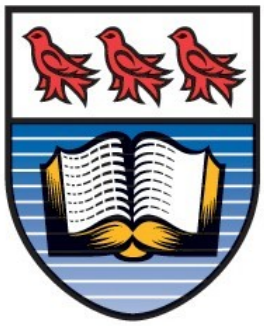
University of Victoria

Electrical and Computer Engineering



CENG 450
Computer Systems and Architecture
Spring 2018

Lab Project Report

Morgan Williams -
Brosnan Yuen -

Submitted: May 15, 2018

Abstract

This report outlines the design process of creating the CENG 450 lab project CPU. As a result of this lab project, a fully functional pipelined 16-bit CPU running the specified ISA has been created. The CPU was realized by programming the lab provided FPGA in VHDL using the Xilinx ISE. The CPU benchmarked remarkably well, despite unoptimized branch handling techniques. The CPU design was successful and bears testament to the robust design approach and implementation. Possible improvements to the CPU design include branch prediction hardware and optimized hazard management.

Contents

List of Figures

List of Tables

Glossary

Central Processing Unit (CPU): Circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions.

Field Programmable Gate Array (FPGA): Integrated circuit designed to be configured by a customer or a designer using hardware description language.

Verilog Hardware Description Language (VHDL): A hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits.

Input/Output (I/O): Digital inputs and outputs.

Arithmetic Logic Unit (ALU): Combinational digital electronic circuit that performs arithmetic and bitwise operations on integer binary numbers.

Instruction Set Architecture (ISA): Document outlining the design and functionality of an instruction scheme.

Random Access Memory (RAM): Volatile digital storage device capable of reading and writing.

Read After Write (RAW): Data hazard associated with overlapping register/memory reading and writing.

Read Only Memory (ROM): Digital storage device with read only capability.

Finite State Machine (FSM): An abstract machine that can be in exactly one of a finite number of states at any given time.

First In First Out (FIFO): Acronym used to describe the ordering of an array in this context.

1 Introduction

The purpose of this lab project was to design and implement a 16-bit CPU for a specified instruction set architecture using a Xilinx Spartan-3E family FPGA. The CPU design is to be implemented by programming the FPGA in VHDL using the Xilinx ISE. The project was introduced January 24th and was to be completed by April 5th 2018.

The specified 16-bit instruction set architecture is comprised of formats A, B, and L. Format A includes arithmetic, logical, as well as, some I/O command instructions to be completed by the ALU. Format B consists of primarily branch instructions but also includes the return instruction. Branches instructions are categorized as relative (BRR), absolute (BR), conditional (BR.(Z or N)), and subroutine (BR.SUB). It is assumed that the Test instruction is used solely for the purpose of resolving conditional branch operations, and as such, is assumed to always be followed by a conditional branch instruction. Format L consists of register and memory instructions like MOV, LOAD, STORE.

2 Considerations

Given the complexity of this project, it is important to outline the criteria and challenges to convey the design approach. This section covers design requirements, project time line, and a brief overview of the inherent design challenges.

2.1 Design Requirements

The goal of this project was to implement a 16-bit pipelined CPU capable of running the provided 16-bit ISA. The project was to be completed according to the project time line outlined in the eponymous section. The basic CPU schematic is shown in Figure 1. The optional implementation of interrupt handling was not implemented in the final design.

The 16-bit ISA is composed of Formats A, B, and L. Format A is composed of primarily of arithmetic operations. Format B consists of branching operations and format L contains all register and memory operations.

Final CPU designs are tested using the provided format tests, as well as, three final test codes that are traditional programs employing all instruction formats.

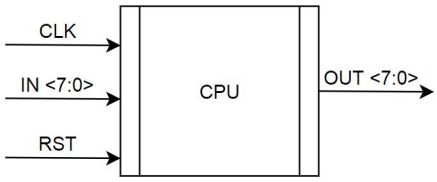


Figure 1: CPU System Schematic

2.2 Project Time Line

1. Preliminary Design Review: February 21, 2018
    - High Level Block Design
    - Format A Instructions
2. Format B Instructions: March 7, 2018
3. Format L Instructions: March 28, 2018
4. Final Design Review and Project Demonstration: April 5, 2018

2.3 Pipeline Hazards

Pipeline hazards are the most difficult design challenge when implementing a pipelined CPU. Hazards are manifested by the pipeline operation itself. Managing pipeline hazards is incumbent when designing a pipelined CPU. These hazards can be categorized into three subsets: structural, control, and data.

2.3.1 Structural Hazards

Pipeline structural hazards occur when two or more separate instructions attempt to access the same hardware component simultaneously. In the example seen in Table 1, a structural hazard occurs as instruction i attempts to access memory at the same time as $i + 3$ attempts to fetch an instruction from memory. This same hazard occurs between $i + 1$ and $i + 4$.

	Clock Cycle Number								
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
i	IF	D	EX	M	WB				
$i + 1$		IF	D	EX	M	WB			
$i + 2$			IF	D	EX	M	WB		
$i + 3$				IF	D	EX	M	WB	
$i + 4$					IF	D	EX	M	WB

Table 1: Pipeline Structural Hazard Example

Structural hazards do not exist within this CPU design because the components used in the fetch and Memory Access stages of the pipeline are unique. In this CPU design, a ROM component stores the program and a RAM module is used as the memory storage solution. A popular method of circumventing this hazard is to employ a dual-ported memory module.

2.3.2 Control Hazards

Control hazards manifest from pipelined branching operations and other operations that change the program counter. The easiest method of handling pipelined branching operations is to implement a stalling mechanism. The purpose of this stalling or "bubbling" mechanism is to permit the resolution of the conditional branch so that the new computed destination can be loaded into the program counter. Table 2 illustrates a stalling mechanism. As a branch instruction is decoded, the successive instruction is stalled and the branch instruction carries through the pipeline to be resolved. Once the branch is resolved, the PC counter is updated and the pipeline continues.

	Clock Cycle Number									
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
Branch	IF	**D**	EX	M	WB					
Branch + 1		**Stall**	**Stall**	**Stall**	IF	D	EX	M	WB	
Branch + 2						IF	D	EX	M	WB

Table 2: Pipeline Control Hazard Example

2.3.3 Data Hazards

Pipeline data hazards occur as dependent instruction results are not executed with correct timing to satisfy data dependencies due to the nature of pipelining. This hazard manifests in a number of scenarios, in particular with RAW errors. Table 3 illustrates a RAW error as the highlighted Decode stage will not deliver the correct ***R1*** into the execute stage, which will result in an erroneous value to be stored in ***R3***.

	Clock Cycle Number									
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
R1 <= R1+R2	IF	D	EX	M	WB					
R3 <= R1 + R2		IF	D	EX	M	WB				

Table 3: Pipeline Data Hazard Example

A method used to solve this common RAW scenario is operand forwarding. This technique derives from the observation that the newly computed value for ***R1*** is available following the execution stage. Control logic detects this hazard by checking if the previous instruction's source destination is the same as the current instruction's source. If this is true, then the previous instruction's execution result is fed back into the current instruction's execute stage, maintaining data integrity. Additional logic is required to select the correct forwarding operand in order to maintain data integrity.

3 Pipeline Components

The pipelined CPU is composed of a number of components that accomplish each stage's function. This section does not cover stage specific controllers and multiplexers, and instead focuses on the functional units.

3.1 ROM Module

The CPU makes use of a ROM module that stores all program instructions. By using a unique ROM module instead of a traditional singular memory unit, all structural hazards are circumvented. Complicated dual-ported memory configurations are also avoided, at the expense of additional hardware. It should be noted that this memory configuration is not the modern approach and is has only been implemented due to its convenience.

The ROM module is found in the fetch stage of the pipeline. The ROM receives an address from the program counter and outputs the corresponding instruction. The ROM array is byte addressable and 2 KB in size.

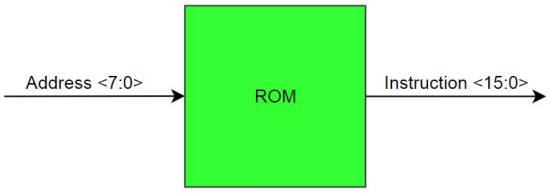


Figure 2: ROM Diagram

3.2 Program Counter

The Program Counter module has undergone a significant evolution over the course of the design process. Initially, the program counter was a self-contained module capable of auto-incrementing and could also update its value based on new branch targets. In the final design, the Program Counter is a signal available to all pipeline stages except Memory Access. The Program Counter is incremented in the Decode stage only as the Control Unit RUN state signal is asserted to the Decode controller. In the event a branch is taken, the Decode controller will load the Program Counter with the computed branch target.

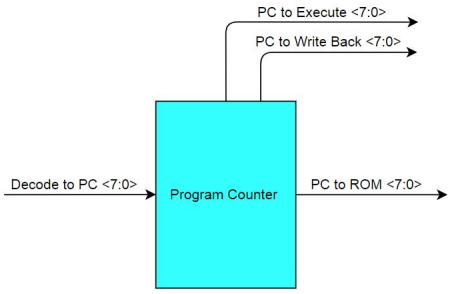


Figure 3: Program Counter Diagram.

3.3 Register File

The Register File is a module that comprises of eight 16-bit registers. This module has three ports used to read contents with corresponding outputs. Indexes 0 and 1 are connected to the Decode Multiplexer and index 2 is connected to the execute state. The purpose of these ports is to ultimately achieve hazard mitigation. The array also has a data input that enables writing to the array. This data input is connected to the write back controller to allow data in the write back stage to be written back into a desired register.

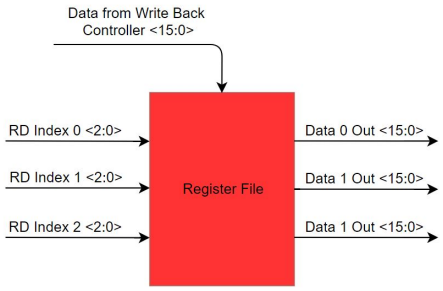


Figure 4: Register File Diargram

3.4 ALU

The ALU is used to responsible for conducting all arithmetic and logical operations. ALU inputs are connected to input multiplexers that perform operand forwarding and other hazard handling. The ALU result and multiply result are fed to the Memory Access latch. N and Z flags are connected to the Decode controller for branch handling purposes. Multiply result contains the upper 16 bits of the 32 bit multiplication resultant. Within the ALU, a temporary multiplication register is used to store the upper half of the multiplication result that is fed out the multiplication port. For this port, it is fed into the Memory Access latch where it is later handled. Section 5.4 covers the multiplication handling design extensively.

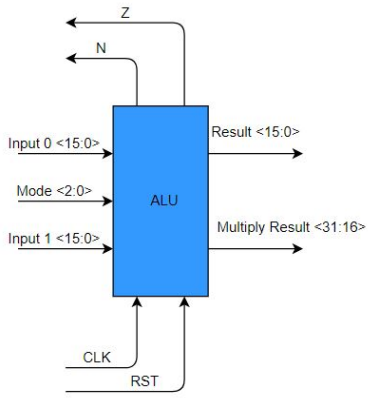


Figure 5: ALU Diagram.

3.5 RAM

The RAM module is used as the CPU memory storage and located in the Memory Access stage. The RAM module is byte addressable and is 4 KB in size. All RAM ports are connected to the Memory Access controller. The Memory Access controller enables normal pipeline stage functionality, as well as load and store hazard mitigation.

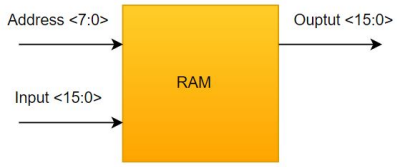


Figure 6: RAM Diagram.

4 Pipeline Design

4.1 Control Unit

The Control unit is a FSM that interacts with all pipeline stages. The FSM is controlled using the previous state and a signal from the Decode stage. The Control Unit FSM consists of several internal states and four output states. The internal states govern the sequence and selection of output state signals. The primary function of the Control Unit is to insert "bubbles" to enable branch handling.

The flow chart seen in Figure 7 illustrates the FSM internal state sequence. Following a Control Unit RESET state, a RUN state is always engaged. The RUN state is only disengaged following a branch or test instruction detected within the Decode stage. The Control Unit inserts three bubbles into the pipeline for a branch instruction and one bubble for a test instruction.

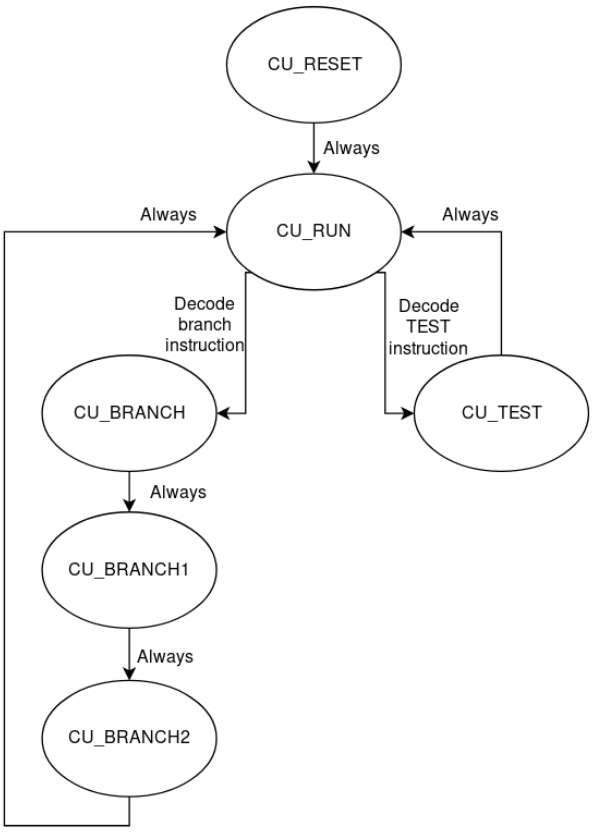


Figure 7: Control Unit FSM Flow Chart.

Tables 4 and 5 depict the resulting Control Unit outputs to the pipeline stages given the Control Unit internal states seen in Figure 7:

	Control Unit State		
Pipeline Stage	*RESET*	*RUN*	*TEST*
Fetch	Reset	Run	Reset
Decode	Reset	Run	Stall
Execute	Reset	Run	Run
Memory	Reset	Run	Run
Write Back	Reset	Run	Run
Next State	*RUN*	*RUN/TEST/BRANCH*	*RUN*

Table 4: Control Unit Outputs to Pipeline Stages.

	Control Unit State		
Pipeline Stage	*BRANCH*	*BRANCH 1*	*BRANCH 2*
Fetch	Stall	stall	Reset
Decode	Reset	Reset	Write PC
Execute	Run	Stall	Stall
Memory	Run	Run	Stall
Write Back	Run	Run	Run
Next State	*BRANCH 1*	*BRANCH 2*	*RUN*

Table 5: Control Unit Outputs to Pipeline Stages Cont'd.

The significance of the Control Unit signalling to each pipeline stage is discussed in the forthcoming pipeline stage sections.

Table 6 shows all possible states of the stage FSM and the interaction with the RAM and Register File, Program Counter value, and stage output. When the RESET state is asserted, all data in all the stages is cleared and the PC value is set to zero. The RUN state operates the stages normally. The STALL state retains the current values inside the stages, as well as their outputs. The WRITE PC state acts identically to STALL, except that the PC value is updated.

	Stage State			
Component	*RESET*	*RUN*	*STALL*	*WRITE PC*
Mem. & Reg.	Set to 0.	Run normally	Keep current value	Keep current value
PC Value	Set to 0.	Run normally	Keep current value	Update PC if branch taken
Stage Output	Set to 0.	Run normally	Keep current value	Output previous value

Table 6: Stage State Interaction with Components.

4.2 Fetch

The purpose of the Fetch stage is to retrieve instructions from the ROM module based on the current Program Counter value. The Fetch controller arbitrates the interaction between the Program Counter and ROM module based on the Control Unit state signals. Figure 8 illustrates the interconnections. The Control Unit can assert output states RESET, RUN, and STALL onto the Fetch Controller. In the event a RESET or RUN state is asserted to the Fetch Controller, the Program Counter contents are loaded into the ROM module. When the STALL state is asserted, the controller does nothing. The Fetch controller is also connected to RST. In the event that RST is toggled, the Program Counter, that has been set to zero by the Decode controller, is fed to the ROM module input to restart the program.

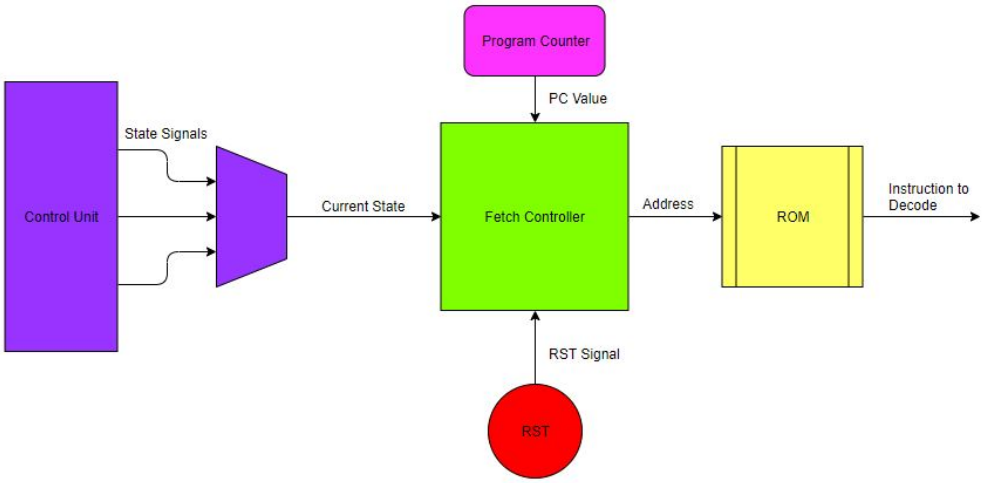


Figure 8: Fetch Block Diagram.

The timing diagram shown in Figure 9 illustrates the asynchronous nature of the ROM module as the instruction address lags the ROM input due to ROM latency.

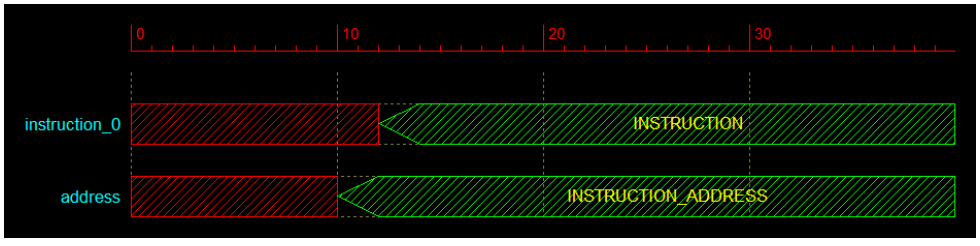


Figure 9: Fetch Stage Timing Diagram.

4.3 Decode

The purpose of the Decode stage is to interpret the instruction for the Fetch stage and coordinate the Execute stage based on the information provided by Hazard Detection Units. Additionally, the Decode controller handles all branching arbitration. Figure 10 illustrates the interconnections and components within the Decode stage.

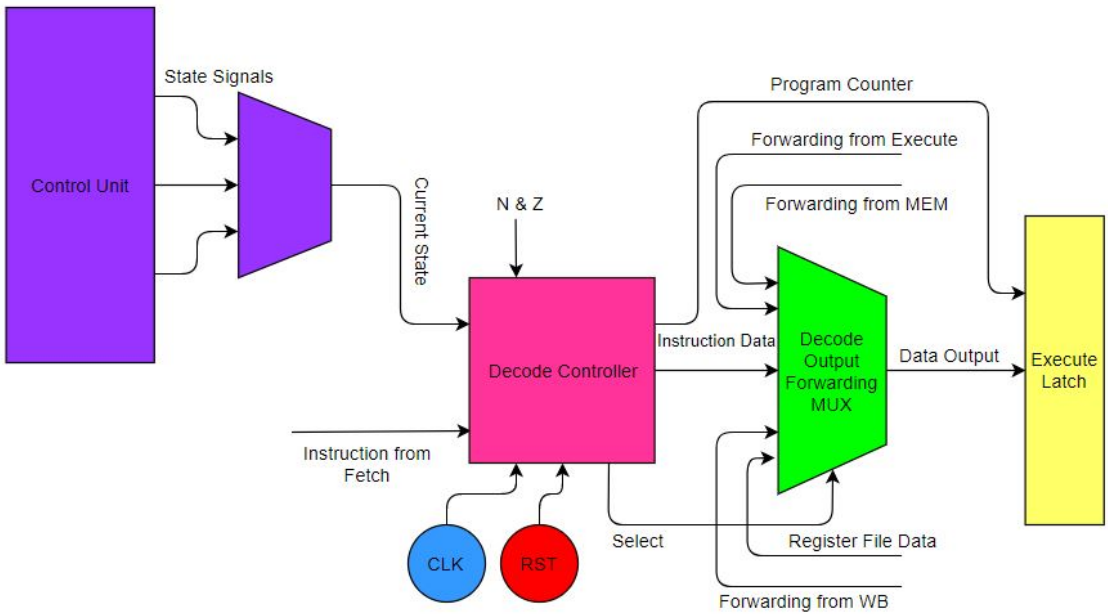


Figure 10: Decode Block Diagram.

When the Control Unit asserts a RESET state onto the Decode controller, Register File inputs and the Decode Output multiplexer are forced to the default state, the Decode Hazard Detection Units are cleared, and N and Z flags are forced to 0. In a STALL state, the intent is to retain the state of the Decode stage. Hazard Detection Units receive a null instruction from Fetch, effectively disengaging hazard detection. Register File inputs and the Decode Output multiplexer is forced to a default state. Both Z and N flags are forced to 0.

The Decode stage runs under normal operation when the Control Unit asserts a RUN state onto the Decode controller. Hazard detection is engaged and acted upon accordingly with the coordination of the Decode controller and Output multiplexer. The Program Counter is incremented if the Decode controller does not detect a Test or branch instruction. Figure 11 illustrates the normal Decode stage operating conditions. The program counter is incremented and the Z and N flags are updated. Instructions are pushed through and the Decode controller and Output multiplexer change state accordingly. The change of Decode Output multiplexer state is indicated as new index values are asserted to the Register File and the data output changes. The delayed output data is to reflect the latency associated with the Decode Output multiplexer logic latency.

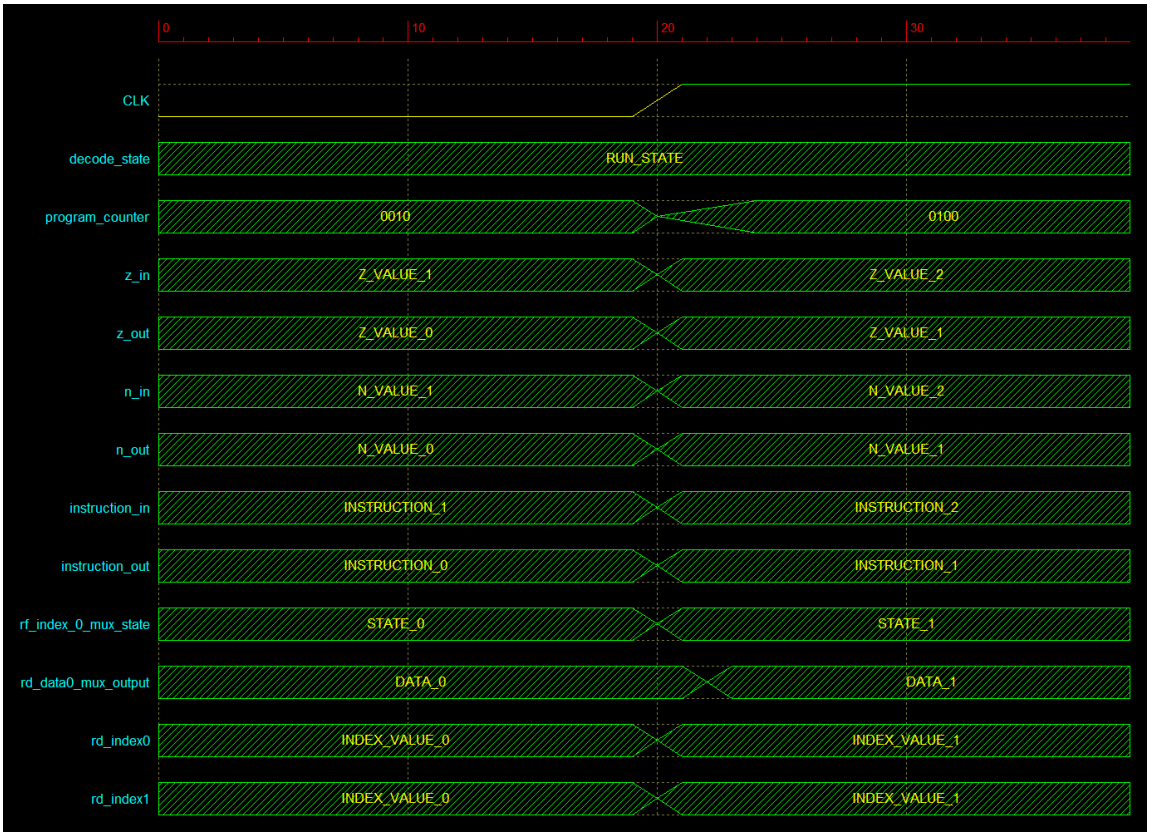


Figure 11: Decode Stage Timing Diagram.

If the Decode stage detects a Test or branch instruction, the Control Unit is signaled accordingly. For branch instructions, the Decode controller asserts a BRANCH state and for a Test instruction a TEST state is asserted onto the Control Unit. For TEST instructions, a 1 cycle delay is inserted into the pipeline. A BRANCH state constitutes a 3 cycle pipeline delay to allow for the branch destination to be calculated. At the final stall cycle, the WRITE PC state is asserted on to the Decode controller. In this state, if the branch conditions are satisfied, the Decode controller updates the Program Counter with the calculated branch destination and the pipeline progresses, otherwise the Program Counter is incremented normally and the pipeline resumes. The timing diagram shown in Figure 12 illustrates the Decode controller when the WRITE PC state is asserted. If the branch is taken, branch destination is loaded into the program counter.

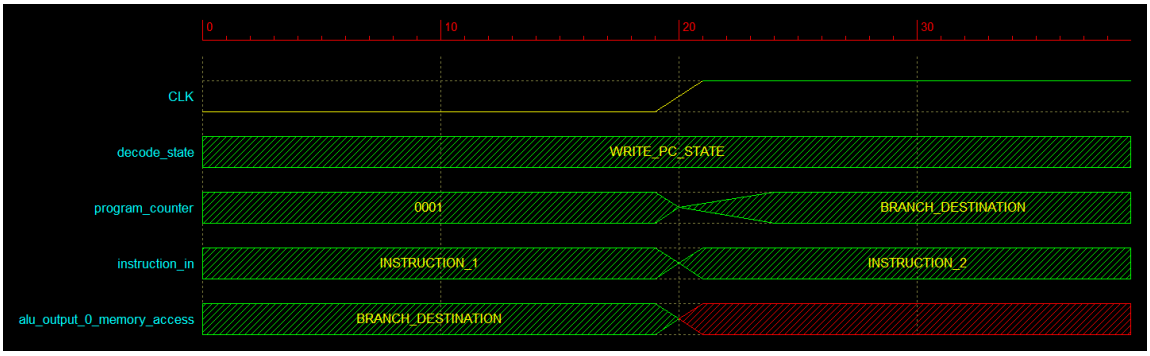


Figure 12: Decode Stage PC Write Timing Diagram.

4.4 Execute

The Execute stage is responsible for conducting all arithmetic and logical operations contained within the specifications of the ISA. Instructions from the Decode stage are brought to Execute controller that arbitrates data fed to the ALU unit. Figure 13 illustrates the interconnections between pertinent Execute stage components.

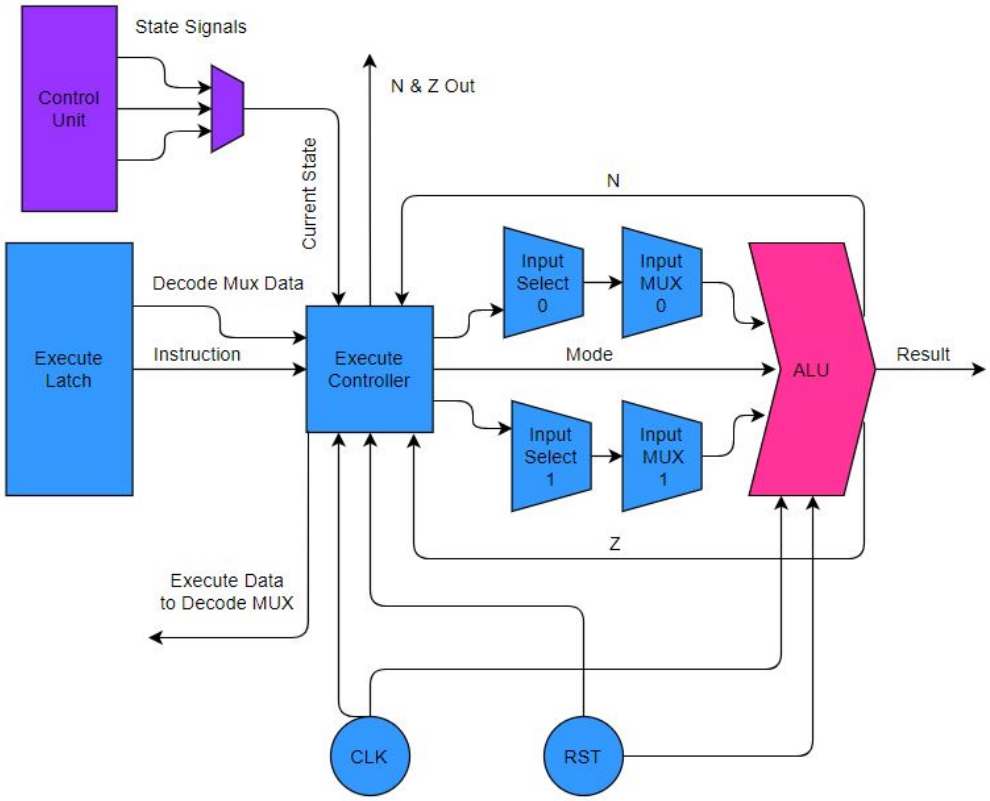


Figure 13: Execute Stage Diagram.

When the RESET or STALL state is asserted onto the Execute controller or a RST signal is toggled, the entire stage is forced to a default state and the ALU input multiplexers are emptied. When the RUN state is asserted, the Execute stage operates normally. The Execute controller feeds instruction data to the ALU input multiplexers based off the information interpreted from the Hazard Detection Units. If a flagged hazard is detected by the Execute controller, the ALU input multiplexers are loaded with the required data and the ALU resultant is directed to the appropriate location to handle the hazard. For example, when a conditional branch instruction arrives, the Execute controller loads an ALU input multiplexer with instruction data to calculate the branch destination. The

resultant is then fed to the Decode controller where a previous Test instruction output flag determines if the Program Counter is loaded with the new branch destination.

The purpose of the ALU input multiplexers is to carry out operand forwarding for hazard mitigation. The two separate multiplexer per input design choice decreases the complexity of each multiplexer but introduces additional hardware and latency. The Execute controller feeds the Input select multiplexers with the required data that is determined by the hazard detection units and the state asserted by the Control Unit. In the event operand forwarding occurs, the Execute controller signals the proper ALU input select multiplexer (determined by the hazard detection units) what forwarding data to select and then passes that data to the ALU input multiplexer.The ALU input multiplexers condition the data supplied to the ALU to enable data forwarding and branch destination calculations.

The Execute stage timing diagram shown in Figure 14 demonstrates the Execute controller loading the ALU input multiplexers with instruction 2 data pass to the ALU for computation. ALU Result 2 delay derives from the Execute controller, ALU input multiplexers, and ALU computation latency.

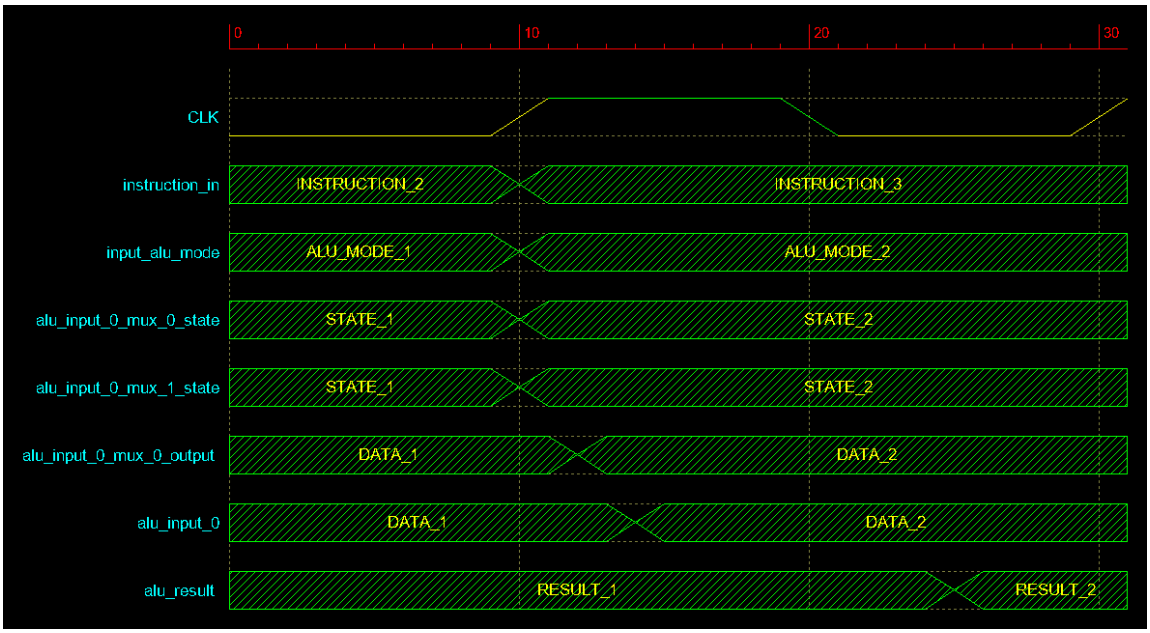


Figure 14: Execute Stage Timing Diagram.

4.5 Memory Access

The primary purpose of the Memory Access stage is to facilitate Load and Store operations. Figure 15 illustrates the Memory Access stage interconnections.

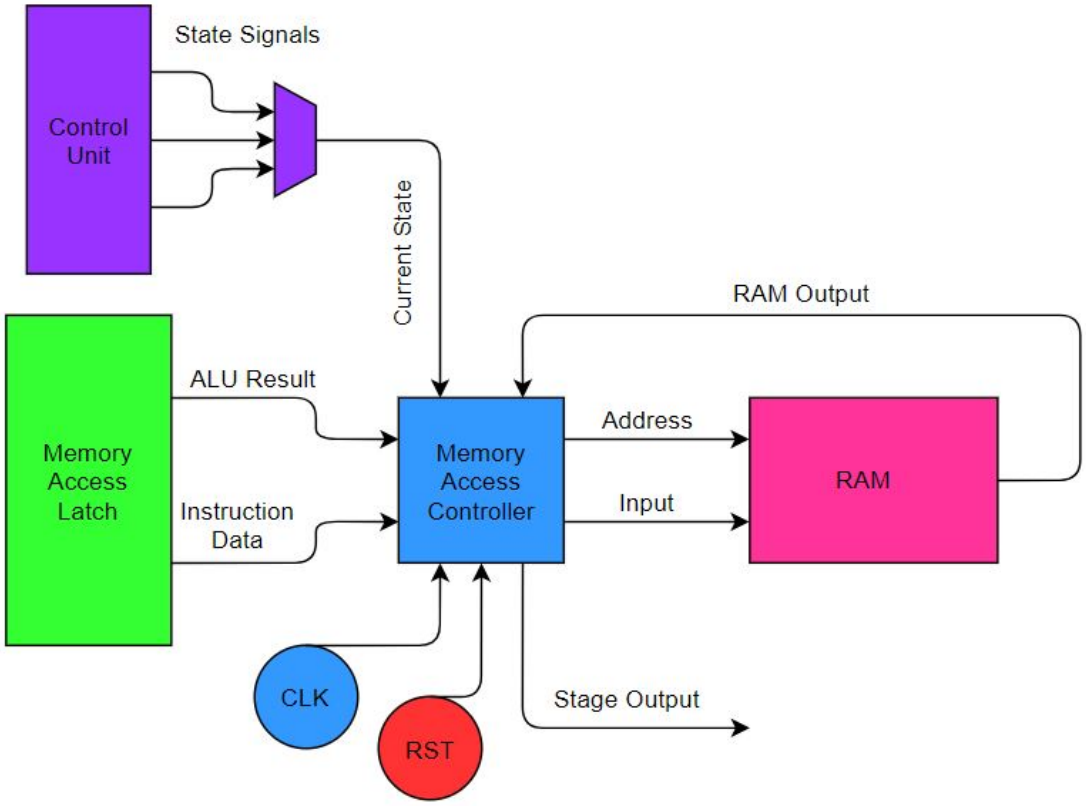


Figure 15: Memory Access Stage Diagram.

The Memory Access controller arbitrates the functionality of the entire stage. When the Control Unit asserts a RESET state or a RST signal is toggled, the Memory controller clears the RAM module and forces all output to a default state. In the RUN state, the Hazard Detection Units are monitored and acted upon accordingly by the Memory Access Controller. Using instruction data from the latch, the Memory Access controller can store and load to and from the RAM module. The RAM output is connected to the Memory Access controller that directs the data for pipelining or forwarding purposes.

The timing digram shown in Figure 16 depicts Data 1 from the Execute latch being stored into the RAM module, triggered at the rising edge. The delay in Data 1 storage is attributed to the combined latency of the Memory Access controller logic and the RAM module.

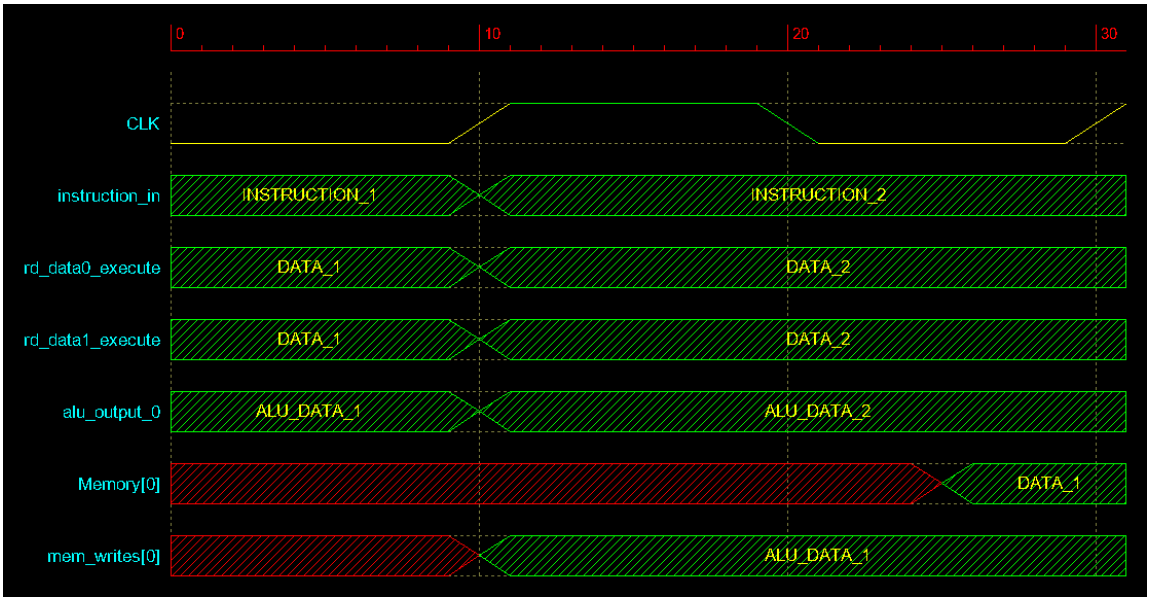


Figure 16: Memory Access Stage Timing Diagram.

4.6 Write Back

The primary function of the Write Back stage is to write results dictated by the ISA to the desired Register File register. Figure 17 depicts the interconnections of the relevant Write Back stage components.

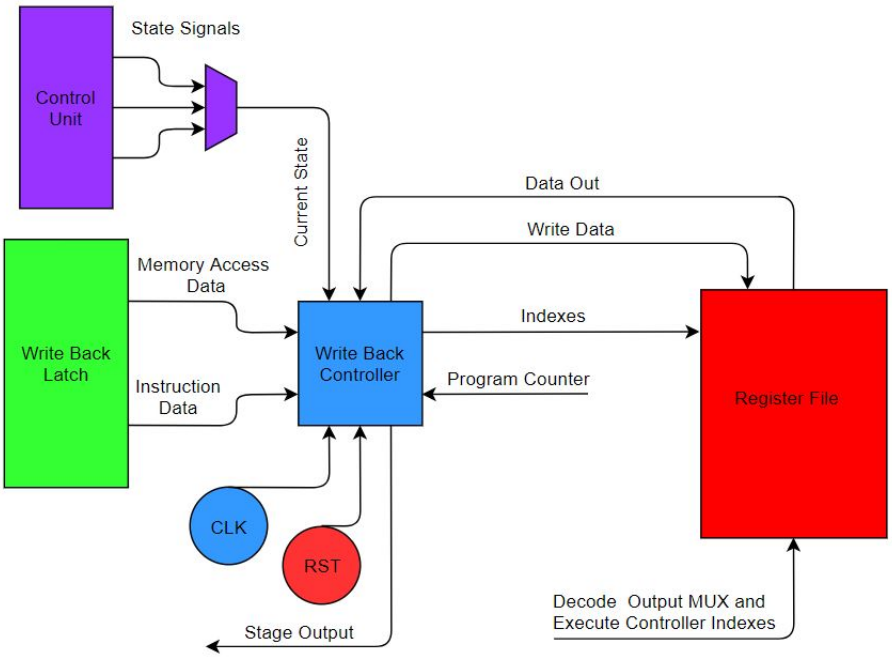


Figure 17: Write Back Stage Diagram.

The Write Back controller arbitrates the functionality of the entire stage. When the Control Unit asserts a RESET state or a RST signal is toggled, the Write Back controller clears the Register File module and forces all output to a default state. In the RUN state, the Hazard Detection Units are not monitored and acted upon because the previous stages have already resolved the current and or forthcoming hazards. Instruction data from the latch is interpreted by the Write Back controller that can read or write to the Register File. For example, the Write Back controller will write RAM output data to the desired register for a Load instruction and for a Load Immediate instruction the constant will be written to R7 by the Write Back controller. In the event a Return instruction is detected, the Write Back controller loads the Program Counter with the value stored in R7 (that is the incremented Program Counter value of the subroutine branch instruction). Additionally, this stage conducts the multiplication handling that is covered in section 5.4.

A Write Back stage timing diagram is shown in Figure 18. The operation illustrated is the Write Back Controller writing Data 1 from the Write Back latch into the register file. Soon after reading register 1 of the Register File, register 1 data is output following a short delay due to the combined latency of the Write Back controller logic and the Register File access.

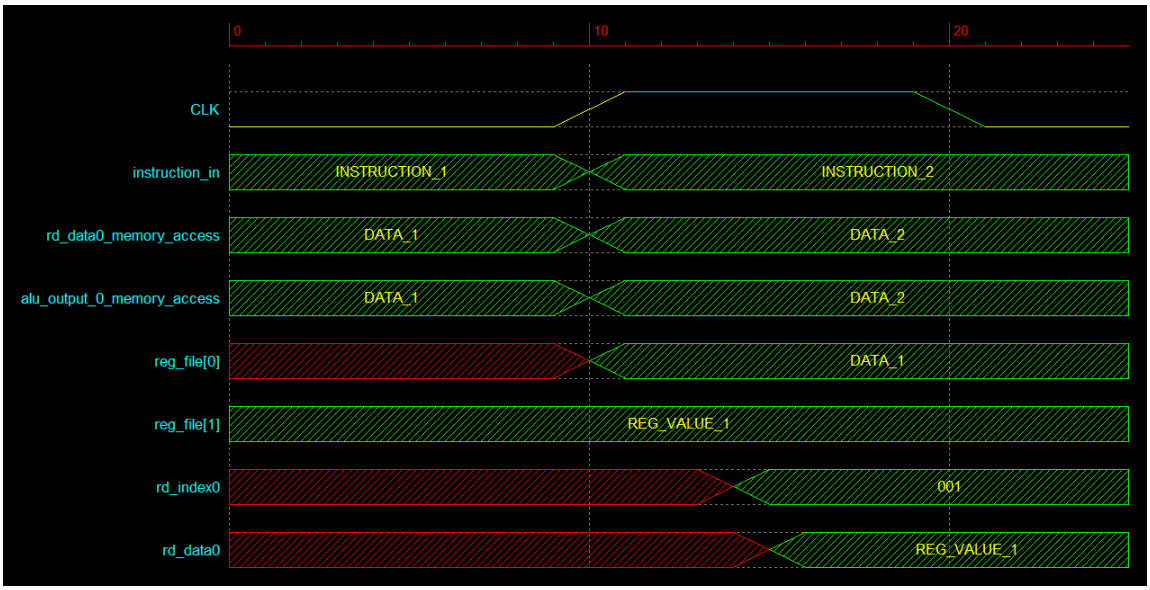


Figure 18: Write Back Stage Timing Diagram.

5 Hazard Mitigation Techniques

5.1 Hazard Detection Units

Decode, Execute, and Memory Access stages have independent Hazard Detection Units. The Hazard Detection Units consist of a write destination FIFO array and a validity indicator vector. The write destination FIFO stores the write destinations of the instructions. The validity indicator vector contains validity bits to indicate which items on the write destination FIFO are valid. At rising edge, the current instruction is checked for hazards. The opcode is decoded and is compared against a list of opcodes that modifies data. If the opcode modifies a register or the RAM, the first bit on the validity indicator vector is marked valid. At the same time, the write destination value is written into the write destinations FIFO. The other data in the write destinations FIFO moves along until they expire. The Hazard Detection Unit will also compare the current instruction's read destination to the write destination FIFO. If a write destination on the write destination FIFO matches the read destination of the current instruction then the hazard is flagged. Flagged hazards will trigger data forwarding in the pipeline.

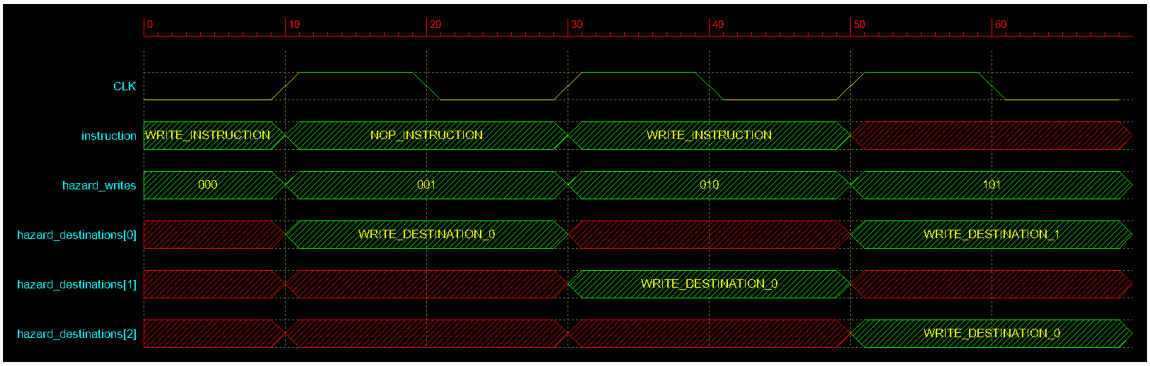


Figure 19: Hazard Detection FIFO Array and Valid Bits.

Figure 19 contains a hazard detection example. The first bit of the validity indicator vector (hazard_writes) is set to 1. All the other bits on the validity indicator vector is set to 0. When the write instruction is clocked in at t = 10 ns, the write destination is stored in the write destination FIFO (hazard_destinations). At t = 30 ns, a NOP instruction is clocked in. Nothing new is written in the FIFO. However, the previous write destination moves through the FIFO. At t = 50 ns, a new write instruction is clocked in. The write destination FIFO and validity indicator vector is updated with new data.

5.2 Operand Forwarding

The most common RAW hazard is an ALU instruction after an ALU instruction. The former ALU instruction does not have enough time to write into the register file before the latter ALU instruction reads the register file. The solution to the problem is operand forwarding. Operands from the former instruction are forwarded to the latter instruction using multiplexers.

	Clock Cycle Number							
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
ADD R1, R4, R0	IF	D	EX	**M**	**WB**			
ADD R0, R1, R3		IF	D	**EX**	**M**	**WB**		
ADD R2, R1, R0			IF	D	**EX**	M	WB	
ADD R3, R1, R0				IF	**D**	**EX**	M	WB

Table 7: ALU Instruction After ALU Instruction Hazard.

Table 7 shows an example of ALU instruction after ALU instruction hazard. At clock 3, R1's value is computed. At clock 4, R1's value is forwarded from the memory access stage (instruction 1) to the execute stage (instruction 2). The forwarding uses muxes from the memory acccess stage to the execute stage. R0's value is also computed at clock 4. At clock 5, R1's value is forwarded from write back stage (instruction 1) to execute stage (instruction 3). At clock 5, R1's value is forwarded from write back stage (instruction 1) to decode stage (instruction 4). At clock 5, R0's value is forwarded from memory access stage (instruction 2) to execute stage (instruction 3). At clock 6, R0's value is forwarded from write back stage (instruction 2) to execute stage (instruction 4). Operands are satisfied at every execute stage of each instruction.

	Clock Cycle Number							
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
IN R1	IF	D	EX	**M**	**WB**			
ADD R0, R1, R3		IF	D	**EX**	M	WB		
ADD R2, R1, R0			IF	D	**EX**	M	WB	
ADD R3, R1, R0				IF	**D**	EX	M	WB

Table 8: IN Instruction Hazard.

Another hazard is any instruction after the IN instruction . The hazard is shown in Table 8. At clock 4, memory access stage (instruction 1) latches the IN data. R1's value becomes the IN data. At the same time in clock 4, the IN data is forwarded to the execute stage (instruction 2). At clock 5, the R1's value is forwarded to instruction 3 and instruction 4. This resolves the IN data hazard.

	Clock Cycle Number						
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***
LOADIMM.lower #9	IF	D	**EX**	M	WB		
LOADIMM.upper #4		IF	**D**	**EX**	M	WB	
MOV R2, R7			IF	**D**	EX	M	WB

Table 9: LOADIMM Hazard.

Table 9 shows the LOADIMM hazard. The hazard occurs when any other instruction arrives after the LOADIMM instruction. In clock 2, R7's value is retrieved and the upper bits are updated to the value 9. In clock 3, R7's value is forwarded from execute stage (instruction 1) to decode stage (instruction 2). At the same time in clock 3, the lower bits of R7 are updated to the value 4. The muxes in the decode stage handle the forwarding. In clock 4, the correct R7 value is forwarded again from execute stage (instruction 2) to decode stage (instruction 3). The forwarding resolves all hazards.

	Clock Cycle Number						
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***
MOV R2, R7	IF	D	**EX**	M	WB		
ADD R1, R2, R3		IF	**D**	EX	M	WB	

Table 10: MOV Hazard.

Table 10 shows the MOV hazard. The hazard occurs when any other instruction arrives after the MOV instruction. At clock 2, R2's value is retrieved. At clock 3, the R2's value is forwarded from execute stage (instruction 1) to decode stage (instruction 2). At clock 4, R2's value is available to the ALU.

	Clock Cycle Number							
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
ADD R1, R4, R0	IF	D	EX	**M**	**WB**			
ADD R0, R1, R3		IF	D	**EX**	M	WB		
TEST R0			IF	D	**EX**	M	WB	
BR.Z R1, #6				IF	**D**	EX	M	WB

Table 11: TEST Branch Instruction Hazard.

Table 11 shows the TEST branch instruction hazard. The hazard occurs when a combination of ALU instructions, TEST instructions, and branch instructions happen in-succession.

5.3 Load and Store Bypassing

If a load/store operation occurs after an ALU instruction, load and store bypassing is done. At the decode stage of the store instruction, the source register value is captured and forwarded. The destination register value is captured in a special FIFO when ALU instruction passes through the memory access stage. At the memory access stage of the store instruction, the correct values for the destination and source register are available. FIFO value retrieval only requires three searches in parallel. The search time is only a few nanoseconds.

	Clock Cycle Number						
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***
ADD R1, R4, R0	IF	D	EX	**M**	WB		
ADD R0, R1, R3		IF	D	EX	**M**	WB	
STORE R1, R0			IF	**D**	EX	**M**	WB

Table 12: Store After ALU Instruction Hazard.

Table 12 shows store after ALU instruction hazard. At clock 3, R1's value is computed by the ALU. At clock 4, decode stage of the store instruction captured R1's value. At the same time in clock 4, R0's value is computed. In clock 5, R0's value captured by a special FIFO in the memory access stage. At clock 6, R0's value and R1's value are available to the memory access stage. This allows the store function to store R1's value at the memory location of R0's value. A similar hazard mitigation procedure happens for the load after ALU hazard.

The second hazard is a store instruction after a load instruction. If the load instruction loads a value into a register and that register contains the value for write memory location in the store instruction then a hazard occurs. The hazard is mitigated when the loaded value is stored in a special FIFO in the memory access stage. At the memory access stage of the store instruction, the value is retrieved from the special FIFO.

	Clock Cycle Number						
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***
LOAD R1, R0	IF	D	EX	**M**	WB		
STORE R4, R1		IF	D	EX	**M**	WB	

Table 13: Store After Load Hazard.

Table 13 demonstrates an example of the store after load hazard. At clock 4, the loaded R1 value is written into a special FIFO. At the clock 5, R1's value is retrieved from the special FIFO. Then R4's value is stored at the memory location of R1's value.

5.4 Multiplication

Since 16-bit multiplication yields 32-bit resultants, the upper 16-bits require management. Multiplication is handled using an additional instruction, modification to the ALU, and a special register.

The modified ALU employs a temporary 32-bit register to store resultants. From this register, the output is split into the upper and lower halves. The lower half (15:0) are output as results and a non-zero upper half (31:16) is stored into the multiply register located in the Write Back stage. Using an available 35 opcode, a new instruction referred to as "MOV_Multi" is specified in Table 14. The purpose of the additional instruction is to enable programmer access and control of the full multiplication resultant. The MOV_Multi instruction allows the programmer to move the upper half of the multiplication resultant to any register. The design approach was used instead of defaulting storage to the link register R7 in favor of less hazard mitigation and programmer control. The MOV_Multi instruction functionality is shown in Figure 20.

OP-CODE = 35								**Destination R[ra]**								
15	14	13	12	11	10	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**X**	**X**	**X**							

Table 14: MOV_Multi Instruction Specification.

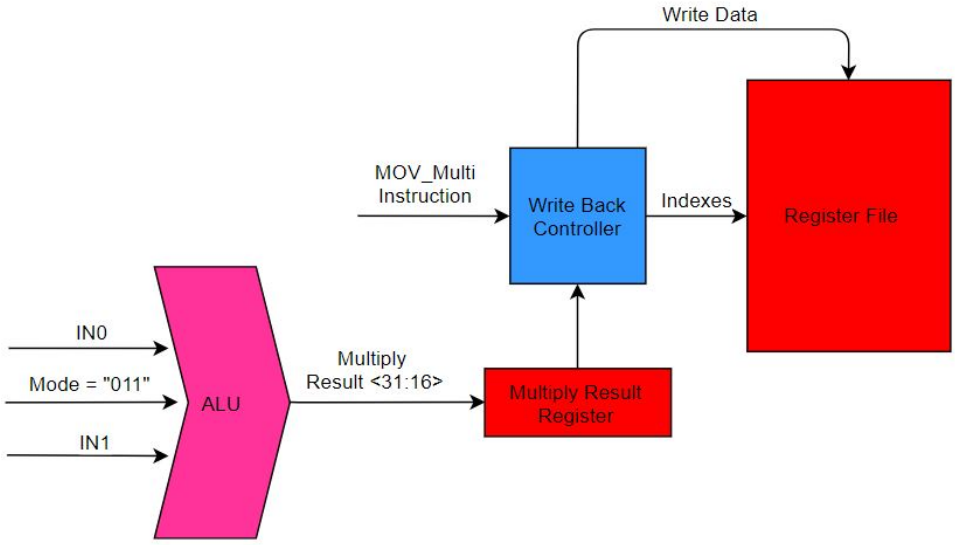


Figure 20: MOV_Multi Instruction Diagram.

6 Performance

6.1 Testing Methodology

Final Tests 1-3 ROM files provided by Lab TA Ibrahim Hazmi will be used as the CPU test code. Additionally, Ibrahim's 7-Segment Display Controller will be used to display CPU output data in hexadecimal format. The 7-segment display is integrated into the FPGA evaluation board. CPU input data is controlled via the mapped FPGA evaluation board SPST switches. A function generator outputting a 3.3V square wave with variable frequency is used as the CPU clock signal. The CPU reset signal is also mapped to an FPGA evaluation board push switch. The Testing system schematic is illustrated in Figure 21.

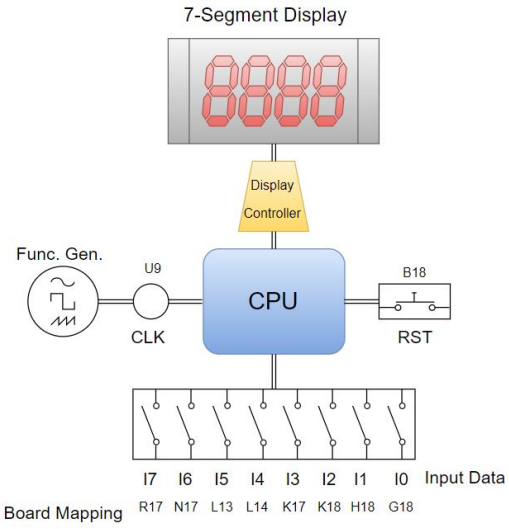


Figure 21: Testing System Schematic.

6.2 Results

The following sections detail the benchmarking results of the final CPU design.

6.2.1 Critical Path

The critical path is the execution stage of the pipeline. The execution stage contains the ALU, 4 muxes, forwarding and a direct path from the register file. The multiplier is the slowest component of the ALU. Multiplier uses the DSP48A block without any pipelining and zero stalls. The DSP48A multiplier mode maximum delay is 6 ns. The maximum delay for the 2 layers of multiplexer is 3 ns. The read period from the register file is 4 ns. Therefore, the maximum delay is 13 ns. As a result, the period of the execute stage correlates with the maximum frequency of 78 MHz.

6.2.2 CPI

The CPI for each of the test programs was calculated using the required instructions to execute each program and timing analysis from the Xilinx Integrated Synthesis Environment (ISE) to determine the number of elapsed clock cycles.

$$CPI = \frac{Elapsed\,Clock\,Cycles}{Required\,Instructions}$$

Given an input of 7, Test 1 program requires 72 instructions to complete. Using timing analysis from the Xilinx ISE, we were able to calculate the number of elapsed clock cycles required to produce the correct resultant. The equation below demonstrate the calculated CPI of 1.33. The same methodology was applied to the calculations for Tests 2 and 3.

$$Test\,1\,CPI = \frac{96}{72} = 1.57$$
$$Test\,2\,CPI = \frac{110}{70} = 1.58$$
$$Test\,3\,CPI = \frac{111}{60} = 1.85$$

6.2.3 Clock Rate

As the maximum clock rate of the CPU exceeds the input buffer of 66 MHz (due to slew rate limitations), an internal Phase-Locked Loop (PLL) was used. The internal PLL increased up the 50 MHz on-board clock to 78 MHz. The 78 MHz clock was used to drive the CPU. 78 MHz is the maximum clock rate of the CPU. Aggressive optimizations was done by the Xilinx ISE to increase the clock rate. All the tests were done on the CPU at 78 MHz. The tests used the LEDs for output instead of the 7-segment display. The 10 segment display has issues displaying digits at high frequencies.

6.3 Simulations

Test 1 simulations results were successful as the correct output for the difference of squares was produced (Input = 7, Output = 0x00F = 15).

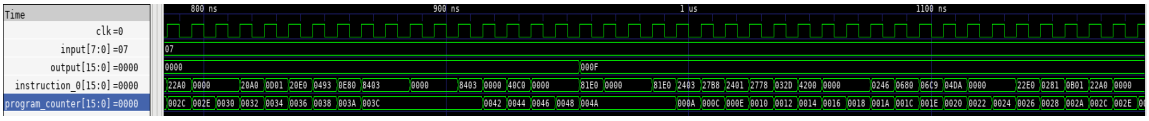


Figure 22: Test 1 Simulation Results.

Tests 2 and 3 simulations results were also successful as the correct output of the factorial was produced (Input = 7, Output = 0x13B0 = 5040).

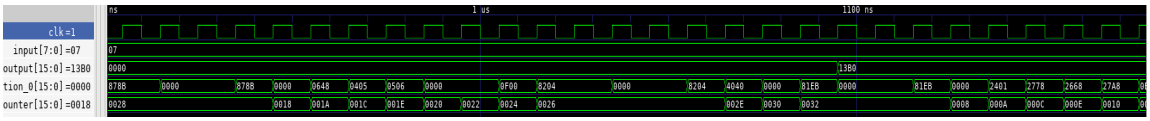


Figure 23: Test 2 Simulation Results.

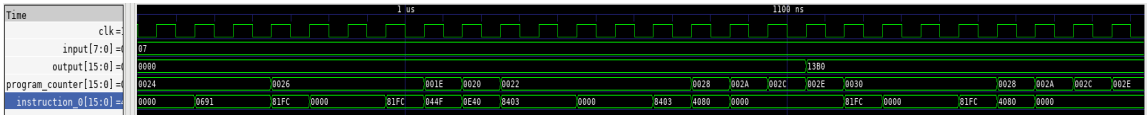


Figure 24: Test 3 Simulation Results.

7 Observations

The following section details some important observations that have manifested from the design and testing process of the CPU.

7.1 Branch Handling

The decision to implement the stalling mechanism for branch handling was made due to its simplicity and the team's inexperience and lack of knowledge regarding branch prediction techniques (taught later in course). The stalling mechanism results in 3 cycle penalties for normal branching operations and 4 cycles for conditional branches. The CPU simulations demonstrated this performance penalty by means of CPI. If our simulations were to have been conducted using programs with more loop iterations, the CPI would increase. In retrospect, branch prediction hardware would alleviate this issue.

7.2 Hazard Management

The Hazard Detection Units employed at Decode, Execute, and Memory Access pipeline stages are a good solution for conducting hazard management. Despite this, during our final design review presentation, Dr. Dimopolous aptly brought it to the Team's attention that the Hazard management arrays are redundant since all the information they store is inherently available at each pipeline stage latch. This additional hardware manifests in more power consumption and perhaps complexity. In a hypothetical design revision this issue would be addressed and the pipeline latches would serve as a means to access instruction data necessary to perform hazard management.

7.3 Tests 2 and 3 Simulation Discrepancy

Test programs 2 and 3 both output a factorial of the input but differ in the number of NOPs inserted. Test 3 is intended to test the capabilities of the CPU hazard management. It is assumed that the Test 3 program should run faster than Test 2 given the lower amount of NOPs. Despite this, we found that Test 2 and 3 ran nearly identically in terms of elapsed clock cycles. This surprising result can be attributed to the branch handling technique.

8 Conclusion

Designing a pipelined 16-bit CPU provides immense challenges and critical experience that is perhaps unparalleled in computer architecture design. The final CPU realized all preliminary design goals and was functional and successful in all performance metrics. The design was successful because of the early start to the project and the intelligent design choices made for the implementation of each pipeline stage, the Control Unit, and the hazard management. The employed branch handling technique resulted in significant performance penalties in certain circumstances. Using predictive or assumption based branch handling hardware would improve performance and eliminate the aforementioned penalties. Despite the excellent functionality of the hazard management system, further improvements could be made. The redundancy of the Hazard Management Arrays could be resolved by employing a method of acquiring the same instruction information for the pipeline stage latches.

9 Recommendation

It is recommended that this CPU design, although sufficient for the purposes of this lab project, is not used in a commercial setting. The CPU requires some significant design changes, particularly in branch handling and hazard management. Despite this, the final design of the CPU is robust and among the best made in the lab section of CENG 450.

10 Contributions

The following briefly overviews the group member project contributions:

1. Pipeline Design:
    - Early Pipeline Design and Implementation - Brosnan and Morgan
    - Final Pipeline Design and Implementation - Brosnan
2. Preliminary Design Review Presentation - Brosnan and Morgan
3. Final Design Review Presentation - Brosnan and Morgan
4. Final Report - Brosnan and Morgan

11 Appendices

VHDL code attached with code sections labelled accordingly.