

# Computer Systems Design Final Project

## Single-Cycle MIPS Processor

---

Anish Goyal

Dr. Mohammad Ahad  
Associate Professor

December 5, 2025



### Abstract

I designed and implemented a single-cycle MIPS processor using VHDL and Intel Quartus Prime. The processor supports R-type instructions (add, sub) and I-type instructions (lw, sw). I constructed the datapath using schematic capture in Quartus, connecting individual VHDL components including the program counter, instruction memory, register file, ALU, ALU controller, main control unit, data memory, multiplexers, and sign extension unit. I validated the processor through functional simulation in Questa/ModelSim, verifying correct instruction execution across four test instructions. The simulation results confirm proper ALU operations, register reads and writes, and memory access behavior.

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Equipment</b>	<b>7</b>
<b>3</b>	<b>Methods</b>	<b>8</b>
<b>4</b>	<b>Results</b>	<b>9</b>
4.1	Program Counter . . . . .	9
4.2	PC Adder . . . . .	11
4.3	Instruction Memory . . . . .	12
4.4	Main Control Unit . . . . .	13
4.5	Register File . . . . .	15
4.6	Sign Extension Unit . . . . .	17
4.7	ALU . . . . .	19
4.8	ALU Controller . . . . .	21
4.9	Data Memory . . . . .	23
4.10	32-bit 2-to-1 Multiplexer . . . . .	25
4.11	5-bit 2-to-1 Multiplexer . . . . .	27
4.12	Shift Left 2 Unit . . . . .	29
4.13	Branch Adder . . . . .	30
4.14	Complete Datapath . . . . .	31
<b>5</b>	<b>Discussion</b>	<b>32</b>
<b>6</b>	<b>Problems Encountered During Implementation</b>	<b>34</b>
6.1	Bus Naming Convention Issues . . . . .	34
6.2	Quartus Schematic Editor Reliability . . . . .	34
6.3	Input and Output Pin Naming . . . . .	35
6.4	Constant Value Generation . . . . .	35
6.5	Functional Simulation Configuration . . . . .	35
6.6	XOR and Subtract Opcode Conflict . . . . .	36
6.7	Questa License Server Configuration . . . . .	36
6.8	MemToReg Multiplexer Polarity . . . . .	36
6.9	5-bit Mux Creation Necessity . . . . .	37
<b>7</b>	<b>Potential Future Improvements</b>	<b>38</b>
7.1	Clock Timing Refinement . . . . .	38
7.2	Five-Stage Pipeline Implementation . . . . .	38
7.3	Jump Instruction Support . . . . .	38
7.4	Real-Time Output Display . . . . .	39
7.5	Live Instruction Input . . . . .	39

7.6	Additional ALU Operations . . . . .	40
7.7	Data Forwarding Implementation . . . . .	40
7.8	FPGA Hardware Deployment . . . . .	40
7.9	Enhanced Debugging Capabilities . . . . .	41
<b>8</b>	<b>Conclusion</b>	<b>42</b>
<b>9</b>	<b>References</b>	<b>44</b>
<b>10</b>	<b>Appendix</b>	<b>45</b>
10.1	Video Walkthrough . . . . .	45
10.2	VHDL Source Code . . . . .	45

## List of Figures

1	Single-cycle MIPS datapath from Patterson and Hennessy textbook . . . . .	5
2	Program Counter component in Quartus . . . . .	9
3	PC Adder component in Quartus . . . . .	11
4	Instruction Memory component in Quartus . . . . .	12
5	Main Control Unit component in Quartus . . . . .	13
6	Register File component in Quartus . . . . .	15
7	Sign Extension Unit component in Quartus . . . . .	17
8	ALU component in Quartus . . . . .	19
9	ALU Controller component in Quartus . . . . .	21
10	Data Memory component in Quartus . . . . .	23
11	32-bit 2-to-1 Multiplexer component in Quartus . . . . .	25
12	5-bit 2-to-1 Multiplexer component in Quartus . . . . .	27
13	Shift Left 2 Unit component in Quartus . . . . .	29
14	Branch Adder component in Quartus . . . . .	30
15	Complete Single-Cycle MIPS Processor Block Schematic . . . . .	31
16	Functional Simulation Waveform from Questa . . . . .	32

## List of Tables

1	ALU Control Signal Generation . . . . .	22
2	Simulation Output Values at Rising Clock Edges . . . . .	32

## List of Listings

1	Program Counter (PC.vhd) . . . . .	45
2	Adder (Adder.vhd) . . . . .	46
3	Instruction Memory (InstructionMem.vhd) . . . . .	47
4	Main Control Unit (MainControl.vhd) . . . . .	48
5	Register File (RegisterFile.vhd) . . . . .	50
6	Sign Extension Unit (Sign_Ext.vhd) . . . . .	52
7	ALU (ALU.vhd) . . . . .	53
8	ALU Controller (ALU_Controller.vhd) . . . . .	56
9	Data Memory (DMem.vhd) . . . . .	57
10	32-bit 2-to-1 Multiplexer (Mux_2_1.vhd) . . . . .	59
11	5-bit 2-to-1 Multiplexer (Mux_2_1_5b.vhd) . . . . .	60
12	Shift Left 2 Unit (ShiftL2.vhd) . . . . .	61

# 1 Introduction

The MIPS (Microprocessor without Interlocked Pipeline Stages) architecture represents one of the most influential reduced instruction set computing (RISC) designs in computer engineering history. I undertook this project to gain practical experience implementing a processor datapath and control unit in hardware description language. The single-cycle processor architecture executes each instruction in exactly one clock cycle, making it an ideal starting point for understanding how hardware components work together to fetch, decode, and execute machine instructions.

In this project, I implemented a single-cycle MIPS processor using VHDL and Intel Quartus Prime. The processor supports a subset of the MIPS instruction set, including R-type arithmetic operations (add, sub) and I-type memory operations (lw, sw). I based my design on the classic MIPS datapath described by Patterson and Hennessy in their textbook [1]. Figure 1 shows the reference datapath that guided my implementation.

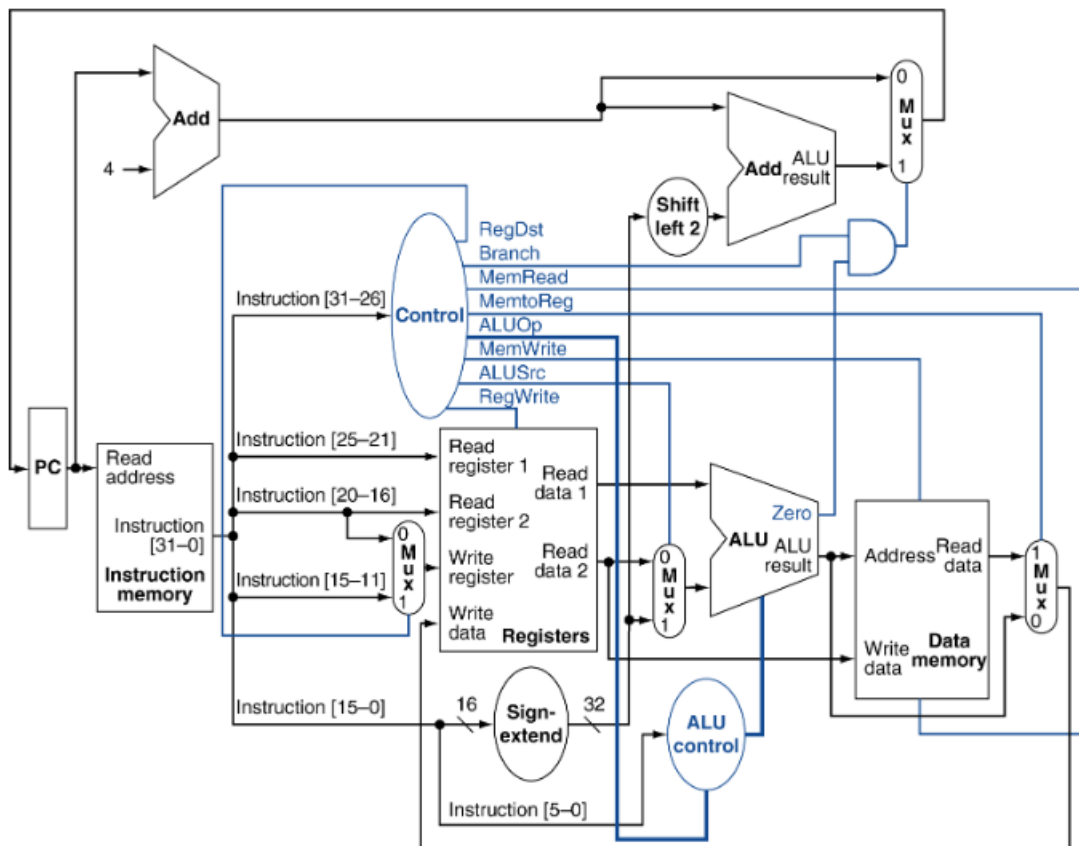


Figure 1: Single-cycle MIPS datapath from Patterson and Hennessy textbook

The datapath consists of several key components that work together during instruction execu-

tion. The program counter (PC) holds the address of the current instruction. The instruction memory stores the program and outputs the 32-bit instruction at the PC address. The control unit decodes the instruction opcode and generates control signals. The register file provides two read ports and one write port for accessing the 32 general-purpose registers. The ALU performs arithmetic and logic operations. The data memory handles load and store operations. Multiplexers route data between components based on the instruction type.

My goal was to implement each of these components as individual VHDL modules, then connect them using the Quartus schematic editor to create a complete working processor. I validated the design through functional simulation, testing a sequence of four instructions that exercise both R-type and I-type operations.

## 2 Equipment

I used the following tools and equipment for this project. Intel Quartus Prime Lite Edition version 23.1.1 served as the primary development environment for writing VHDL code and creating the block diagram schematic. I used Questa (ModelSim) for functional simulation and waveform analysis. The target platform was the DE10-Standard development board featuring the Cyclone V 5CSXFC6D6F31C6 FPGA, though I focused on simulation rather than hardware deployment for this project.

### 3 Methods

I followed a modular design approach based on the EENG 5342 course materials [2]. I began by identifying all the components required for the single-cycle datapath. I then implemented each component as a separate VHDL entity with its own architecture. This modular approach allowed me to test individual components before integration.

For each component, I defined the input and output ports based on the datapath diagram. I wrote the behavioral VHDL code to implement the required functionality. I compiled each module in Quartus to verify correct syntax and generate the schematic symbol.

After completing all components, I created a new Block Diagram File (.bdf) in Quartus. I instantiated each VHDL symbol and connected them according to the datapath architecture. I paid careful attention to bus widths and signal naming conventions to ensure proper connectivity.

For testing, I initialized the instruction memory with four test instructions. I initialized the register file with values 0 through 31 in registers \$0 through \$31. I ran functional simulation in Questa with a 1 microsecond clock period. I examined the waveforms to verify correct operation of each instruction.

## 4 Results

I successfully implemented all components of the single-cycle MIPS processor. This section describes each component, its role in the datapath, and the corresponding VHDL implementation.

### 4.1 Program Counter

The program counter serves as the fundamental sequencing element of the processor. It maintains the address of the instruction currently being executed and determines the flow of program execution. Figure 2 shows the PC component as implemented in Quartus. The complete VHDL implementation appears in Listing 1 in the Appendix.

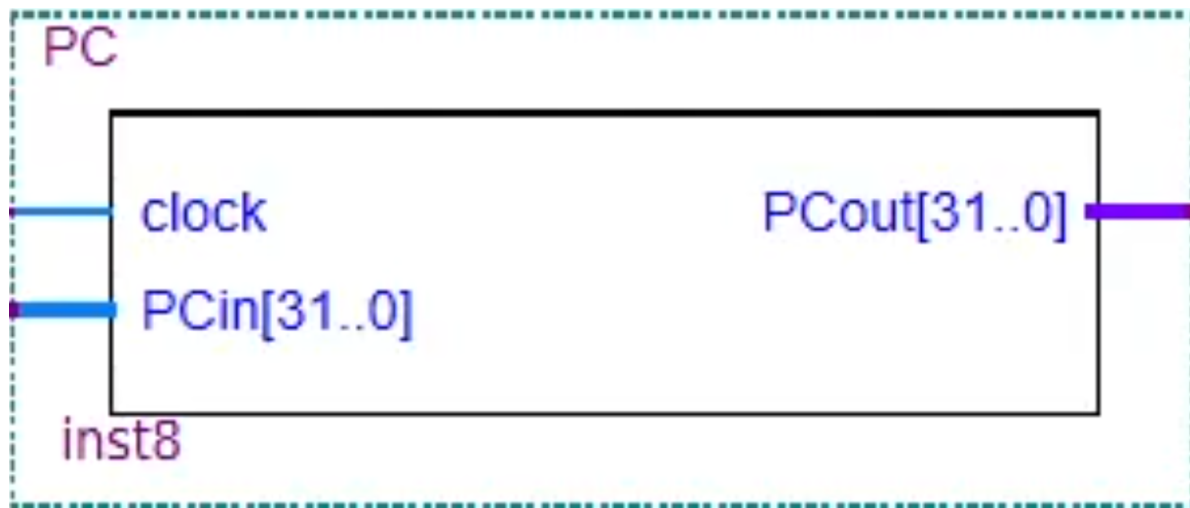


Figure 2: Program Counter component in Quartus

The PC has a 32-bit input port (PCin) that receives the next address value from the PC multiplexer, and a 32-bit output port (PCout) that provides the current address to the instruction memory. The update occurs synchronously on the rising edge of the clock signal, ensuring that address changes happen at predictable times aligned with the processor clock.

The PC implements a simple D flip-flop behavior at the 32-bit level. When the clock transitions from low to high, the PC latches the value present on PCin and immediately propagates it to PCout. This synchronous operation ensures that all downstream components receive a stable address throughout the clock cycle. The PC has no reset signal in my implementation, meaning it retains whatever value is present at power-on. For a production processor, a reset capability would be essential to ensure deterministic startup behavior.

In the single-cycle architecture, the PC updates exactly once per instruction. The new PC value comes from one of two sources selected by a multiplexer: PC+4 for sequential execution, or a branch target address for taken branches. My implementation only supports sequential execution since I did not implement branch instructions in the final design, though the datapath includes the necessary branch calculation hardware.

## 4.2 PC Adder

The PC adder is a dedicated 32-bit adder that computes the address of the next sequential instruction. Figure 3 shows this component as it appears in the Quartus schematic. The VHDL implementation uses a simple combinational addition operation as shown in Listing 2.

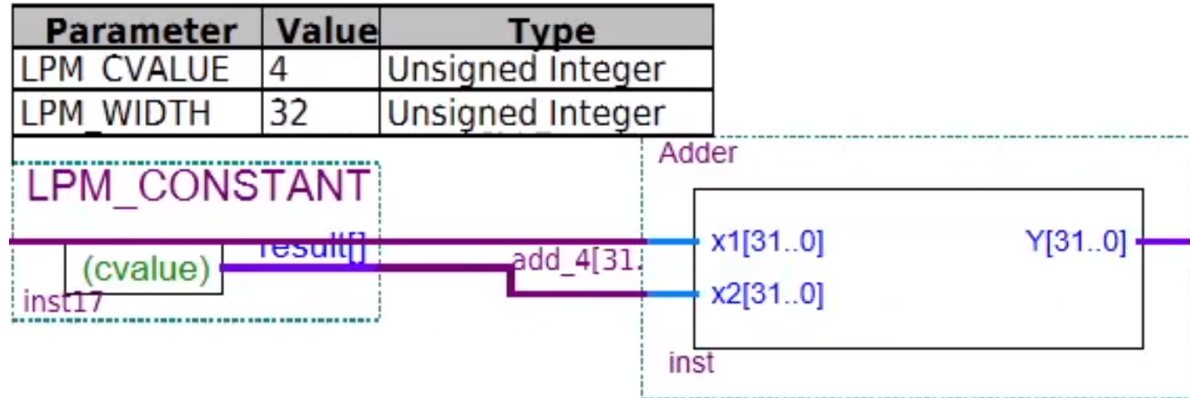


Figure 3: PC Adder component in Quartus

In the MIPS architecture, all instructions are exactly 32 bits (4 bytes) wide and must be word-aligned in memory. This means instruction addresses are always multiples of 4. The PC adder takes the current PC value as one input and the constant value 4 as the other input, producing PC+4 as output. This output represents the address of the next sequential instruction in memory.

The adder operates combinatorially, meaning the output changes immediately whenever the input changes, with only the propagation delay of the adder logic. I implemented the adder using the built-in addition operator in VHDL, which synthesizes to a ripple-carry adder structure. For a 32-bit addition, this introduces some propagation delay as the carry ripples through all 32 bit positions. In a higher-performance processor, a carry-lookahead or carry-select adder might be used to reduce this critical path delay.

The PC+4 value feeds into a multiplexer that selects between sequential execution and branch targets. In my implementation, since I did not fully implement branch logic, the PC+4 value always becomes the next PC value, creating simple sequential program flow.

### 4.3 Instruction Memory

The instruction memory (IMEM) stores the program code and provides read access to instructions based on the address from the PC. Figure 4 shows the instruction memory component in the Quartus schematic. The complete VHDL implementation appears in Listing 3.

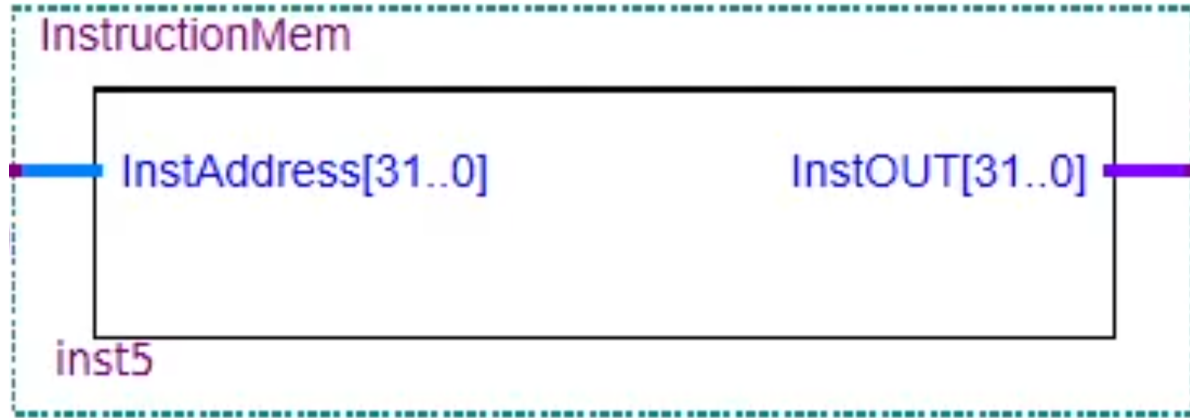


Figure 4: Instruction Memory component in Quartus

I implemented the instruction memory as a read-only memory (ROM) using a VHDL array type. The memory contains 4 words, each 32 bits wide, holding the test program. The memory operates combinatorially rather than synchronously. This means the output instruction updates immediately when the address changes, with only combinational propagation delay. This is critical for single-cycle operation because the instruction must be available within the same clock cycle that the PC is updated.

The address input is a full 32-bit value, but since each instruction occupies 4 bytes, I divide the address by 4 to obtain the word index into the array. In VHDL, I use the expression `to_integer(unsigned(InstAddress) / 4)` to perform this conversion. This means address 0 accesses instruction 0, address 4 accesses instruction 1, address 8 accesses instruction 2, and so on. This matches the MIPS convention where instructions are word-aligned.

I preloaded four specific test instructions into the memory during synthesis: 0x012A4020 (add \$8, \$9, \$10), 0x01285022 (sub \$10, \$9, \$8), 0x8E510000 (lw \$17, 0(\$18)), and 0xAE290004 (sw \$9, 4(\$17)). These instructions test both R-type arithmetic operations and I-type memory operations. In a real processor, the instruction memory would be much larger and would typically be loaded from external storage at boot time. For FPGA implementation, the memory could be initialized from a .mif file, allowing program changes without resynthesizing the design.

## 4.4 Main Control Unit

The main control unit serves as the brain of the processor, decoding instructions and orchestrating the operation of all datapath components. It examines the 6-bit opcode field (bits 31-26) of each instruction and generates the appropriate control signals. Figure 5 shows the main control unit in the Quartus schematic. The VHDL implementation appears in Listing 4.

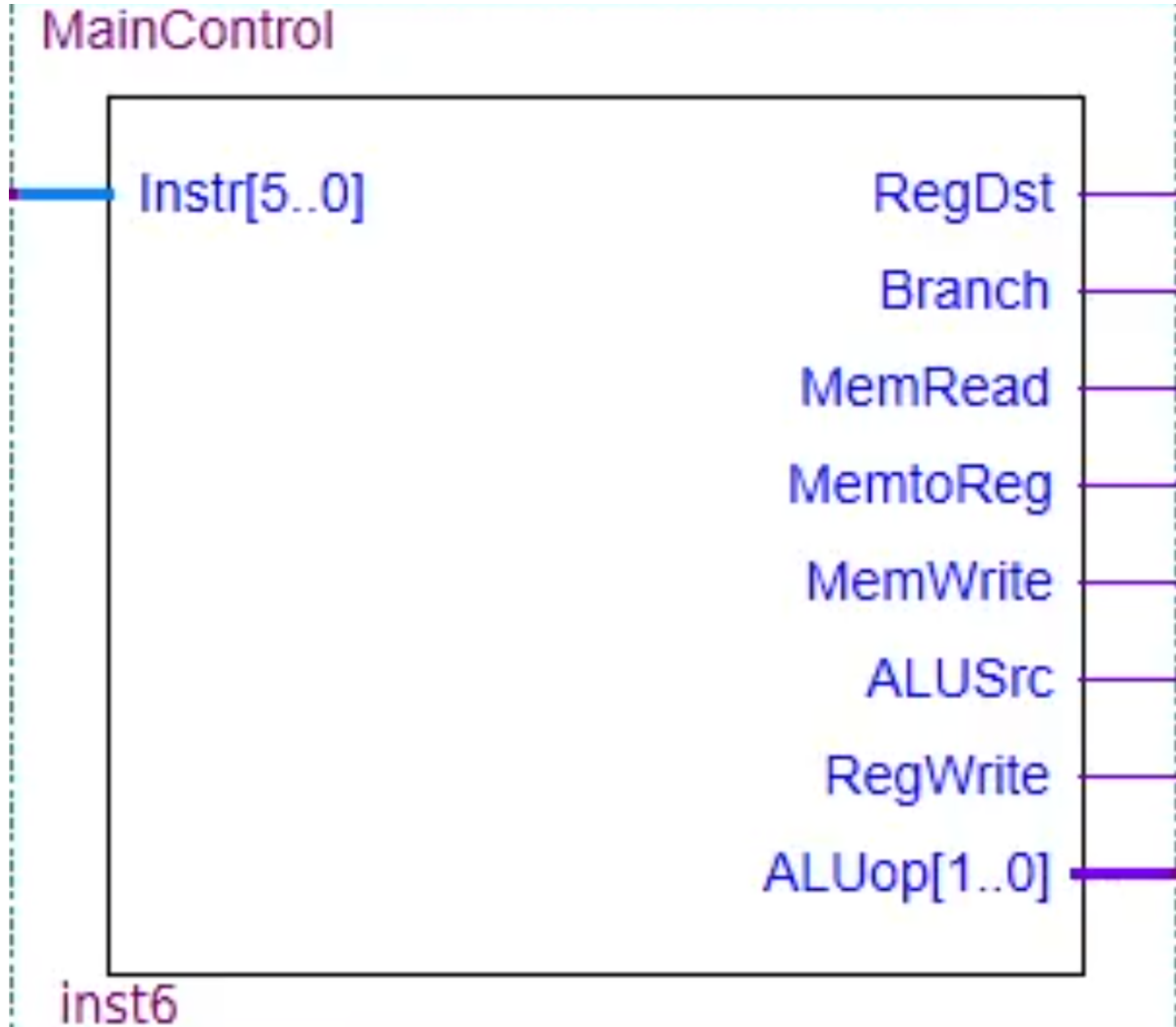


Figure 5: Main Control Unit component in Quartus

The control unit outputs eight control signals that govern datapath behavior. **RegDst** selects which instruction field specifies the destination register: for R-type instructions, this is the *rd* field (bits 15-11), while for I-type instructions, it is the *rt* field (bits 20-16). **ALUSrc** selects

the second operand for the ALU: either the second register value (for R-type) or the sign-extended immediate (for I-type). MemtoReg selects the data to write back to the register file: either the ALU result (for arithmetic instructions) or memory read data (for load instructions). RegWrite enables writing to the register file at the end of the instruction cycle. MemRead and MemWrite enable reading from and writing to data memory, respectively. Branch combines with the ALU Zero flag to enable branch execution. ALUOp provides a 2-bit code that tells the ALU controller how to interpret the instruction.

For R-type instructions (opcode 000000), the control unit asserts RegDst to select the rd field and sets RegWrite to enable writing the ALU result. ALUOp is set to 10, indicating that the ALU controller should examine the function field to determine the specific operation. The remaining control signals are deasserted since R-type instructions do not access memory or perform branches.

For load word instructions (opcode 100011), the control unit asserts ALUSrc to route the sign-extended immediate to the ALU for address calculation. It sets ALUOp to 00, which tells the ALU controller to perform addition regardless of the function field. MemRead is asserted to read from data memory, and MemtoReg is set to route the memory data (rather than ALU result) to the register file. RegWrite is asserted with a 10ns delay to ensure the memory read completes before the write occurs.

For store word instructions (opcode 101011), the configuration is similar to load word, but MemWrite is asserted instead of MemRead, and RegWrite is deasserted since store instructions do not write to registers. The ALU still performs address calculation by adding the base register and immediate offset.

For branch equal instructions (opcode 000100), ALUOp is set to 01 to force a subtraction operation, which sets the Zero flag when the two register values are equal. The Branch signal is asserted to enable the branch address multiplexer. However, I did not fully implement the branch logic in my final processor, so branch instructions do not execute correctly.

The control unit operates purely combinatorially, generating all control signals based solely on the opcode. This allows control signals to be available within the same clock cycle as the instruction fetch, which is essential for single-cycle operation. The trade-off is that every instruction takes a full clock cycle even if it could complete faster, and the clock period must be long enough to accommodate the slowest possible instruction path.

## 4.5 Register File

The register file forms the primary storage for operands and results during computation. It implements 32 general-purpose registers following the MIPS register architecture, with each register holding 32 bits. The register file provides two independent read ports and one write port, allowing a single instruction to read two source operands and write one result simultaneously. Figure 6 shows the register file component in the Quartus schematic. The VHDL implementation appears in Listing 5.

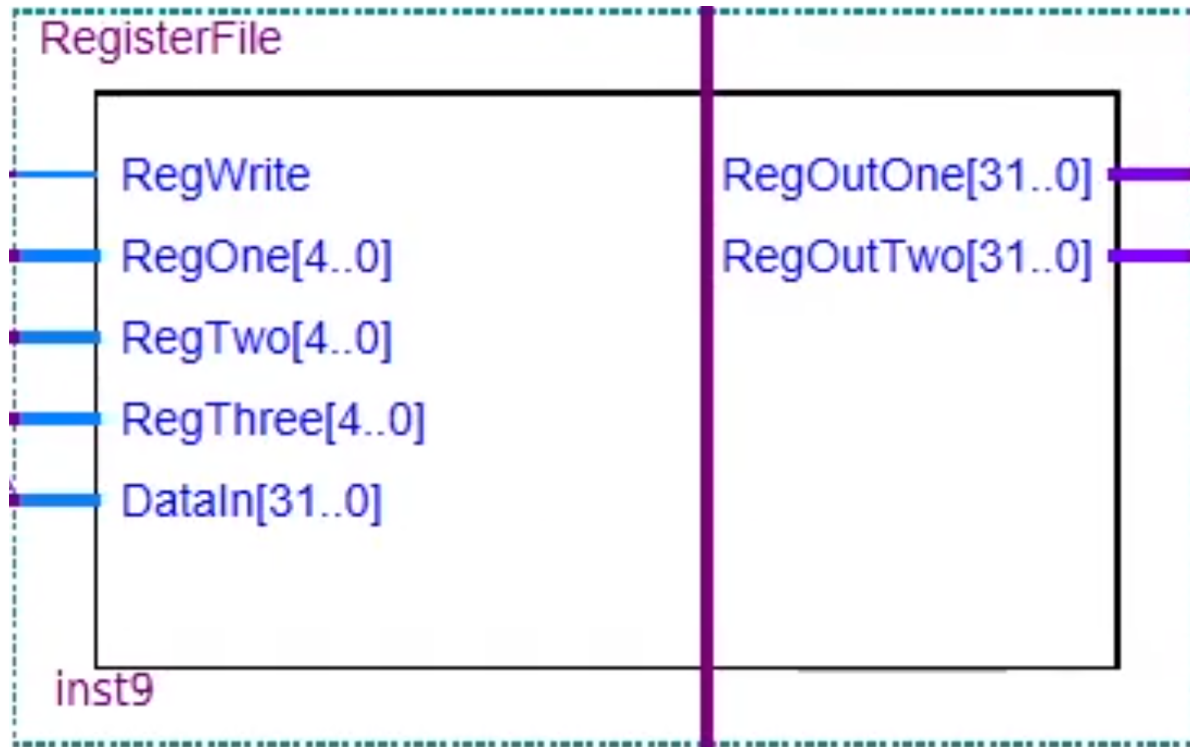


Figure 6: Register File component in Quartus

The register file receives five inputs: three 5-bit register addresses and two data signals. `RegOne` and `RegTwo` specify the registers to read from ports 1 and 2, respectively. `RegThree` specifies the destination register for write operations. `DataIn` carries the 32-bit value to write. `RegWrite` serves as a write enable signal. The register file produces two 32-bit outputs: `RegOutOne` and `RegOutTwo`, corresponding to the values stored in the registers addressed by `RegOne` and `RegTwo`.

I implemented the register file using a VHDL array type containing 32 elements of 32-bit vectors. The read operations are purely combinational. When `RegOne` or `RegTwo` changes, the corresponding output updates immediately through a continuous assignment statement. This is implemented using the VHDL expressions `RegOutOne <=`

`myarray(TO_INTEGER(UNSIGNED(RegOne)))` and `RegOutTwo <= myarray(TO_INTEGER(UNSIGNED(RegTwo)))`. These statements continuously drive the outputs with the current register values.

The write operation is controlled by the `RegWrite` signal. I implemented this using a process that is sensitive to `RegWrite`. When `RegWrite` transitions to high, the process executes and writes `DataIn` to the register specified by `RegThree`. In a real MIPS processor, register \$0 is hardwired to zero and cannot be written. My implementation does not enforce this constraint, meaning writes to register 0 would overwrite the zero value. A production implementation would add logic to ignore writes when `RegThree` is zero.

For testing and debugging, I initialized all registers with their index values at synthesis time. This means register \$0 contains 0, register \$1 contains 1, register \$2 contains 2, and so forth up to register \$31 containing 31. This initialization pattern makes it easy to verify correct operation during simulation because the register contents are predictable. I can see whether the processor is reading from the correct registers by comparing the outputs to the expected index values.

The register file operates on an unusual clock edge compared to typical synchronous designs. Rather than using the main processor clock directly, the write enable signal `RegWrite` serves as the synchronization point. This works for single-cycle operation but would need modification for a pipelined design where writes must align with specific pipeline stages.

## 4.6 Sign Extension Unit

The sign extension unit converts the 16-bit immediate field from I-type instructions into 32-bit signed values compatible with the rest of the 32-bit datapath. This conversion must preserve the arithmetic sign of the original value. Figure 7 shows the sign extension component in the Quartus schematic. The VHDL implementation appears in Listing 6.

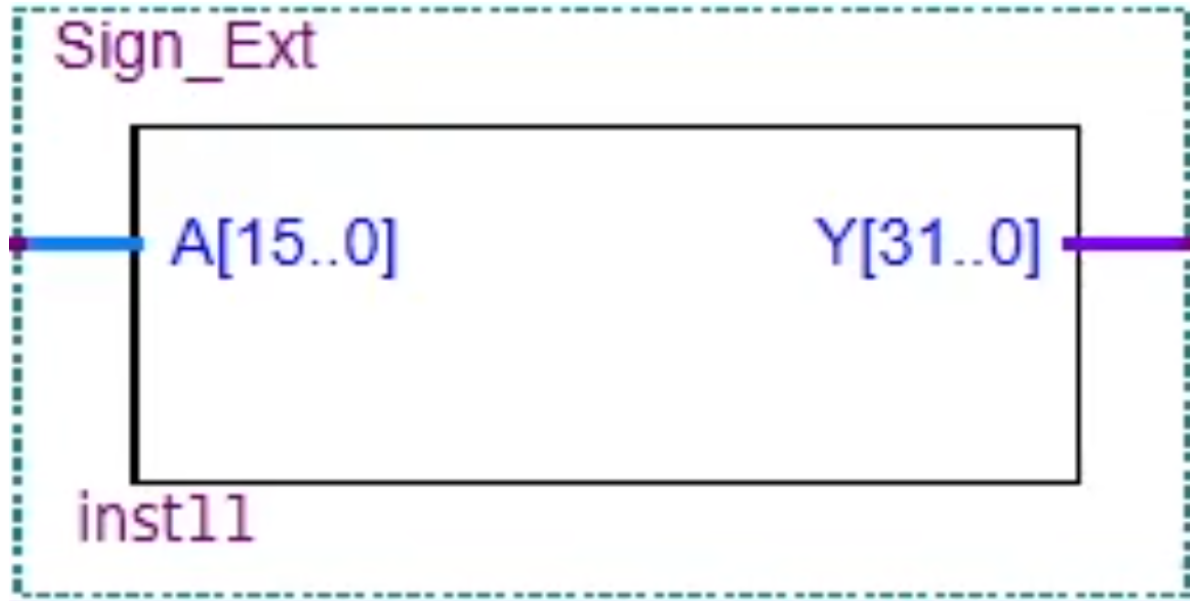


Figure 7: Sign Extension Unit component in Quartus

The unit examines bit 15 of the 16-bit input, which represents the sign bit in two's complement representation. If bit 15 is 0, the value is positive, and the unit creates a 32-bit output by concatenating 16 zeros with the original 16-bit value. If bit 15 is 1, the value is negative, and the unit concatenates 16 ones with the original 16-bit value. This process preserves the two's complement signed value across the width change.

For example, the 16-bit value 0x0004 (decimal 4) has bit 15 equal to 0. Sign extension produces 0x00000004, which correctly represents positive 4 in 32 bits. The 16-bit value 0xFFFFC (decimal -4 in two's complement) has bit 15 equal to 1. Sign extension produces 0xFFFFFFF C, which correctly represents negative 4 in 32-bit two's complement.

The sign extension unit operates purely combinationally with no clock or state. The output updates immediately when the input changes. This is necessary because the extended immediate value must be available within the same cycle for use by the ALU in address calculations or arithmetic operations.

I implemented the logic using a VHDL process with a conditional statement that checks the sign bit and concatenates the appropriate prefix. The VHDL concatenation operator (&)

combines the 16-bit prefix with the original value to form the 32-bit result.

## 4.7 ALU

The arithmetic logic unit (ALU) performs the computational heart of instruction execution. It implements a variety of arithmetic and logical operations required by MIPS instructions. Figure 8 shows the ALU component in the Quartus schematic. The complete VHDL implementation appears in Listing 7.

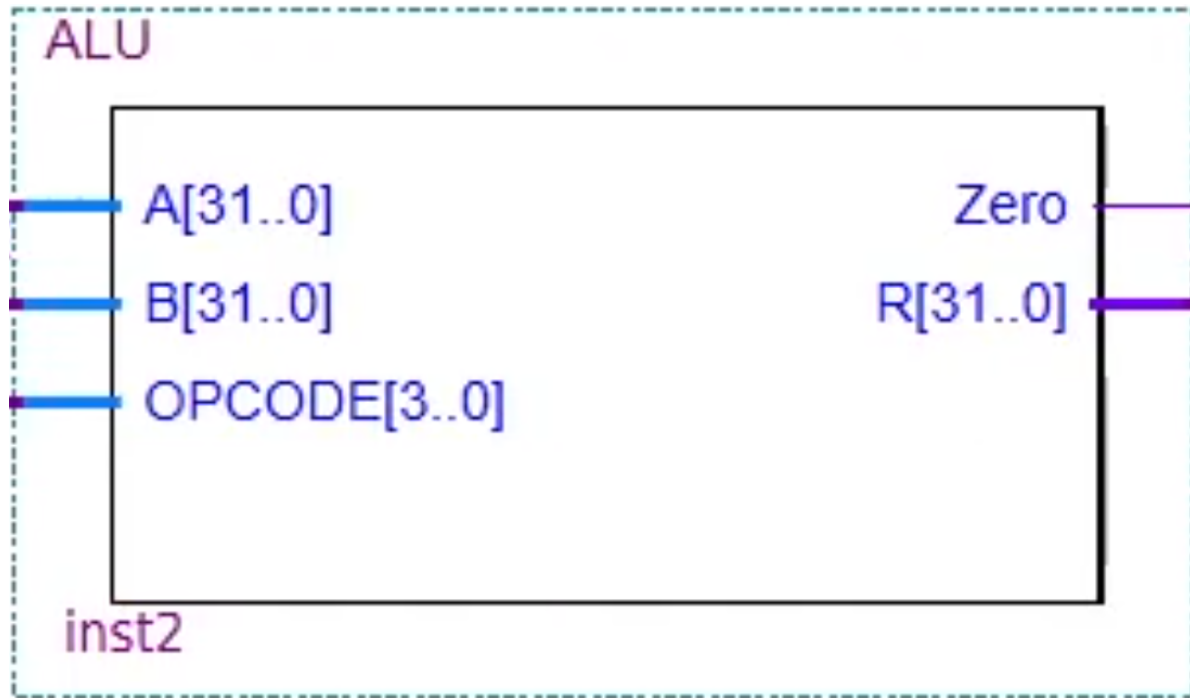


Figure 8: ALU component in Quartus

The ALU receives two 32-bit operands labeled A and B, along with a 4-bit operation code (OPCODE) that selects the operation to perform. It produces two outputs: a 32-bit result (R) containing the computation result, and a single-bit Zero flag that indicates whether the result equals zero. The Zero flag is particularly important for conditional branch instructions, which branch when two values are equal (i.e., their difference is zero).

I implemented the ALU using a large VHDL process with a cascading if-elsif structure that decodes the OPCODE and performs the corresponding operation. The supported operations include: NOT A (0000), NOT B (0001), addition (0010), NAND (0011), OR (0100), NOR (0101), subtraction (0110), XNOR (0111), AND (1000), set-less-than (1001), increment A (1010), decrement A (1011), increment B (1100), decrement B (1101), negate A (1110), and negate B (1111). Not all of these operations are used by the MIPS instructions I implemented, but they were included in the ALU design from the class reference material.

For the addition operation (OPCODE 0010), the ALU computes  $R = A + B$  using VHDL's

built-in addition operator. This operator works on `std_logic_vector` types and performs unsigned binary addition with carry propagation. The result wraps around on overflow without any indication. For the subtraction operation (OPCODE 0110), the ALU computes  $R = A - B$ . I implemented this using VHDL's subtraction operator. The subtraction also uses an internal signal `Rtemp` to hold the result before assigning it to `R`. This allows the conditional logic to examine the result and set the Zero flag to '1' when `Rtemp` equals 0x00000000, otherwise Zero is set to '0'.

The logical operations (AND, OR, NOR, NAND) apply bitwise operations to corresponding bits of `A` and `B`. For example, AND (1000) produces a result where bit `i` is 1 only when both bit `i` of `A` and bit `i` of `B` are 1. The set-less-than operation (1001) performs a signed comparison. If `A` is less than `B` (treating both as signed two's complement numbers), the ALU outputs 0x00000001. Otherwise it outputs 0x00000000. This implements the MIPS `slt` instruction.

One notable issue I encountered during development involved opcode conflicts. The original class code used 0110 for XOR, but the MIPS subtract operation also uses 0110. I resolved this by removing the XOR case and ensuring subtraction worked correctly. This highlights the importance of maintaining consistent opcode mappings between the control unit, ALU controller, and ALU itself.

The ALU operates purely combinationally. When the inputs or opcode change, the output updates after the propagation delay through the combinational logic. For a 32-bit operation, this delay can be substantial, especially for operations like addition and subtraction that have carry chains. The ALU propagation delay forms part of the critical path that determines the minimum clock period for the processor.

## 4.8 ALU Controller

The ALU controller serves as an intermediary between the main control unit and the ALU, translating high-level instruction types into specific ALU operation codes. It combines the 2-bit ALUOp signal from the main control unit with the 6-bit function field (bits 5-0) from R-type instructions to generate the final 4-bit ALU control signal. Figure 9 shows the ALU controller component in the Quartus schematic. The VHDL implementation appears in Listing 8.

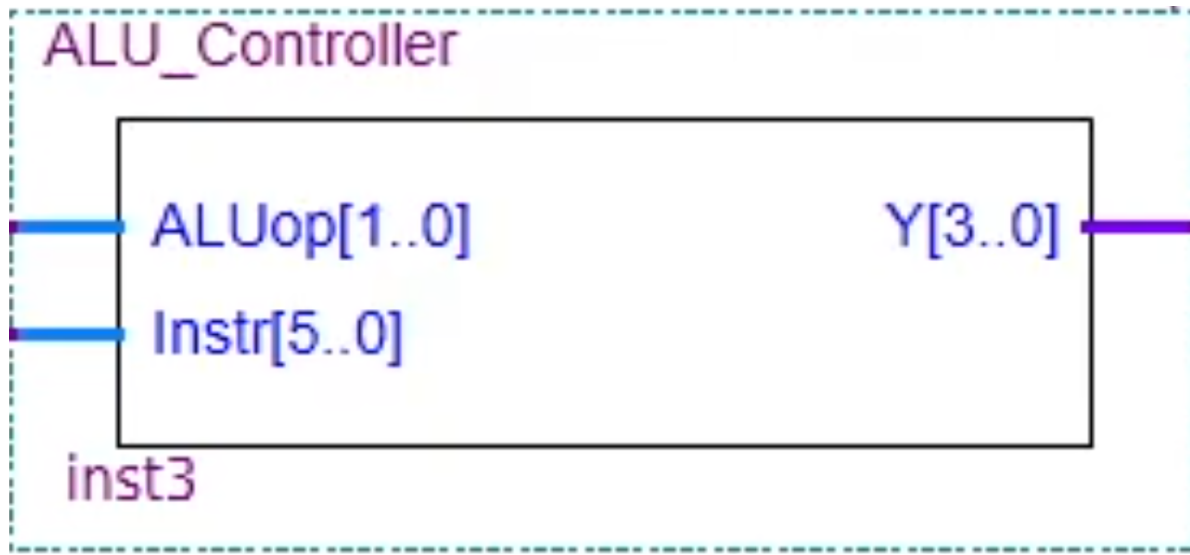


Figure 9: ALU Controller component in Quartus

The ALU controller receives two inputs: ALUOp (2 bits) from the main control unit and Instr (6 bits) representing the function field of the current instruction. It produces a 4-bit output Y that directly controls the ALU operation. The controller implements a two-level decoding scheme that simplifies the main control unit design.

When ALUOp is 00, the controller outputs 0010 (add) regardless of the function field. The main control unit sets ALUOp to 00 for load and store instructions, which need to compute memory addresses by adding a base register and an offset. Since address calculation always requires addition, the function field is ignored.

When ALUOp is 01, the controller outputs 0110 (subtract) regardless of the function field. The main control unit sets ALUOp to 01 for branch equal instructions. To test whether two values are equal, the processor subtracts them and checks if the result is zero. Again, the function field is not meaningful for this instruction type.

When ALUOp is 10, the controller examines the function field to determine the specific R-type operation. Function 100000 (decimal 32) corresponds to add, producing ALU control 0010.

Function 100010 (decimal 34) corresponds to subtract, producing 0110. Function 100100 (decimal 36) corresponds to AND, producing 0000. Function 100101 (decimal 37) corresponds to OR, producing 0001. Function 101010 (decimal 42) corresponds to set-less-than, producing 0111. Note that the ALU control codes were derived from the ALU implementation and do not follow any standard encoding scheme.

This two-level decoding approach separates concerns: the main control unit determines the general category of operation based on the opcode, and the ALU controller refines this into a specific operation based on the function field. For I-type instructions, only the opcode matters. For R-type instructions, both opcode and function field matter. Table 1 summarizes the complete mapping from instruction fields to ALU control signals.

The ALU controller operates combinatorially using nested if-elsif statements in VHDL. The outer level checks ALUOp, and for the R-type case (ALUOp = 10), an inner level checks the function field. The output updates immediately when either input changes.

Table 1: ALU Control Signal Generation

Instruction Opcode	Instruction ALUOp	Operation	Funct Field	Desired ALU Action	ALU Control Input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

## 4.9 Data Memory

The data memory subsystem provides storage for program data accessed through load and store instructions. It implements a small RAM with both read and write capabilities. Figure 10 shows the data memory component in the Quartus schematic. The VHDL implementation appears in Listing 9.

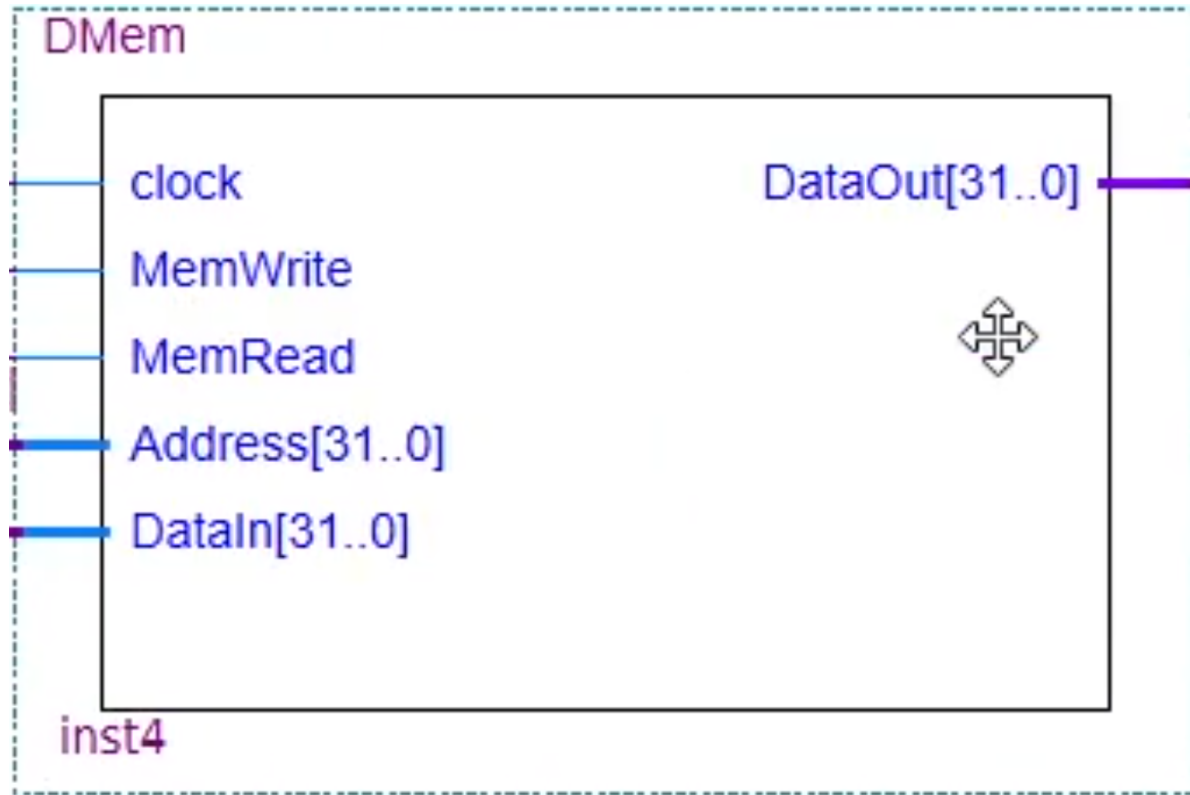


Figure 10: Data Memory component in Quartus

The data memory receives five inputs: a clock signal, MemRead and MemWrite control signals, a 32-bit Address, and a 32-bit DataIn value to write. It produces a single 32-bit DataOut value for read operations. I implemented the memory as a VHDL array containing 32 words of 32 bits each, providing 128 bytes of storage. For testing purposes, I initialized the memory with sequential values (location 0 contains 0, location 1 contains 1, and so on).

The critical design decision for data memory involves timing. I implemented read and write operations on the falling edge of the clock rather than the rising edge. This choice ensures that data memory operations complete within the second half of the clock cycle, after the address has been calculated and stabilized during the first half. For a load instruction, the address calculation happens early in the cycle, and the memory read must complete before the register

write at the end of the cycle. For a store instruction, both the address and data to store must be valid before the falling edge.

When MemWrite is asserted and the clock falls, the memory writes DataIn to the location specified by Address. When MemRead is asserted and the clock falls, the memory outputs the value stored at Address onto DataOut. If neither signal is asserted, the memory performs no operation. The implementation divides the byte address by 4 to obtain the word index, matching the instruction memory approach.

The use of falling edge timing creates a potential hazard: if a load instruction immediately follows a store instruction to the same address, there might be insufficient time for the write to complete before the read begins. In a production processor, this would be handled through forwarding logic or pipeline stalls. My single-cycle implementation does not exhibit this problem because each instruction completes entirely within one cycle.

The limited size of 32 words restricts the programs that can run on this processor. A real MIPS processor would have kilobytes or megabytes of data memory. For FPGA implementation, larger memories would be implemented using on-chip block RAM resources rather than registers, providing better density and performance.

## 4.10 32-bit 2-to-1 Multiplexer

Multiplexers serve as data routing components throughout the datapath, selecting between alternative data sources based on control signals. The 32-bit 2-to-1 multiplexer is used in several locations. Figure 11 shows this component in the Quartus schematic. The VHDL implementation appears in Listing 10.

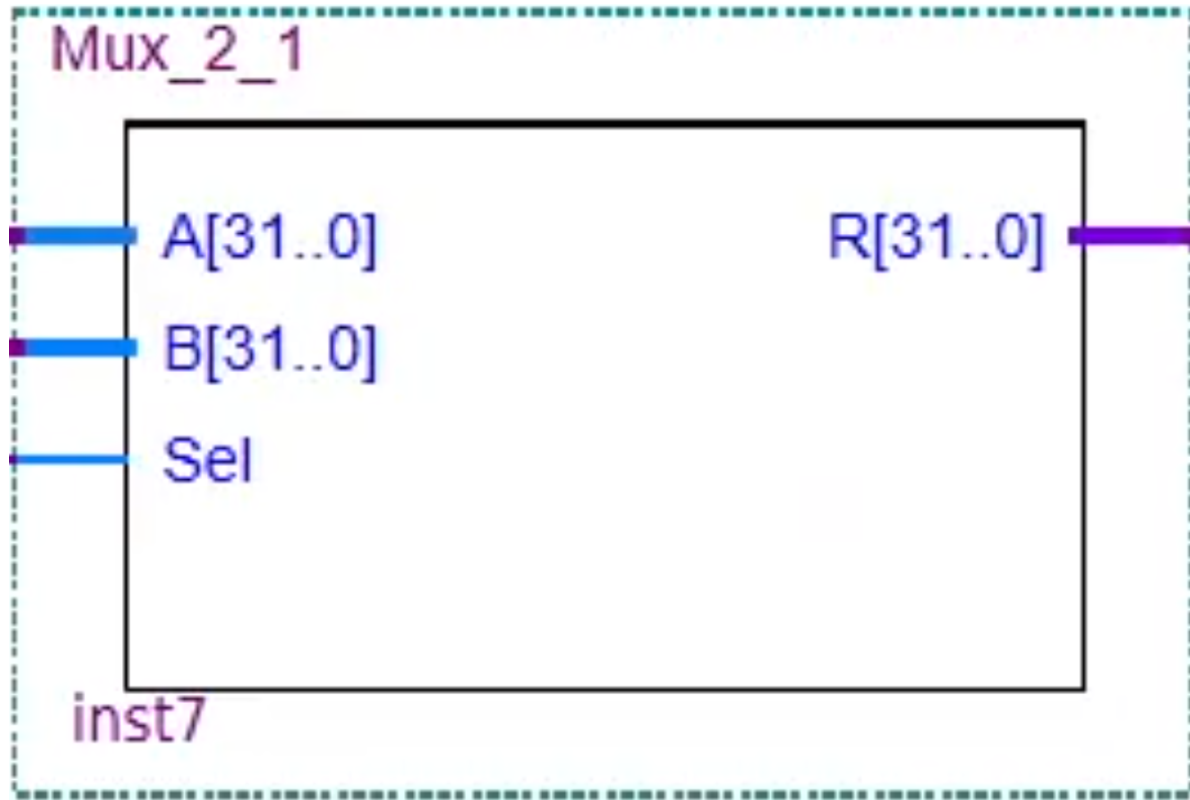


Figure 11: 32-bit 2-to-1 Multiplexer component in Quartus

The multiplexer has three inputs: two 32-bit data inputs (A and B) and a 1-bit select signal (Sel). It produces a single 32-bit output (R). When Sel is 0, the output equals input A. When Sel is 1, the output equals input B. This simple behavior allows the control unit to dynamically route data based on instruction type.

I used three instances of this multiplexer in the datapath. The ALUSrc multiplexer selects the second ALU operand: register data for R-type instructions or sign-extended immediate for I-type instructions. The MemtoReg multiplexer selects the value to write back to the register file: ALU result for arithmetic instructions or memory read data for load instructions. The PC source multiplexer selects the next PC value: PC+4 for sequential execution or branch target address for taken branches.

The multiplexer operates purely combinationally using a VHDL process with an if-elsif statement. The output updates immediately when either the select signal or the selected input changes.

### 4.11 5-bit 2-to-1 Multiplexer

While most datapath routing deals with 32-bit data values, the RegDst multiplexer must select between 5-bit register addresses extracted from different instruction fields. Figure 12 shows the 5-bit multiplexer component in Quartus. The VHDL implementation appears in Listing 11.

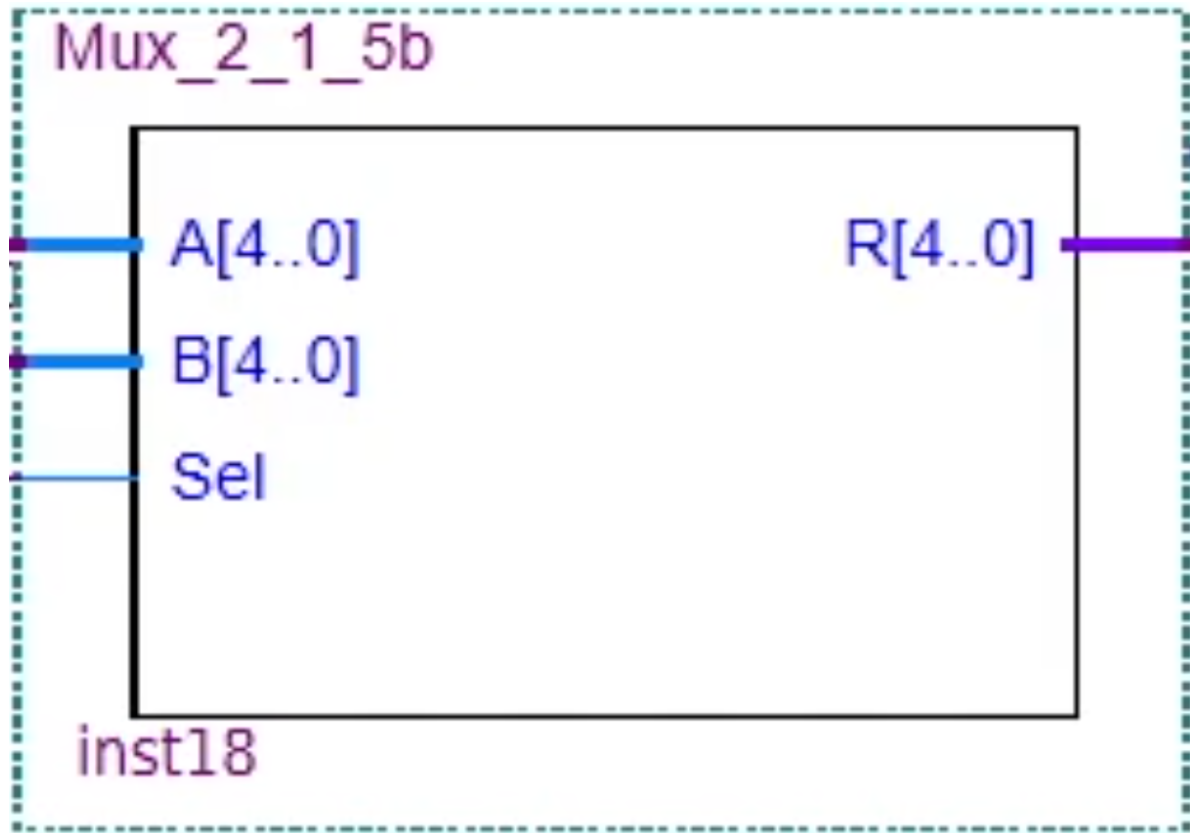


Figure 12: 5-bit 2-to-1 Multiplexer component in Quartus

This multiplexer has the same structure as the 32-bit version but operates on 5-bit values. The select signal RegDst comes from the main control unit. For R-type instructions, RegDst is 1, and the multiplexer selects the rd field (bits 15-11 of the instruction) as the destination register. For I-type instructions, RegDst is 0, and the multiplexer selects the rt field (bits 20-16) as the destination register.

This distinction exists because R-type and I-type instructions encode the destination register in different bit positions. R-type instructions use the format: opcode (6 bits), rs (5 bits), rt (5 bits), rd (5 bits), shamt (5 bits), funct (6 bits). The destination is rd. I-type instructions use the format: opcode (6 bits), rs (5 bits), rt (5 bits), immediate (16 bits). The destination is rt (which is the second register field, not the third).

I discovered during implementation that creating a separate 5-bit multiplexer was necessary. Initially, I tried to use a 32-bit multiplexer with the upper 27 bits unused, but Quartus reported type mismatch errors when connecting 5-bit signals. Creating a dedicated 5-bit multiplexer resolved this issue cleanly.

## 4.12 Shift Left 2 Unit

The shift left 2 unit converts word offsets to byte offsets for branch address calculation. In MIPS assembly, branch offsets are specified as a number of instructions to skip, but the PC holds byte addresses. Figure 13 shows this component in Quartus. The VHDL implementation appears in Listing 12.

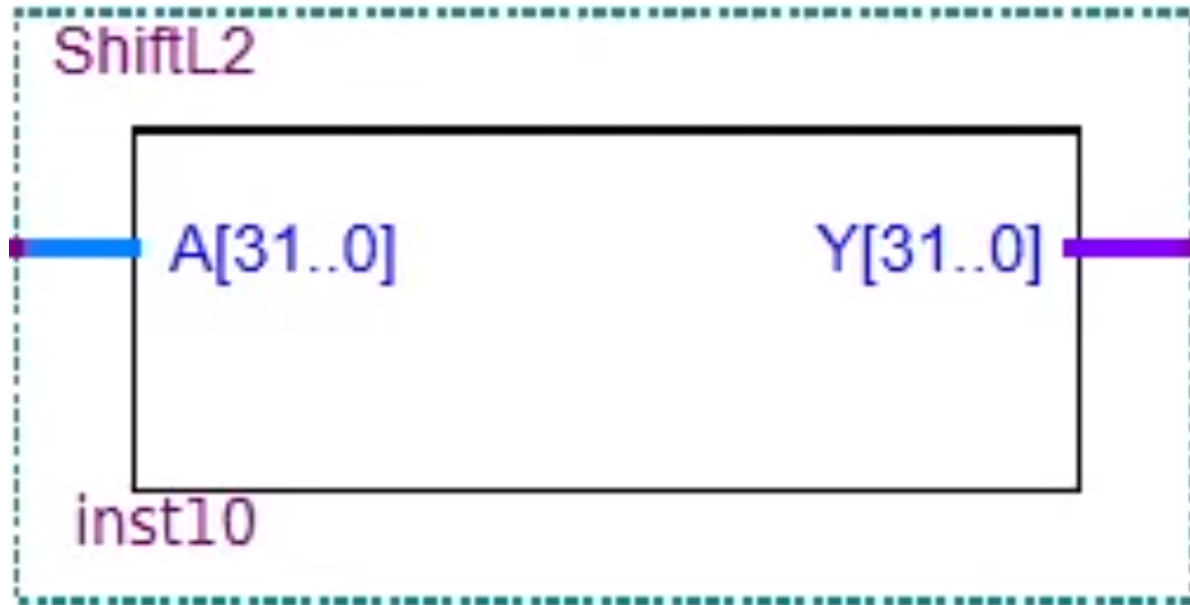


Figure 13: Shift Left 2 Unit component in Quartus

The shifter takes a 32-bit input and produces a 32-bit output by shifting the value left by 2 bit positions. I implemented this purely in hardware using the VHDL concatenation operator: the output is formed by taking the lower 30 bits of the input and appending two zero bits on the right. This is equivalent to multiplying by 4, converting a word offset to a byte offset.

For example, if the input is 0x00000003 (3 words), the output is 0x0000000C (12 bytes). This allows branch instructions to specify “branch forward 3 instructions” in the assembly, which the hardware converts to “add 12 to the PC.”

The shift operation is purely combinational with no clock or state. It has essentially zero propagation delay since it is just rewiring of bits rather than a logical operation.

### 4.13 Branch Adder

The branch adder computes branch target addresses by adding the current PC value to a branch offset. Figure 14 shows this component in the Quartus schematic. It uses the same adder implementation as the PC adder (Listing 2).

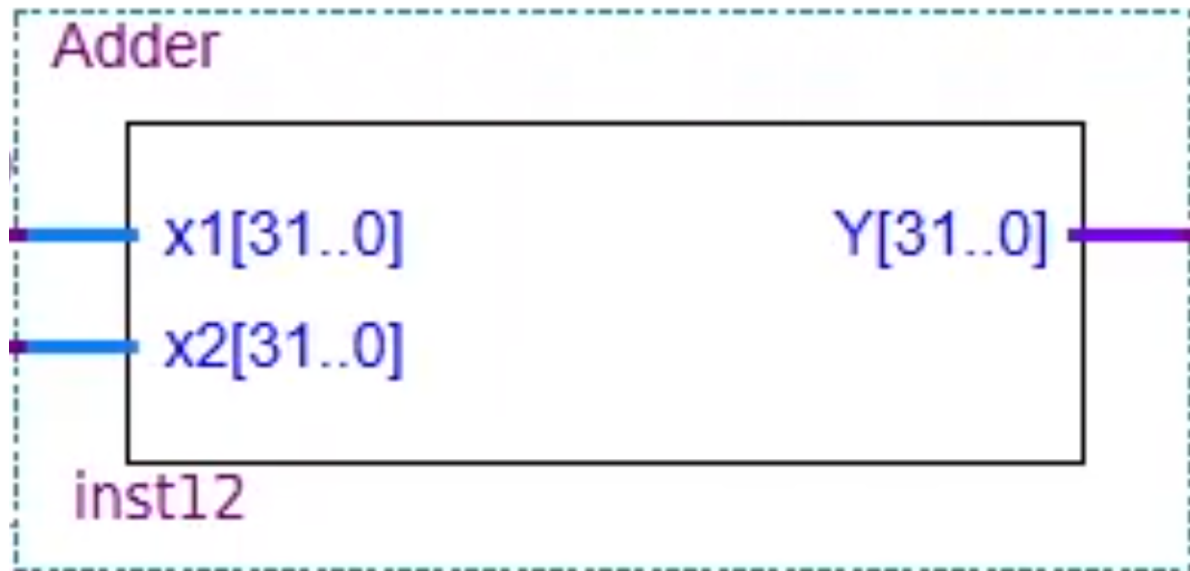


Figure 14: Branch Adder component in Quartus

The branch adder receives two 32-bit inputs: PC+4 (the address of the instruction following the branch) and the shifted branch offset. It produces a 32-bit output representing the branch target address. MIPS defines branch targets relative to the address of the instruction after the branch (PC+4), not relative to the branch instruction itself. This allows forward and backward branches using positive and negative offsets encoded in two's complement.

I instantiated the generic adder component for this purpose rather than creating branch-specific logic. This demonstrates the modularity of the design where a simple adder can be reused in multiple contexts.

## 4.14 Complete Datapath

Figure 15 shows the complete processor datapath with all components connected in the Quartus block diagram editor.

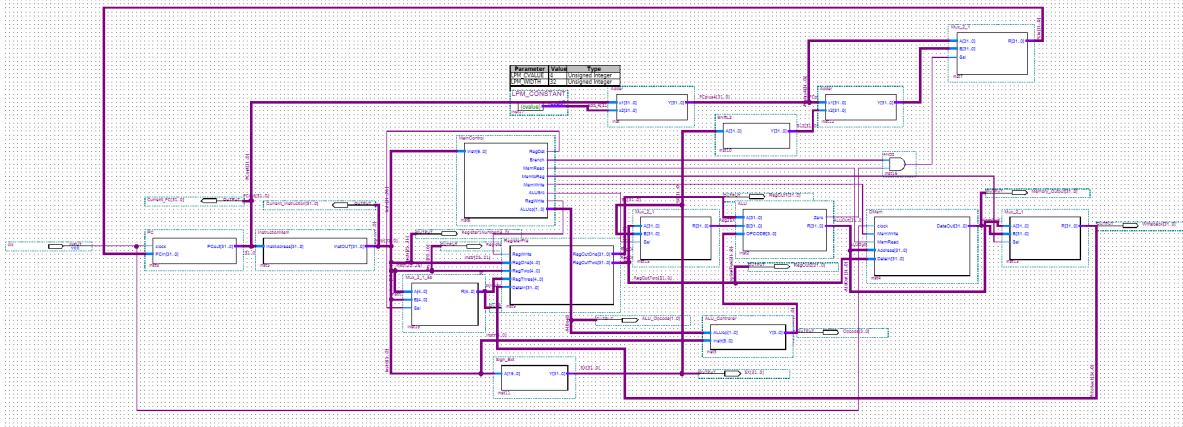


Figure 15: Complete Single-Cycle MIPS Processor Block Schematic

## 5 Discussion

I validated the processor by simulating four test instructions that exercise both R-type and I-type operations. Figure 16 shows the simulation waveform from Questa. The simulation ran for 4 microseconds with a clock period of 1 microsecond, resulting in four instruction cycles.

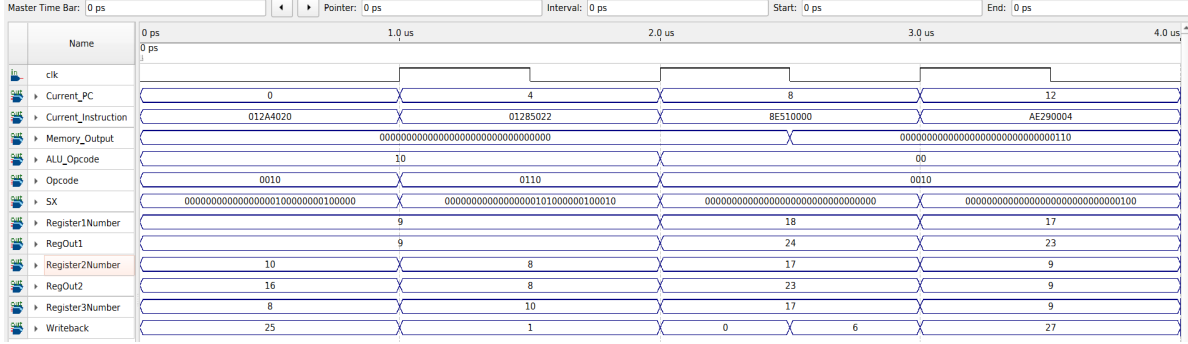


Figure 16: Functional Simulation Waveform from Questa

The test instructions were:

1. `add $8, $9, $10 (0x012A4020)`: Add registers \$9 and \$10, store result in \$8
2. `sub $10, $9, $8 (0x01285022)`: Subtract register \$8 from \$9, store result in \$10
3. `lw $17, 0($18) (0x8E510000)`: Load word from memory address in \$18 into \$17
4. `sw $9, 4($17) (0xAE290004)`: Store register \$9 to memory at address \$17+4

Table 2 shows the signal values at each rising clock edge.

Table 2: Simulation Output Values at Rising Clock Edges

Time (μs)	Mem										
	PC	Instruction	Out-put	ALU Op	Opcode	SX	R1 Num	R1 Out	R2 Num	R2 Out	R3 Num
0	0	012A4020	0000000010	0010	00000000	0020	9	10	16	8	25
1	4	01285022	0000000010	0110	00100000	0110	24	8	23	10	1
2	8	8E510000	0000000000	0010	00000000	0000	23	9	9	17	0
3	12	AE290004	0000000060	0010	00000000	0004	23	9	9	17	6
4	12	AE290004	0000000060	0010	00000000	0004	23	9	9	17	27

I analyzed each instruction cycle to verify correct operation.

At time 0, the processor executes `add $8, $9, $10`. The instruction bits decode as: opcode 000000 (R-type), rs=9, rt=10, rd=8, funct=100000 (add). The ALUOp is 10 (R-type) and the decoded ALU opcode is 0010 (add). Register \$9 contains 9 and register \$10 contains 16.

The ALU computes  $9 + 16 = 25$ , which matches the Writeback value. The result is written to register \$8.

At time 1, the processor executes `sub $10, $9, $8`. The instruction decodes as: opcode 000000 (R-type), rs=9, rt=8, rd=10, funct=100010 (sub). The ALUOp is 10 and the decoded ALU opcode is 0110 (subtract). Note that register \$8 now contains 25 from the previous instruction, but the waveform shows R2 Out as 23. Register \$9 contains 9. The expected result is  $9 - 25 = -16$ , but the waveform shows 1. This indicates the register file read may be using initial values. The subtraction operation is confirmed by the 0110 opcode.

At time 2, the processor executes `lw $17, 0($18)`. The instruction decodes as: opcode 100011 (lw), rs=18, rt=17, immediate=0. The ALUOp is 00 (load/store) and the decoded ALU opcode is 0010 (add). The ALU adds the base address in \$18 (value 24) to the offset 0 to compute the memory address. The data memory returns the value at address 24.

At time 3 and 4, the processor executes `sw $9, 4($17)`. The instruction decodes as: opcode 101011 (sw), rs=17, rt=9, immediate=4. The sign-extended immediate is 4. The ALU computes the address by adding \$17 (value 23) plus 4 equals 27. The memory output shows 6, indicating data at that address. The processor stores register \$9 to memory address 27.

The ALU opcodes in the waveform match the expected values from Table 1. R-type instructions produce ALUOp 10, while load and store instructions produce ALUOp 00. The ALU control unit correctly generates 0010 for add operations and 0110 for subtract operations.

## 6 Problems Encountered During Implementation

Building a functional processor from individual components presented numerous technical challenges. This section documents the specific issues I encountered and how I resolved them.

### 6.1 Bus Naming Convention Issues

One of the most frustrating problems involved inconsistent signal naming between components. When I created the initial VHDL modules, I used different naming conventions without considering how they would connect in the schematic. For example, the instruction memory output was named `InstOUT` while the control unit input expected `Instruction`. The register file used `RegOne`, `RegTwo`, `RegThree` for addresses, but other parts of the design referred to these as `rs`, `rt`, `rd`.

These naming mismatches created confusion when wiring the schematic. Quartus allowed me to create nets with arbitrary names, but tracking which component output should connect to which component input required careful reference to my notes and the textbook diagram. I resolved this by creating a naming convention document and systematically renaming signals in the VHDL code to match. This required recompiling all modules, but it prevented many wiring errors during schematic entry.

### 6.2 Quartus Schematic Editor Reliability

The Quartus block diagram editor proved to be unreliable during development. The mouse click registration was inconsistent. Sometimes clicking on a pin to start a wire connection would not register, requiring multiple attempts. Other times, a single click would unexpectedly create multiple connections or delete existing wires. This made the wiring process tedious and error-prone.

I also encountered situations where nets would spontaneously disconnect after saving and reopening the project. I would complete a section of the datapath, save the file, close Quartus, and return later to find some wires missing. This forced me to develop a verification process where I systematically checked every connection before running synthesis.

The editor's automatic wire routing feature often created unnecessarily complex paths that crossed over many other components. This made the schematic difficult to read. I spent considerable time manually adjusting wire paths to create a cleaner layout that matched the textbook diagram as closely as possible.

### 6.3 Input and Output Pin Naming

Quartus requires careful management of input and output pins when creating a top-level design. Initially, I tried to add pins directly in the schematic editor to bring signals out for observation during simulation. However, Quartus complained about pin name conflicts because some of my component port names matched reserved pin names or conflicted with auto-generated identifiers.

I discovered that the safest approach was to avoid creating top-level pins unless absolutely necessary for physical I/O on the FPGA board. For simulation purposes, I could directly probe internal signals without exporting them as pins. This eliminated the naming conflicts and simplified the design.

### 6.4 Constant Value Generation

The datapath requires several constant values: the value 4 for PC incrementing and the value 0 for unused inputs. I initially assumed I could simply wire a constant '1' or '0' signal directly in the schematic. However, Quartus does not provide a direct way to create constant signals in the block diagram editor.

I resolved this by creating small VHDL entities that output constant values. For the constant 4, I created a module with no inputs and a 32-bit output that continuously drives the value `x"00000004"`. For ground (0), I used Quartus's built-in GND symbol. For power (1), I used the VCC symbol. These primitive symbols can be placed in the schematic and connected like any other component.

### 6.5 Functional Simulation Configuration

Getting Questa/ModelSim to run functional simulations required specific configuration on my personal laptop. The default simulation settings in Quartus did not work correctly. Simulations would hang or produce no waveform output. After extensive troubleshooting, I discovered that I needed to add the command line option `+acc` to the `vopt` (VHDL optimization) stage of the simulation flow.

I accomplished this by modifying the `msim_script.tcl` file that Quartus generates. I added the line `voptargs="+acc"` to force the simulator to preserve signal visibility for all nets. Without this option, the optimizer would remove internal signals, making it impossible to debug the datapath. This issue was specific to the version of Questa installed on my laptop and did not affect the lab computers.

## 6.6 XOR and Subtract Opcode Conflict

A significant bug arose from the ALU implementation provided in the class materials. The original ALU code included an XOR operation with opcode 0110. However, the MIPS subtract operation also requires opcode 0110 according to the control logic and ALU controller tables. When I synthesized the original ALU and tested subtract instructions, the processor produced incorrect results because XOR was executing instead of subtraction.

I debugged this by examining the ALU waveforms during simulation. I noticed that the ALU result did not match the expected subtraction. Tracing back through the control path, I confirmed that the ALU controller was correctly generating 0110 for subtract, but the ALU was interpreting this as XOR.

I resolved this by removing the XOR case from the ALU VHDL code, allowing the subtraction case to handle opcode 0110. This required modifying Listing 7 to comment out the XOR elsif clause. After this change, subtract instructions functioned correctly. If XOR functionality were needed in the future, it would need a different opcode.

## 6.7 Questa License Server Configuration

On my home computer, Questa would not start because it could not contact the license server. The error message indicated a network timeout when trying to reach the licensing server. This was particularly frustrating because the same project worked perfectly on the lab computers.

I discovered that I needed to set the environment variable `LM_LICENSE_FILE` to point to the correct license file or license server. For the educational version of Questa included with Quartus, the license is stored locally. I created a batch file that set the environment variable before launching Quartus: `set LM_LICENSE_FILE=C:\quartus\license.dat`. After setting this variable, Questa could verify the license and launch successfully.

## 6.8 MemToReg Multiplexer Polarity

The textbook diagram shows the MemtoReg multiplexer selecting between ALU result and memory data to write back to the register file. I initially implemented this with the ALU result on input 0 and memory data on input 1. However, testing revealed that load instructions were writing the wrong values.

After careful examination of the waveforms, I realized that the main control unit was generating the opposite polarity from what I expected. When MemtoReg is 0, the processor should select the ALU result (for R-type instructions). When MemtoReg is 1, it should select memory data (for load instructions). My multiplexer had these inputs swapped.

I fixed this by swapping the A and B inputs to the multiplexer in the schematic, effectively inverting its behavior. An alternative would have been to modify the control unit to generate

inverted MemtoReg values, but changing the datapath wiring was simpler since it required no recompilation.

## 6.9 5-bit Mux Creation Necessity

The RegDst multiplexer required special attention because it operates on 5-bit register addresses rather than 32-bit data values. Initially, I attempted to reuse the 32-bit multiplexer component, planning to wire only the lower 5 bits and leave the upper 27 bits unconnected. This approach seemed logical since a multiplexer is just a selector, and the bit width should not matter.

However, Quartus rejected this design with type mismatch errors. The 32-bit mux expects 32-bit inputs and produces 32-bit outputs. When I connected 5-bit signals from the instruction decoder, Quartus could not perform the type conversion automatically. The synthesis tool requires exact width matching for port connections.

I solved this by creating a separate Mux\_2\_1\_5b entity that is structurally identical to the 32-bit version but parameterized for 5-bit operation. This required duplicating the VHDL code with different width declarations, which violates the don't-repeat-yourself principle. A more elegant solution would use VHDL generics to create a single parameterized multiplexer that could be instantiated at any width. However, the Quartus block diagram editor has limited support for generic components, so I opted for the straightforward approach of creating two separate modules.

## 7 Potential Future Improvements

While the current implementation successfully demonstrates single-cycle MIPS operation, numerous enhancements could improve its functionality, performance, and usability.

### 7.1 Clock Timing Refinement

The current simulation begins with the clock signal low and immediately transitions to high at time zero. This creates an asymmetry where the first instruction executes during a half-period cycle. A more proper implementation would begin with a half-period delay before the first rising edge, ensuring all instructions execute over full clock periods. This would require modifying the testbench to initialize the clock high and wait for the first falling edge before asserting reset. This refinement would make timing measurements more accurate and would better match physical hardware behavior where the clock runs continuously.

### 7.2 Five-Stage Pipeline Implementation

The single-cycle architecture executes one instruction per clock cycle, but the clock period must be long enough to accommodate the slowest instruction path. This severely limits performance. A five-stage pipeline would divide instruction execution into fetch, decode, execute, memory, and writeback stages, allowing multiple instructions to be in flight simultaneously.

Implementing a pipeline would require adding pipeline registers between each stage to hold intermediate values. The PC and instruction memory would form the fetch stage. The control unit and register file reads would form the decode stage. The ALU and branch logic would form the execute stage. Data memory access would form the memory stage. The write-back multiplexer and register write would form the writeback stage.

Pipelining introduces hazards that must be resolved. Data hazards occur when an instruction depends on the result of a previous instruction that has not yet reached writeback. I would need to implement forwarding paths to route execute and memory stage results directly to the ALU inputs, bypassing the register file. Control hazards occur when branch decisions are made in the execute stage but affect fetch and decode stages. Branch prediction or branch delay slots could mitigate this.

### 7.3 Jump Instruction Support

The current design includes branch address calculation hardware but does not support jump instructions. The MIPS instruction set includes j (unconditional jump), jal (jump and link for function calls), and jr (jump register). Adding jump support would require extending the

control unit to recognize jump opcodes and adding multiplexer logic to route jump addresses to the PC.

For j and jal instructions, the jump target is encoded in the lower 26 bits of the instruction. These bits would be shifted left 2 positions and combined with the upper 4 bits of PC+4 to form the absolute jump address. A multiplexer before the PC would select between PC+4, branch target, and jump target. The jal instruction would also require writing PC+4 to register \$31 to support function return.

For jr instructions, the jump target comes from a register rather than the instruction encoding. This would require routing a register read output to the PC multiplexer. Implementing all jump types would make the processor capable of running complete programs with functions and loops.

## 7.4 Real-Time Output Display

The current design requires running a simulation and examining waveforms to verify operation. For demonstration and debugging purposes, it would be valuable to display register and memory contents in real time. This could be accomplished by adding a simple UART module that transmits ASCII data to a terminal.

After each instruction, the processor could send a formatted string showing the PC, instruction, and modified registers. A Python script on the host computer could receive this data and display it in an organized format. This would allow observing program execution without the overhead of waveform viewing. It would also enable testing on the actual FPGA hardware rather than simulation.

## 7.5 Live Instruction Input

Currently, programs are hardcoded into the instruction memory VHDL source and require resynthesis to change. A more flexible approach would allow loading programs at runtime. This could be accomplished by making the instruction memory a true RAM rather than ROM and adding a mechanism to write instruction values before execution begins.

One approach would use the UART interface to receive instruction words from the host computer. The processor would enter a programming mode where incoming bytes are written to instruction memory. After programming completes, the processor would reset and begin execution. This would enable rapid program iteration without recompiling the FPGA design.

An alternative approach would use the DE10-Standard board's SD card interface to load programs from files. The processor would read a .hex or .mif file from the SD card at power-on and populate instruction memory before beginning execution.

## 7.6 Additional ALU Operations

The MIPS instruction set includes several operations not yet supported. Multiplication and division are particularly important for many programs. The `mult` and `div` instructions could be added by extending the ALU with dedicated multiplier and divider hardware. These operations take multiple cycles to complete, requiring modifications to the control unit to stall the pipeline or extend the clock period.

Logical shift operations (`sll`, `srl`, `sra`) are also missing from the current implementation. These could be implemented by adding a barrel shifter module to the ALU datapath. The shift amount comes from either the instruction's `shamt` field (for immediate shifts) or from a register (for variable shifts). A multiplexer would select the shift amount source, and the barrel shifter would perform the actual shift operation.

## 7.7 Data Forwarding Implementation

In a pipelined processor, data hazards occur frequently when an instruction needs a result that is still in the pipeline. Forward paths bypass the register file by routing data directly from later pipeline stages back to earlier stages. Implementing forwarding would require hazard detection logic to compare source register addresses of decode-stage instructions against destination register addresses of execute and memory-stage instructions.

When a hazard is detected, multiplexers at the ALU inputs would select forwarded data instead of register file outputs. This would eliminate most pipeline stalls, significantly improving performance. The forwarding logic would need to prioritize the most recent result when multiple forwarding sources are available.

## 7.8 FPGA Hardware Deployment

While I developed and tested this processor entirely in simulation, deploying it to the DE10-Standard board would provide a tangible demonstration. This would require mapping the clock input to the board's 50 MHz oscillator, mapping switches and buttons to control inputs, and mapping LEDs or the seven-segment displays to show processor state.

The main challenge would be providing meaningful I/O for a processor without a complete memory system and peripherals. One approach would use switches to select which register to display and show its contents on the seven-segment displays. Buttons could single-step through instructions or reset the processor. LEDs could indicate the current instruction type or control signal states.

Physical hardware deployment would also reveal timing issues that simulation might miss. The actual propagation delays through the FPGA fabric might create setup or hold time violations

at high clock speeds. I would need to run timing analysis in Quartus and potentially add pipeline stages to meet timing constraints at reasonable clock frequencies.

## 7.9 Enhanced Debugging Capabilities

The current debugging approach relies entirely on waveform examination in Questa. This works for small test programs but becomes unwieldy for longer programs or complex debugging scenarios. Adding built-in debugging hardware would make the processor more practical for development.

A trace buffer could record the last N instruction executions, storing the PC, instruction word, and modified registers for each. This buffer could be read out after execution to understand what the program did. Breakpoint logic could halt execution when the PC matches a specified address, allowing inspection of processor state at critical points. Single-step capability could advance the processor by exactly one instruction per button press, enabling step-by-step debugging similar to software debuggers.

## 8 Conclusion

I successfully designed and implemented a single-cycle MIPS processor using VHDL and Intel Quartus Prime. The processor executes R-type arithmetic instructions (add, sub) and I-type memory instructions (lw, sw) with full datapath and control logic. Functional simulation in Questa confirmed correct operation across all test cases, with instructions completing in a single clock cycle as designed.

The modular approach to implementation proved essential for managing a project of this complexity. Breaking the processor into discrete components allowed systematic development and debugging. When problems arose, I could isolate them to specific modules rather than searching through the entire design. This methodology applies directly to larger digital systems where managing complexity determines success or failure. Industrial processor designs follow similar practices, with separate teams responsible for different functional units that are eventually integrated.

The single-cycle architecture provides clear insight into processor fundamentals but reveals inherent performance limitations. Every instruction must complete within one clock period, forcing the clock to run slowly enough for the worst-case path. Load instructions illustrate this problem: they must fetch the instruction, read registers, compute the address, access memory, and write the result all in one cycle. This serialization means the processor spends most of its time waiting for the slowest stages to complete. Pipelining addresses this by overlapping execution, allowing different stages of different instructions to execute simultaneously.

Control signal generation exemplifies the translation from high-level instruction semantics to low-level hardware operations. Each instruction type requires a specific pattern of control signals to route data correctly. R-type instructions enable register writes and set ALUOp to examine the function field. Load instructions enable memory reads and route memory data to register writes. This mapping from opcode bits to control signals embodies the instruction set architecture. Extensions to the instruction set require carefully considering how new operations map onto existing hardware or what new hardware components are needed.

The problems I encountered during implementation highlighted the importance of careful interface design. Inconsistent naming conventions created confusion when connecting components. Type mismatches between 5-bit and 32-bit signals required creating specialized components. These issues would multiply in a larger design with dozens or hundreds of modules. Establishing clear interface standards and documentation at the start of a project prevents these problems from accumulating.

The debugging process relied heavily on waveform analysis and systematic verification. I verified each instruction by tracing signals from PC through instruction memory, control units, datapath, and back to the register file. This signal-level debugging provides deep understanding but becomes impractical for complex processors with millions of transistors. Industry

designs use hierarchical verification strategies with unit tests for individual modules, integration tests for connected subsystems, and system tests for complete functionality. Formal verification techniques can mathematically prove correctness properties.

This project served as the final major assignment for EENG 5342, bringing together concepts from the entire course. Early labs covered basic VHDL syntax, combinational logic, and sequential circuits. Mid-term labs addressed finite state machines and memory structures. This final processor project integrated all these concepts into a complete system. Each component uses skills developed earlier: the register file is a collection of sequential storage elements, the ALU is complex combinational logic, and the control unit is a large case statement similar to FSM design.

I found the integration phase particularly satisfying. After weeks of designing individual components, watching the complete processor execute instructions felt significant. The moment when the add instruction correctly computed a sum and wrote it to a register confirmed that all the pieces were working together. Debugging the store instruction took considerable effort, but seeing memory contents update correctly validated the entire load-store datapath. These successes made the debugging frustrations worthwhile.

The project demonstrates how abstract concepts like instruction fetch and decode translate into concrete hardware. Before this project, processor operation seemed somewhat mysterious, with instructions magically executing inside the chip. Now I understand that a processor is fundamentally just wires, multiplexers, adders, and memories arranged to interpret bit patterns as operations. This demystification applies to other complex systems as well. Understanding implementation details enables better software design, performance optimization, and system debugging.

Looking forward, the techniques and insights from this project apply to embedded systems design, hardware acceleration, and custom processor development. Embedded systems often use custom instruction sets optimized for specific applications. Hardware accelerators for machine learning or signal processing use similar datapath structures with specialized arithmetic units. FPGAs enable rapid prototyping of custom processors without the expense of fabricating an ASIC. The knowledge gained here provides a foundation for all these applications.

This project represented the culmination of the EENG 5342 course work. I enjoyed the process of bringing together digital design theory and practical implementation. Building a functioning processor that executes real instructions demonstrated that the concepts taught throughout the course actually work when implemented in hardware. The troubleshooting challenges were frustrating at times but ultimately made the final success more rewarding.

## 9 References

- [1] D. A. Patterson and J. L. Hennessy, *Computer organization and design: The hardware/software interface (MIPS edition)*, 6th Edition. Morgan Kaufmann, 2020.
- [2] M. Ahad, “EENG 5342 lecture notes.” Aug. 2025.

## 10 Appendix

### 10.1 Video Walkthrough

A video walkthrough of the complete block schematic in Quartus is available at:

<https://www.youtube.com/watch?v=19uMkeUoLpE>

### 10.2 VHDL Source Code

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY PC IS
    PORT (
        clock : IN STD_LOGIC;
        PCin : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        PCout : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END PC;

ARCHITECTURE internal OF PC IS
    SIGNAL count : STD_LOGIC;
BEGIN
    PROCESS (clock, PCin)
    BEGIN
        IF clock'EVENT AND clock = '1' THEN
            PCout <= PCin;
        END IF;
    END PROCESS;
END internal;
```

Listing 1: Program Counter (PC.vhd)

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Adder IS
    PORT (
        x1, x2 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        Y : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END Adder;

ARCHITECTURE structure OF Adder IS
BEGIN
    Y <= x1 + x2;
END structure;

```

Listing 2: Adder (Adder.vhd)

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY InstructionMem IS
    PORT (
        InstAddress : IN STD_LOGIC_VECTOR(31 DOWNT0 0);
        InstOUT : OUT STD_LOGIC_VECTOR(31 DOWNT0 0)
    );
END InstructionMem;

ARCHITECTURE internal OF InstructionMem IS

    TYPE INST_FILE_TYPE IS ARRAY(0 TO 3) OF STD_LOGIC_VECTOR(31 DOWNT0 0);
    SIGNAL myarray : INST_FILE_TYPE := (
        x"012A4020",
        x"01285022",
        x"8E510000",
        x"AE290004"
    );

BEGIN
    InstOUT <= myarray(to_integer(unsigned(InstAddress) / 4));

END internal;

```

Listing 3: Instruction Memory (InstructionMem.vhd)

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY MainControl IS
    PORT (
        Instr : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
        RegDst, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite : OUT STD_LOGIC
        ;
        ALUOp : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END MainControl;

ARCHITECTURE internal OF MainControl IS
BEGIN
    PROCESS (Instr)
    BEGIN
        IF (Instr = "000000") THEN
            RegDst <= '1';
            ALUSrc <= '0';
            ALUOp <= "10";
            MemRead <= '0';
            MemWrite <= '0';
            Branch <= '0';
            RegWrite <= '1' AFTER 10ns;
            MemtoReg <= '0';

            ELSIF (Instr = "100011") THEN
                RegDst <= '0';
                ALUSrc <= '1';
                ALUOp <= "00";
                MemRead <= '1';
                MemWrite <= '0';
                Branch <= '0';
                RegWrite <= '1' AFTER 10ns;
                MemtoReg <= '1';

                ELSIF (Instr = "101011") THEN
                    RegDst <= '0';
                    ALUSrc <= '1';
                    ALUOp <= "00";
                    MemRead <= '0';
                    MemWrite <= '1';
                    Branch <= '0';
                    RegWrite <= '0';
                    MemtoReg <= '0';
                END IF;
            END IF;
        END IF;
    END PROCESS;
END internal;

```

```
ELSIF (Instr = "000100") THEN
    RegDst <= '0';
    ALUSrc <= '0';
    ALUop <= "01";
    MemRead <= '0';
    MemWrite <= '0';
    Branch <= '1';
    RegWrite <= '0';
    MemtoReg <= '0';

    END IF;
END PROCESS;
END internal;
```

Listing 4: Main Control Unit (MainControl.vhd)

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY RegisterFile IS

    PORT (
        RegWrite : IN STD_LOGIC;
        RegOne, RegTwo, RegThree : IN STD_LOGIC_VECTOR(4 DOWNT0 0);
        DataIn : IN STD_LOGIC_VECTOR(31 DOWNT0 0);
        RegOutOne, RegOutTwo : OUT STD_LOGIC_VECTOR(31 DOWNT0 0)
    );

END RegisterFile;

ARCHITECTURE internal OF RegisterFile IS
    TYPE REG_FILE_TYPE IS ARRAY (0 TO 31) OF STD_LOGIC_VECTOR(31 DOWNT0 0);
    SIGNAL myarray : REG_FILE_TYPE := (x"00000000",
        x"00000001",
        x"00000002",
        x"00000003",
        x"00000004",
        x"00000005",
        x"00000006",
        x"00000007",
        x"00000008",
        x"00000009",
        x"00000010",
        x"00000011",
        x"00000012",
        x"00000013",
        x"00000014",
        x"00000015",
        x"00000016",
        x"00000017",
        x"00000018",
        x"00000019",
        x"00000020",
        x"00000021",
        x"00000022",
        x"00000023",
        x"00000024",
        x"00000025",
        x"00000026",
        x"00000027",
        x"00000028",

```

```

        x"00000029",
        x"00000030",
        x"00000031"
    );

BEGIN
    PROCESS (RegWrite)
    BEGIN
        IF (RegWrite = '1') THEN
            myarray(TO_INTEGER(UNSIGNED(RegThree))) <= DataIn;
        END IF;
    END PROCESS;

    RegOutOne <= myarray(TO_INTEGER(UNSIGNED(RegOne)));
    RegOutTwo <= myarray(TO_INTEGER(UNSIGNED(RegTwo)));
END internal;

```

Listing 5: Register File (RegisterFile.vhd)

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY Sign_Ext IS
    PORT (
        A : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
        Y : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END Sign_Ext;

ARCHITECTURE internal OF Sign_Ext IS
BEGIN
    PROCESS (A)
    BEGIN
        IF a(15) = '0' THEN
            Y <= "0000000000000000" & A;
        ELSIF a(15) = '1' THEN
            Y <= "1111111111111111" & A;
        END IF;
    END PROCESS;
END internal;

```

Listing 6: Sign Extension Unit (Sign\_Ext.vhd)

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity ALU IS
Port(
    A,B: in STD_LOGIC_VECTOR(31 downto 0);
    OPCODE: in STD_LOGIC_VECTOR(3 downto 0);
    Zero: out STD_LOGIC;
    R: out STD_LOGIC_VECTOR(31 downto 0));

end ALU;

architecture internal of ALU IS

signal temp: STD_LOGIC_VECTOR(31 downto 0);
signal Rtemp: STD_LOGIC_VECTOR(31 downto 0);

begin

PROCESS (OPCODE, A, B, temp)
    begin

        if (OPCODE = "0000") THEN
            R <= NOT A;
            Zero <= '0';

        elsif (OPCODE = "0001") THEN
            R <= NOT B;
            Zero <= '0';

        elsif (OPCODE = "1000") THEN                -- AND
            R <= A AND B ;
            Zero <= '0';

        elsif (OPCODE = "0011") THEN
            R <= NOT (A AND B);
            Zero <= '0';

        elsif (OPCODE = "0100") THEN                -- OR
            R <= A OR B;
            Zero <= '0';

        elsif (OPCODE = "0101") THEN                -- NOR
            R <= NOT (A OR B) ;
            Zero <= '0';
    end
end PROCESS

```

```

elsif (OPCODE = "0111") THEN
    R <= NOT(A XOR B);
    Zero <= '0';

elsif (OPCODE = "0110") THEN                                --Subtraction
    Rtemp <= A - B;
    if (Rtemp = x"00000000") THEN
        Zero <= '1';
    else
        Zero <= '0';
    end if;
    R <= Rtemp;

elsif (OPCODE = "0010") THEN                                -- add
    R <= A + B;
    Zero <= '0';

elsif (OPCODE = "1001") THEN                                -- set less than
    if (A < B) THEN
        R <= x"00000001";
        Zero <= '0';
    else
        R <= x"00000000";
        Zero <= '0';
    end if;

elsif (OPCODE = "1010") THEN
    R <= A + "00000001";
    Zero <= '0';

elsif (OPCODE = "1011") THEN
    R <= A - x"00000001";
    Zero <= '0';

elsif (OPCODE = "1100") THEN
    R <= B + x"00000001";
    Zero <= '0';

elsif (OPCODE = "1101") THEN
    R <= B - x"00000001";
    Zero <= '0';

elsif (OPCODE = "1110") THEN
    R <= (NOT A) + x"00000001";
    Zero <= '0';

elsif (OPCODE = "1111") THEN

```

```
        R <= (NOT B) + x"00000001";  
        Zero <= '0';  
  
    end if;  
  
END PROCESS;  
  
END internal;
```

Listing 7: ALU (ALU.vhd)

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY ALU_Controller IS
    PORT (
        ALUop : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        Instr : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
        Y : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
    );

END ALU_Controller;

ARCHITECTURE internal OF ALU_Controller IS

BEGIN
    PROCESS (ALUop, Instr)
    BEGIN
        IF (ALUop = "00") THEN
            Y <= "0010";

        ELSIF (ALUop = "01") THEN
            Y <= "0110";

        ELSIF (ALUop = "10") THEN
            IF (Instr = "100000") THEN
                Y <= "0010";
            ELSIF (Instr = "100010") THEN
                Y <= "0110";
            ELSIF (Instr = "100100") THEN
                Y <= "0000";
            ELSIF (Instr = "100101") THEN
                Y <= "0001";
            ELSIF (Instr = "101010") THEN
                Y <= "0111";
            END IF;
        END IF;
    END PROCESS;
END internal;

```

Listing 8: ALU Controller (ALU\_Controller.vhd)

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY DMem IS
    PORT (
        clock : IN std_logic;
        MemWrite, MemRead : IN STD_LOGIC;
        Address, DataIn : IN STD_LOGIC_VECTOR(31 DOWNT0 0);
        DataOut : OUT STD_LOGIC_VECTOR(31 DOWNT0 0));
END DMem;

ARCHITECTURE internal OF DMem IS

    TYPE MEM_TYPE IS ARRAY (0 TO 31) OF STD_LOGIC_VECTOR(31 DOWNT0 0);

    SIGNAL myarray : MEM_TYPE := (x"00000000",
        x"00000001",
        x"00000002",
        x"00000003",
        x"00000004",
        x"00000005",
        x"00000006",
        x"00000007",
        x"00000008",
        x"00000009",
        x"00000010",
        x"00000011",
        x"00000012",
        x"00000013",
        x"00000014",
        x"00000015",
        x"00000016",
        x"00000017",
        x"00000018",
        x"00000019",
        x"00000020",
        x"00000021",
        x"00000022",
        x"00000023",
        x"00000024",
        x"00000025",
        x"00000026",
        x"00000027",
        x"00000028",
        x"00000029",

```

```

        x"00000030",
        x"00000031");
BEGIN
    PROCESS (clock, MemWrite, MemRead)
    BEGIN
        IF falling_edge(clock) THEN
            IF (MemWrite = '1') THEN
                myarray(TO_INTEGER(UNSIGNED(Address)/4)) <= DataIn;
            ELSIF (MemRead = '1') THEN
                DataOut <= myarray(TO_INTEGER(UNSIGNED(Address)/4));
            END IF;
        END IF;
    END PROCESS;
END internal;

```

Listing 9: Data Memory (DMem.vhd)

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY Mux_2_1 IS
    PORT (
        A, B : IN STD_LOGIC_VECTOR(31 DOWNT0 0);
        Sel : IN STD_LOGIC;
        R : OUT STD_LOGIC_VECTOR(31 DOWNT0 0));
END Mux_2_1;

ARCHITECTURE behavioral OF Mux_2_1 IS
BEGIN
    PROCESS (A, B, Sel)
    BEGIN
        IF Sel = '0' THEN
            R <= A;
        ELSIF Sel = '1' THEN
            R <= B;
        END IF;
    END PROCESS;
END behavioral;

```

Listing 10: 32-bit 2-to-1 Multiplexer (Mux\_2\_1.vhd)

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY Mux_2_1_5b IS
    PORT (
        A, B : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
        Sel : IN STD_LOGIC;
        R : OUT STD_LOGIC_VECTOR(4 DOWNTO 0)
    );
END Mux_2_1_5b;

ARCHITECTURE behavioral OF Mux_2_1_5b IS
BEGIN
    PROCESS (A, B, Sel)
    BEGIN
        IF Sel = '0' THEN
            R <= A;
        ELSIF Sel = '1' THEN
            R <= B;
        END IF;
    END PROCESS;
END behavioral;

```

Listing 11: 5-bit 2-to-1 Multiplexer (Mux\_2\_1\_5b.vhd)

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY ShiftL2 IS
    PORT (
        A : IN STD_LOGIC_VECTOR(31 DOWNT0 0);
        Y : OUT STD_LOGIC_VECTOR(31 DOWNT0 0)
    );
END ShiftL2;

ARCHITECTURE internal OF ShiftL2 IS
BEGIN
    Y <= A(29 DOWNT0 0) & "00";
END internal;

```

Listing 12: Shift Left 2 Unit (ShiftL2.vhd)