CSE 490/590

Computer Architecture

Project 1 - Report

Team 19

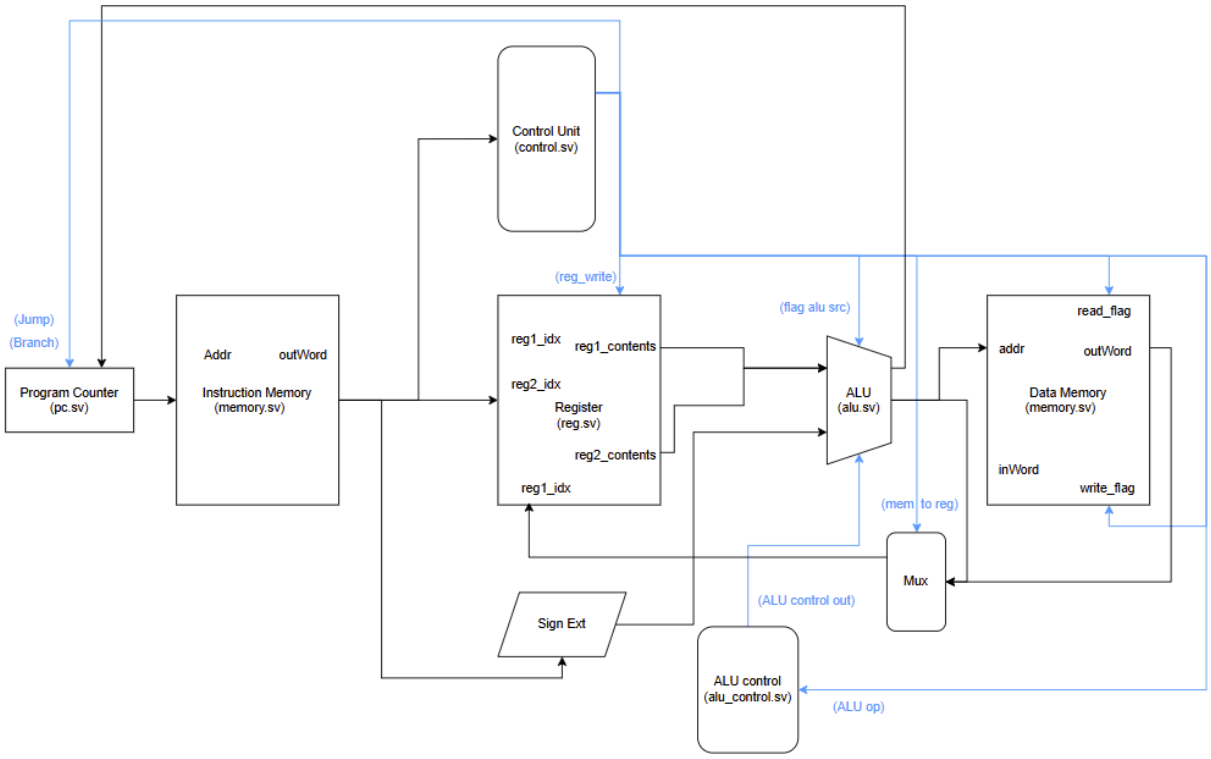
Herbert Acheampong, Nathan Fox, Dhiraj Patil, Osmond Wan

Table of Contents

Overview **3**
Components **4**
Program Counter 4
Instruction Memory 5
Registers 5
Control Unit 5
Arithmetic and Logical Unit (ALU) 5
ALU Control Unit 5
Data Memory 6
Datapath and Control Path **6**
Datapath 6
Control Path 8
Control Unit 8
R-type Instructions 10
I-type: LW 10
I-type: SW 10
I-type: ADDI 10
I-type: BEQ 10
I-type: BNE 11
J-type Instruction 11
ALU Control Unit 11
Control Unit (“alu_op” Flag) ALU Control Unit 11
ALU Control Unit {alu_op + funct} ALU 12
Simulations **13**
CPU Waveform 13
ALU Waveform 14
ALU Control Waveform 15
Control Waveform 16
Data Memory Waveform 17
Instruction Memory Waveform 17
Program Counter Waveform 18
Register File Waveform 18
Instruction/Data Memory and Registers **19**
How to Run the Simulation and Hardware **20**
Improvements Since the Previous Submission **21**
Work Distribution **22**

Overview

The following report gives an generalized idea on our approach to create and design a 16-bit processor, single cycle and non-pipelined, in system verilog. This processor design can perform a few operations on the R, I and J type of instructions. The processor follows all of the process sequence (fetch, decode, execute, memory, and writeback) but is able to complete all the phases in a single clock cycle for each instruction.



Components

Component	File
Program Counter	pc.sv
Jump address calculation	pc.sv
Program counter offset addition	pc.sv
Branch MUX	pc.sv
Jump MUX	pc.sv
Instruction Memory	memory.sv
Register File	reg.sv
Sign Extension Unit(s)	pc.sv/alu.sv
Control Unit	control.sv
ALUSrc MUX	alu.sv
ALU Control Unit	alu_control.sv
ALU	alu.sv
Data Memory	memory.sv
MemtoReg MUX	cpu.sv

Program Counter

The program counter manages the current instruction and switches to next instructions similar to a MIPS processor. Upon each clock cycle it calculates PC + 2 value as the default next address to move to the next instruction. Our implementation also handles the branching multiplexers where it produces the branch address by sign extending a 4 bit immediate value from the instruction, shifts it by 1 and adds it to the next address value. This lets it support both branch equal and branch not equal conditions based on the flag_branch_select and the ALU zero flag. To handle jumps we have included the jump multiplexer implementation, where it calculates the jump target by shifting the instructions 12 bit jump address to the left by 1 bit and adding it to the next address, and based on the control unit signals, it selects the next output of the program counter between the branch, jump and the next address.

Instruction Memory

The instruction memory serves the process of storing the instructions. It combines two 8 bit word values from consecutive memory locations to form a complete 16 bit instruction. It utilizes the big-endian byte ordering where the most significant byte comes from addr value and the least significant byte comes from addr + 1. We also store the hexadecimal values which help us to represent the actual instructions in machine code.

Registers

The register module stores the incoming instructions from instruction memory into two register indexes which store the rs and rt values respectively. The rd value is stored in the register index 1. The always_comb block allows the register to continuously read and output the data stored in respective registers based on the indices. This allows other components to access the registers at any time. The code also handles write operations on the registers on the falling edge of the clock. When the reg_write is activated, it writes the rd value to the register index 1.

Control Unit

The Control Unit is like the brain of the processor. It reads the instruction opcode and then sends out signals to control the rest of the components. Based on the instruction, it decides what needs to happen next, whether it's performing an operation, accessing memory, or doing a jump. It makes sure everything works together smoothly while the processor is running.

Arithmetic and Logical Unit (ALU)

The ALU is responsible for doing all the math and logic operations. It takes in two inputs and performs operations like addition, subtraction, AND, OR, and shifting. The results are used for things like calculating addresses or comparing values to make decisions in the program.

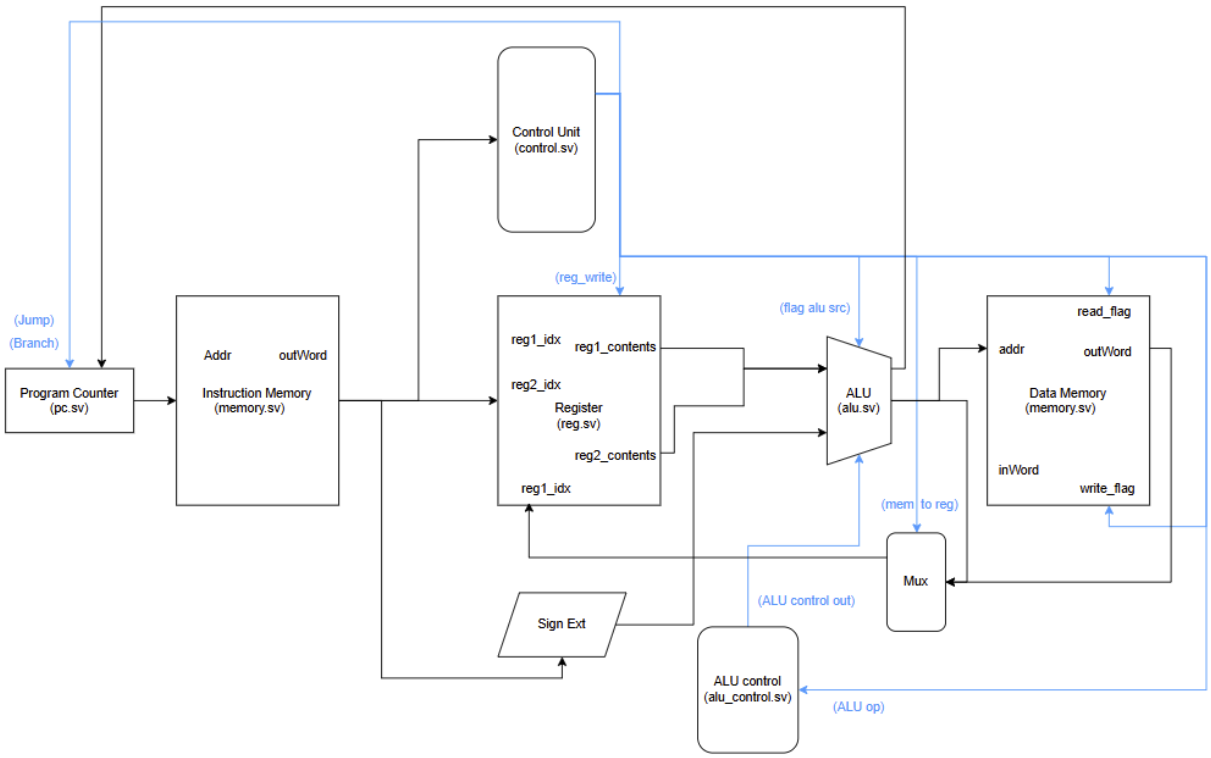
ALU Control Unit

The ALU Control Unit tells the ALU exactly what operation to perform. It looks at the instruction and decides whether the ALU should add, subtract, or do some other calculation. Without the ALU Control Unit, the ALU wouldn't know what to do with the inputs.

Data Memory

The data memory’s primary object is to serve as a read/write memory unit for the processor during execution. The always_comb block deals with read operations whereas the always @ with negative edge clock handles the write operations on the falling edge of the clock.

Datapath and Control Path



Datapath

1. The instruction address for the current cycle is taken from the **Program Counter**'s next instruction address output and passed into the Instruction Memory. The address is also fed back into the Program Counter's current instruction address input. The next instruction address was computed in the previous cycle and stored in the pc register.

2. The address is passed into the **Instruction Memory** to fetch the instruction in which the instruction memory address corresponds to. The full fetched instruction is passed back into the program counter to calculate its next address. The fetched instruction is broken up and passed into the Control Unit, Register File, and ALU control unit (it is also passed into the ALU because the sign extension unit logic was broken up and placed in the modules it was needed).

3. The **Control Unit** takes the opcode portion of the fetched instruction and sets the output control flags. These flags are passed to the Program Counter, ALU Control Unit, ALU, Data Memory, MemToRegMUX, and Register File.

4. The **Register File** takes bits [11:8] and [7:4] of the fetched instruction as the index for both registers. It takes those indices, reads the internal register array, and returns the 16-bit data in its two reg1_contents (R[rt]) and reg2_contents (R[rs]) outputs. These outputs are passed to the ALU, Data Memory, and MemToReg MUX.

5. The **ALU Control Unit** receives inputs from the Control Unit and the Instruction Memory (fetched instruction). Based on these inputs, the ALU Control Unit outputs a 3-bit code to the ALU to indicate which ALU operation the ALU should perform on its reg1_contents and reg2_contents inputs.

6. The **ALU** takes the flags from the Control Unit and ALU Control Unit and performs an operation on the outputs from the Register File. If the result from that operation is 0, the output zero_flag is set. The zero_flag output is passed to the Program Counter. The result from the operation is passed to the Data Memory and the MemToReg MUX.

7. The **Data Memory** uses the Control Unit flags that were passed into it to determine what to do with the ALU result and Register File output (R[rt] data). It will either read data from the memory address calculated from the ALU result, or write the Register File output data to the memory address calculated from the ALU result. For data memory reads, the Data Memory will have a meaningful output - the data read from the memory address. For data memory writes, the output will not be meaningful. Either way, the output is passed to the MemToRegMUX.

8. The **MemToReg MUX** takes the output flag from the Control Unit flag and chooses between the ALU result output or the Data Memory output to pass back to the Register File. The MUX chooses which output is being written back to R[rt/rd] if the instruction indicates that a write back to a register is necessary.

9. The **Register File** receives the output from the MemToReg MUX and the Control Unit output flag. Based on the flag, the Register File will take the MemToReg MUX output and either do nothing with it or write it to R[rt/rd] in the internal register array.

Control Path

Control Unit

1. The control path starts at the Control Unit where it receives the opcode, bits [15:12], from the fetched instruction output of the Instruction Memory. Based on the opcode, certain flags are set to make sure the rest of the datapath performs the correct operation on the data and produces a proper output.

   Below is a table explaining all the different control flags and what they are supposed to represent. Then afterwards are screenshots of which flags are set or left unset when a specific instruction's opcode is passed into the Control Unit.

Flag Name	Component it is Used in	Use Case	
jump	Program Counter	0	Program Counter will not choose calculated jump address
		1	Program Counter sets next instruction address to calculated jump address
branch	Program Counter	0	Program Counter will not choose calculated branch address
		1	Program Counter will set next instruction address to calculated branch address
branch_select	Program Counter	0	Use BEQ logic to select between pc+2 or calculated branch address
		1	Use BNE logic to select between pc+2 or calculated branch address
mem_read	Data Memory	0	Data Memory will not read data from array
		1	Data Memory reads data from array and returns a significant output

mem_to_reg	MemToReg MUX	0	Selects ALU result to pass back to Register File to write
		1	Selects Data Memory output to pass back to Register File to write
alu_op	ALU Control Unit	Used in conjunction with “funct” to determine which ALU operation to perform. Refer to the next section about the ALU Control Unit.	
mem_write	Data Memory	0	Data Memory will not write data to array
		1	Data Memory will write data to array on clock edge
aluSrc	ALU	0	ALU selects register 1 (R[rt/rd]) contents to use in operation
		1	ALU selects sign-extended immediate to use in operation
reg_write	Register File	0	Register File will not write anything
		1	Register File will write the data to R[rt/rd]

R-type Instructions

```
// R-type (opcode = 0000: add, sub, sll, and)
4'b0000: begin
  alu_op        = 2'b10;   // R-type => 10
  aluSrc        = 1'b0;    // ALU operand from register - use input1
  reg_write     = 1'b1;    // Enable write to register - write to rt/rd
end
```

I-type: LW

```
// lw (opcode = 0001)
4'b0001: begin
  alu_op        = 2'b00;    // I-type (non branch) => 00
  aluSrc        = 1'b1;     // Use immediate as second ALU operand
  mem_read      = 1'b1;     // Enable memory read
  reg_write     = 1'b1;     // Write loaded data to register
  mem_to_reg    = 1'b1;     // Choose memory unit output to write back
end
```

I-type: SW

```
// sw (opcode = 0010)
4'b0010: begin
  alu_op        = 2'b00;    // I-type (non branch) => 00
  aluSrc        = 1'b1;     // Use immediate as second ALU operand
  mem_write = 1'b1;         // Enable memory write
end
```

I-type: ADDI

```
// addi (opcode = 0011)
4'b0011: begin
  alu_op    = 2'b00;   // I-type (non branch) => 00
  aluSrc    = 1'b1;    // Use immediate as second ALU operand
  reg_write = 1'b1;    // Write result to register
end
```

I-type: BEQ

```
// beq (opcode = 0100)
4'b0100: begin
  alu_op        = 2'b01;    // I-type (branch) => 01
  branch        = 1'b1;     // Branch instruction
  branch_select = 1'b0;     // 0 => BEQ path (branch if Zero == 1)
  aluSrc        = 1'b0;     // ALU operand from register - use input1
end
```

I-type: BNE

```
// bne (opcode = 0101)
4'b0101: begin
  alu_op        = 2'b01;    // I-type (branch) => 01
  branch        = 1'b1;     // Branch instruction
  branch_select = 1'b1;     // 1 => BNE path (branch if Zero == 0)
  aluSrc        = 1'b0;     // ALU operand from register - use input1
end
```

J-type Instruction

```
// jump (opcode = 0110)
4'b0110: begin
  jump = 1'b1;  // Jump instruction
end
```

ALU Control Unit

2. The ALU Control Unit receives the “alu_op” output from the Control Unit and also receives the last 4 bits of the fetched instruction output from the Instruction Memory as the “function code” for an R-type instruction. It concatenates the alu_op and the funct to make a 6-bit code that determines which operation the ALU will perform.

   The four ALU operations (ADD, SUB, SLL, AND) are represented by a 3-bit code. That 3-bit code is what the ALU Control Unit determines and is the output of the unit which ends up being passed into the ALU.

   Below are tables which detail how the 2-bit “alu_op” flag travels from the Control Unit, to the ALU Control Unit, and then produces a 3-bit code that the ALU uses to determine which operation to perform.

Control Unit (“alu_op” Flag) → ALU Control Unit

Instruction	Instruction Type	alu_op
ADD	R	10
SUB	R	10
SLL	R	10

AND	R	10
LW	I	00
SW	I	00
ADDI	I	00
BEQ	I	01
BNE	I	01
JMP	J	xx

ALU Control Unit {alu_op + funct} → ALU			
Instruction	**alu_op + funct**	**ALU Control Output**	**ALU Operation**
ADD	10 0000	010	Addition
SUB	10 0001	110	Subtraction
SLL	10 0010	011	Logical Left Shift
AND	10 0011	000	Bitwise AND
LW	00 xxxx	010	Addition
SW	00 xxxx	010	Addition
ADDI	00 xxxx	010	Addition
BEQ	01 xxxx	110	Subtraction
BNE	01 xxxx	110	Subtraction
JMP*	xx xxxx	xxx	xxx

** JMP is essentially a "don't care" in theory, but in our implementation JMP corresponds to an addition operation by the ALU. The result will not be used anywhere or be significant because the "mem_to_reg" flag will be set to 0 so the ALU result will not write.*

Simulations

CPU Waveform

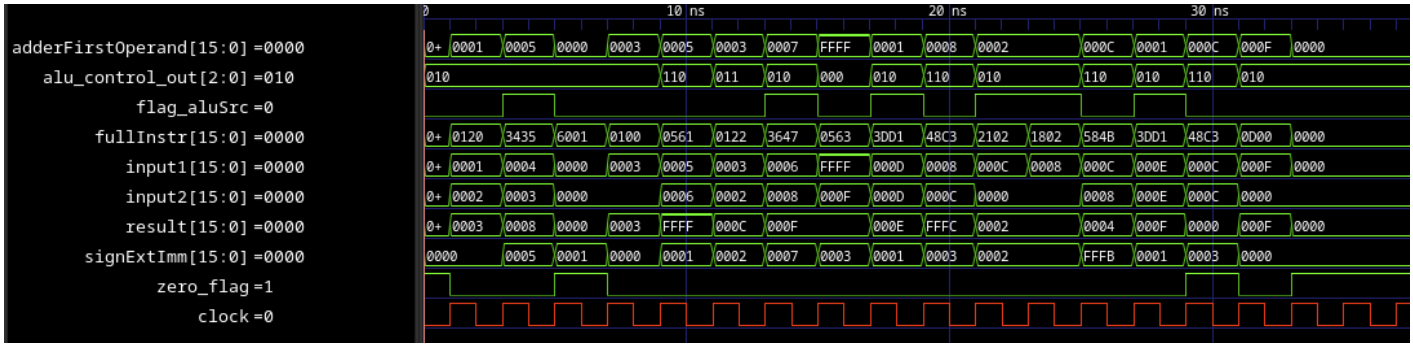
Inputs:

- **clock -** The clock that alternates between 0 and 1. On the positive edge of the clock, the pc register in cpu.sv will be updated with the current instruction address and that will trigger the instruction memory to fetch the instruction from memory. This causes a chain reaction for all the other combinational logic modules that will eventually compute a result. Also on the positive edge of the clock, the previous instruction's register or data memory writes occur. The clocked procedural blocks always run first before any combinational logic is run so there are no race conditions.

Outputs:

- **rd_before -** The contents of R[rt/rd] before the instruction fully executes. For jump instructions specifically, the output will always be 0x0000 in this implementation. This is due to our use of the default case in case statements.
- **rd_after -** The contents of R[rt/rd] after the instruction executes. This will be different from the "rd_before" output whenever the instruction writes back to a register. For instructions that do not have a write back stage, "rd_after" will be the same as "rd_before". For jump instructions specifically, the output will always be 0x0000 in this implementation. This is due to our use of the default case in case statements.
- **current_pc -** The current cycle's instruction address. This is just for clarity and to see the program counter increment or decrement depending on the instruction.
- **current_instr -** The current cycle's fetched instruction from the Instruction Memory output. It is to check which instruction is running this current cycle.

ALU Waveform



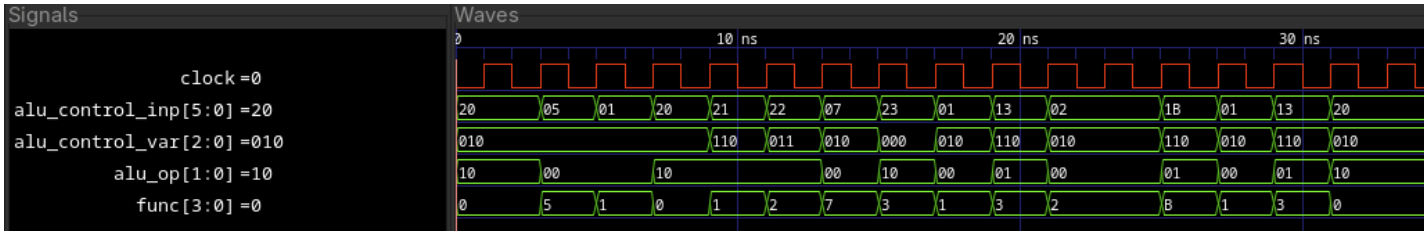
Inputs:

- **Input1 -** Used to store the 16 bit instruction content read from register 1 (first operand)
- **Input2 -** Used to store the 16 bit instruction content read from register 2 (second operand)
- **fullInstr -** Stores the 16 bit decoded instruction from the instruction memory. It's used to extract the 4-bit immediate value.
- **flag_aluSrc -** Determines whether the ALU uses the immediate value for the instruction or a register value as the second operand. (1) means alu uses immediate value and (0) means the alu uses value from a register.
- **alu_control_out -** It is a 3-bit logic used to dictate which operation to perform. (010) for ADD, (110) for SUB, (011) for SLL (shift), (000) for AND.
- **adderFirstOperand -** It is used to store and substitute the values of either input1 or signExt based on the aluSrc flag. If flag is (1) sign extension immediate value is used in adder, in case of (0) input1 is used in adder.
- **signExtImm -** used to perform sign extension on the 4-bit value to create a 16-bit by copying the sign bit [3] across the upper 12 bits.

Outputs:

- **result -** Used to store the result of the operation performed on the two inputs.
- **zero_flag -** The zero flag is set to (1) when the result of the operation is equal to 0 other is set to (1).

ALU Control Waveform



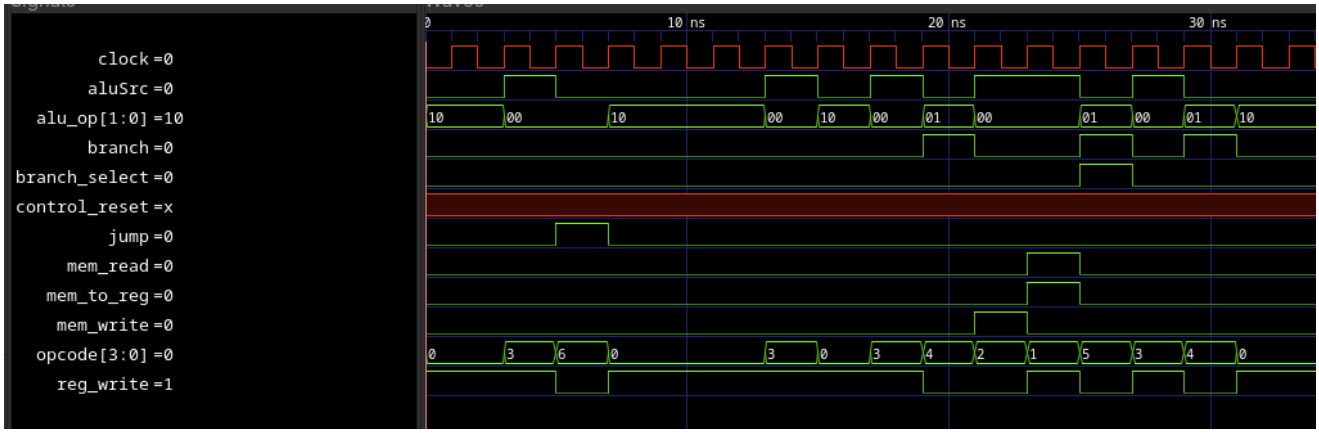
Input:

- **Alu_control_inp** - This combination of two inputs, alu for 2 bits and func for 4 bits, so together 6 bits. Used to select the correct ALU operation. The alu_op part defines the general ALU operation, while the func part helps specify the specific instruction, such as addition.
- **Alu_op** - Signal is generated by the control unit and specifies the type of operation the ALU should perform (10 for R-type, 00 for I-type (non-branch), and 01 for I-type (branch)). (2 bit instruction)
- **Func** - Specifies the detailed operation for R-type instructions. It determines whether the ALU performs an addition (0000), subtraction (0001), shift left (0010), AND (0011), etc. (4 bit instruction)

Output:

- **Alu_control_var** - Specifies the exact operation the ALU should perform. (010 for ADD, 110 for SUB, 011 for SLL (shift left), and 000 for AND) (3 bit instruction)

Control Waveform



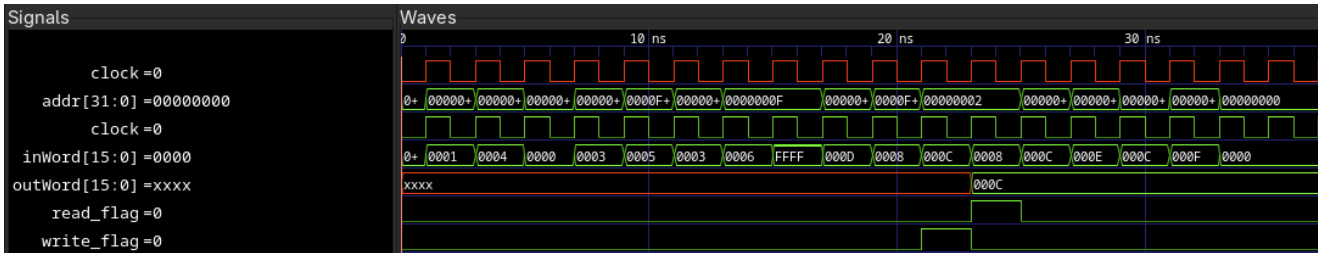
Inputs:

- **Opcode** - Determines whether the operation is a load, store, arithmetic operation, or branch. (4 bit instruction)
- **Control_reset** - When signal is high (1), it resets all the control outputs to their default values, regardless of the opcode.

Outputs:

- **Jump** - Controls whether a jump operation should occur. (1) enables the jump else it's (0).
- **AluSrc** - Determines whether the ALU uses the immediate value for the instruction or a register value as the second operand. (1) means alu uses immediate value and (0) means the alu uses value from a register.
- **Alu_op** - Determines which operation the Alu will perform. (10 for R-type, 00 for I-type (non-branch), and 01 for I-type (branch)).
- **Branch** - Signal is set to (1) for branch operations like BEQ and BNE.
- **Branch_select** - Used to differentiate between BEQ and BNE. (0) selects the BEQ path, and (1) selects the BNE path.
- **Mem_read** - Signal is used for load instruction. It enables reading from memory when set to (1). If (0), memory read operations are disabled.
- **Mem_to_reg** - Signal controls whether data coming from the memory should be written into a register. For load instructions, it is set to (1) to allow data from memory to be written into the register file.
- **Mem_write** - Signal is set to (1) for store instructions like sw. It enables writing data to memory.
- **Reg_write** - Signal indicates whether data should be written back to the register file. It is set to 1 for R-type instructions, lw, and addi instructions, enabling the write-back to registers.

Data Memory Waveform



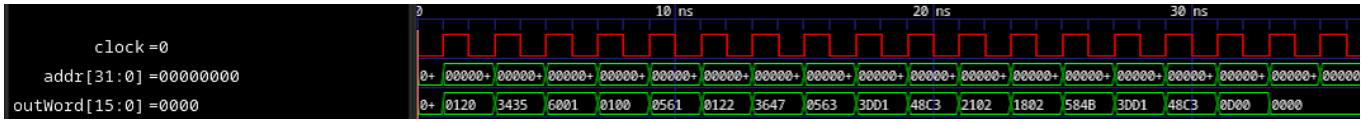
Input:

- **InWord** - This is the word that is to be written to memory when the optype is write. It contains the 16 bit data that will be stored in the memory at the specified address.
- **Addr** - This is the memory address to access. It specifies the location in memory where data is read or written.
- **Optype** - This is the operation typeread or write. I tell the memory whether to read from the memory or write to it.

Output:

- **Outward** - This is the word that is read from memory or returned after a write operation. During read, it contains data read from memory. During write, it contains the same data that was written to inWord.

Instruction Memory Waveform



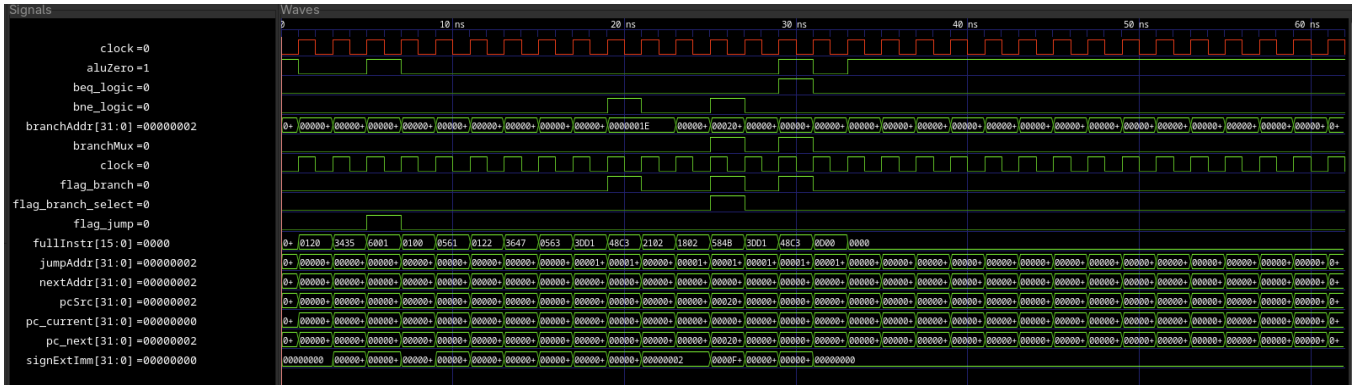
Inputs:

- **Addr -** Used to store the significant byte address value to utilize as the memory address to access the instruction from instruction memory.

Outputs:

- **outWord** - It stores the 16 bit instruction value which is read from memory for a specific addr value. It is formed by the combination of two bytes, significant (addr) and least significant (addr + 1)

Program Counter Waveform



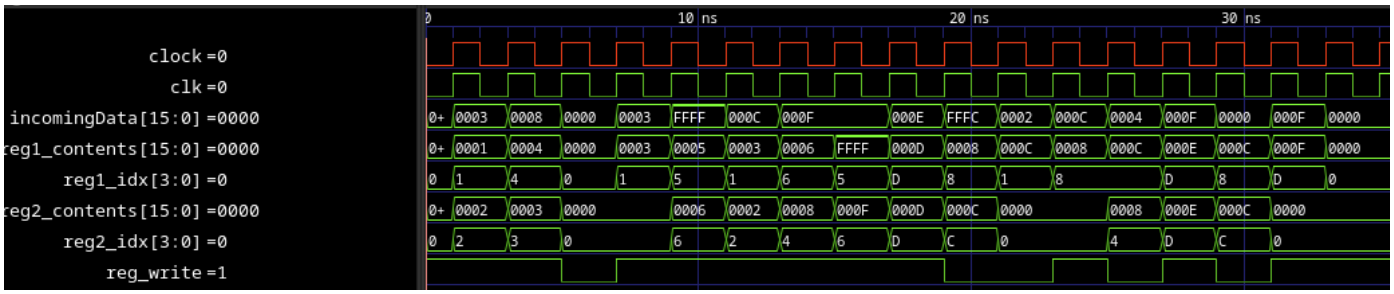
Input:

- **Pc_current** - The current value of the program counter, representing the address of the current instruction in memory
- **Flag_branch** - Flag is set by the control unit. (1) if the current instruction is a branch instruction.
- **Flag_branch_select** - Flag controls which type of branch to select. If (0), it selects the beg logic, and if (1), it selects the bne logic.
- **Aluzero** - 1 bit instruction which signal is generated by the Alu. It is (1) if the ALU's result was zero.
- **Flag_jump** - Stores a 1 bit instruction. This flag, when set to (1), shows that the current instruction is a jump instruction.
- **FullInstr** - Stores a 16 bit instruction which is the decoded instruction from the instruction memory. It contains information like branch offset and the jump address.

Output:

- **Pc_next** -This is the next value for the program counter. I determine the address of the next instruction to fetch, which could either be the next sequential instruction or a new address after jump or branch.

Register File Waveform



Inputs:

- **incomingData -** Used to store the instruction data which is written to registers. It is only used when the reg_write flag is set to (1) and is on a negative edge cycle.
- **Reg_write -** Signal indicates whether data should be written back to the register file. It is set to 1 for R-type instructions, lw, and addi instructions, enabling the write-back to registers.
- **Reg1_idx** - register variable used to store the index of the first register which is utilized for both read and write.
- **Reg2_idx -** register variable used to store the index of the second register and is only utilized for read.

Outputs:

- **Reg1_contents** - data that was read from the register index 1.
- **Reg2_contents** - data that was read from the register index 2.

Instruction/Data Memory and Registers

Instruction Memory (Read-Only): Instruction memory is typically read-only, which means you hard-code the values in the begin block. You manually specify the instruction values at specific memory locations, ensuring that when the processor reads from memory, it fetches the instructions from the hardcoded values.

Data Memory (Read-Write): Data memory, on the other hand, can both read and write values. To set a value in data memory, you pass the value inword that you want to store into a specific address in the data memory. The write operation is controlled by a write flag, which enables writing to memory. When you want to write, you must set the write flag to enable writing, and the data will be stored at the specified address.

Registers (Read-Write): Processors are used to hold temporary data. To set the register values, you pass the value inword to a specific register address, which is determined by the RegWrite signal and the register number. The write operation occurs when the RegWrite flag is enabled, indicating that a write operation is allowed. Registers are updated on the positive edge of the clock, ensuring synchronization with the system's timing. Once updated, the values are held in the registers until a new write operation takes place, allowing the data to persist until changed again.

Synchronization: The actual write operation happens on the positive edge of the clock. This ensures proper synchronization with the system's clock and prevents accidental data writes during the wrong cycle. The flag for reading read flag should be disabled when writing data to prevent conflicting operations.

How to Run the Simulation and Hardware

Things to set up before running your own test cases and simulations:

1. Go into reg.sv and change any initial register values in the “registers” array if you so desire.
2. Go into memory.sv, in the data_memory module, change any initial data memory if you so desire.
3. Go into memory.sv, in the instruction_memory module module, replace the contents of the “instruction_mem” array with your own instruction in hex or binary. You will have to format it to the desired type of data you want.
    a. Remember that it should be byte aligned so only half an instruction can be placed in each instruction memory location.
    b. Put a NOP as the first instruction because cpu_tb.sv starts on a low clock so it will not run the first instruction properly since when the clock turns high, the next instruction will be fetched and executed.

How to set up the hardware:

1. Get a working usb-a to micro-usb cable and plug it into the computer running Vivado
2. In Vivado, run the Synthesis and Implementation and then generate the bitstream. This gets the Verilog RTL into a readable state for the FPGA board
3. Open the hardware manager and auto connect to the board. Vivado should say “There are no debug cores”, but that is irrelevant. Just program the board with “program device”
4. After you program the device, hit the center button U18 and it’ll trigger a change in the clock, both posedge and negedge

Improvements Since the Previous Submission

During the first submission, each team member worked on their own component, and while each component worked separately, there were issues when it came time to connect them. Due to a lack of communication, there was some overlap, which prevented the system from working as intended. Since then, not only did we improve our communication, we successfully connected all the components, and everything is now working properly on hardware.

Improvements on code:

- Fixed a lot of bugs and logic to reflect the specifications of the document.
- Moved away from an always block main cpu module and now the main module uses a mostly wire driven implementation instead.
- It's actually working this time!
- Moved continuous in sub-modules inside the always_comb combinational blocks.
- Now using clock driven register writes and data memory writes.
- Moved away from using a generalized register module and updated it to reflect the 2 input register file we see on datapaths.
- Stopped modifying local variables using blocking assignments in the main module in hopes of the combinational blocks triggering. Instead, we directly hooked up the local variable wires to each module so that when one local variable updates, the connecting modules will automatically trigger.

Work Distribution

Dhiraj Patil	ALU - Worked on designing subtraction, shift left, load word and add immediate modules of the logical unit. Created separate functional modules for each ALU operation. This inevitably added redundancy to the code and made it harder to integrate, which was later optimized with a better implementation. Control Unit - Helped in the implementation of the control unit module maintaining absolute decoding of opcode operations inherited from instructions to control unit flags utilized by other components. ALU Control - Wrote the initial implementation of the ALU Control. Modified and updated later with the defined control flags. Misc. - Described a few components of the processor design on the report. - Wrote briefly on the workings of the simulations of each component.
Herbert Acheampong	Control Unit - Worked on implementing the control unit, which decodes the opcode from instruction memory and generates control signals to guide its behavior.
Nathan Fox	ALU, Registers, Memory, Hardware - Implemented the adder - Implemented the and - Implemented memory - Implemented registers - Assisted with final testbench and reworked code as necessary - Ran the synthesis and verified the project worked on hardware - Created makefile to easily compile and test the code
Osmond Wan	Program Counter - Worked on implementation of the program counter functionality. - The implementation includes incrementing the program counter, calculating the branch address from branch offset, calculating the branch MUX selector bit (for both BNE and BEQ), and the MUX logic to choose between the pc+2, branch address, or jump address as the next instruction address to output. ALU - Revised the alu.sv file to condense all the operations into a single

	module. - Added all the extra logic for the ALU including the sign extension specifically for the ALU, the aluSrc MUX that chooses between an immediate and R[rt/rd] data, and setting the output “zero” flag. ALU Control Unit - Defined the meaning of the control flags - which 6-bit (alu_op + funct) combination corresponds to which 3-bit ALU operation. Control Unit - Checked over code and fixed any irregular logic or unneeded inputs/outputs. Data Memory & Register File - Revised to make writes clocked CPU - Wrote the final working implementation of the main module that holds the instantiation of all the sub-modules and the logic that produces the outputs. - Made the set of instructions that are included as a comprehensive test for all the instructions of the cpu. - Using the tests, simulated and tested to fix any bugs or logic in any module that need to be fixed to get the cpu working. Misc. - Wrote basic and complex tests for the cpu. The complex tests are the set of instructions that are included with the submission. They test all instructions and have comments about expected inputs and outputs. - Wrote almost all the comprehensive comments you see in every module. - Wrote “Datapath and Control Path” section of the report. - Cowrote “How to Run the Simulation and Hardware” section of the report. - Wrote “CPU Waveform” section of the report. - Wrote a script to turn assembly instructions to byte sized hex format to help in testing.