other words, the clocks per instruction (CPI) factor in the performance equation is reduced from 5.0 to 1.0.

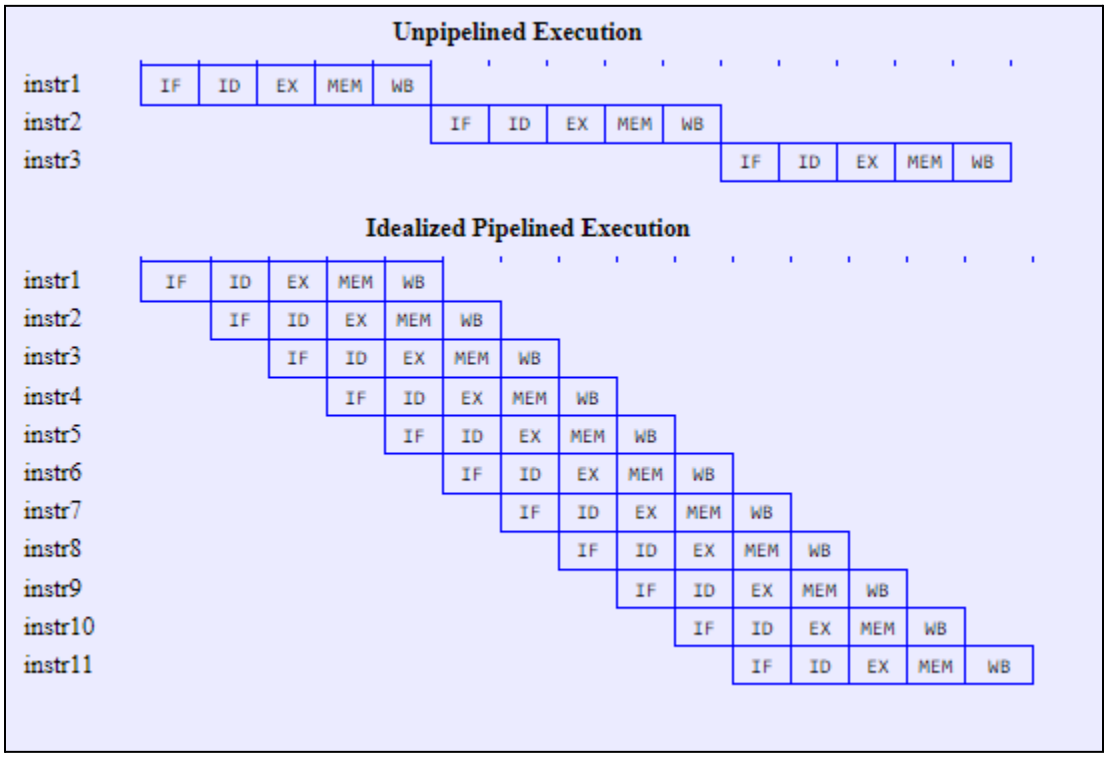


Figure 2.3: Pipeline benefits

Pipeline Implementation - source: (https://www.d.umn.edu/~gshute/mips/mips-intro.xhtml)

The best starting point for a pipelined implementation is a single-cycle implementation. For example, for a MIPS pipeline you could start with an implementation whose high-level data path is shown as the "Before Pipelining" diagram below.

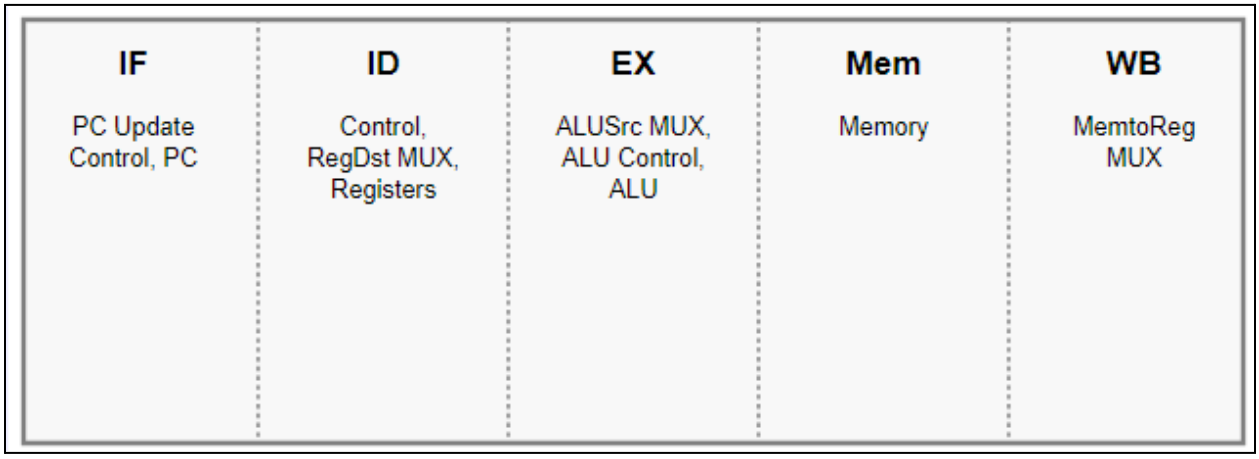


Figure 2.4: Before pipeline implementation

To implement pipelining registers are added between stages. The pipelining registers are shown in light green in the "After Pipelining" diagram below. The pipelining registers hold data and control signals that are produced in an early stage for use in later stages.

Signals generated in a stage cannot be held for more than one cycle. A signal that is generated in an early stage and used several stages later must pass through all of the intermediate pipeline registers. For example, a control signal that is produced in the ID stage and used in the WB stage must pass through 3 pipelining registers: the ID/EX registers, the EX/MEM registers, and the MEM/WB registers.

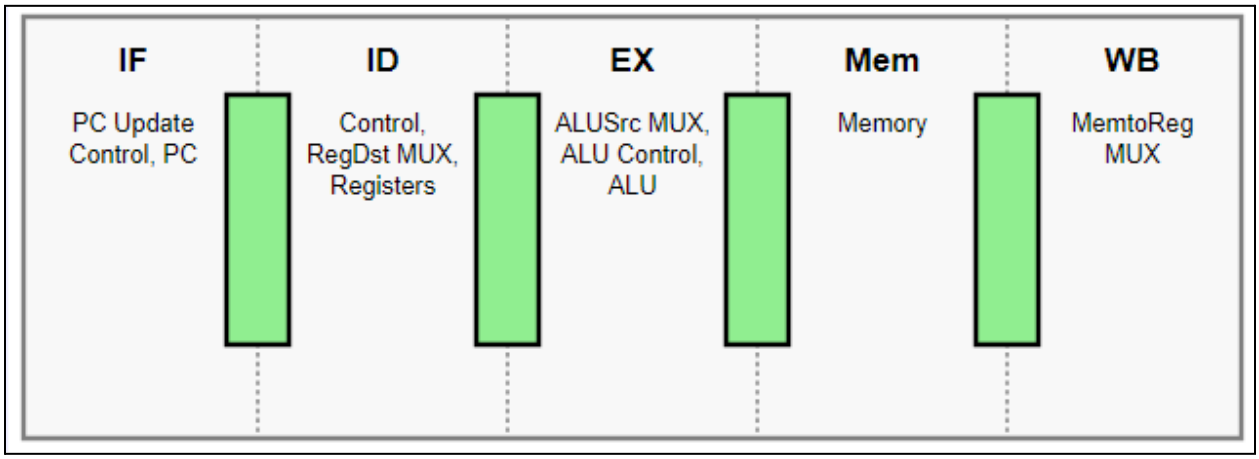


Figure 2.5: After pipeline implementation

Obstacles to Pipelining - source: (https://www.d.umn.edu/~gshute/mips/mips-intro.xhtml)

The analogy between a pipeline and an assembly line breaks down in one important respect. Putting together a door for a car does not depend on cars further along in the assembly line.

But there are dependencies between instructions. These can be seen in the diagram below where data is passed back from a later stage to an earlier stage. The ones that involve updating the PC (red) are called control hazards. The ones that involve writing data back to registers (purple) are called data hazards.

Both of these dependencies are inherent in the instruction set. Compiler writers call them control and data dependencies. In both cases the execution of a later instruction depends on the results of earlier instruction. There are other obstacles, called structural hazards that arise from the starting point of the pipelining implementation

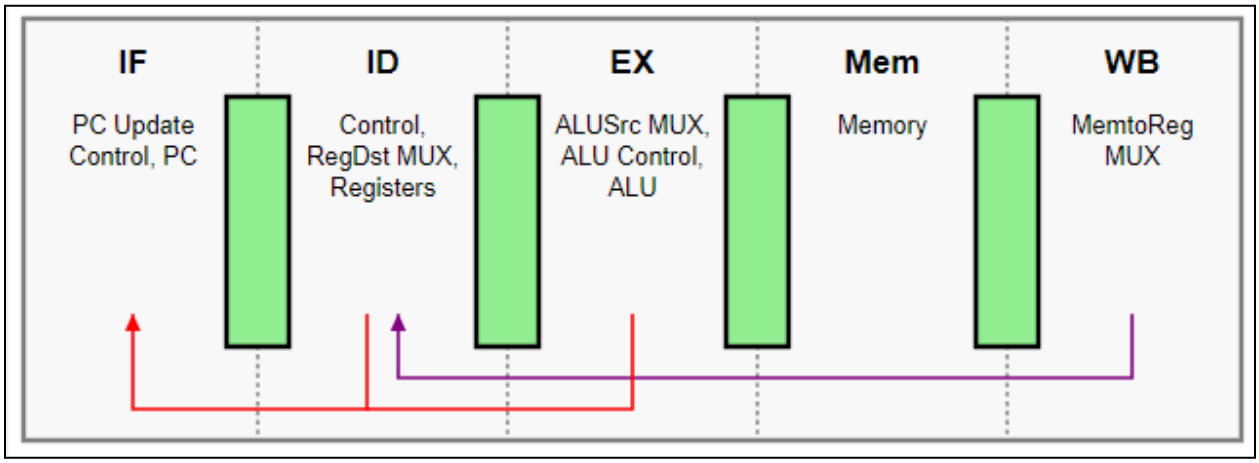


Figure 2.6: Dependencies

Control Hazards - source: (https://www.d.umn.edu/~gshute/mips/mips-intro.xhtml)

Control hazards arise from branches and jumps. They involve signals that are passed from a later stage to an earlier stage:

- A branch or jump target address is not available until the ID stage but it needs to be passed back to the PC for the IF stage of the next instruction.
- The condition for a branch instruction is not tested until the EX stage but it needs to be passed back to the PC Update Control for the IF stage of the next instruction.

Data Hazards - source: (https://www.d.umn.edu/~gshute/mips/mips-intro.xhtml)

Data hazards arise from instructions producing data that is used in later instructions. They involve signals that are selected by the MemtoReg multiplexer in the WB stage to be written to a register. The register may be read by a later instruction in its ID stage.

Structural Hazards - source: (https://www.d.umn.edu/~gshute/mips/mips-intro.xhtml)

Structural hazards are hazards that depend on the starting point for the implementation. For example, if we started with a multicycle implementation, we would have problems in a pipeline because the ALU is used in more than one stage by the same instruction. Executing a branch instruction, the ALU is used to increment the PC, compute a branch target address, and compare two source operands. These uses are going to prevent other instructions in the pipeline from using the ALU.

Pipelining and the Instruction Set

Pipelining is one of the primary reasons why RISC processors have a significant speed advantage over CISC processors. If arithmetic and logical instructions can access memory for source or destination operands then it is much more difficult to break down instruction execution into stages with equal durations. If memory addressing modes are complex then this problem just gets harder. If instructions have varying lengths it is more difficult to start a new instruction every cycle.

When pipelining is done with a CISC processor it is done at a different level. The execution of instructions is broken down into smaller parts which can then be pipelined. In effect, The CISC instructions are translated into a sequence of internal RISC instructions, which are then pipelined. This adds complexity to the processor and generally does not produce as much benefit. For upward compatibility, the Intel 80x86 family of processors, including Pentium processors since the early 1990s, have used this approach.

5-Stage-Pipeline MIPS - source: (“VHDL MIPS 5 stage pipeline Bug”)

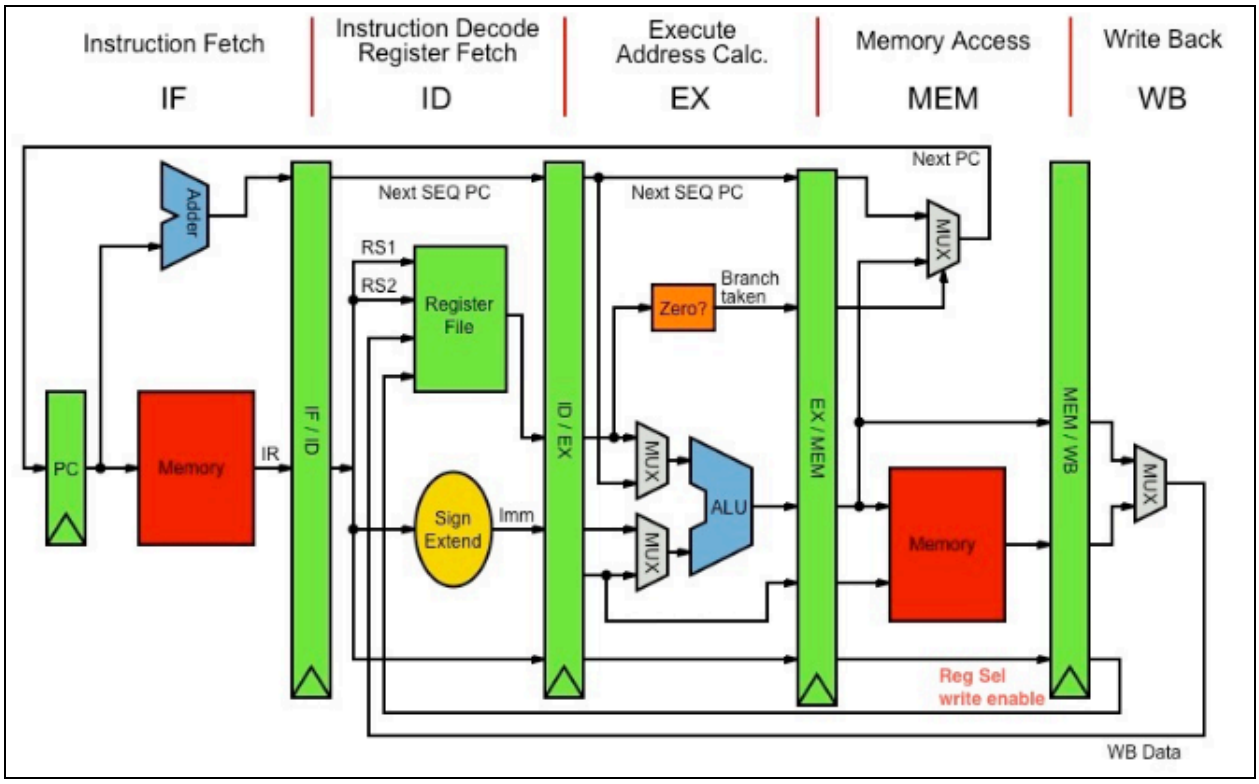


Figure 2.7: 5-stage pipeline for MIPS

The figure above **Figure 2.7** shows the complete 5 stages required for the 32-bit mips processor. The figure isn't a complete list of requirements for completing the processor, muxes, control units, and hazard block isn't displayed. However, it is a general idea for completing the processor.

Understanding Mips Instructions - source: ("VHDL MIPS 5 stage pipeline Bug")

Data types:

1. Instructions are all 32 bits
2. byte(8 bits), halfword (2 bytes), word (4 bytes)
3. a character requires 1 byte of storage
4. an integer requires 1 word (4 bytes) of storage

Registers

- 32 general-purpose registers
- register preceded by $ in assembly language instruction

- two formats for addressing:
    - using register number e.g. $0 through $31
    - using equivalent names e.g. $t1, $sp

- special registers Lo and Hi used to store result of multiplication and division
    - not directly addressable; contents accessed with special instruction mfhi ("move from Hi") and mflo ("move from Lo")
    - stack grows from high memory to low memory

Register Number	Alternative Name	Description
0	zero	the value 0
1	$at	(assembler temporary) reserved by the assembler
2-3	$v0 - $v1	(values) from expression evaluation and function results
4-7	$a0 - $a3	(arguments) First four parameters for subroutine. Not preserved across procedure calls
8-15	$t0 - $t7	(temporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls
16-23	$s0 - $s7	(saved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls
24-25	$t8 - $t9	(temporaries) Caller saved if needed. Subroutines can use w/out saving. These are in addition to $t0 - $t7 above. Not preserved across procedure calls.
26-27	$k0 - $k1	reserved for use by the interrupt/trap handler
28	$gp	global pointer. Points to the middle of the 64K block of memory in the static data segment.
29	$sp	stack pointer Points to last location on the stack.
30	$s8/$fp	saved value / frame pointer Preserved across procedure calls
31	$ra	return address

Figure 2.8: Register descriptions

Program Structure - source: ("VHDL MIPS 5 stage pipeline Bug")

- just plain text file with data declarations, program code (name of file should end in suffix .s to be used with SPIM simulator)
- data declaration section followed by program code section

Data Declarations

- placed in section of program identified with assembler directive .data
- declares variable names used in program; storage allocated in main memory (RAM)

Code

- placed in section of text identified with assembler directive .text
- contains program code (instructions)
- starting point for code execution given label main:
- ending point of main code should use exit system call (see below under System Calls)

Comments

- anything following # on a line
- # This stuff would be considered a comment

Load / Store Instructions - source: (Lecture slides)

RAM access is only allowed with load and store instructions. all other instructions use register operands.

load:

```
lw      register_destination, RAM_source
```

#copy word (4 bytes) at source RAM location to destination register.

```
lb      register_destination, RAM_source
```

#copy byte at source RAM location to low-order byte of destination register,
and sign-extend to higher-order bytes

store word:

```
sw      register_source, RAM_destination
```

#store word in source register into RAM destination

```
sb      register_source, RAM_destination
```

#store byte (low-order) in source register into RAM destination

load immediate:

```
li      register_destination, value
```

#load immediate value into destination register

Figure 2.9: lw/sw instructions

Arithmetic Instructions

- most use 3 operands
- all operands are registers; no RAM or indirect addressing
- operand size is word (4 bytes)

```
add     $t0,$t1,$t2     #  $t0 = $t1 + $t2;   add as signed (2's complement) integers
sub     $t2,$t3,$t4     #  $t2 = $t3 Ð $t4
addi    $t2,$t3, 5      #  $t2 = $t3 + 5;   "add immediate" (no sub immediate)
addu    $t1,$t6,$t7     #  $t1 = $t6 + $t7;   add as unsigned integers
subu    $t1,$t6,$t7     #  $t1 = $t6 + $t7;   subtract as unsigned integers

mult    $t3,$t4         #  multiply 32-bit quantities in $t3 and $t4, and store 64-bit
                        #  result in special registers Lo and Hi:  (Hi,Lo) = $t3 * $t4
div     $t5,$t6         #  Lo = $t5 / $t6   (integer quotient)
                        #  Hi = $t5 mod $t6   (remainder)
mfhi    $t0             #  move quantity in special register Hi to $t0:   $t0 = Hi
mflo    $t1             #  move quantity in special register Lo to $t1:   $t1 = Lo
                        #  used to get at result of product or quotient

move    $t2,$t3 #  $t2 = $t3
```

Figure 2.10: Arithmetic operations

Control Structures

Branches: comparison for conditional branches is built into instruction

```
b       target          #  unconditional branch to program label target
beq     $t0,$t1,target  #  branch to target if  $t0 = $t1
blt     $t0,$t1,target  #  branch to target if  $t0 < $t1
ble     $t0,$t1,target  #  branch to target if  $t0 <= $t1
bgt     $t0,$t1,target  #  branch to target if  $t0 > $t1
bge     $t0,$t1,target  #  branch to target if  $t0 >= $t1
bne     $t0,$t1,target  #  branch to target if  $t0 <> $t1
```

Figure 2.11: Control operations

Jumps

```
j       target  #  unconditional jump to program label target
jr      $t3             #  jump to address contained in $t3 ("jump register")
```

Figure 2.12: Control operations 2

MIPS Instruction Types - source: (Lecture notes)

When MIPS instructions are classified according to coding format, they fall into four categories: R-type, I-type, J-type, and coprocessor. The coprocessor instructions are not considered here.

The classification below refines the classification according to coding format, taking into account the way that the various instruction fields are used in the instruction. The details of the execution activities and the required control signal values depend almost entirely on the instruction type in this classification.

- Non-Jump R-Type
- Immediate Arithmetic and Logic
- Branch
- Load
- Store
- Non-Register Jump
- Register Jump

In the remainder of this web page, the instruction fetch and instruction decode activities are omitted since they are the same for all instructions. The PC update activity only shows updates beyond the standard PC increment (PC ← PC + 4).

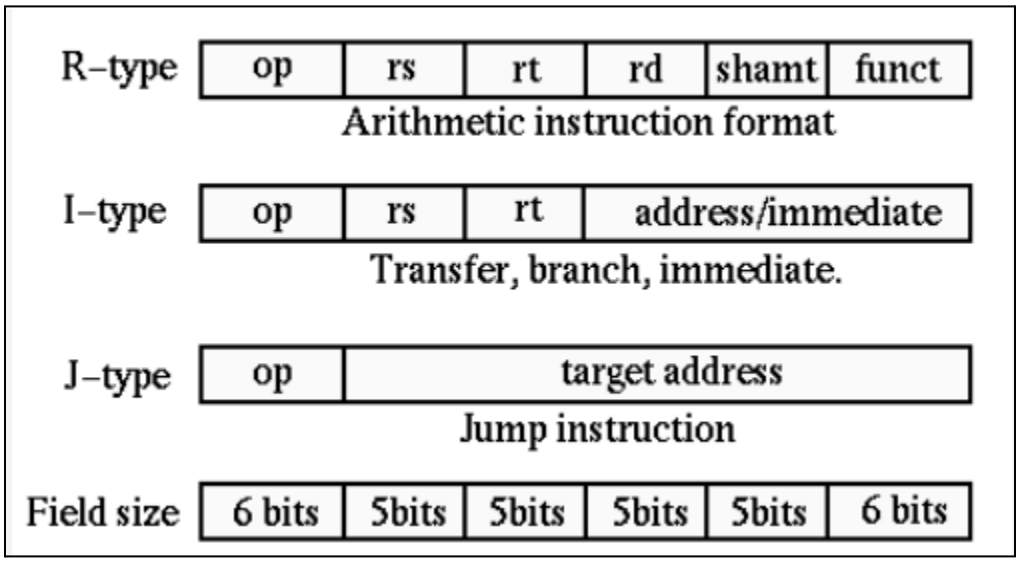


Figure 2.12: Instruction information

Non-Jump R-Type

Non-jump R-type instructions include all R-type instructions except jr and jalr. This includes all of the integer arithmetic and bitwise operations, along with the non-branching compare instructions such as slt, sgt, and seq. They use the R coding format. The opcode bits are all 0.

- PC update: There is no update beyond the normal increment.
- Source operand fetch: The two source operands are rs and rt.
- ALU operation: The ALU operation is determined by the function field.
- Memory access: There is no memory access for data.
- Register write: The result from the ALU is written to rd.

Immediate Operand

Most immediate operand instructions perform arithmetic or logical operations using one operand that is coded into the instruction. The immediate operand group also includes the comparison instructions slti and sltiu and the lui instruction. Immediate operand instructions use the I coding format.

- PC update: There is no update beyond the normal increment.
- Source operand fetch: The two source operands are rs and the immediate field. For all instructions except sltiu the immediate field is sign extended. For sltiu the immediate field is zero extended. This instruction is not considered in Patterson and Hennessey.
- ALU operation: The ALU operation is determined by the opcode.
- Memory access: There is no memory access for data.
- Register write: The result from the ALU is written to rt.

Branch

Branch instructions conditionally branch to an address whose distance is coded into the instruction. Branch instructions use the I coding format.

- PC update: If the branch condition is true (see ALU operation), PC ← PC + 4 + (sign-extended immediate field)<<2.
- Source operand fetch: The two source operands are rs and rt.
- ALU operation: The source operands are subtracted for comparison.
- Memory access: There is no memory access for data.
- Register write: There is no register write.

3. EXPERIMENTAL PROCEDURE

3.1 Equipment used

1. Personal Computer
2. Modelsim
3. Lecture slides and ICAs

4. Online resources

3.2 Project Procedure Description

3.2.1. Task 1: Implementation of individual blocks for single-cycle MIPS processor

1. When it comes to designing a MIPS processor, it's always good to start off with individual blocks required rather than trying to implement it all together.
2. The first glance of implementation on the MIPS processor is focused on the single cycle because it is easier to debug if there are any errors in individual blocks at the beginning.
3. Therefore, the processor is divided into small modules as follows:
    a. ALU
    b. ALU_Control
    c. Branch_Adder
    d. Control_Unit
    e. Data_Memory
    f. Instruction_Memory
    g. Jump_Calc
    h. Program_Counter
    i. Register_File
    j. Sign_Extend
    k. Mux
        i. Operand_mux
        ii. Pc_mux
        iii. Reg_file_mux
        iv. Write_back_mux
    l. Processor_top: which is the top module to connect all the modules for single-cycle MIPS Processor.
4. Once all the modules are connected via the top module, there will be simulations that will be analyzed to check if there are any errors with individual modules before moving to the next stage of the process, forwarding and pipelining.

3.2.2. Task 2: Implementation of Forwarding logic to transform into pipeline logic

1. Once configured the single-cycle MIPS processor from task 1, we will go ahead and implement the forwarding logic by implementing the following modules:
    a. IFID
    b. IDEX
    c. EXMEM
    d. MEMWB

2. Once these modules are completed, then we will configure the processor_top module accordingly to use these blocks for forwarding logic.
3. Once done, simulate the results and check if there are any errors in individual modules and overall wave outputs.

3.2.3. Task 3: Implementation of Hazard Control unit to detect any hazards

1. As mentioned under the theoretical section of this report, while doing forwarding, we will encounter hazards along the way.
2. Therefore, in this step of the project, we have implemented hazard control module to detect any hazard found
3. If it does, then we will generate “bubbles” to make sure pipelining is successfully operating with no problem.

3.2.4. Task 4: Simulations

1. Once at this stage, the 5-stage pipelined MIPS processor design is complete
2. Now we are doing simulations and testing the instructions listed in the project manual to see if we are getting correct results and our processor is doing what we want.
3. The findings for this step will be under the **Analysis section.**

3.2.5. Task 5: Implementation of the processor on FPGA

1. At this stage of the project, we will upload the processor onto the FPGA, specifically the DE2-115 development board.
2. We will test our instructions one by one and check our results
3. It must be noted that we will use FPGAs memory instead of registers while testing our processor.

3.3 Project Execution

3.3.1. Task 1: Implementation of individual blocks for single-cycle MIPS processor

ALU: ALU stands for Arithmetic Logic Unit, which is a fundamental component of a computer's central processing unit (CPU). The ALU is responsible for performing arithmetic and logical operations on binary numbers.

The arithmetic operations include addition, subtraction, multiplication, and division, while the logical operations include AND, OR, NOT, and XOR. The ALU can perform these operations on single bits or on groups of bits, depending on the instruction provided to it by the CPU.

The ALU has inputs for two binary numbers and a control signal that determines the operation to be performed on the numbers. It then performs the operation and outputs the result. The result can be used by other components of the CPU or stored in memory.

Now let's take a look at the ALU Module that is implemented for this project:

```
1   module alu (
2     input      [31:0] operand_a,
3     input      [31:0] operand_b,
4     input      [3: 0] alu_operation,
5     output reg [31:0] alu_result,
6     output            zero
7   );
```

Figure 3.3.1: Initialize

We first must initialize our input, output and output reg ports just like given above, responsible for being a placeholder for operand a, b and output. Then we will start initializing our arithmetic and logical operations one by one with a case number assigned to them.

```
// Calculate the multiplication result
assign mult_result = operand_a * operand_b;

// perform the ALU operation based on the ALU control signal
always @* begin
  case (alu_operation)
    4'b0000: begin // And
      alu_result = operand_a & operand_b;
    end
    4'b0001: begin // Or
      alu_result = operand_a | operand_b;
    end
    4'b0010: begin // Add, ADDI, LW, SW
      alu_result = operand_a + operand_b;
    end
```

Figure 3.3.2: Cases

After initialization of each individual case that has a corresponding arithmetic, logic, or data flow operation, our ALU module is completed. However, we still need to configure the control module that will interact with the ALU module.

ALU CONTROL: The ALU Control Unit (ALU CU) is responsible for controlling the operation of the Arithmetic Logic Unit (ALU) within a CPU. The ALU CU takes in the instruction from the CPU, which specifies the type of arithmetic or logical operation that needs to be performed on the data.

The ALU CU is responsible for generating the appropriate control signals that direct the ALU to perform the correct operation. It determines which arithmetic or logical operation needs to be performed, based on the instruction provided, and generates the appropriate signals to control the ALU's operation.

The ALU CU is also responsible for deciding how the result of the operation should be handled, including setting the appropriate condition codes to indicate whether the result is negative, zero, or positive.

```
module alu_control_unit (
  input      [31:0] instruction,
  input      [1: 0] alu_op,
  output reg [3: 0] alu_control
);

  // Extract the opcode and function fields from the instruction
  reg [5:0] opcode, func;

  // Select function field from the given instruction
  always @* begin
    func = instruction [5:0];
  end
```

Figure 3.3.3: Initialize

The module takes in an instruction and an ALU operation code as input and produces a 4-bit output signal called alu_control, which specifies the operation to be performed by the Arithmetic Logic Unit (ALU).

The code first extracts the opcode and function fields from the instruction. The opcode is a 6-bit field that specifies the type of instruction, and the function field is a 6-bit field that specifies the specific operation to be performed by the ALU for R-Type instructions.

The module then uses a case statement to determine the appropriate value for alu_control based on the input instruction and alu_op. The case statement handles three cases based on the value of alu_op:

Finally, the module outputs the value of alu_control, which is used to control the operation of the ALU. The value of alu_control specifies the specific operation to be performed by the ALU, such as addition, subtraction, multiplication, or logical operations like AND, OR, and NOR.

```
// Control signals for R, I and J-TYPE instructions
always @* begin
  case (alu_op)
      2'b00: begin
          // I and J-Type
          case (instruction[31:26]) // Checking opcode for I-Type instructions as they don't have the function field
              6'b001000: begin // ADDI
                  alu_control = 4'b0010;
              end
              6'b100011: begin // LW
                  alu_control = 4'b0010;
              end
              6'b101011: begin // SW
                  alu_control = 4'b0010;
              end
              default: alu_control = 4'b1111; // NOP
          endcase
      end
      2'b01: begin
          alu_control = 4'b0110; // BEQ
      end
      2'b10: begin
          case (func)
              6'b100000: begin // Add
                  alu_control = 4'b0010;
              end
              6'b100010: begin // Subtract
                  alu_control = 4'b0110;
              end
              6'b100100: begin // And
                  alu_control = 4'b0000;
              end
              6'b100101: begin // Or
                  alu_control = 4'b0001;
              end
              6'b101010: begin // Set on less than
                  alu_control = 4'b0111;
              end
              6'b100111: begin // Nor
                  alu_control = 4'b1000;
              end
              6'b011010: begin // Div
                  alu_control = 4'b1010;
              end
              6'b011000: begin // Mult
                  alu_control = 4'b1011;
              end
              6'b010000: begin // Mfhi
                  alu_control = 4'b0100;
              end
              6'b010010: begin // Mflo
                  alu_control = 4'b0101;
              end
          endcase
      end
      2'b11: begin
          alu_control = 4'b0;
```

Figure 3.3.4: Control signals for R, I and J-TYPE instructions

BRANCH ADDER: The "branch_adder" module performs a critical function in a processor's pipeline to calculate the address of the next instruction after a branch instruction is executed. This module is used in conjunction with other components of a processor's pipeline to ensure correct execution of branch instructions.

```
1   module branch_adder (
2     input             branch,
3     input      [31:0] current_pc,
4     input      [31:0] sign_extend,
5     output reg [31:0] pc_branch
6   );
7
8   // Calculate the address for the branch
9   always @* begin
10      pc_branch = branch ? current_pc + sign_extend : '0;
11  end
12
13  endmodule
```

Figure 3.3.5: Branch adder

The module takes in three input signals: "branch", "current_pc", and "sign_extend", and produces an output signal "pc_branch", which is the address of the next instruction to be executed after a branch instruction is encountered.

The "branch" signal is a single bit that indicates whether a branch is being taken or not. The "current_pc" signal is a 32-bit input that represents the current program counter (PC), which is the address of the current instruction being executed. The "sign_extend" signal is a 32-bit input that represents the sign-extended immediate value of the branch instruction.

The module uses an "always" block to calculate the address of the next instruction after a branch. If "branch" is high, indicating that a branch is being taken, the module adds the "sign_extend" value to the "current_pc" value to calculate the address of the next instruction. If "branch" is low, indicating that a branch is not being taken, the module sets the "pc_branch" value to '0'.

The output signal "pc_branch" is a 32-bit value that represents the address of the next instruction to be executed after a branch. If the branch is taken, the value of "pc_branch" will be equal to "current_pc" plus the "sign_extend" value. If the branch is not taken, the value of "pc_branch" will be 0.

DATA MEMORY: This module essentially implements a simple memory unit where data can be written to or read from specific memory locations based on an address input.

```
module data_memory (
  input             clk,
  input      [31:0] address,
  input      [31:0] write_data,
  input             mem_write,
  input             mem_read,
  output reg [31:0] read_data
);

  reg [31:0] RAM [0:1023];

  // Initial value of the memory (for testing purposes)
  initial begin
   // RAM [102] = 32'h000000FF;
  end

  // Memory read and write operation (The write operation gets priority over the read operation)
  always @* begin
    if (mem_write) begin
      RAM [address] = write_data;
    end
    if (mem_read) begin
      read_data = RAM [address];
    end
  end

endmodule
```

Figure 3.3.6: Data memory

This code defines a module for a data memory unit in a digital system. The module has a synchronous interface consisting of an address input (address), a write data input (write_data), and read/write control signals (mem_read and mem_write) that are triggered by a clock signal (clk). The module also has an output read_data which represents the data read from the memory location specified by the address input.

The module implements the memory using an array of 1024 32-bit registers (RAM), initialized to zero. The address input is used to index the RAM array to perform memory read or write operations. When mem_write is asserted, the write_data input is written to the memory location specified by the address input. When mem_read is asserted, the read_data output is updated with the data stored in the memory location specified by the address input.

INSTRUCTION MEMORY: The instruction_memory module provides an interface for a processor to read instructions from a memory based on a specified address.

```
module instruction_memory (
  input            clk,        // clock input (active-high)
  input      [31:0] address,
  output reg [31:0] instruction
);

  reg [7:0] mem[0:4095];

  initial begin

        // R-TYPE
        {mem[3 ],mem[2 ],mem[1 ],mem[0 ]}    =   32'h00430820; // add
        {mem[7 ],mem[6 ],mem[5 ],mem[4 ]}    =   32'h00430824; // and
        {mem[11],mem[10],mem[9 ],mem[8 ]}    =   32'h00430825; // or
        {mem[15],mem[14],mem[13],mem[12]}    =   32'h00430827; // nor
        {mem[19],mem[18],mem[17],mem[16]}    =   32'h00430822; // sub
        {mem[23],mem[22],mem[21],mem[20]}    =   32'h0043082A; // slt
        {mem[27],mem[26],mem[25],mem[24]}    =   32'h0043081A; // div
        {mem[31],mem[30],mem[29],mem[28]}    =   32'h00430818; // mul
        {mem[35],mem[34],mem[33],mem[32]}    =   32'h00000810; // mfhi
        {mem[39],mem[38],mem[37],mem[36]}    =   32'h00000812; // mflo

    // I-Type
        {mem[43],mem[42],mem[41],mem[40]}    =   32'h20410003; // addi
        {mem[47],mem[46],mem[45],mem[44]}    =   32'h10850019; // beq
        {mem[51],mem[50],mem[49],mem[48]}    =   32'h8c410064; // lw
        {mem[55],mem[54],mem[53],mem[52]}    =   32'hAC410064; // sw

    // J-Type
    {mem[59],mem[58],mem[57],mem[56]}    =   32'h08000101; // j

  end

  always @* begin
    instruction = {mem[address+3], mem[address+2], mem[address+1], mem[address+0]};
  end

endmodule
```

Figure 3.3.7: Instructions

The instruction_memory module is a memory unit that stores instructions for a processor. It has three inputs:

1. clk: a clock signal
2. address: a 32-bit input representing the memory address to read from
3. instruction: a 32-bit output representing the instruction read from the memory

The module uses a reg array called mem to store the instructions. The initial values of this array are set in the initial block of the module. Each instruction is stored as a 32-bit value in the mem array.

The always block in the module is triggered whenever the address changes. It reads the 32-bit instruction stored in the mem array at the address specified by address, and assigns this instruction to the instruction output. This means that the instruction output will always contain the instruction stored at the memory address specified by address.

JUMP CALCULATION: Calculates the target jump address and determines whether the jump should be taken or not.

```
module jum_calc(
  input      [31:0] old_pc,
  input      [25:0] target_address,
  input             jump,
  output reg [31:0] jump_pc
);

  always @* begin
    jump_pc = (jump) ? {old_pc[31:28], target_address, 2'b00} : '0;
  end

endmodule
```

Figure 3.3.7: Jump calculation

The old_pc input is the current address of the instruction being executed. The target_address input is the 26-bit address of the instruction to which the program should jump. The jump input is a signal that determines whether the jump should be taken.

The jump_pc output is the computed target address of the jump instruction. The module uses the jump signal to decide whether to take the jump or not. If the jump signal is asserted, the module constructs the jump address by concatenating the upper 4 bits of the current address (old_pc[31:28]) with the 26-bit target address (target_address) and appending 2 bits of zero (2'b00) to the least significant end. Otherwise, the module sets jump_pc to zero.

PROGRAM COUNTER: The program_counter module is responsible for keeping track of the program counter (PC) for a processor. The program counter is a register that stores the address of the current instruction being executed, and it is updated after each instruction is executed.

```
module program_counter (
  input             clk,       // clock input (active-high)
  input             reset,     // reset input (active-low)
  input      [31:0] pc_in,     // updated pc for the branch/jump instruction(s)
  output reg [31:0] pc_adder,  // pc of the next instruction to be executed
  output reg [31:0] pc_out     // pc of the next instruction to be executed
);

  always @(posedge clk or negedge reset) begin
    if (reset) begin
      pc_out <= 32'd0;
    end else begin
      pc_out <= pc_in;
    end
  end

  always @* begin
    pc_adder = pc_out + 4;
  end

endmodule
```

Figure 3.3.8: Program counter

The module has four ports: clk, reset, pc_in, pc_adder, and pc_out. clk is the clock signal for the processor, reset is the reset signal, pc_in is the updated program counter value, pc_adder is the program counter for the next instruction to be executed, and pc_out is the program counter for the current instruction being executed.

The always @(posedge clk or negedge reset) block handles the clock and reset signals. When reset is active low, the pc_out is set to 0. Otherwise, the pc_out is set to the updated program counter value pc_in during the positive edge of the clk signal.

The always @* block is a combinational logic that calculates the pc_adder value by adding 4 to pc_out. This value is used to determine the address of the next instruction to be executed after the current instruction.

REGISTER FILE: A register file is a collection of registers that are used in a digital circuit to temporarily store and manipulate data during computation. The registers can be accessed by the circuit's control unit, which retrieves data from the registers, performs arithmetic or logical operations on the data, and stores the results back into the registers.

```
module register_file (
  input         clk,             // clock input (active-high)
  input  [4: 0] read_register_1,
  input  [4: 0] read_register_2,
  input  [4: 0] write_register,
  input  [31:0] write_data,
  input         reg_write,
  output [31:0] read_data_1,
  output [31:0] read_data_2
);

  reg [31:0] registers[0:31];


  // initial values of the registers (for testing purposes)
  initial begin
    registers [1] = 32'h2;
    registers [2] = 32'h2;
    registers [3] = 32'h3;
    registers [4] = 32'h5;
    registers [5] = 32'h5;
  end

  always @(posedge clk) begin
    if (reg_write) begin
      registers[write_register] <= write_data;
    end
  end

  assign read_data_1 = registers[read_register_1];
  assign read_data_2 = registers[read_register_2];

endmodule
```

Figure 3.3.9: Register file

The module has the following inputs and outputs:

Inputs:

- clk: the clock input signal.
- read_register_1: a 5-bit input signal that specifies the first register to be read.
- read_register_2: a 5-bit input signal that specifies the second register to be read.
- write_register: a 5-bit input signal that specifies the register to be written to.
- write_data: a 32-bit input signal that specifies the data to be written to the register.
- reg_write: an input signal that specifies whether the write operation is to be performed.

Outputs:

- read_data_1: a 32-bit output signal that contains the data read from the first register.
- read_data_2: a 32-bit output signal that contains the data read from the second register.

The module contains a 32-bit array of registers called "registers", with one register for each of the 32 possible register addresses. The initial values of the registers are set in an "initial" block, which is used for testing purposes.

The "always" block is triggered by the positive edge of the clock signal. If the "reg_write" signal is high, the data specified by the "write_data" signal is written to the register specified by the "write_register" signal.

The "assign" statements are used to assign the data read from the specified registers to the "read_data_1" and "read_data_2" output signals.

REGISTER FILE MUX: The reg_file_mux module is a multiplexer that selects the destination register for a write operation in a MIPS processor's register file based on the instruction being executed.

```
1  module reg_file_mux (
2    input  [31:0] instruction,
3    input         reg_dst,
4    output [4: 0] mux_output
5  );
6
7    assign mux_output = reg_dst ? instruction[20:16] : instruction[15:11];
8
9  endmodule
```

Figure 3.3.10: Register file mux

The module takes in a 32-bit instruction signal and a control signal reg_dst. The reg_dst signal determines which of two fields in the instruction contains the destination register number. If reg_dst is high, then bits 20-16 of the instruction contain the register number, otherwise bits 15-11 are used. The module outputs the selected register number on a 5-bit signal mux_output.

SIGN EXTEND: "sign_extend" takes a 16-bit input "in_data" and extends the sign bit (the leftmost bit) to fill the remaining 16 bits, creating a 32-bit output "out_data".

```
module sign_extend(
  input  [15:0] in_data,
  output [31:0] out_data
);

  assign out_data = { {16{in_data[15]}}, in_data };

endmodule
```

Figure 3.3.11: Sign extend

The sign bit is typically used to indicate whether the number is positive or negative. If the sign bit is 0, the number is positive; if it is 1, the number is negative. In a two's complement system, the sign bit is also used to represent the magnitude of the number.

The code uses a concatenation operator ({ }) to create a 32-bit value. The first argument is an array of 16 copies of the sign bit (in_data[15]), which is used to fill the upper 16 bits of the output. The second argument is the original 16-bit input (in_data), which is used to fill the lower 16 bits of the output. The resulting 32-bit output is then assigned to "out_data".

PROGRAM COUNTER MUX: PC (program counter) multiplexer selects the next program counter address. It has five inputs and one output.

```
module pc_mux (
  input  [31:0] pc_in,
  input         branch,
  input         jump,
  input         zero,
  input  [31:0] jump_pc,
  input  [31:0] jump_address,
  output [31:0] imem_address
);

  assign imem_address = (pc_in == '0) ? pc_in : (branch & zero) ? jump_address : (jump) ? jump_pc : pc_in;

endmodule
```

Figure 3.3.12: PC Mux

The pc_in input is the current program counter address, which is the address of the instruction to be executed next. The branch input indicates whether the current instruction is a branch instruction or not. The jump input indicates whether the current instruction is a jump instruction or not. The zero input is the result of the ALU's zero flag, which indicates whether the previous ALU operation resulted in a zero value. The jump_pc input is the program counter value that should be used if the current instruction is a

jump instruction.The jump_address input is the jump target address, which is computed based on the instruction's offset value.

The imem_address output is the selected program counter address that should be used to fetch the next instruction. It is computed based on the current program counter address, branch condition, jump condition, and jump target address. If pc_in is zero, the output is also set to zero. Otherwise, if the current instruction is a branch instruction and the zero flag is set, the output is set to the jump target address. If the current instruction is a jump instruction, the output is set to the jump program counter address. Otherwise, the output is set to the next sequential program counter address.

OPERAND MUX: Operand_mux implements a multiplexer that selects between two input operands based on a control signal alu_src. If alu_src is high, then the output operand_b_out is set to the immediate value immediate, otherwise it is set to the value of the second input operand_b. This multiplexer is typically used in a CPU's datapath to select between two sources of operands for an arithmetic or logical operation, depending on the instruction being executed.

```
1   module operand_mux (
2     input  [31:0] operand_b,
3     input  [31:0] immediate,
4     input         alu_src,
5     output [31:0] operand_b_out
6   );
7
8     assign operand_b_out = alu_src ? immediate : operand_b;
9
10  endmodule
```

Figure 3.3.13: Operand mux

WRITE BACK MUX: This module implements a multiplexer (mux) used for selecting data to be written back to a register in a processor's write-back stage. The module has three inputs: read_data, alu_result, and mem_to_reg, and one output write_data. The read_data input represents the data read from memory in the memory access stage, alu_result is the output of the arithmetic and logic unit (ALU) in the execution stage, and mem_to_reg is a control signal that determines whether the output of the memory access stage or the output of the ALU is selected for writing back to a register.

```
1   module write_back_mux (
2     input  [31:0] read_data,
3     input  [31:0] alu_result,
4     input         mem_to_reg,
5     output [31:0] write_data
6   );
7
8     assign write_data = mem_to_reg ? read_data : alu_result;
9
10  endmodule
```

Figure 3.3.14: Write back mux

The write_data output of the module is set to read_data if mem_to_reg is asserted (i.e., is equal to 1), otherwise it is set to alu_result. This mux is used to select the correct data to be written back to a register in a processor's write-back stage, depending on whether the instruction requires data from memory or the output of the ALU.

CONTROL UNIT: The control unit takes an instruction as input and generates various control signals based on the opcode and function fields of the instruction.

The module defines output signals for reg_dst, branch, jump, mem_read, mem_to_reg, alu_op, mem_write, alu_src, and reg_write. These signals are used to control other parts of the processor, such as the register file, ALU, and memory.

```
module control_unit (
  input      [31:0] instruction,
  output reg        reg_dst,
  output reg        branch,
  output reg        jump,
  output reg        mem_read,
  output reg        mem_to_reg,
  output reg [1: 0] alu_op,
  output reg        mem_write,
  output reg        alu_src,
  output reg        reg_write
);
```

Figure 3.3.15: Initialize

The control unit extracts the opcode field from the instruction and uses it to determine which instruction type it is dealing with: R-Type, I-Type, or J-Type. It then sets the control signals accordingly.

```
// Extract the opcode and function fields from the instruction
reg [5:0] opcode;

// Shuffle unit
always @* begin
  opcode = instruction [31:26];
end

// control signals for R, I and J-TYPE instructions
always @* begin
  case (opcode)
    // R-Type
    6'b000000: begin
      reg_dst    = 1'b0;
      branch     = 1'b0;
      jump       = 1'b0;
      mem_read   = 1'b0;
      mem_to_reg = 1'b0;
      alu_op     = 2'b10;
      mem_write  = 1'b0;
      alu_src    = 1'b0;
      reg_write  = 1'b1;
    end
```

Figure 3.3.16: R-type instruction

This part of the code is defining the values of various control signals based on the opcode field of the input instruction. These control signals are used to enable or disable various components of the processor during the execution of the instruction.

The case statement checks the value of the opcode variable and executes the corresponding code block. In this particular code block, which corresponds to R-Type instructions, the control signals are set as follows:

- reg_dst: This signal indicates whether the instruction writes to a register or not. In this case, it is set to 0, indicating that the instruction does not write to a register.
- branch: This signal indicates whether the instruction is a branch instruction. In this case, it is set to 0, indicating that the instruction is not a branch instruction.
- jump: This signal indicates whether the instruction is a jump instruction. In this case, it is set to 0, indicating that the instruction is not a jump instruction.
- mem_read: This signal indicates whether the instruction performs a memory read operation. In this case, it is set to 0, indicating that the instruction does not perform a memory read operation.
- mem_to_reg: This signal indicates whether the data read from memory should be written to a register or not. In this case, it is set to 0, indicating that the data should not be written to a register.
- alu_op: This signal indicates the operation that should be performed by the ALU. In this case, it is set to 2'b10, indicating that the ALU should perform a subtraction operation.
- mem_write: This signal indicates whether the instruction performs a memory write operation. In this case, it is set to 0, indicating that the instruction does not perform a memory write operation.
- alu_src: This signal indicates whether the second operand of the ALU should come from the immediate field or the second register file. In this case, it is set to 0, indicating that the second operand should come from the second register file.
- reg_write: This signal indicates whether the instruction writes to a register or not. In this case, it is set to 1, indicating that the instruction writes to a register.

```
// I-Type
6'b001000: begin // ADDI
  reg_dst    = 1'b1;
  branch     = 1'b0;
  jump       = 1'b0;
  mem_read   = 1'b0;
  mem_to_reg = 1'b0;
  alu_op     = 2'b00;
  mem_write  = 1'b0;
  alu_src    = 1'b1;
  reg_write  = 1'b1;
end
6'b000100: begin // BEQ
  reg_dst    = 1'b0;
  branch     = 1'b1;
  jump       = 1'b0;
  mem_read   = 1'b0;
  mem_to_reg = 1'b0;
  alu_op     = 2'b01;
  mem_write  = 1'b0;
  alu_src    = 1'b0;
  reg_write  = 1'b0;
end
```

```
6'b100011: begin // LW
  reg_dst    = 1'b1;
  branch     = 1'b0;
  jump       = 1'b0;
  mem_read   = 1'b1;
  mem_to_reg = 1'b1;
  alu_op     = 2'b00;
  mem_write  = 1'b0;
  alu_src    = 1'b1;
  reg_write  = 1'b1;
end
6'b101011: begin // SW
  reg_dst    = 1'b1;
  branch     = 1'b0;
  jump       = 1'b0;
  mem_read   = 1'b0;
  mem_to_reg = 1'b0;
  alu_op     = 2'b00;
  mem_write  = 1'b1;
  alu_src    = 1'b1;
  reg_write  = 1'b0;
end
```

Figure 3.3.17: I-type instruction

This is a block of code within the always block that assigns control signals to various outputs based on the value of the opcode signal, which determines the type of instruction being executed.

For example the BEQ instructions, the code sets the following signals:

- reg_dst to 1'b0, indicating that this instruction does not specify a destination register
- branch to 1'b1, indicating that this instruction is a branch instruction
- jump to 1'b0, indicating that this instruction is not a jump instruction
- mem_read to 1'b0, indicating that this instruction does not read from memory
- mem_to_reg to 1'b0, indicating that this instruction does not write to memory
- alu_op to 2'b01, indicating that the ALU operation for this instruction is subtraction
- mem_write to 1'b0, indicating that this instruction does not write to memory
- alu_src to 1'b0, indicating that the second operand of the ALU operation comes from the register file
- reg_write to 1'b0, indicating that this instruction does not write to a register.

PROCESSOR TOP MODULE: This module is an implementation of a processor's datapath, containing several sub-modules for the various stages of the processor's operation, such as instruction fetch, decode, execute, and memory access. Each sub-module is set to be interconnected using several input and output wires, which are used to pass data and control signals between modules.

Here are some of the key sub-modules and wires in this design:

- PC Selection MUX: selects between the output of the branch adder, jump calculator, and instruction memory to determine the next address to fetch from
- Program Counter: generates the next address to fetch from, based on the output of the PC Selection MUX
- Instruction Memory: reads the instruction at the current address specified by the Program Counter
- Control Unit: decodes the instruction and generates control signals for the rest of the datapath, such as whether to read from or write to memory or registers
- Register File MUX: selects between the destination register specified by the instruction and a register specified by the Control Unit
- Register File: reads from or writes to the register file based on the input signals
- ALU Control Unit: generates the appropriate ALU operation based on the instruction and control signals
- Sign Extend: extends the immediate value in the instruction to a full 32-bit value
- Branch Adder: computes the branch address based on the current PC and the sign-extended immediate value in the instruction
- Jump Calculator: computes the jump address based on the current PC and the immediate value in the instruction

In summary, all of the modules that had been created so far for the single-cycle processor, come all together and get wired in this module to be interconnected with one another to pass the data, address, and values generated by the instructions.

3.3.2. Task 2: Implementation of Forwarding logic to transform into pipeline logic

1. Since the single-cycle MIPS processor from task 1, we will go ahead and implement the forwarding logic by implementing the following modules:

IF_ID MODULE: This module is responsible for storing the fetched instruction and its associated program counter value in registers so that they can be passed to the next stage of the pipeline.

```
module IFID (
    input             clk,
    input             reset,
    input             flush,
    input      [31:0] instruction,
    input      [31:0] pc_in,
    output reg [31:0] instruction_ifid,
    output reg [31:0] pc_ifid
);

always @(posedge clk) begin
    if (reset | flush) begin
        instruction_ifid <= '0;
        pc_ifid          <= '0;
    end else begin
        instruction_ifid <= instruction;
        pc_ifid          <= pc_in;
    end
end

endmodule
```

Figure 3.3.18: IF/ID Module

The inputs to the module are clk, reset, flush, instruction, and pc_in. The clk input is a clock signal that is used to synchronize the operations of the module. The reset and flush inputs are used to reset and flush the pipeline, respectively. The instruction input is the 32-bit instruction fetched from memory, and the pc_in input is the program counter value associated with the fetched instruction.

The module has two outputs: instruction_ifid and pc_ifid. These are both 32-bit registers that store the instruction and program counter values, respectively, in the instruction fetch/decode stage of the pipeline.

The module uses an always block that triggers on the positive edge of the clk signal. If either the reset or flush inputs are asserted, the registers are reset to zero. Otherwise, the values of instruction and pc_in are loaded into instruction_ifid and pc_ifid, respectively.

ID_EX MODULE: The IDEX stage takes the instruction fetched in the Instruction Fetch stage and decodes it, preparing it for execution in the Execute stage.

```
module IDEX (
    input             clk,
    input             reset,
    input             flush,
    input             reg_dst,
    input             branch,
    input             jump,
    input             mem_read,
    input             mem_to_reg,
    input      [1: 0] alu_op,
    input             mem_write,
    input             alu_src,
    input             reg_write,
    input      [31:0] operand_a,
    input      [31:0] operand_b,
    input      [31:0] instruction_ifid,
    input      [31:0] pc_ifid,
    input      [4: 0] wb_address,
    output reg        reg_dst_idex,
    output reg        branch_idex,
    output reg        jump_idex,
    output reg        mem_read_idex,
    output reg        mem_to_reg_idex,
    output reg [1: 0] alu_op_idex,
    output reg        mem_write_idex,
    output reg        alu_src_idex,
    output reg        reg_write_idex,
    output reg [31:0] operand_a_idex,
    output reg [31:0] operand_b_idex,
    output reg [31:0] instruction_idex,
    output reg [31:0] pc_idex,
    output reg [4: 0] wb_address_idex
);
```

Figure 3.3.19: IDEX initialization

The module takes several input signals representing various control signals and data from the IFID stage and provides the necessary control signals and data for the Execute stage. The inputs include clk for the clock, reset for reset, flush to clear the pipeline in case of a branch misprediction, instruction_ifid and pc_ifid to receive instruction and program counter from the previous stage, and operand_a and operand_b to receive data from the register file, among others.

The output signals are derived from the inputs after some computations and include instruction_idex which carries the decoded instruction, pc_idex which carries the program counter for the current instruction, and operand_a_idex and operand_b_idex which carry the operands for the current instruction. Other output signals include various control signals for
execution such as reg_write_idex, mem_read_idex, mem_write_idex, and alu_op_idex, among others.

```
always @(posedge clk) begin
    if (reset | flush) begin
        reg_dst_idex     <= '0;
        branch_idex      <= '0;
        jump_idex        <= '0;
        mem_read_idex    <= '0;
        mem_to_reg_idex  <= '0;
        alu_op_idex      <= '0;
        mem_write_idex   <= '0;
        alu_src_idex     <= '0;
        reg_write_idex   <= '0;
        operand_a_idex   <= '0;
        operand_b_idex   <= '0;
        instruction_idex <= '0;
        pc_idex          <= '0;
        wb_address_idex  <= '0;
    end else begin
        reg_dst_idex     <= reg_dst;
        branch_idex      <= branch;
        jump_idex        <= jump;
        mem_read_idex    <= mem_read;
        mem_to_reg_idex  <= mem_to_reg;
        alu_op_idex      <= alu_op;
        mem_write_idex   <= mem_write;
        alu_src_idex     <= alu_src;
        reg_write_idex   <= reg_write;
        operand_a_idex   <= operand_a;
        operand_b_idex   <= operand_b;
        instruction_idex <= instruction_ifid;
        pc_idex          <= pc_ifid;
        wb_address_idex  <= wb_address;
    end
end
```

Figure 3.3.20: IDEX always module

EX_MEM MODULE: The EXMEM module is a module that represents a stage in a pipelined processor. Specifically, this module represents the execution-to-memory stage of the pipeline, where results from the execution stage are written to the memory stage.

```
module EXMEM (
    input             clk,
    input             reset,
    input             reg_dst_idex,
    input             branch_idex,
    input             jump_idex,
    input             mem_read_idex,
    input             mem_to_reg_idex,
    input             alu_op_idex,
    input             mem_write_idex,
    input      [1: 0] alu_src_idex,
    input             reg_write_idex,
    input      [31:0] operand_a_idex,
    input      [31:0] operand_b_idex,
    input      [31:0] instruction_idex,
    input      [31:0] alu_result,
    input             zero,
    input      [31:0] pc_idex,
    input      [4: 0] wb_address_idex,
    output reg        reg_dst_exmem,
    output reg        branch_exmem,
    output reg        jump_exmem,
    output reg        mem_read_exmem,
    output reg        mem_to_reg_exmem,
    output reg        alu_op_exmem,
    output reg        mem_write_exmem,
    output reg [1: 0] alu_src_exmem,
    output reg        reg_write_exmem,
    output reg [31:0] operand_a_exmem,
    output reg [31:0] operand_b_exmem,
    output reg [31:0] instruction_exmem,
    output reg [31:0] alu_result_exmem,
    output reg        zero_exmem,
    output reg [31:0] pc_exmem,
    output reg [4: 0] wb_address_exmem
);
```

Figure 3.3.21: EXMEM initialization

The module has several inputs, including control signals from the previous instruction decode-to-execution stage (IDEX), as well as the result of the arithmetic/logic unit (ALU) operation, the program counter (PC), and the address of the register being written to (wb_address). There are also several outputs, which are the same control signals and data values but are passed on to the next stage of the pipeline (the memory-to-writeback stage, or MEMWB).

The always block is a synchronous always block that is sensitive to the rising edge of the clock signal (posedge clk). When the reset input is asserted, all output registers are reset to zero. Otherwise, the output registers are updated with the input values on the rising edge of the clock.

```
38  always @(posedge clk) begin
39      if (reset) begin
40          reg_dst_exmem     <= '0;
41          branch_exmem      <= '0;
42          jump_exmem        <= '0;
43          mem_read_exmem    <= '0;
44          mem_to_reg_exmem  <= '0;
45          alu_op_exmem      <= '0;
46          mem_write_exmem   <= '0;
47          alu_src_exmem     <= '0;
48          reg_write_exmem   <= '0;
49          operand_a_exmem   <= '0;
50          operand_b_exmem   <= '0;
51          instruction_exmem <= '0;
52          alu_result_exmem  <= '0;
53          zero_exmem        <= '0;
54          pc_exmem          <= '0;
55          wb_address_exmem  <= '0;
56      end else begin
57          reg_dst_exmem     <= reg_dst_idex;
58          branch_exmem      <= branch_idex;
59          jump_exmem        <= jump_idex;
60          mem_read_exmem    <= mem_read_idex;
61          mem_to_reg_exmem  <= mem_to_reg_idex;
62          alu_op_exmem      <= alu_op_idex;
63          mem_write_exmem   <= mem_write_idex;
64          alu_src_exmem     <= alu_src_idex;
65          reg_write_exmem   <= reg_write_idex;
66          operand_a_exmem   <= operand_a_idex;
67          operand_b_exmem   <= operand_b_idex;
68          instruction_exmem <= instruction_idex;
69          alu_result_exmem  <= alu_result;
70          zero_exmem        <= zero;
71          pc_exmem          <= pc_idex;
72          wb_address_exmem  <= wb_address_idex;
73      end
74  end
```

Figure 3.3.22: EXMEM always block

The always block is executed on the positive edge of the clock signal (@(posedge clk)), and it synchronously transfers the input signals from the previous pipeline stage (IDEX) to the current pipeline stage (EXMEM). The transfer is controlled by a reset signal, which sets all the output registers to zero when asserted.

When the reset signal is not asserted, the values of the input signals are assigned to the output registers. The output registers store the control signals and data values that are needed by the next pipeline stage to perform the memory access operation or to calculate the next instruction address.

Each output register corresponds to a control signal or data value, and it is assigned the value of the corresponding input signal. For example, the reg_dst_exmem output register stores the control signal that indicates whether the destination register for the result of the ALU operation is the register file or the immediate value, and it is assigned the value of the reg_dst_idex input signal. Similarly, the alu_result_exmem output register stores the result of the ALU operation and is assigned the value of the alu_result input signal.

MEM_WB MODULE: The purpose of this stage is to write data back to the register file or memory, completing the execution of the instruction that was started in the previous pipeline stage.

```
module MEMWB (
    input            clk,
    input            reset,
    input            reg_dst_exmem,
    input            branch_exmem,
    input            jump_exmem,
    input            mem_read_exmem,
    input            mem_to_reg_exmem,
    input            alu_op_exmem,
    input            mem_write_exmem,
    input      [1: 0] alu_src_exmem,
    input            reg_write_exmem,
    input      [31:0] operand_a_exmem,
    input      [31:0] operand_b_exmem,
    input      [31:0] instruction_exmem,
    input      [31:0] alu_result_exmem,
    input            zero_exmem,
    input      [31:0] read_data,
    input      [31:0] pc_exmem,
    input      [4: 0] wb_address_exmem,
    output reg       reg_dst_memwb,
    output reg       branch_memwb,
    output reg       jump_memwb,
    output reg       mem_read_memwb,
    output reg       mem_to_reg_memwb,
    output reg       alu_op_memwb,
    output reg       mem_write_memwb,
    output reg [1: 0] alu_src_memwb,
    output reg       reg_write_memwb,
    output reg [31:0] operand_a_memwb,
    output reg [31:0] operand_b_memwb,
    output reg [31:0] instruction_memwb,
    output reg [31:0] alu_result_memwb,
    output reg       zero_memwb,
    output reg [31:0] read_data_memwb,
    output reg [31:0] pc_memwb,
    output reg [4: 0] wb_address_memwb
);
```

Figure 3.3.23: MEMWB initialization

The always @(posedge clk) block is a synchronous design that is triggered by the positive edge of the clock signal. If the reset input is high, then all the output signals are reset to zero. Otherwise, the output signals are updated to match the corresponding input signals from the previous pipeline stage.

The input signals from the previous pipeline stage include the results of memory reads, the result of an ALU operation, and various control signals that determine what operation should be performed in the current stage. The output signals are used to write data back to the register file or memory, depending on the instruction that was executed in the previous stage.

```
40  always @(posedge clk) begin
41      if (reset) begin
42          reg_dst_memwb     <= '0;
43          branch_memwb      <= '0;
44          jump_memwb        <= '0;
45          mem_read_memwb    <= '0;
46          mem_to_reg_memwb  <= '0;
47          alu_op_memwb      <= '0;
48          mem_write_memwb   <= '0;
49          alu_src_memwb     <= '0;
50          reg_write_memwb   <= '0;
51          operand_a_memwb   <= '0;
52          operand_b_memwb   <= '0;
53          instruction_memwb <= '0;
54          alu_result_memwb  <= '0;
55          zero_memwb        <= '0;
56          read_data_memwb   <= '0;
57          pc_memwb          <= '0;
58          wb_address_memwb  <= '0;
59      end else begin
60          reg_dst_memwb     <= reg_dst_exmem;
61          branch_memwb      <= branch_exmem;
62          jump_memwb        <= jump_exmem;
63          mem_read_memwb    <= mem_read_exmem;
64          mem_to_reg_memwb  <= mem_to_reg_exmem;
65          alu_op_memwb      <= alu_op_exmem;
66          mem_write_memwb   <= mem_write_exmem;
67          alu_src_memwb     <= alu_src_exmem;
68          reg_write_memwb   <= reg_write_exmem;
69          operand_a_memwb   <= operand_a_exmem;
70          operand_b_memwb   <= operand_b_exmem;
71          instruction_memwb <= instruction_exmem;
72          alu_result_memwb  <= alu_result_exmem;
73          zero_memwb        <= zero_exmem;
74          read_data_memwb   <= read_data;
75          pc_memwb          <= pc_exmem;
76          wb_address_memwb  <= wb_address_exmem;
77      end
78  end
```

Figure 3.3.24: MEMWB always block

UPDATING PROCESSOR TOP MODULE: After completing all individual blocks required for the pipeline logic, we must interconnect the wires within the top module to make it work. Therefore, we have updated the previous "single-cycle processor top" module to now operate as a forwarding supported pipeline processor.

```
76  ////////////////////////////////
77  /// PIPELINE CONNECTION WIRES ///
78  ////////////////////////////////
79
80  // if/id stage variables
81  wire [31:0] instruction_ifid;
82  wire [31:0] pc_ifid;
83
```

Figure 3.3.25: IFID Wires

These two lines declare wire variables instruction_ifid and pc_ifid with 32-bit widths. They will be used to pass data between the IF (instruction fetch) stage and the ID (instruction decode) stage in your top-level module.

The instruction_ifid wire will carry the 32-bit instruction fetched from memory during the IF stage to the ID stage for decoding and execution.

The pc_ifid wire will carry the 32-bit program counter value for the next instruction to be fetched during the IF stage. It is passed to the ID stage so that it can be used to calculate the address of the next instruction in memory to be fetched.

The module below are responsible for wiring the required input/output ports of the id/ex stage in our top module.

```
84  // id/ex stage variables
85  wire        reg_dst_idex;
86  wire        branch_idex;
87  wire        jump_idex;
88  wire        mem_read_idex;
89  wire        mem_to_reg_idex;
90  wire [1: 0] alu_op_idex;
91  wire        mem_write_idex;
92  wire        alu_src_idex;
93  wire        reg_write_idex;
94  wire [31:0] operand_a_idex;
95  wire [31:0] operand_b_idex;
96  wire [31:0] instruction_idex;
97  wire [31:0] pc_idex;
98  wire [4: 0] wb_address_idex;
99
```

Figure 3.3.26: IDEX Wires

These wires represent various signals that are used in the ID/EX stage of the pipeline in a processor design.

- reg_dst_idex: Selects the destination register for the writeback stage (either rt or rd)
- branch_idex: Indicates if the current instruction is a branch
- jump_idex: Indicates if the current instruction is a jump
- mem_read_idex: Indicates if the current instruction is a memory read operation
- mem_to_reg_idex: Selects the data source for the writeback stage (either memory or ALU)
- alu_op_idex: Specifies the type of operation to be performed by the ALU in the EX stage
- mem_write_idex: Indicates if the current instruction is a memory write operation
- alu_src_idex: Selects the second operand for the ALU in the EX stage (either rt or immediate value)
- reg_write_idex: Indicates if the current instruction is a register write operation
- operand_a_idex: The value of the first operand for the ALU in the EX stage
- operand_b_idex: The value of the second operand for the ALU in the EX stage
- instruction_idex: The current instruction being executed
- pc_idex: The current program counter value for the current instruction being executed
- wb_address_idex: The register address to be written back to in the writeback stage.

The module below is responsible for wiring the required input/output ports of the ex/mem stage in our top module.

```
100   // ex/mem stage variables
101   wire        reg_dst_exmem;
102   wire        branch_exmem;
103   wire        jump_exmem;
104   wire        mem_read_exmem;
105   wire        mem_to_reg_exmem;
106   wire        alu_op_exmem;
107   wire        mem_write_exmem;
108   wire [1: 0] alu_src_exmem;
109   wire        reg_write_exmem;
110   wire [31:0] operand_a_exmem;
111   wire [31:0] operand_b_exmem;
112   wire [31:0] instruction_exmem;
113   wire [31:0] alu_result_exmem;
114   wire        zero_exmem;
115   wire [31:0] pc_exmem;
116   wire [4: 0] wb_address_exmem;
```

Figure 3.3.27: EXMEM Wires

During this stage, the processor executes the instruction that was fetched and decoded in the previous stages. It calculates the address of memory for any memory read or write operations, performs any necessary arithmetic or logical operations, and updates any registers that need to be written to.

The signals and data carried by these wires are used to perform these operations and to pass data and control signals between the various stages of the pipeline. For example, alu_result_exmem carries the result of the arithmetic or logical operation that was performed during this stage, and wb_address_exmem contains the address of the register to which the result should be written.

Now, the final stage of pipelining is the MEM\WB stage, which is given below and wired in top module.

```
118    // mem/wb stage variables
119    wire        reg_dst_memwb;
120    wire        branch_memwb;
121    wire        jump_memwb;
122    wire        mem_read_memwb;
123    wire        mem_to_reg_memwb;
124    wire        alu_op_memwb;
125    wire        mem_write_memwb;
126    wire [1: 0] alu_src_memwb;
127    wire        reg_write_memwb;
128    wire [31:0] operand_a_memwb;
129    wire [31:0] operand_b_memwb;
130    wire [31:0] instruction_memwb;
131    wire [31:0] alu_result_memwb;
132    wire        zero_memwb;
133    wire [31:0] read_data_memwb;
134    wire [31:0] pc_memwb;
135    wire [4: 0] wb_address_memwb;
```

Figure 3.3.28: MEMWB Wires

These are the wire declarations for the "mem/wb" stage variables in the processor. These wires are used to transfer the results of the memory stage and write back stage of the processor.

Here's what each of these wires is used for:

- reg_dst_memwb: A control signal that determines whether the destination register for the current instruction is rt (register 2) or rd (register 1).
- branch_memwb: A control signal that indicates whether the current instruction is a branch instruction.
- jump_memwb: A control signal that indicates whether the current instruction is a jump instruction.
- mem_read_memwb: A control signal that indicates whether the current instruction is a memory read instruction.

- mem_to_reg_memwb: A control signal that determines whether the value to be written back to the register file comes from the ALU or memory.
- alu_op_memwb: A control signal that specifies the operation to be performed by the ALU in the write-back stage.
- mem_write_memwb: A control signal that indicates whether the current instruction is a memory write instruction.
- alu_src_memwb: A control signal that determines whether the second operand to the ALU comes from
- the register file or is an immediate value.
- reg_write_memwb: A control signal that indicates whether the current instruction writes back to the register file.
- operand_a_memwb: The first operand to the ALU.
- operand_b_memwb: The second operand to the ALU.
- instruction_memwb: The current instruction being executed.
- alu_result_memwb: The result of the ALU operation performed in the write-back stage.
- zero_memwb: A control signal that indicates whether the result of the ALU operation was zero.
- read_data_memwb: The data read from memory in the memory stage.
- pc_memwb: The program counter value for the current instruction in the write-back stage.
- wb_address_memwb: The address of the register to be written to in the write-back stage.

2. Once we have all 5 stages of pipeline interconnected and wired in our top processor module, we will move on to the next step of the process in which we have to identify their registers under their specified parts.

```
185   ////////////////////////////////
186   /// IF/ID PIPELINE REGISTER ///
187   ////////////////////////////////
188
189   IFID if_id (
190     .clk             (clk             ),
191     .reset           (reset           ),
192     .flush           (flush           ),
193     .instruction     (instruction     ),
194     .pc_in           (pc_out          ),
195     .instruction_ifid (instruction_ifid),
196     .pc_ifid         (pc_ifid         )
197   );
```

Figure 3.3.29: IF/ID Pipeline registers

This code block instantiates a module called IFID and connects its input and output ports to the corresponding signals in the top-level module.

The IFID module takes the following inputs:

- clk: a clock signal used to synchronize the internal operations of the module
- reset: a signal used to reset the module to its initial state
- flush: a signal used to flush the instruction in the current pipeline stage when a branch misprediction occurs
- instruction: the current instruction being fetched from memory
- pc_out: the current program counter value
- instruction_ifid: the instruction that will be passed from the instruction fetch stage to the instruction decode stage
- pc_ifid: the program counter value that will be passed from the instruction fetch stage to the instruction decode stage

The IFID module has internal logic to latch the input values on each clock cycle, store them in registers, and pass them to the output ports for use in the next stage of the pipeline.

```
245  ////////////////////////////////
246  /// ID/EX PIPELINE REGISTER ///
247  ////////////////////////////////
248
249  IDEX id_ex (
250    .clk             (clk            ),
251    .reset           (reset          ),
252    .flush           (flush          ),
253    .reg_dst         (reg_dst        ),
254    .branch          (branch         ),
255    .jump            (jump           ),
256    .mem_read        (mem_read       ),
257    .mem_to_reg      (mem_to_reg     ),
258    .alu_op          (alu_op         ),
259    .mem_write       (mem_write      ),
260    .alu_src         (alu_src        ),
261    .reg_write       (reg_write      ),
262    .operand_a       (operand_a      ),
263    .operand_b       (operand_b      ),
264    .instruction_ifid (instruction_ifid),
265    .pc_ifid         (pc_ifid        ),
266    .wb_address      (mux_output     ),
267    .reg_dst_idex    (reg_dst_idex   ), // do not need to forward this to next stages
268    .branch_idex     (branch_idex    ),
269    .jump_idex       (jump_idex      ),
270    .mem_read_idex   (mem_read_idex  ),
271    .mem_to_reg_idex (mem_to_reg_idex ),
272    .alu_op_idex     (alu_op_idex    ),
273    .mem_write_idex  (mem_write_idex ),
274    .alu_src_idex    (alu_src_idex   ),
275    .reg_write_idex  (reg_write_idex ),
276    .operand_a_idex  (operand_a_idex ),
277    .operand_b_idex  (operand_b_idex ),
278    .instruction_idex (instruction_idex),
279    .pc_idex         (pc_idex        ),
280    .wb_address_idex (wb_address_idex )
281  );
```

Figure 3.3.30: ID/EX Pipeline registers

This code block is instantiating an instance of the IDEX module and connecting its inputs and outputs to various signals from the parent module.

The IDEX module represents the second stage of a pipelined processor and stands for "Instruction Decode/Execute". It takes the instruction and program counter from the previous stage (IF/ID) and decodes the instruction, generates the necessary control signals, and prepares the operands for the ALU. The resulting output of this stage includes the updated control signals, ALU operands, and other relevant data, which are forwarded to the next stage (EX/MEM) and also saved in latches for forwarding to subsequent stages.

Therefore, the code block is passing several inputs such as clk, reset, flush, instruction, and pc_out to the IDEX module, as well as receiving several outputs such as reg_dst_exmem, branch_exmem,

jump_exmem, mem_read_exmem, mem_to_reg_exmem, alu_op_exmem, and so on. These outputs represent the necessary control signals and ALU operands for the next stage (EX/MEM) to execute the decoded instruction. Additionally, some of the outputs are saved in latches for forwarding to later stages in the pipeline.

```
362   ////////////////////////////////
363   /// EX/MEM PIPELINE REGISTER ///
364   ////////////////////////////////
365
366   EXMEM ex_mem (
367     .clk               (clk             ),
368     .reset             (reset           ),
369     .reg_dst_idex      (reg_dst_idex    ),
370     .branch_idex       (branch_idex     ),
371     .jump_idex         (jump_idex       ),
372     .mem_read_idex     (mem_read_idex   ),
373     .mem_to_reg_idex   (mem_to_reg_idex ),
374     .alu_op_idex       (alu_op_idex     ),
375     .mem_write_idex    (mem_write_idex  ),
376     .alu_src_idex      (alu_src_idex    ),
377     .reg_write_idex    (reg_write_idex  ),
378     .operand_a_idex    (operand_a_idex  ),
379     .operand_b_idex    (operand_b_idex  ),
380     .instruction_idex  (instruction_idex ),
381     .alu_result        (alu_result      ),
382     .zero              (zero            ),
383     .pc_idex           (pc_idex         ),
384     .wb_address_idex   (wb_address_idex ),
385     .reg_dst_exmem     (reg_dst_exmem   ),
386     .branch_exmem      (branch_exmem    ),
387     .jump_exmem        (jump_exmem      ),
388     .mem_read_exmem    (mem_read_exmem  ),
389     .mem_to_reg_exmem  (mem_to_reg_exmem ),
390     .alu_op_exmem      (alu_op_exmem    ),
391     .mem_write_exmem   (mem_write_exmem ),
392     .alu_src_exmem     (alu_src_exmem   ),
393     .reg_write_exmem   (reg_write_exmem ),
394     .operand_a_exmem   (operand_a_exmem ),
395     .operand_b_exmem   (operand_b_exmem ),
396     .instruction_exmem (instruction_exmem),
397     .alu_result_exmem  (alu_result_exmem ),
398     .zero_exmem        (zero_exmem      ),
399     .pc_exmem          (pc_exmem        ),
400     .wb_address_exmem  (wb_address_exmem )
401   );
402
```

Figure 3.3.31: EX/MEM Pipeline registers

This code block instantiates an "EXMEM" module that represents the third stage of a pipelined processor. The module takes in various control and data signals from the previous pipeline stage (IDEX), as well as the current clock and reset signals.

The EXMEM module performs the execution of the instruction, such as arithmetic, logical or comparison operations, on the operands received from the previous pipeline stage. It then forwards the results to the next pipeline stage (MEMWB). It also generates control signals such as whether a branch should be taken or not based on the instruction's opcode.

The signals passed between the modules include control signals such as register destination, whether to branch or jump, whether to read or write memory, ALU operation, and whether to write to the register file. Data signals include the two operands, the instruction to be executed, the ALU result, whether the result is zero, the program counter, and the address to write back to the register file.

```
420   ////////////////////////////////
421   /// MEM/WB PIPELINE REGISTER ///
422   ////////////////////////////////
423
424   MEMWB mem_wb (
425     .clk              (clk              ),
426     .reset            (reset            ),
427     .reg_dst_exmem    (reg_dst_exmem    ),
428     .branch_exmem     (branch_exmem     ),
429     .jump_exmem       (jump_exmem       ),
430     .mem_read_exmem   (mem_read_exmem   ),
431     .mem_to_reg_exmem (mem_to_reg_exmem ),
432     .alu_op_exmem     (alu_op_exmem     ),
433     .mem_write_exmem  (mem_write_exmem  ),
434     .alu_src_exmem    (alu_src_exmem    ),
435     .reg_write_exmem  (reg_write_exmem  ),
436     .operand_a_exmem  (operand_a_exmem  ),
437     .operand_b_exmem  (operand_b_exmem  ),
438     .instruction_exmem(instruction_exmem),
439     .alu_result_exmem (alu_result_exmem ),
440     .zero_exmem       (zero_exmem       ),
441     .read_data        (read_data        ),
442     .pc_exmem         (pc_exmem         ),
443     .wb_address_exmem (wb_address_exmem ),
444     .reg_dst_memwb    (reg_dst_memwb    ),
445     .branch_memwb     (branch_memwb     ),
446     .jump_memwb       (jump_memwb       ),
447     .mem_read_memwb   (mem_read_memwb   ),
448     .mem_to_reg_memwb (mem_to_reg_memwb ),
449     .alu_op_memwb     (alu_op_memwb     ),
450     .mem_write_memwb  (mem_write_memwb  ),
451     .alu_src_memwb    (alu_src_memwb    ),
452     .reg_write_memwb  (reg_write_memwb  ),
453     .operand_a_memwb  (operand_a_memwb  ),
454     .operand_b_memwb  (operand_b_memwb  ),
455     .instruction_memwb(instruction_memwb),
456     .alu_result_memwb (alu_result_memwb ),
457     .zero_memwb       (zero_memwb       ),
458     .read_data_memwb  (read_data_memwb  ),
459     .pc_memwb         (pc_memwb         ),
460     .wb_address_memwb (wb_address_memwb )
461   );
```

Figure 3.3.32: MEM/MWB Pipeline registers

This code block is defining a module called MEMWB which has inputs and outputs for various signals used in a computer's memory and write-back stages.

The inputs to the module include the clock signal (clk), reset signal (reset), various control signals such as whether to write to a register (reg_write_exmem) or perform a memory write (mem_write_exmem), and data signals such as the ALU result (alu_result_exmem) and read data (read_data).

The outputs from the module include signals that will be used in the next pipeline stage (reg_dst_memwb, jump_memwb, etc.), the data that will be written to memory (write_data_memwb), and signals indicating whether a memory read was performed (mem_read_memwb) or whether the ALU result was zero (zero_memwb).

This module is typically used in a pipelined CPU architecture, where each stage of the pipeline is responsible for a different part of the instruction execution process. In this case, the MEMWB module is responsible for handling the memory access and write-back stages.

3.3.3. Task 3: Implementation of Hazard Control unit to detect any hazards

1. Pipelining in a processor allows multiple instructions to be executed simultaneously, which increases the overall throughput of the processor. However, pipelining introduces new hazards that need to be controlled, such as data hazards and control hazards.

   Data hazards occur when instructions in the pipeline require data from a previous instruction that has not yet completed. This can cause stalls in the pipeline, which reduces the throughput of the processor. A hazard control unit is responsible for detecting and resolving data hazards by forwarding data from the output of one pipeline stage to the input of another pipeline stage, or by inserting pipeline stalls when necessary.

   Control hazards occur when the outcome of a conditional branch instruction is not yet known when the next instruction enters the pipeline. This can cause the pipeline to execute incorrect instructions, which can result in incorrect program behavior. A hazard control unit can detect and resolve control hazards by inserting pipeline stalls or by predicting the outcome of conditional branch instructions.

   In summary, a hazard control unit is necessary in pipelining to ensure correct and efficient execution of instructions in the pipeline by detecting and resolving data hazards and control hazards.

```
1   module hazard_detection_unit (
2     input             clk,
3     input      [4:0]  rs1_addr,
4     input      [4:0]  rs2_addr,
5     input             mem_read_idex,
6     input             reg_write,
7     input             reg_write_idex,
8     input      [31:0] instruction_ifid,
9     input      [31:0] instruction_idex,
10    output reg        stall
11  );
12
13    // variables needed for different pipes
14    reg [4:0] if_id_rd, id_ex_rd;
```

Figure 3.3.33: Hazard control initialization

This Verilog module implements a hazard detection unit for a pipeline processor. The module has the following inputs:

- clk: a clock signal used for synchronization
- rs1_addr: the address of the first source register
- rs2_addr: the address of the second source register
- mem_read_idex: a signal indicating whether a memory read is performed in the ID/EX pipeline stage
- reg_write: a signal indicating whether a register write is performed in the pipeline

- reg_write_idex: a signal indicating whether a register write is performed in the ID/EX pipeline stage
- instruction_ifid: the instruction in the IF/ID pipeline stage
- instruction_idex: the instruction in the ID/EX pipeline stage

The module has one output:

- stall: a signal indicating whether a stall is required in the pipeline

The hazard detection unit is responsible for detecting data hazards in the pipeline by checking whether the source registers of an instruction being fetched in the IF/ID stage match the destination registers of an instruction being decoded in the ID/EX stage. If a hazard is detected, the hazard detection unit asserts the stall signal, which stalls the pipeline and prevents further instructions from being fetched until the hazard is resolved.

```
16    // Check the destination register of the mem/wb pipe instruction
17    always @* begin
18      if (mem_read_idex) begin
19        id_ex_rd = instruction_idex [20:16];
20      end else begin
21        id_ex_rd = instruction_idex [15:11];
22      end
23    end
24
25    // Selection of the destination register of the ex/mem pipe instruction
26    always @* begin
27      if_id_rd = instruction_ifid [15:11];
28    end
29
30    // Check if there is any hazard found + bubble generation logic
31    always @* begin
32
33      stall = 1'b0;
34
35      // Generate bubble for memory instruction's dependency
36      if (id_ex_rd != 0 && id_ex_rd == rs1_addr && reg_write_idex) begin
37        stall = 1'b1;
38      end
39
40      if (id_ex_rd != 0 && id_ex_rd == rs2_addr && reg_write_idex) begin
41        stall = 1'b1;
42      end
43
44    end
45
46  endmodule
```

Figure 3.3.34: Hazard control always blocks

In the block of code given above, the module uses another always block to check for data hazards between the current instruction in the ID/EX stage and the previous memory-read instruction in the MEM/WB stage. If a dependency is detected between the source operands of the current instruction and the destination register of the previous memory-read instruction, stall is set to 1 to insert a bubble into the pipeline.

FLUSH UNIT: The flush unit in a MIPS processor is responsible for clearing or "flushing" the contents of the instruction pipeline in certain situations. Specifically, it is used when a branch or jump instruction is encountered, as these instructions may cause the processor to execute instructions out of order, which can lead to incorrect results.

```
module flush_unit (
  input         zero,
  input         branch_idex,
  input         jump_idex,
  output reg    flush
);

  // Check whether we need to FLUSH the Fetch and Decode Stage Pipes
  // (For Branch and Jump Instructions)
  always @* begin
    flush = ((zero & branch_idex) || jump_idex) ? 1'b1 : 1'b0;
  end

endmodule
```

Figure 3.3.35: Flush unit

The code block defines a flush unit module that takes in inputs zero, branch_idex, and jump_idex, and outputs flush.

The always @* block uses a ternary operator to assign the value of flush. If zero is asserted (1) and branch_idex is asserted (1), or if jump_idex is asserted (1), then flush is set to 1, indicating that the pipeline should be flushed. Otherwise, flush is set to 0, indicating that the pipeline should not be flushed.

3.3.4. Task 4: Simulations

1. In order to simulate our working process, we must write a test bench to be used while running modelsim built in simulation tool to generate clk.

```
module test_bench();

  reg clk;   // clock input (active-high)
  reg reset; // reset input (active-low)

  initial begin
    clk       = 0;
    reset     = 1; // Reset is active low
    #15 reset = 0; // Assert reset after 10 time units
  end

  always #5 clk = ~clk; // Toggle clk every 5 time units

  // Instantiate DUT (Design Under Test) module
  processor_top mips_processor(
    .clk   (clk  ),
    .reset (reset)
  );

endmodule
```

Figure 3.3.36: Test bench simulation

This code block is a test bench module for a MIPS processor design. It instantiates the DUT (Design Under Test) module, which is the actual processor implementation that needs to be tested. The test bench provides the clock and reset signals to the DUT and sets their values according to a predefined sequence of events.

The clk signal is a clock input that is toggled every 5 time units using an always block with a delay of 5 time units. The reset signal is an active low reset input that is set to 1 initially and then set to 0 after 15 time units using delay operator #.

The processor_top is the module being tested, which takes the clk and reset signals as inputs. By instantiating the processor_top module and providing the input signals, the test bench can test the functionality of the MIPS processor implementation.

The outcome of the simulations could be found under the **Analysis** section.

4. ANALYSIS

4.1 Experimental Results

Let's start with first uploading our files onto the modelsim environment to be run and simulated.

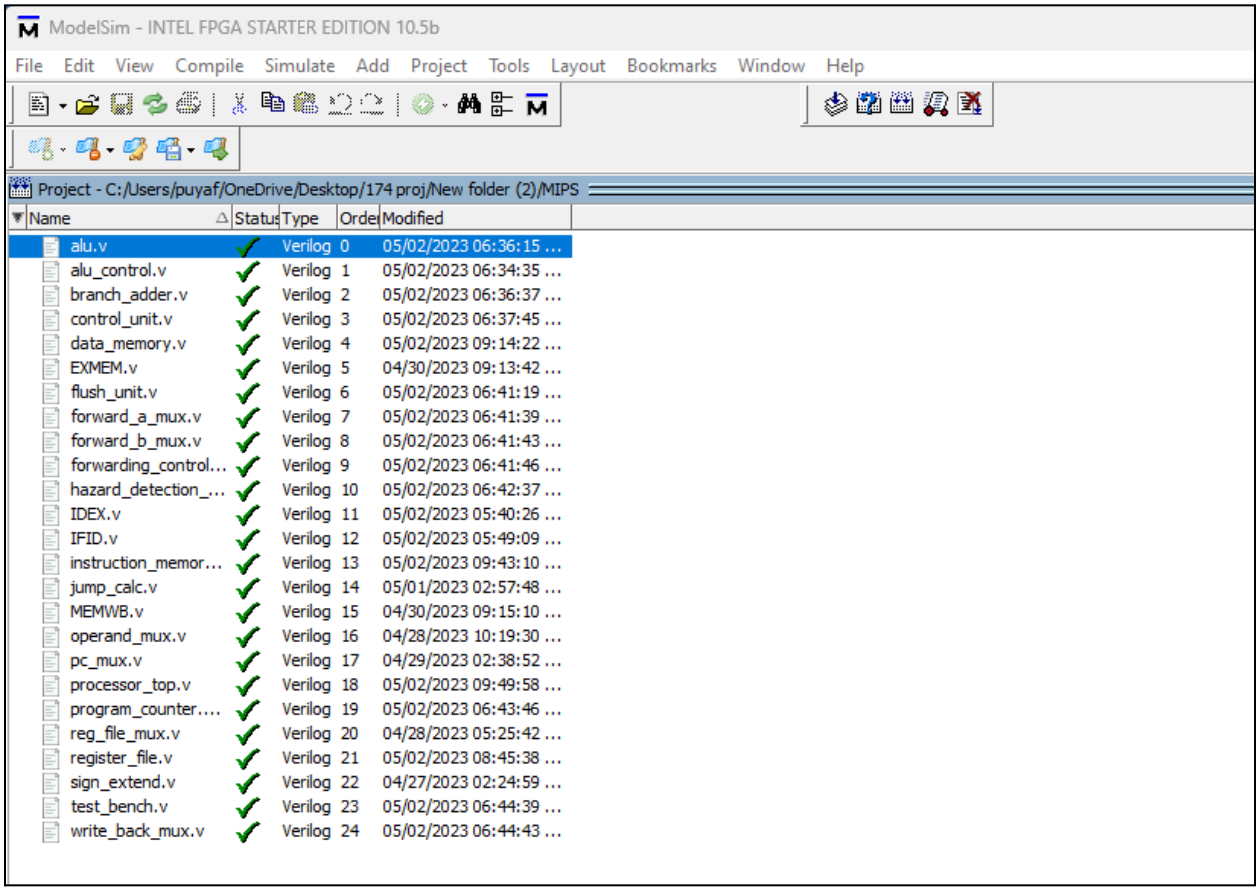


Figure 4.1.1: Uploading .v files

As could be seen above, we uploaded all of the fully developed modules including the top module to a file in modelsim, named **work**. This file will be physically located in the computer disk as well.

We will then simulate our test bench using ModelSim's simulation tool, and add waves to the corresponding modules to analyze the working processor.

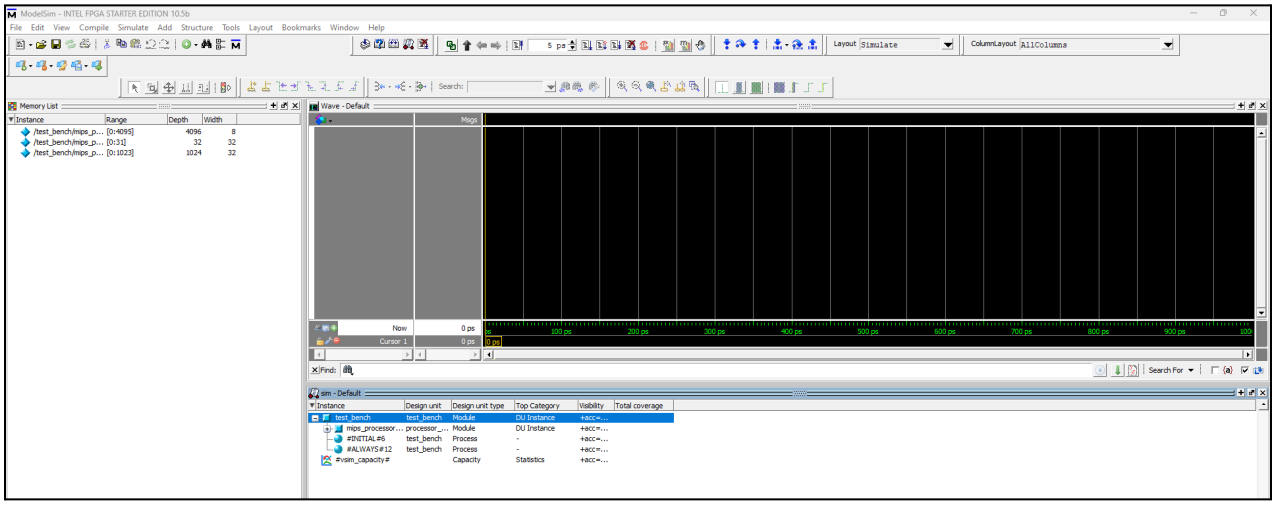


Figure 4.1.2: Running the simulation tool

For the simulation tool, we must select the test bench file and click simulate. The program will automatically add a wave and have the required files ready to be used under the simulation wave tool. We will first select the simulation to be at a 5ps time cycle to analyze it step by step.

In order to check the simulation process, we must select the modules that will be necessary from our list, in this case we will select the ALU module first.

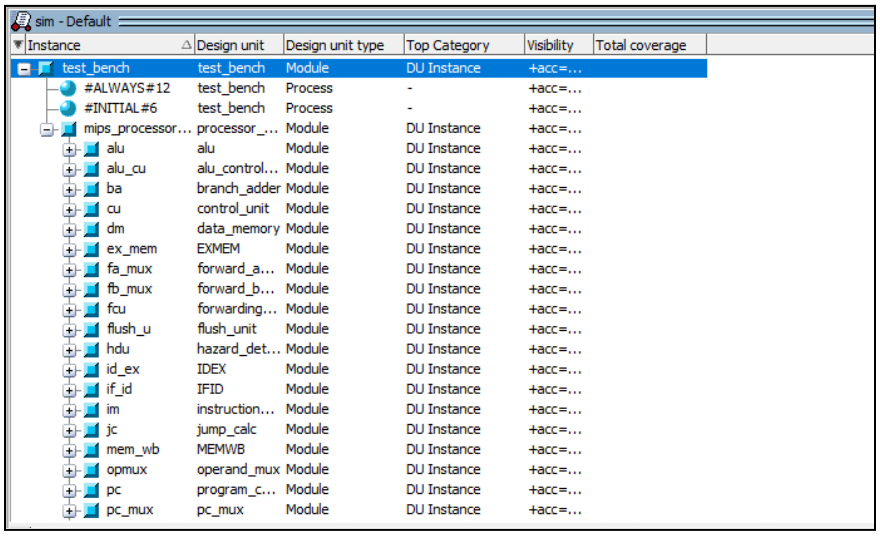


Figure 4.1.3: Selecting alu module

Once selected, we will add waves to the module and using our wave simulation tool, we will run the processor in a 5ps time cycle at first to check how it interacts with the overall processor. The next step of the process at this point is to click **Run**.

Testing instruction ADD:

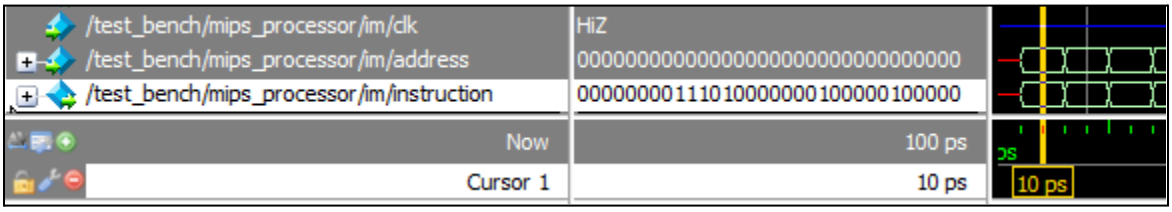


Figure 4.1.4: Instruction fetch of **add**

As we can see, the instruction is being fetched from the instruction memory correctly. We define the instruction 32'h00E80820; // add to be fetched first. The 32 bit binary version of the instruction is given to us in the simulation. If we translate it to binary and separate its section we will see that 32'h00E80820 in binary is 000000 00111 01000 00001 00000 100000

As we know, add is a R-Type instruction, therefore if we split the 32 bit binary values into its corresponding sections, we will have:
000000 = op code, 00111 = operand a, 01000 = operand b, 00001 = destination register, function: 100000

```
15  // Initial values of the registers (For testing purposes)
16  initial begin
17    registers [1] = 32'h1;
18    registers [2] = 32'h7;
19    registers [3] = 32'h3;
20    registers [4] = 32'h4;
21    registers [5] = 32'h8;
22    registers [6] = 32'h7;
23    registers [7] = 32'h7;
24    registers [8] = 32'h6;
25  end
```

Figure 4.1.5: Register values

In the figure above, we can see that our register file holds arbitrary register values to be tested during these operations fetched from instruction memory. In this case, we are using register [7]'s value 7, and adding it to register [8]'s value 6. The resulting add operation will be 13.

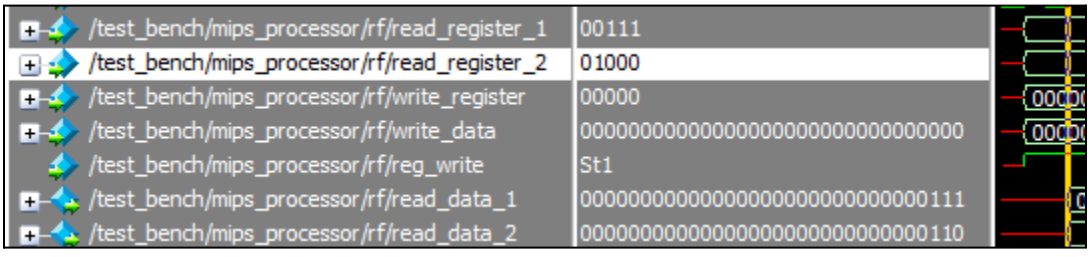


Figure 4.1.6: Register values simulation

In the figure above, we can observe that during the next cycle our instruction memory has been updated with the corresponding register numbers and register data. It is reading register 7 and 8, with values 7 and 6 respectively. **Which is correct.**

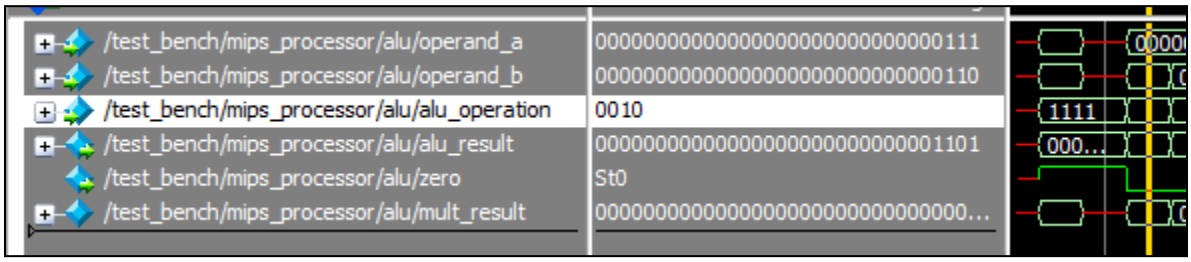


Figure 4.1.7: ADD Operation ALU

Now moving to the execution stage of our process, we have **ADD** operation between two operands which had values hardcoded in the register file module to them. We have so far fetched the instruction, and moved the register address with their corresponding values. Now let's check our simulation results and see if we get the resulting 13 from our signals. We can observe that our first fetched instruction from instruction memory is resulting in **1101** at alu_result register defined above. **Therefore it is in fact properly working because 6 + 7 = 13.**

Now let's take a look at Write back data at our Write back mux.

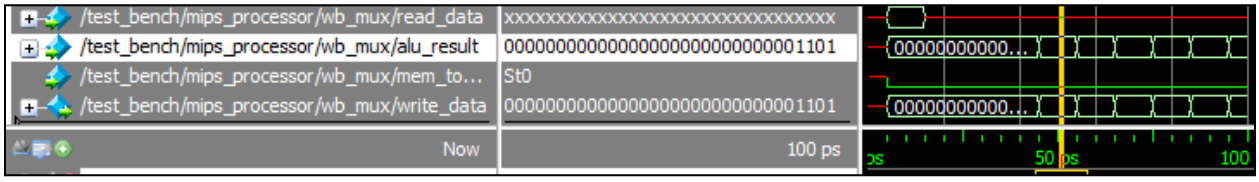


Figure 4.1.8: ADD Operation WB
It is writing back the result **1101**.

Now let's check if at the end of all 5 stages our result is written back at register file module,

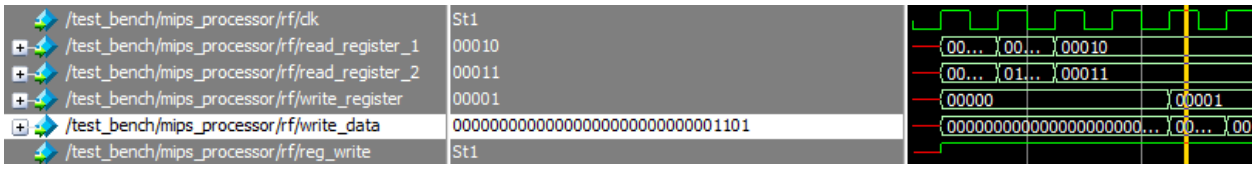


Figure 4.1.9: WB to register file module

Yes, it did in fact write back the 01101 to the write_data location of the register module. Which will satisfy the working instruction test for ADD instruction.

Testing instruction AND:

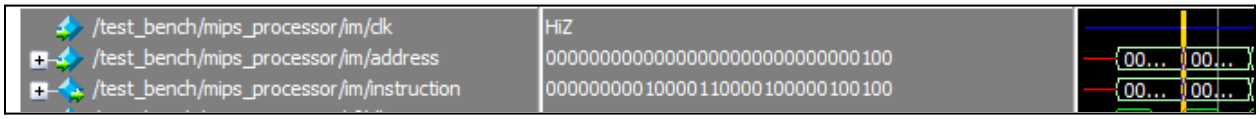


Figure 4.1.10: Instruction mem AND

As we can see, the instruction is being fetched from the instruction memory correctly. We define the instruction 32'h00430824; // and to be fetched next. The 32 bit binary version of the instruction is given to us in the simulation. If we translate it to binary and separate its section we will see that 32'h00430824 in binary is 000000 00010 00011 00001 00000 100100

As we know, and is a R-Type instruction, therefore if we split the 32 bit binary values into its corresponding sections, we will have:
000000 = op code, 00010 = operand a, 00011 = operand b, 00001 = destination register, function: 100100

```
15  // Initial values of the registers (For testing purposes)
16  initial begin
17    registers [1] = 32'h1;
18    registers [2] = 32'h7;
19    registers [3] = 32'h3;
20    registers [4] = 32'h4;
21    registers [5] = 32'h8;
22    registers [6] = 32'h7;
23    registers [7] = 32'h7;
24    registers [8] = 32'h6;
25  end
```

Figure 4.1.11: Register values

In the figure above, we can see that our register file holds arbitrary register values to be tested during these operations fetched from instruction memory. In this case, we are using register [2]'s value 7, and anding it to register [3]'s value 3. The resulting operation will be 3.

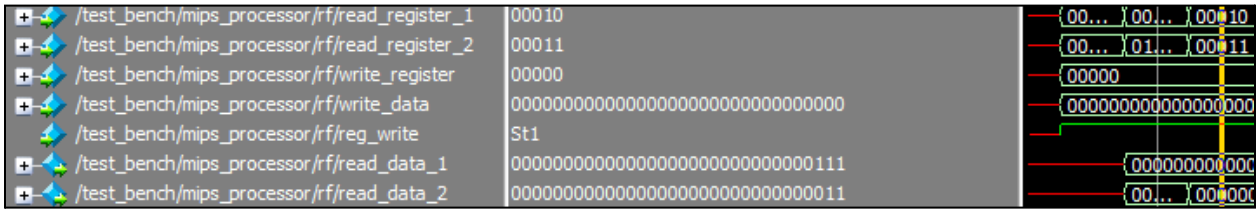


Figure 4.1.12: Register values simulation

In the figure above, we can observe that during the next cycle our instruction memory has been updated with the corresponding register numbers and register data. It is reading register 2 and 3, with values 7 and 3 respectively. **Which is correct.**

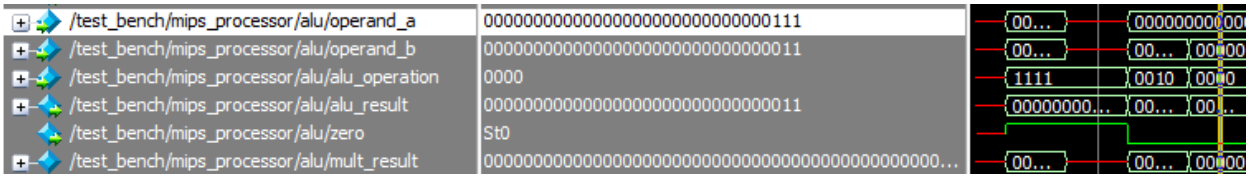


Figure 4.1.13: AND Operation ALU

Now moving to the execution stage of our process, we have **AND** operation between two operands which had values hardcoded in the register file module to them. We have so far fetched the instruction, and moved the register address with their corresponding values. Now let's check our simulation results and see if we get the resulting3 from our signals. We can observe that our first fetched instruction from instruction memory is resulting in **0011** at the alu_result register defined above. **Therefore it is in fact properly working.**

Now let's take a look at our Data memory and Write Back module at our Write back mux.

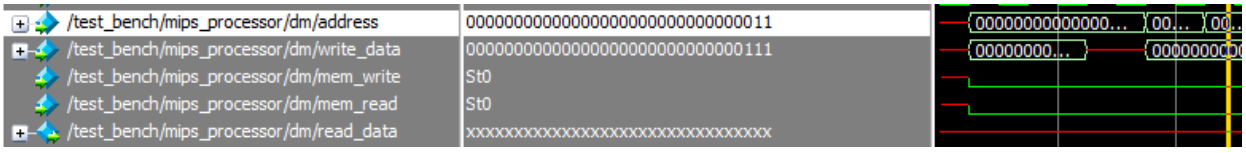


Figure 4.1.14: Data memory block

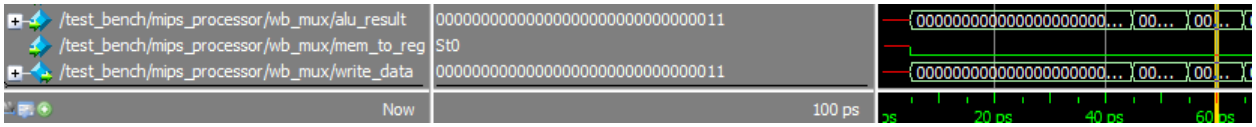


Figure 4.1.15: Write back MUX

It could be observed that these two modules are also operating successfully for the first four stages of the pipeline, now let's check if the final result is written back to our write register of our register module:

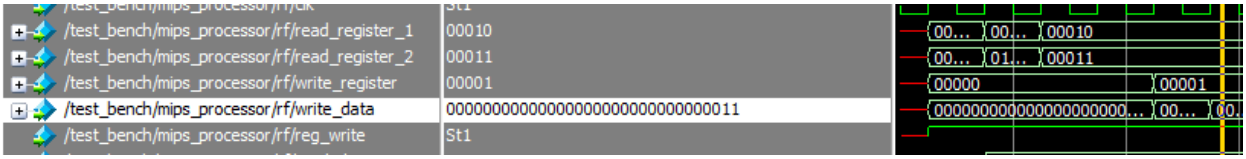


Figure 4.1.15: Write back to register file

Yes, it did in fact write back the 0011 to the write_data location of the register module. Which will satisfy the working instruction test for AND instruction.

Testing instruction OR:

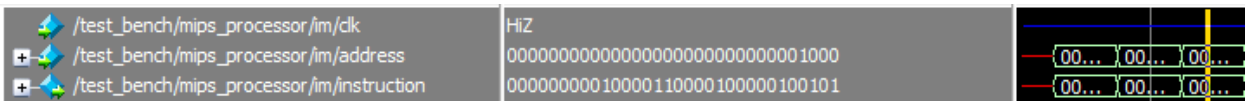


Figure 4.1.15: Instruction fetch OR

As we can see, the instruction is being fetched from the instruction memory correctly. We define the instruction32'h00430825; // or to be fetched next. The 32 bit binary version of the instruction is given to us in the simulation. If we translate it to binary and separate its section we will see that 32'h00430825 in binary is 000000 00010 00011 00001 00000 100101

As we know, and is a R-Type instruction, therefore if we split the 32 bit binary values into its corresponding sections, we will have:
000000 = op code, 00010 = operand a, 00011 = operand b, 00001 = destination register, function: 100101

```
15  // Initial values of the registers (For testing purposes)
16  initial begin
17    registers [1] = 32'h1;
18    registers [2] = 32'h7;
19    registers [3] = 32'h3;
20    registers [4] = 32'h4;
21    registers [5] = 32'h8;
22    registers [6] = 32'h7;
23    registers [7] = 32'h7;
24    registers [8] = 32'h6;
25  end
```

Figure 4.1.16: Register values

In the figure above, we can see that our register file holds arbitrary register values to be tested during these operations fetched from instruction memory. In this case, we are using register [2]'s value 7, and anding it to register [3]'s value 3. The resulting operation will be 3.

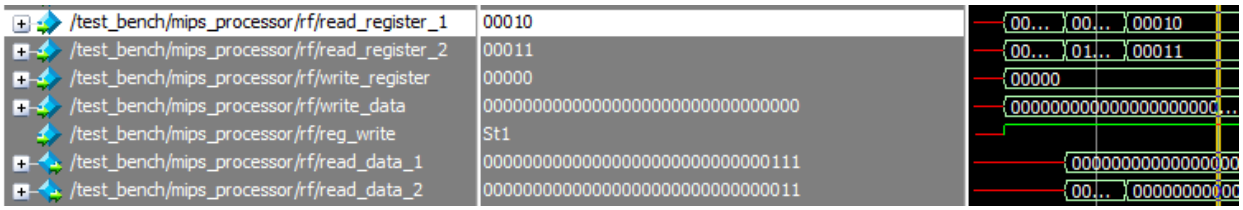


Figure 4.1.17: Register values simulation

In the figure above, we can observe that during the next cycle our instruction memory has been updated with the corresponding register numbers and register data. It is reading register 2 and 3, with values 7 and 3 respectively. **Which is correct.**

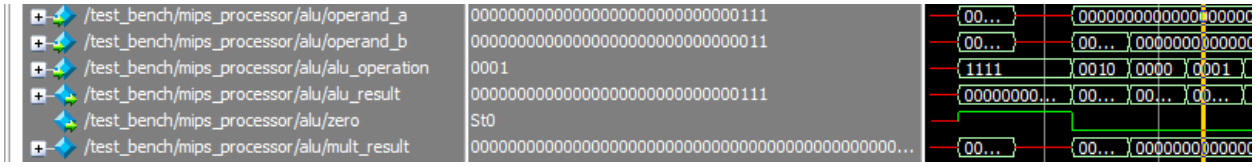


Figure 4.1.18: ALU Operation OR

Now moving to the execution stage of our process, we have **OR** operation between two operands which had values hardcoded in the register file module to them. We have so far fetched the instruction, and moved the register address with their corresponding values. Now let's check our simulation results and see if we get the resulting 7 from our signals. We can observe that our first fetched instruction from instruction memory is resulting in **0111** at the alu_result register defined above. **Therefore it is in fact properly working.**

Now let's take a look at our Data memory and Write Back module at our Write back mux.

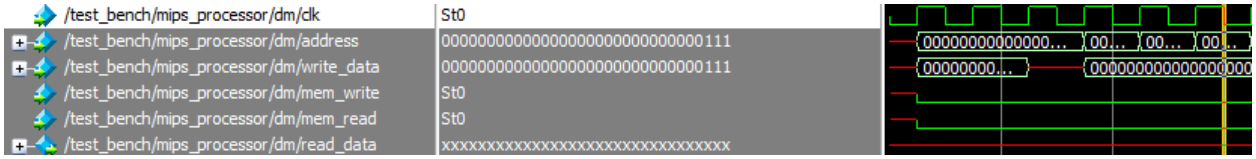


Figure 4.1.19: Data memory

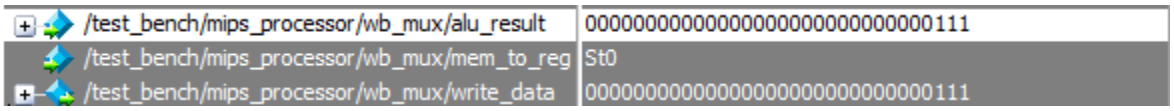


Figure 4.1.20: Write Back

It could be observed that these two modules are also operating successfully for the first four stages of the pipeline, now let's check if the final result is written back to our write register of our register module:

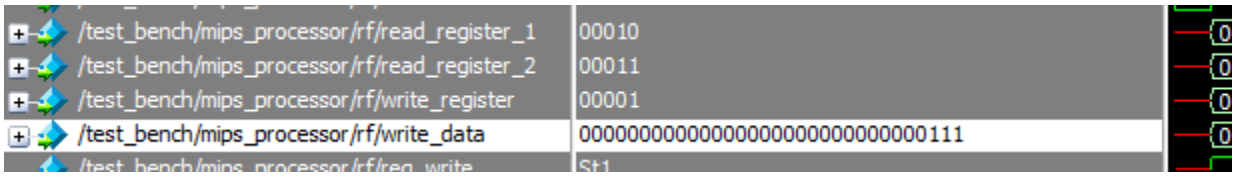


Figure 4.1.21: Regfile write data

Yes, it did in fact write back the 0011 to the write_data location of the register module. Which will satisfy the working instruction test for OR instruction.

Testing instruction NOR:

Figure 4.1.22: Instruction fetch NOR

As we can see, the instruction is being fetched from the instruction memory correctly. We define the instruction 32'h00430827; // nor to be fetched next. The 32 bit binary version of the instruction is given to us in the simulation. If we translate it to binary and separate its section we will see that 32'h00430827 in binary is 000000 00010 00011 00001 00000 100111

As we know, and is a R-Type instruction, therefore if we split the 32 bit binary values into its corresponding sections, we will have:
000000 = op code, 00010 = operand a, 00011 = operand b, 00001 = destination register, function: 100111

```
// Initial values of the registers (For testing purposes)
initial begin
  registers [1] = 32'h1;
  registers [2] = 32'h9;
  registers [3] = 32'h2;
  registers [4] = 32'h4;
  registers [5] = 32'h8;
  registers [6] = 32'h7;
  registers [7] = 32'h7;
  registers [8] = 32'h6;
end
```

Figure 4.1.23: Register values

In the figure above, we can see that our register file holds arbitrary register values to be tested during these operations fetched from instruction memory. In this case, we are using register [2]'s value 9, and anding it to register [3]'s value 2.

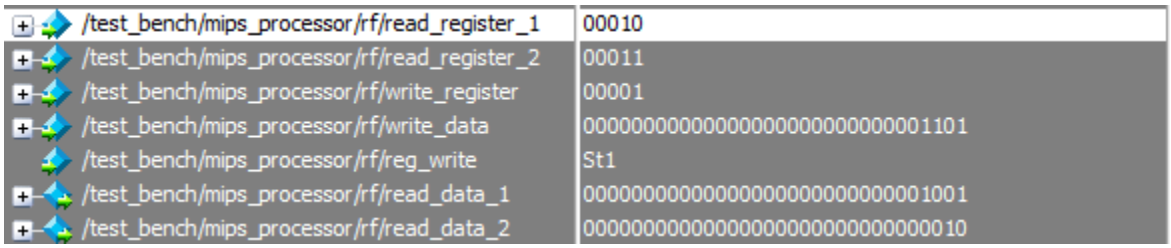


Figure 4.1.24: Register values simulation

In the figure above, we can observe that during the next cycle our instruction memory has been updated with the corresponding register numbers and register data. It is reading register 2 and 3, with values 9 and 2 respectively. **Which is correct.**

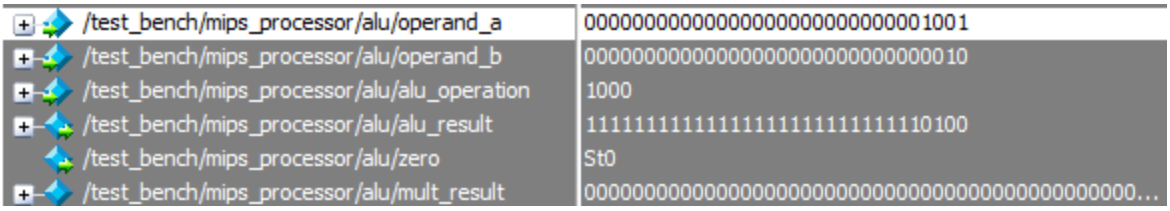


Figure 4.1.24: ALU module for NOR

Now moving to the execution stage of our process, we have **NOR** operation between two operands which had values hardcoded in the register file module to them. We have so far fetched the instruction, and moved the register address with their corresponding values. Now let's check our simulation results and see if we get the resulting values from our signals. We can observe that our first fetched instruction from instruction memory is resulting in **32b'111…10100** at the alu_result register defined above. **Therefore it is in fact properly working.**

Now let's take a look at our Data memory and Write Back module at our Write back mux.

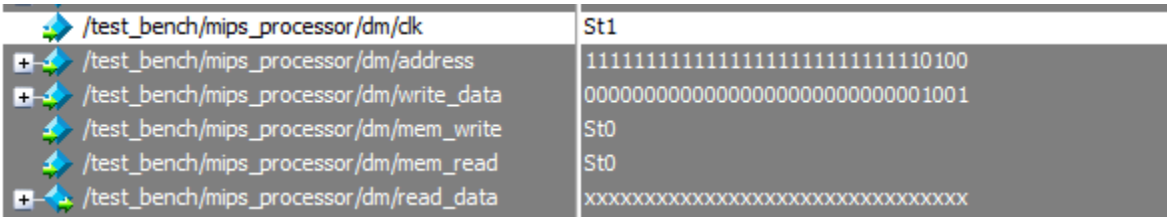


Figure 4.1.25: Data memory for NOR

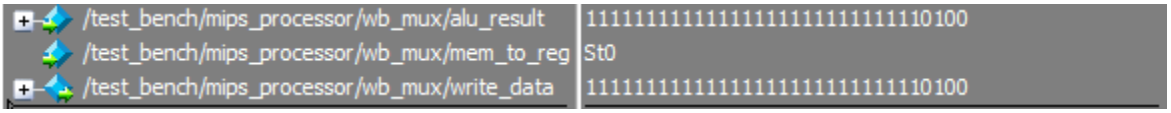


Figure 4.1.26: WB Mux for NOR

It could be observed that these two modules are also operating successfully for the first four stages of the pipeline, now let's check if the final result is written back to our write register of our register module:

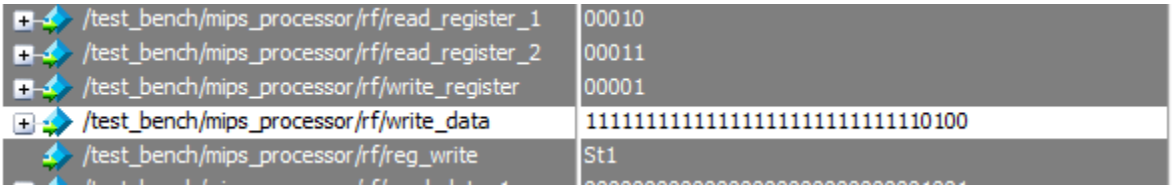


Figure 4.1.27: Write data register at the end for NOR

Yes, it did in fact write back the32b'111…10100 to the write_data location of the register module. Which will satisfy the working instruction test for NOR instruction.

Testing instruction SUB:

Figure 4.1.28: Instruction fetch for SUB

As we can see, the instruction is being fetched from the instruction memory correctly. We define the instruction 32'h00430822; // sub to be fetched next. The 32 bit binary version of the instruction is given to us in the simulation. If we translate it to binary and separate its section we will see that 32'h00430822 in binary is 000000 00010 00011 00001 00000 100010

As we know, and is a R-Type instruction, therefore if we split the 32 bit binary values into its corresponding sections, we will have:
000000 = op code, 00010 = operand a, 00011 = operand b, 00001 = destination register, function: 100010

```
// Initial values of the registers (For testing purposes)
initial begin
  registers [1] = 32'h1;
  registers [2] = 32'h9;
  registers [3] = 32'h2;
  registers [4] = 32'h4;
  registers [5] = 32'h8;
  registers [6] = 32'h7;
  registers [7] = 32'h7;
  registers [8] = 32'h6;
end
```

Figure 4.1.29: Register values

In the figure above, we can see that our register file holds arbitrary register values to be tested during these operations fetched from instruction memory. In this case, we are using register [2]'s value 9, and anding it to register [3]'s value 2.

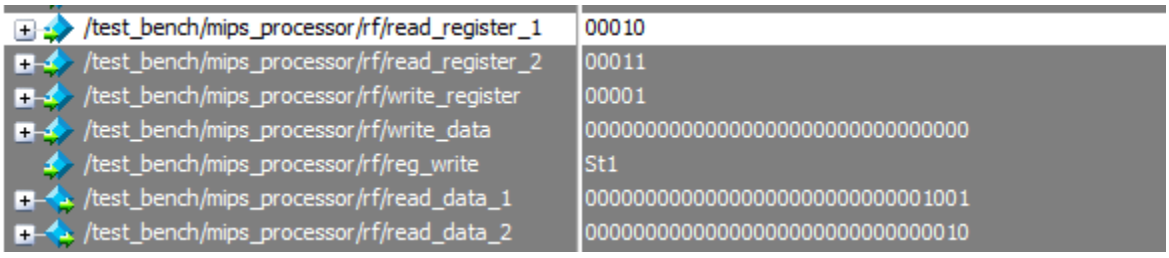


Figure 4.1.30: Register values simulation

In the figure above, we can observe that during the next cycle our instruction memory has been updated with the corresponding register numbers and register data. It is reading register 2 and 3, with values 9 and 2 respectively. **Which is correct.**

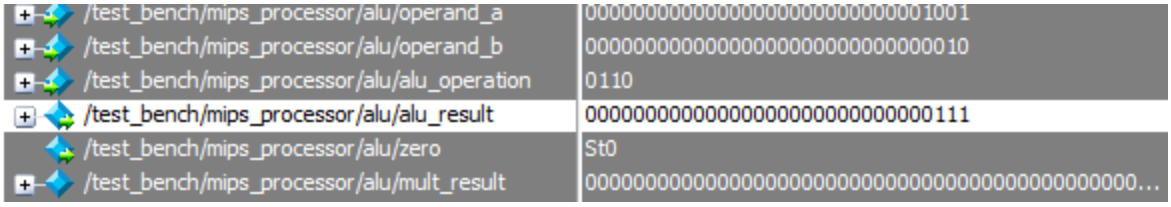


Figure 4.1.31: ALU Operation for SUB

Now moving to the execution stage of our process, we have **SUB** operation between two operands which had values hardcoded in the register file module to them. We have so far fetched the instruction, and moved the register address with their corresponding values. Now let's check our simulation results and see if we get the resulting values from our signals. We can observe that our first fetched instruction from instruction memory is resulting in **111 = 7** at the alu_result register defined above. **Therefore it is in fact properly working.**

Now let's take a look at our Data memory and Write Back module at our Write back mux.

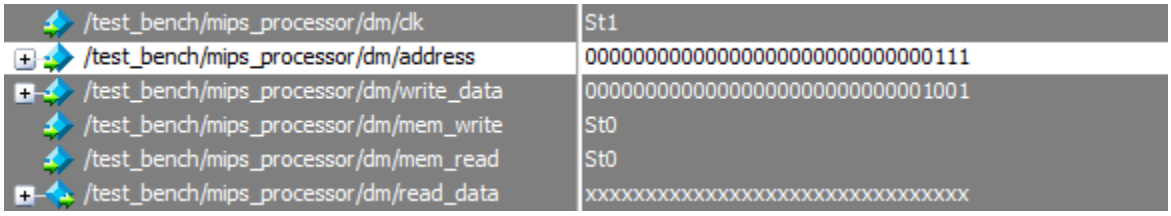


Figure 4.1.32: Data memory SUB

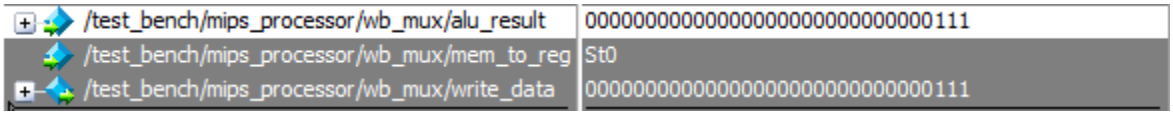


Figure 4.1.33: AWB Mux SUB

It could be observed that these two modules are also operating successfully for the first four stages of the pipeline, now let's check if the final result is written back to our write register of our register module:

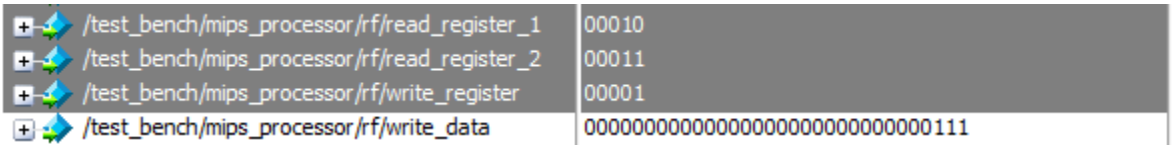


Figure 4.1.34: Reg write back SUB

Yes, it did in fact write back the 000..111 to the write_data location of the register module. Which will satisfy the working instruction test for SUB instruction.

Testing instruction SLT:

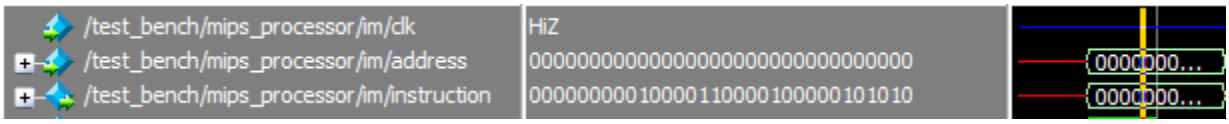


Figure 4.1.35: Instruction fetch of SLT

As we can see, the instruction is being fetched from the instruction memory correctly. We define the instruction 32'h0043082A; // sub to be fetched next. The 32 bit binary version of the instruction is given to us in the simulation. If we translate it to binary and separate its section we will see that 32'h0043082A in binary is 000000 00010 00011 00001 00000 101010

As we know, and is a R-Type instruction, therefore if we split the 32 bit binary values into its corresponding sections, we will have:
000000 = op code, 00010 = operand a, 00011 = operand b, 00001 = destination register, function: 101010

```
// Initial values of the registers (For testing purposes)
initial begin
  registers [1] = 32'h1;
  registers [2] = 32'h9;
  registers [3] = 32'h2;
  registers [4] = 32'h4;
  registers [5] = 32'h8;
  registers [6] = 32'h7;
  registers [7] = 32'h7;
  registers [8] = 32'h6;
end
```

Figure 4.1.36: Register values

In the figure above, we can see that our register file holds arbitrary register values to be tested during these operations fetched from instruction memory. In this case, we are using register [2]'s value 9, and anding it to register [3]'s value 2.

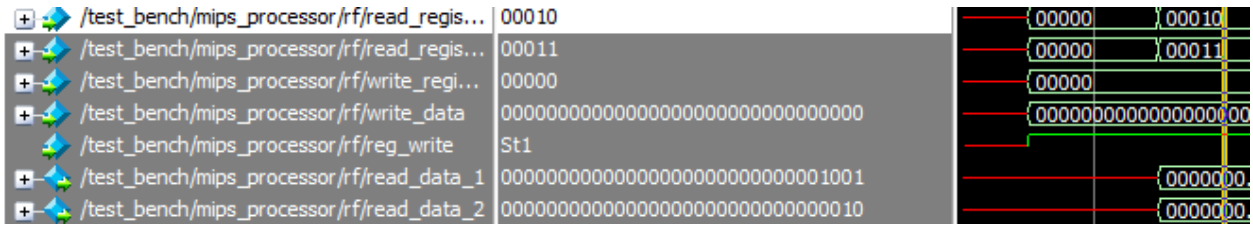


Figure 4.1.37: Register values simulation

In the figure above, we can observe that during the next cycle our instruction memory has been updated with the corresponding register numbers and register data. It is reading register 2 and 3, with values 9 and 2 respectively. **Which is correct.**

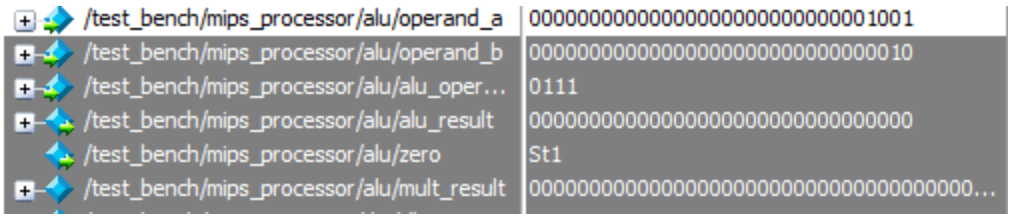


Figure 4.1.38: ALU Operation for SLT

Now moving to the execution stage of our process, we have **SLT** operation between two operands which had values hardcoded in the register file module to them. We have so far fetched the instruction, and moved the register address with their corresponding values. Now let's check our simulation results and see if we get the resulting values from our signals. We can observe that our first fetched instruction from instruction memory is resulting in 00.000 at the alu_result register defined above **because 9 is not less than 2. Therefore it is in fact properly working.**

Now let's take a look at our Data memory and Write Back module at our Write back mux.

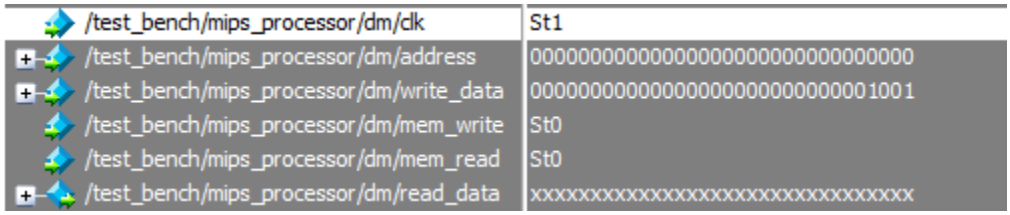


Figure 4.1.39: SLT for Data memory

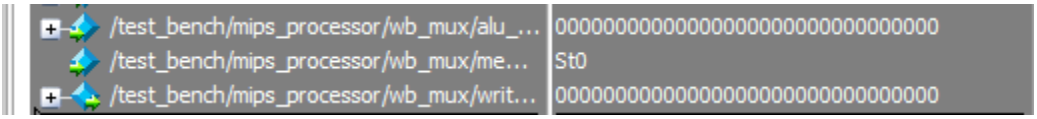


Figure 4.1.40: WB mux for Data memory

It could be observed that these two modules are also operating successfully for the first four stages of the pipeline, now let's check if the final result is written back to our write register of our register module:

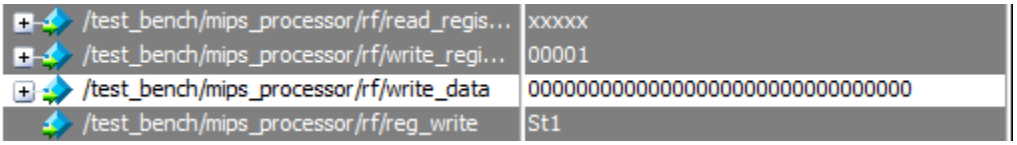


Figure 4.1.41: Reg write back SLT

Yes, it did in fact write back the 000..000 to the write_data location of the register module. Which will satisfy the working instruction test for SLT instruction.

Testing instruction DIV:

Figure 4.1.42: Instruction fetch for DIV

As we can see, the instruction is being fetched from the instruction memory correctly. We define the instruction 32'h0043082A; // sub to be fetched next. The 32 bit binary version of the instruction is given to us in the simulation. If we translate it to binary and separate its section we will see that 32'h0043082A in binary is 000000 00010 00011 00001 00000 101010

As we know, and is a R-Type instruction, therefore if we split the 32 bit binary values into its corresponding sections, we will have:
000000 = op code, 00010 = operand a, 00011 = operand b, 00001 = destination register, function: 011010

```
// Initial values of the registers (For testing purposes)
initial begin
  registers [1] = 32'h1;
  registers [2] = 32'h9;
  registers [3] = 32'h2;
  registers [4] = 32'h4;
  registers [5] = 32'h8;
  registers [6] = 32'h7;
  registers [7] = 32'h7;
  registers [8] = 32'h6;
end
```

Figure 4.1.43: Register values

In the figure above, we can see that our register file holds arbitrary register values to be tested during these operations fetched from instruction memory. In this case, we are using register [2]'s value 9, and anding it to register [3]'s value 2.

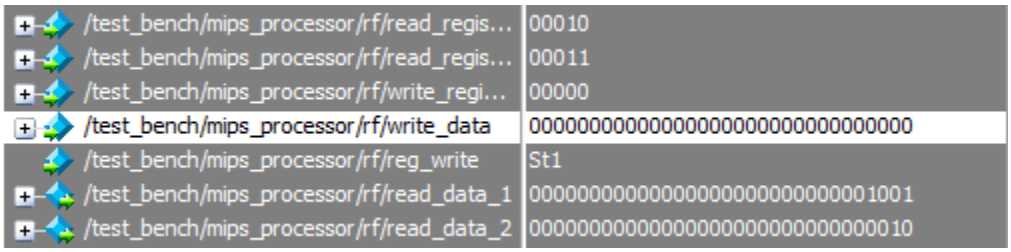


Figure 4.1.44: Register values simulation

In the figure above, we can observe that during the next cycle our instruction memory has been updated with the corresponding register numbers and register data. It is reading register 2 and 3, with values 9 and 2 respectively. **Which is correct.**

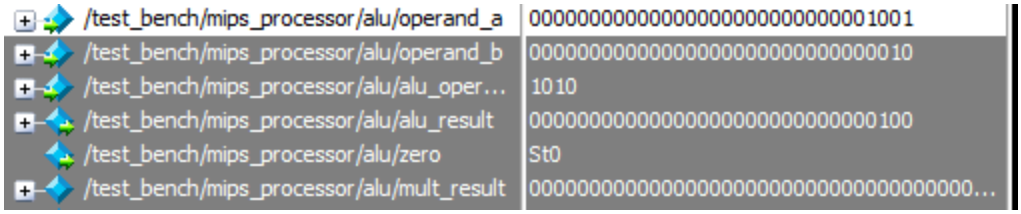


Figure 4.1.45: ALU Operation for DIV

Now moving to the execution stage of our process, we have a **DIV** operation between two operands which had values hardcoded in the register file module to them. We have so far fetched the instruction, and moved the register address with their corresponding values. Now let's check our simulation results and see if we get the resulting values from our signals. We can observe that our first fetched instruction from instruction memory is resulting in 00…100 at the alu_result register defined above **because 9/2=4.5 and 4 is what it will be displayed. Therefore it is in fact properly working.**

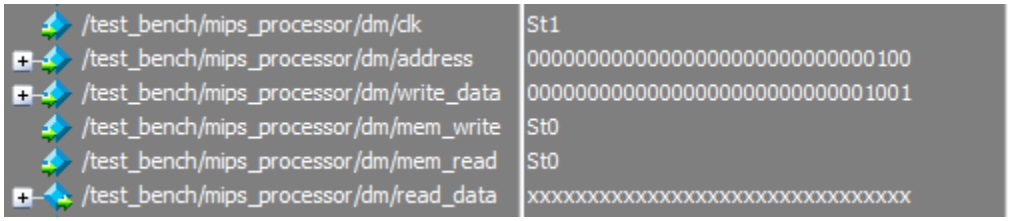


Figure 4.1.46: Datamemory for DIV

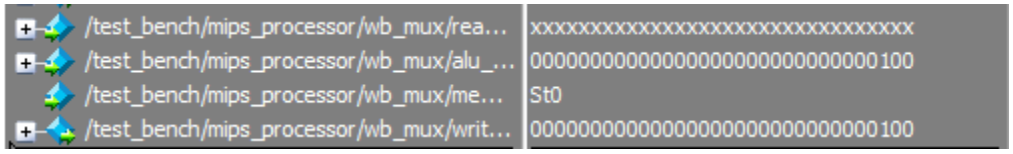


Figure 4.1.47: WB mux for DIV

It could be observed that these two modules are also operating successfully for the first four stages of the pipeline, now let's check if the final result is written back to our write register of our register module:

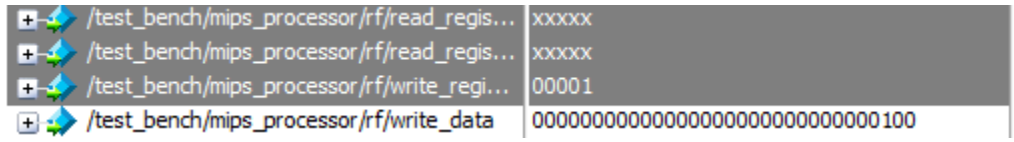


Figure 4.1.48: Reg write back DIV

Yes, it did in fact write back the 000..100 to the write_data location of the register module. Which will satisfy the working instruction test for DIV instruction.

Testing instruction MUL:

Figure 4.1.49: Instruction fetch for MUL

As we can see, the instruction is being fetched from the instruction memory correctly. We define the instruction 32'h00430818; // MUL to be fetched next. The 32 bit binary version of the instruction is given to us in the simulation. If we translate it to binary and separate its section we will see that 32'h00430818 in binary is 000000 00010 00011 00001 00000 011000

As we know, and is a R-Type instruction, therefore if we split the 32 bit binary values into its corresponding sections, we will have:
000000 = op code, 00010 = operand a, 00011 = operand b, 00001 = destination register, function: 011000

```
// Initial values of the registers (For testing purposes)
initial begin
  registers [1] = 32'h1;
  registers [2] = 32'h9;
  registers [3] = 32'h2;
  registers [4] = 32'h4;
  registers [5] = 32'h8;
  registers [6] = 32'h7;
  registers [7] = 32'h7;
  registers [8] = 32'h6;
end
```

Figure 4.1.50: Register values

In the figure above, we can see that our register file holds arbitrary register values to be tested during these operations fetched from instruction memory. In this case, we are using register [2]'s value 9, and anding it to register [3]'s value 2.

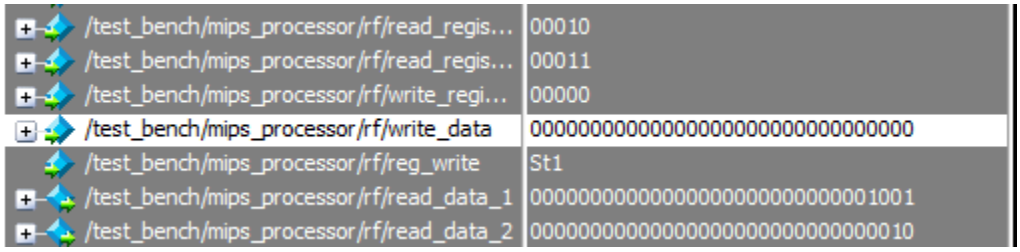


Figure 4.1.51: Register values simulation

In the figure above, we can observe that during the next cycle our instruction memory has been updated with the corresponding register numbers and register data. It is reading register 2 and 3, with values 9 and 2 respectively. **Which is correct.**

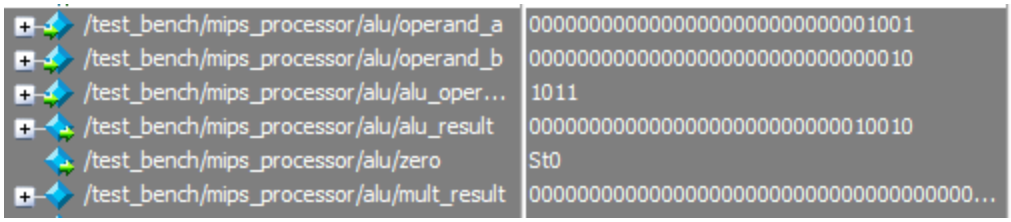


Figure 4.1.52: ALU operation for MUL

Now moving to the execution stage of our process, we have a **MUL** operation between two operands which had values hardcoded in the register file module to them. We have so far fetched the instruction, and moved the register address with their corresponding values. Now let's check our simulation results and see if we get the resulting values from our signals. We can observe that our first fetched instruction from instruction memory is resulting in 00…10010 at the alu_result register defined above **because 9*2=18 and 18 is what will be displayed at alu_result. Therefore it is in fact properly working.**

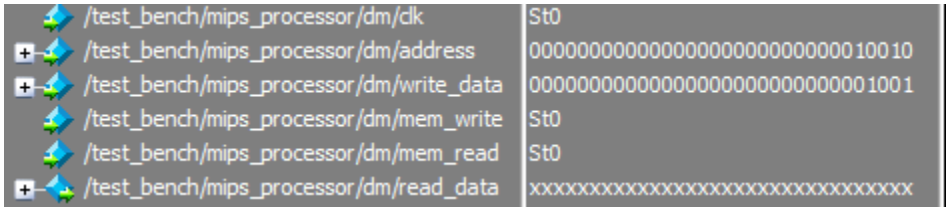


Figure 4.1.53: Datamemory for MUL

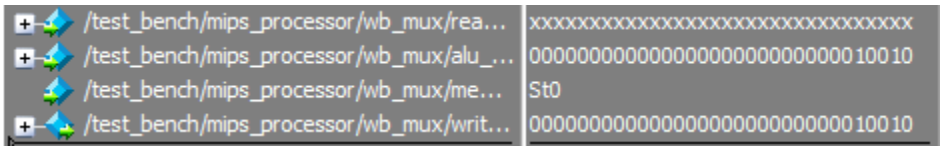


Figure 4.1.54: WB mux for MUL

It could be observed that these two modules are also operating successfully for the first four stages of the pipeline, now let's check if the final result is written back to our write register of our register module:

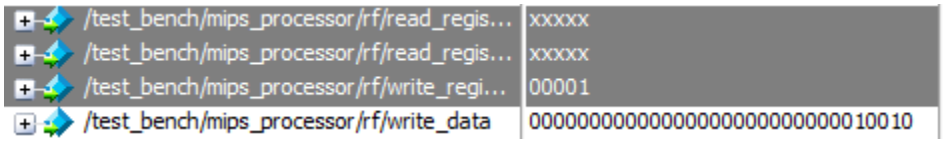


Figure 4.1.55: Reg write back DIV

Yes, it did in fact write back the 000..10010 to the write_data location of the register #1 module. Which will satisfy the working instruction test for MUL instruction.

Testing instruction ADDI:

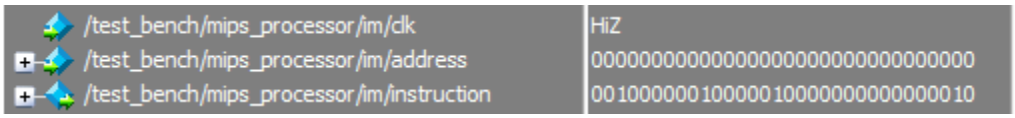


Figure 4.1.56: Instruction fetch for ADDI

As we can see, the instruction is being fetched from the instruction memory correctly. We define the instruction 32'h20410002; // ADDI to be fetched next. The 32 bit binary version of the instruction is given to us in the simulation. If we translate it to binary and separate its section we will see that 32'h20410002 in binary is 001000 00010 00001 0000000000000010

As we know, and is a I-Type instruction, therefore if we split the 32 bit binary values into its corresponding sections, we will have:
001000 = op code, 00010 = operand a, 00001 = destination register, Imm = 10.

```
// Initial values of the registers (For testing purposes)
initial begin
  registers [1] = 32'h1;
  registers [2] = 32'h9;
  registers [3] = 32'h2;
  registers [4] = 32'h4;
  registers [5] = 32'h8;
  registers [6] = 32'h7;
  registers [7] = 32'h7;
  registers [8] = 32'h6;
end
```

Figure 4.1.57: Register values

In the figure above, we can see that our register file holds arbitrary register values to be tested during these operations fetched from instruction memory. In this case, we are using register [2]'s value 9.

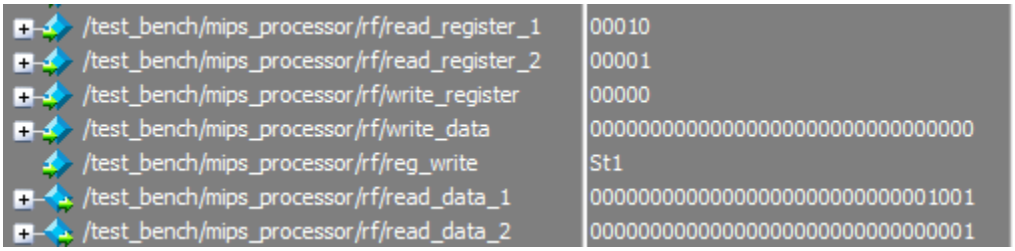


Figure 4.1.58: Register values simulation

In the figure above, we can observe that during the next cycle our instruction memory has been updated with the corresponding register numbers and register data. It is reading register 2 and 1, with values 9 and 3 respectively. **Which is correct.**

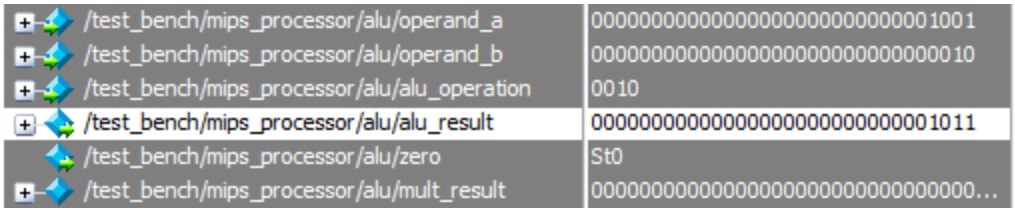


Figure 4.1.59: ALU operation for ADDI

Now moving to the execution stage of our process, we have an ADDI operation between an operand and immediate value, which has values hardcoded in the register file module to them. We have so far fetched the instruction, and moved the register address with their corresponding values. Now let's check our simulation results and see if we get the resulting values from our signals. We can observe that our first fetched instruction from instruction memory is resulting in **1011** at the alu_result register defined above **because 9+2=11 and 11 is what will be displayed at alu_result. Therefore it is in fact properly working.**

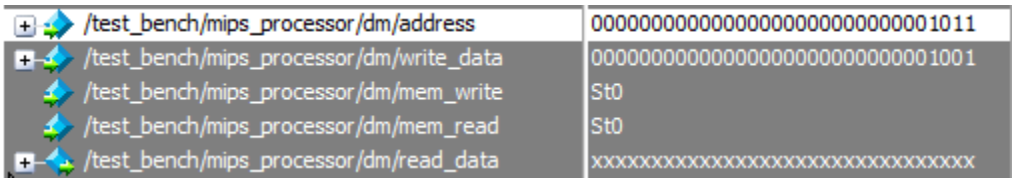


Figure 4.1.60: Datamemory for ADDI

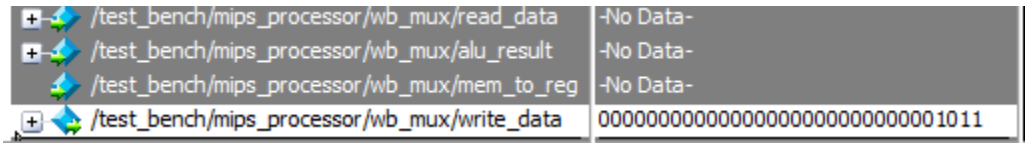


Figure 4.1.61: WB mux for ADDI

It could be observed that these two modules are also operating successfully for the first four stages of the pipeline, now let's check if the final result is written back to our write register of our register module:

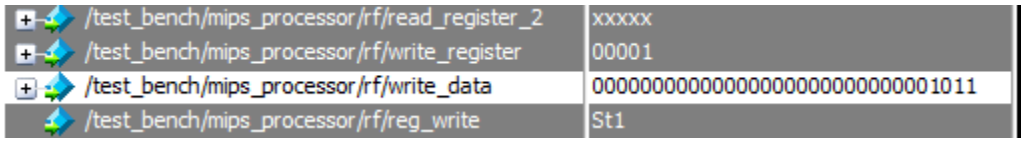


Figure 4.1.62: WB mux for ADDI

Yes, it did in fact write back the 000..1011 to the write_data location of the register #1 module. Which will satisfy the working instruction test for ADDI instruction.

Testing instruction BEQ:

Figure 4.1.63: Instruction fetch for BEQ

As we can see, the instruction is being fetched from the instruction memory correctly. We define the instruction 32'h10850019; //BEQ to be fetched next. The 32 bit binary version of the instruction is given to us in the simulation. If we translate it to binary and separate its section we will see that 32'h10850019 in binary is 000000 01000 01000 01010000000000011001

As we know, and is a I-Type instruction, therefore if we split the 32 bit binary values into its corresponding sections, we will have:
000000 = op code, 01000 = operand a, 01000 = destination register, Imm = 01010000000000011001.

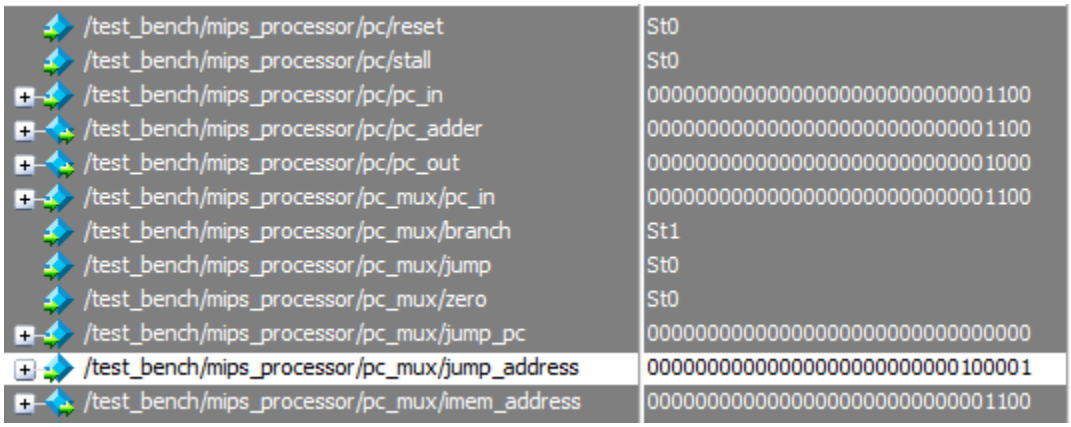


Figure 4.1.64: PC for BEQ

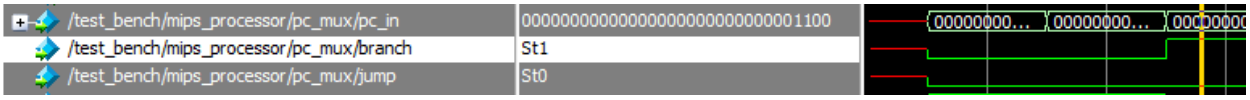


Figure 4.1.65: PC_mux for BEQ

As could be seen above, once BEQ instruction is passed thru the PC Mux, the signal for branch goes high, and will have a branch operation successfully conducted afterwards.

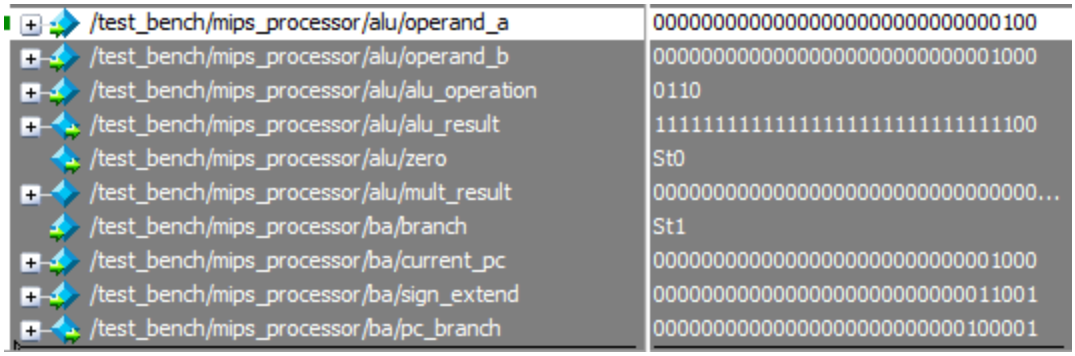


Figure 4.1.65: Branch calculation

The above figure shows how the branch is calculated in ALU module, and our branch adder module's current pc with sign extend and pc_branch signals which shows where to branch.

Testing instruction J:

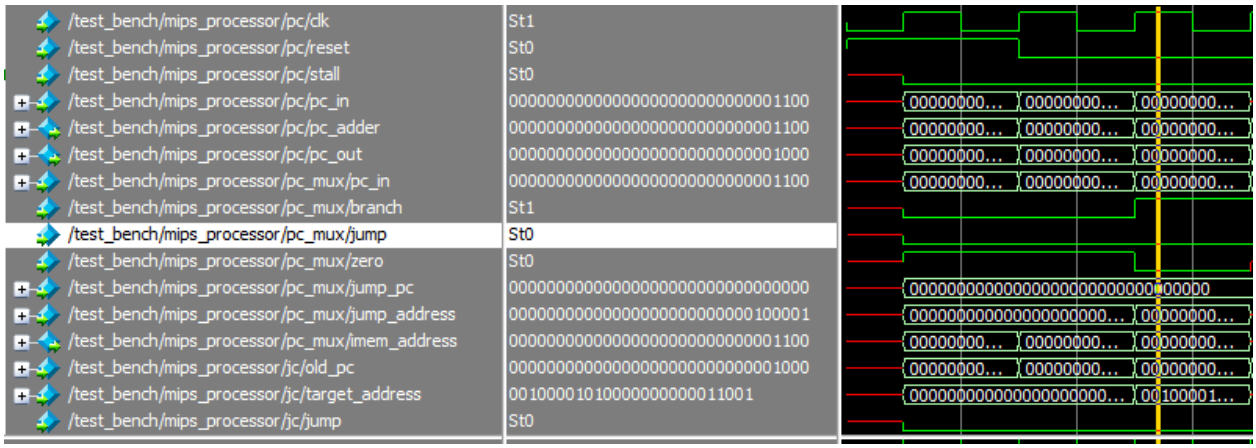


Figure 4.1.66: Jump instruction

As we can see in the figure above, we will have our jump instruction fetch and execute. We will fetch the instruction 32'h08000101; // JUMP. The 32 bit binary version of the instruction is given to us in the simulation. If we translate it to binary and separate its section we will see that 32'h08000101 in binary is 000010 00000000000000000100000001

As we know, and is a J-Type instruction, therefore if we split the 32 bit binary values into its corresponding sections, we will have:
000010 = op code, Imm = 00000000000000000100000001.

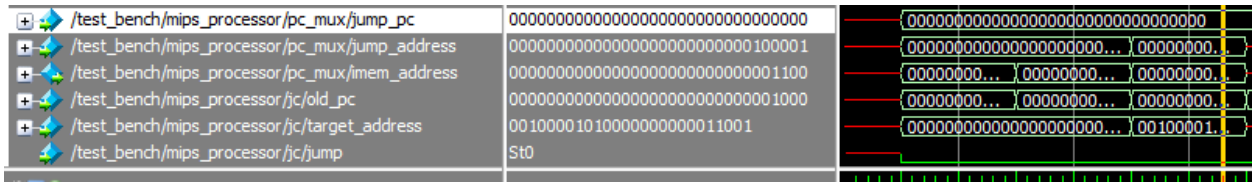


Figure 4.1.66: Jump calculation

As we can see in the above figure, our module will calculate the target address and jump from the previous PC value. Therefore, this instruction is working properly.

Testing instruction LW:

Figure 4.1.67: Instruction fetch for LW

As we can see, the instruction is being fetched from the instruction memory correctly. We define the instruction 32'h8c410064; // LW to be fetched next. The 32 bit binary version of the instruction is given to us in the simulation. If we translate it to binary and separate its section we will see that 32'h8c410064 in binary is 100011 00010 00001 0000000001100100

As we know, and is a I-Type instruction, therefore if we split the 32 bit binary values into its corresponding sections, we will have:
100011 = op code, 00010 = operand a, 00001 = destination register, Imm = 0000000001100100.

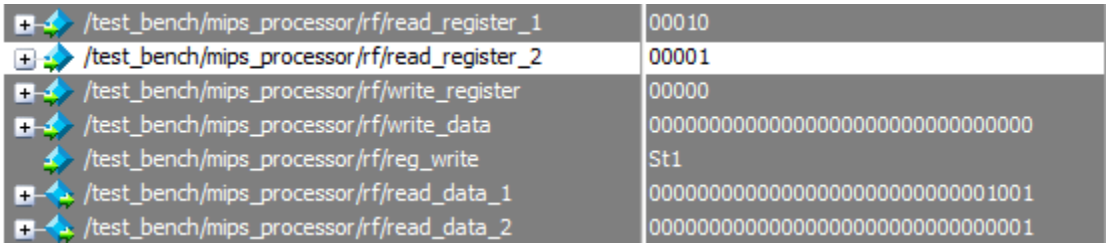


Figure 4.1.68: Register values simulation

As we can see in the figure above, we have our source register displayed under read_register1, which is register 2. And our destination register at register 1.

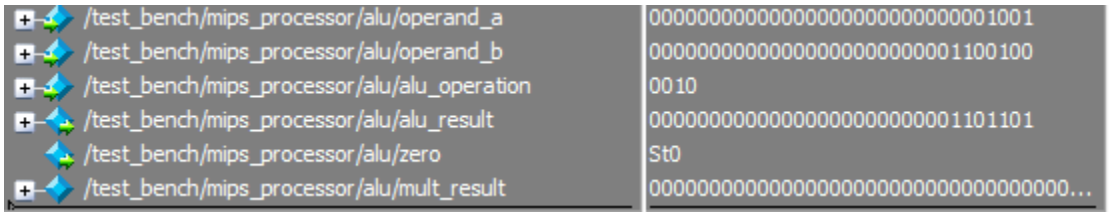


Figure 4.1.69: ALU for LW

Now moving to the execution stage of our process, we have an LW operation between an operand and immediate value, which has values hardcoded in the register file module to them. We have so far fetched the instruction, and moved the register address with their corresponding values. Now let's check our simulation results and see if we get the resulting values from our signals. We can observe that our first fetched instruction from instruction memory is resulting in **1011** at the alu_result register defined above **because 9+100=109 and 1101101 is what will be displayed at alu_result. Therefore it is in fact properly working.**

Testing instruction SW:

Figure 4.1.70: Instruction fetch for SW

As we can see, the instruction is being fetched from the instruction memory correctly. We define the instruction 32'hAC610064; // SW to be fetched next. The 32 bit binary version of the instruction is given to us in the simulation. If we translate it to binary and separate its section we will see that 32'hAC610064 in binary is 101011 00011 00001 0000000001100100

As we know, and is a I-Type instruction, therefore if we split the 32 bit binary values into its corresponding sections, we will have:
101011 = op code, 00011 = operand a, 00001 = destination register, Imm = 0000000001100100.

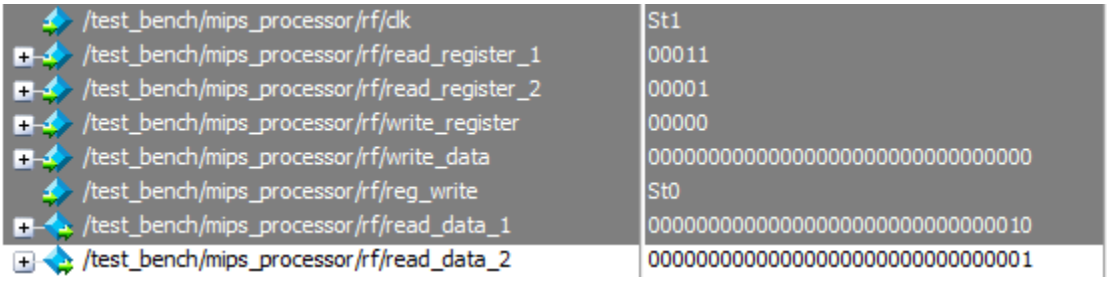


Figure 4.1.71: Register values simulation

As we can see in the figure above, we have our source register displayed under read_register1, which is register 2. And our destination register at register 1.

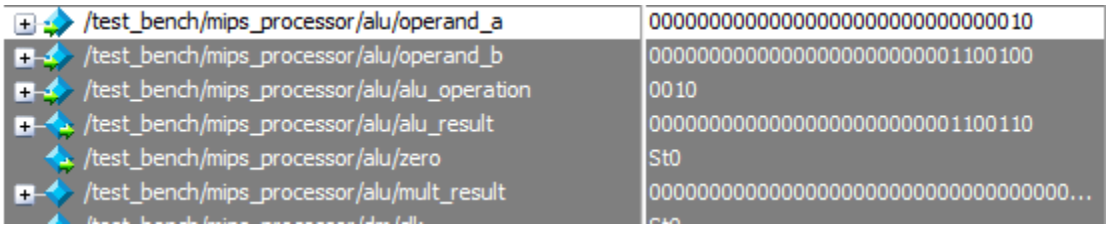


Figure 4.1.72: ALU for SW

We can observe that our first fetched instruction from instruction memory is resulting in **1011** at the alu_result register defined above **because 2+100=102 and 1100110 is what will be displayed at alu_result. Therefore it is in fact properly working.**

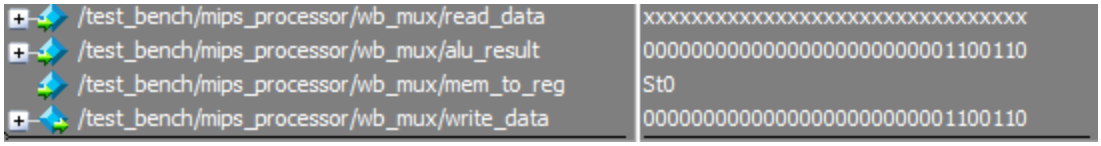


Figure 4.1.73: WB Mux for SW

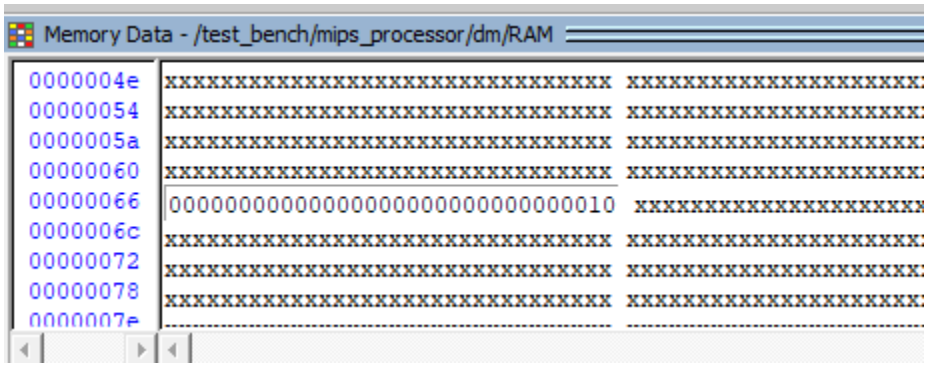


Figure 4.1.73: RAM results

As we can see, our RAM block and WB mux, will in fact include 000…10 results on the corresponding RAM block of 102. Therefore, SW instruction is working properly.

Testing instruction MFHI and MFLO:

The MFHI (Move From HI) and MFLO (Move From LO) instructions are used in MIPS assembly language to retrieve the contents of the special-purpose registers HI (High) and LO (Low), respectively. These registers are used for storing the high and low parts of the result when performing multiplication or division operations in MIPS.

Here's a brief explanation of each instruction:

MFHI (Move From HI):

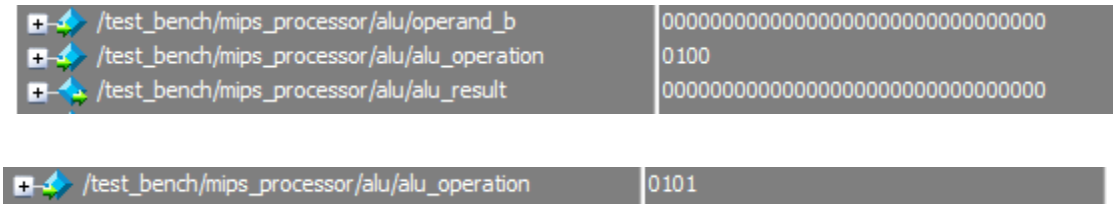
Syntax: mfhi rd
Operation: Copies the value from the HI register to the destination register rd.
Example: mfhi $t0 copies the value of HI into register $t0.
MFLO (Move From LO):

Syntax: mflo rd
Operation: Copies the value from the LO register to the destination register rd.
Example: mflo $t1 copies the value of LO into register $t1.
These instructions are particularly useful when performing multiplication or division operations that require a 64-bit result or when working with large numbers in MIPS assembly language.

```
4'b0100: begin // MFHI
  alu_result = $hi;
end
4'b0101: begin // MFLO
  alu_result = $lo;
end
```

Figure 4.1.74: MFHI & MFLO results

As could be seen from the Verilog code above, we have instantiated our hi and lo alu results accordingly, Now let's ake a look at our ALU operation.



Therefore, it is in fact passing the instruction MFHI and MFLO to our alu block successfully to execute.

Testing hazard:

In order to test our hazard detect unit, we will have two add operations back to back that will require the first add operations result on the second operations. As we know, this won't be possible until the first add operation's ALU result is successfully achieved by the processor. Therefore, we are expecting a stall signal here.

Figure 4.1.75: First ADD operation

Our first operation is 32'h00E80820 // ADD, as we know it is an R type operation. If we break this instruction apart we will get: 000000 00111 01000 00001 00000 100000

000000 = op code, 00111 = operand a, 01000 = operand b, 00001 = destination register, function = 100000.

In this operation we are adding the value of register [7] + register [8] and storing it in register [1].
Now let's take a look our register file to see what the values in those registers are:

```
registers [7] = 32'h7;
registers [8] = 32'h6;
```

Figure 4.1.76: Register values

Now, let's take a look at our second ADD operation that will be fetched right after first ADD operation.

Figure 4.1.75: Second ADD operation

Our second operation is 32'h00280820; // ADD, as we know it is an R type operation. If we break this instruction apart we will get: 000000 00001 01000 00001 00000 100000

000000 = op code, 00001 = operand a, 01000 = operand b, 00001 = destination register, function = 100000.

Now, since we have a dependence at our register 1 value, our system will have to stall once to get the correct value for register [1].

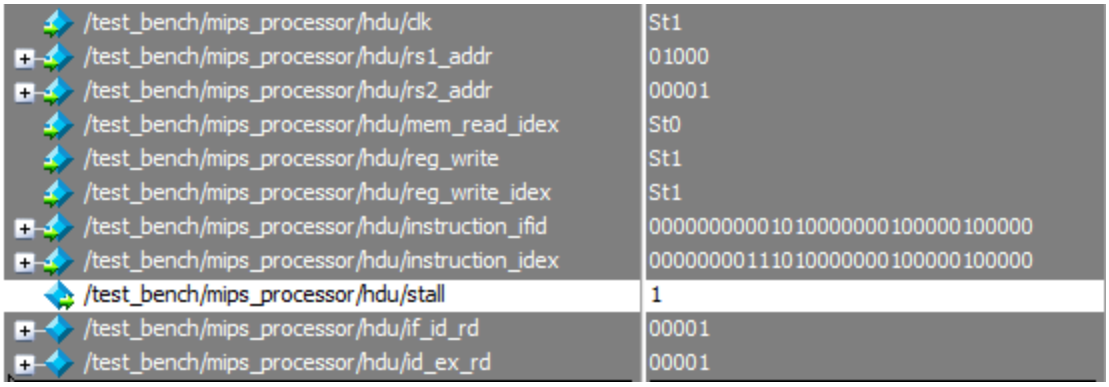


Figure 4.1.75: Hazard control module

As we can see, our hazard module will generate **stall = 1 signal** for the following operation to be stalled until our register[1] value is updated from our first instruction. Once it does, it will continue to finish the second instruction going into the ALU module and will give us the correct value for the operation. **Therefore, our Hazard control is working properly as well.**

4.2 Data Analysis

We have conducted a 10 minute youtube video to demonstrate the basics of our processor and how it works. It could be reached from this link: https://youtu.be/u5yX6-xm8oE

Now, to analyze how our instructions are being fetched to our mips processor, let's analyze our instructions.txt file that is available in the same directory as of our processor files.

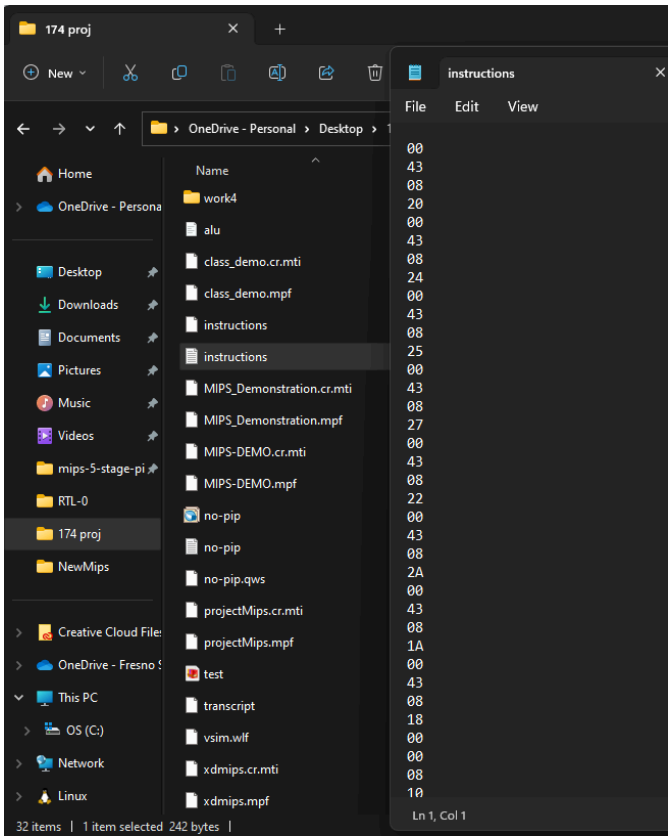


Figure 4.2.1: Instructions.txt file

We are inserting our instructions via an output text file to our processor's instruction_memory module in order to fetch instructions. This way, we are simulating our processor at all times, and the only file we need to make adjustments is this text file when we want to upload new sets of instructions for the program.

Now, let's take a look at our instructions_memory module to understand how this is allowed and possible to accomplish such feature:

C: > Users > puyaf > OneDrive > Desktop > 174 proj > 1 > NewMips > instruction_memory.v

```
1   module instruction_memory (
2       input             clk,
3       input      [31:0] address,
4       output reg [31:0] instruction
5   );
6
7       reg [7:0] mem[0:4095];
8
9       integer fp;
10      integer status;
11      integer i;
12
13      initial begin
14          fp = $fopen("instructions.txt", "r");
15          if (fp != 0) begin
16              for (i = 0; i < 4096; i = i+1) begin
17                  status = $fscanf(fp, "%h", mem[i]);
18                  if (status == 0) begin
19                      $display("Error: Could not read instruction at address %d", i*4);
20                      $finish;
21                  end
22              end
23              $fclose(fp);
24          end
25          else begin
26              $display("Error: Could not open file 'instructions.txt'");
27              $finish;
28          end
29      end
30
31      always @(address) begin
32          instruction = {mem[address+0], mem[address+1], mem[address+2], mem[address+3]};
33      end
34
35  endmodule
```

Figure 4.2.1: Instructions_memory module

The given code represents a Verilog module called "instruction_memory" that models an instruction memory block. It reads instructions from a text file called "instructions.txt" and stores them in an internal memory array called "mem". The module provides an interface to access instructions based on the provided address.

Module and Memory Declaration:

The module declaration specifies the module name (instruction_memory) and its ports: clk (a clock input), address (a 32-bit input representing the memory address), and instruction (a 32-bit output representing the retrieved instruction).

```
module instruction_memory (
    input             clk,
    input      [31:0] address,
    output reg [31:0] instruction
);
```

Figure 4.2.2: module declaration

The mem declaration creates an internal memory array of size 4096, where each element is an 8-bit register. It represents the instruction memory.

```
reg [7:0] mem[0:4095];
```

Figure 4.2.3: Memory declaration

Initial Block:

```
initial begin
    fp = $fopen("instructions.txt", "r");
    if (fp != 0) begin
        for (i = 0; i < 4096; i = i+1) begin
            status = $fscanf(fp, "%h", mem[i]);
            if (status == 0) begin
                $display("Error: Could not read instruction at address %d",
                $finish;
            end
        end
        $fclose(fp);
    end
    else begin
        $display("Error: Could not open file 'instructions.txt'");
        $finish;
    end
end
```

Figure 4.2.4: Initial block

The initial block is executed once at the start of simulation. Inside the block, it attempts to open the file "instructions.txt" using $fopen function, which returns a file pointer (fp). If the file is successfully opened (fp != 0), the block reads instructions from the file and stores them in the memory array mem.

The for loop iterates from 0 to 4095, reading a hexadecimal value from the file using $fscanf and storing it in mem[i]. If the read operation fails (status == 0), it displays an error message and terminates the simulation using $finish. Finally, the file is closed using $fclose(fp).

If the file cannot be opened (fp == 0), it displays an error message and terminates the simulation.

Always Block:

```
always @(address) begin
    instruction = {mem[address+0], mem[address+1], mem[address+2], mem[addre
end
```

Figure 4.2.5: Always block

The always block triggers whenever there is a change in the address input. It assigns the corresponding instruction to the instruction output by concatenating four bytes from the mem array based on the given address. The concatenation is performed using the {} syntax, which combines the four elements into a 32-bit value.

In summary, this Verilog code represents an instruction memory module that reads instructions from a file during initialization and provides an interface to retrieve instructions based on the provided memory address.

Once the instructions get fetched, the processor will continue doing what its doing with the rest of its blocks and completing the instruction as it is displayed in **4.1 Experimental results** sections of this report.

4.3 Demo Analysis

I uploaded our demonstration videos to YouTube, you can access them from these links, although every instruction has been analyzed individually at our **Experimental Result** section of this report. Since If have already done over 90 pages for the report, I didn't want to go way more beyond this point. Therefore everything regarding analysis and demo is found under these youtube links, as they are also submitted separately to **Canvas**.

Demonstration of r = a - [a/b]*b || Link: https://youtu.be/-FvuESWfeQ8
Demonstration of FORWARDING Logic and Analysis: https://youtu.be/mS1JrcWa30A
Demonstration of HAZARD Control unit and Analysis: https://youtu.be/zPa9nVMNQyU
Demonstration of Sample Program: https://youtu.be/OAL-ecyAuaA

Quartus Analysis for the processor:

For some reason, it seems like that the FPGA memory is not showing up on Quartus Hardware analysis that was synthesized. Here is our results for Hardware analysis:

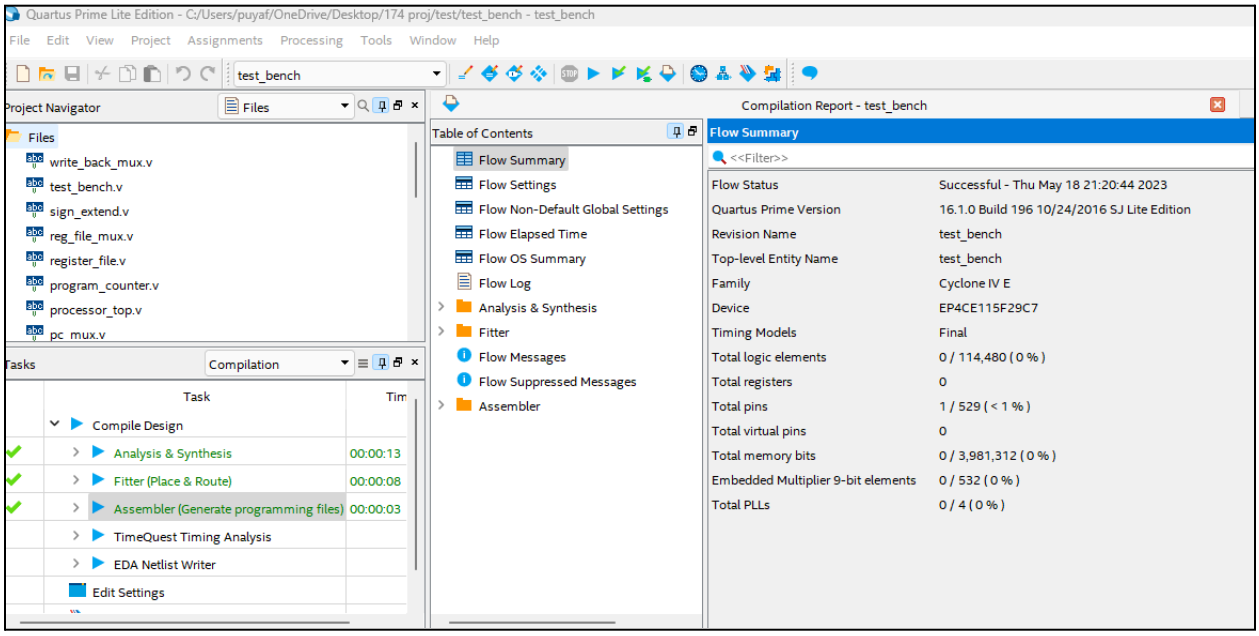


Figure 4.3.1: Quartus analysis

I have tried working on it and fixing it, however I didn't get any positive results. Therefore, this part of the project is not synthesizing correctly where it shows 0 total registers and logical elements for our FPGA syntheses.

General Program instructions, and how to manage the demonstration is given below with directions and logic behind it:

For given program: r = a - [a/b] * b

I-Type: 001000 00000 00010 0000000000000010 // addi r2, r0, 2
I-Type: 001000 00000 00001 0000000000001001 // addi r1, r0, 9
R-Type: 000000 00010 00001 00011 00000 011010 // div r3, r2, r1
R-Type: 000000 00011 00010 00100 00000 011000 // mul r4, r3, r2
R-Type: 000000 00001 00100 00101 00000 100010 // sub r5, r1, r4

Our result (r) will be in Register 5 location stored in memory.

To run the code above, we must store the hex conversions of binary operations in our instructions.txt file and run the simulation. Here below is the given instructions in hex that was demonstrated in the video as well. Below is given how to store these instructions in our **instruction.txt** file.

20
02
00
02
20
01
00
09
00
41
18
1A
00
62
20
18
00
24
28
22

Now Let's test our processor with a different set of operations, A simple multiplication and a longer multiplication.

below does 2 x 9 = 18 saves result into r4

001000 00000 00010 0000000000000010 // 20020002
001000 00000 00011 0000000000001001 // 20030009
000000 00011 00010 00100 00000 011000 // 00622018

below does 25631 x 7543 = 193334633 saves result into r4

001000 00000 00010 0000000000000010 // 2002641F
001000 00000 00011 0000000000001001 // 20031D77
000000 00011 00010 00100 00000 011000 // 00622018

I already included a hazard control unit working in demonstration, but I will add the instructions that I checked here as well.

ADD r3, r1, r2
ADD r4, r1, r3

This causes hazard unit to stall 1 because our r3 value needs to complete updating before it uses it for the next instruction's source operand.

5. CONCLUSIONS

As a conclusion we have integrated and prototype the datapath and the control units of the simple 32-bit MIPS processor with five pipeline stages. This processor is written in Verilog hardware language and able to perform arithmetic/logic such as AND, ADD, ADDI, OR, NOR, SUB, MUL, DIV, data movement such as LW, SW, MFHI, MFLO, and flow control such as J and BEQ instructions. Moreover, all instructions had passed the test successfully and were working accordingly. Finally, The learning outcomes of this project resulted in having hands- on experience on building a computer processor and implementing it on hardware.

6. REFERENCES

Introduction to the MIPS Implementation, https://www.d.umn.edu/~gshute/mips/mips-intro.xhtml. Accessed 4 May 2023.

MIPS Quick Tutorial, https://minnie.tuhs.org/CompArch/Resources/mips_quick_tutorial.html. Accessed 4 May 2023.

"ModelSim HDL simulator | Siemens Software." *Siemens EDA*, https://eda.sw.siemens.com/en-US/ic/modelsim/. Accessed 4 May 2023.

"VHDL MIPS 5 stage pipeline Bug." *Stack Overflow*, 8 December 2011, https://stackoverflow.com/questions/8434743/vhdl-mips-5-stage-pipeline-bug. Accessed 4 May 2023.