Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Computer Architecture – ENCS4370

Project 2

Designing Simple RISC Processor – Multi-Cycle Approach

Group Members:

Doaa Hatu 1211088

Jana Qatousa 1210331

Shiyar DarMousa 1210766

Instructor:

Aziz Qaroush

Section: 1

Date: 13/06/2024

Abstract

This project report details the design and implementation of a multicycle processor architecture, which executes instructions over multiple cycles to optimize resource utilization and flexibility. The processor's architecture includes essential components such as the Program Counter (PC), Instruction Memory, Register File, ALU, Data Memory, and various multiplexers. These components work together under the control of a centralized control unit that generates the necessary control signals based on the current instruction and state. The multicycle approach breaks down instruction execution into fetch, decode, execute, memory access, and write-back stages, each occurring in a separate clock cycle, thereby efficiently utilizing hardware resources. Through this method, the processor achieves a balance between complexity and performance, demonstrating advantages in resource optimization and efficient instruction processing. The report provides a comprehensive overview of the processor's datapath and control logic, validated through simulation and testing for correctness and efficiency

Contents

Table of Figures

List of Tables

Design and Implementation

1. Design specification

1.1.Motivation

This project is inspired by MIPS architecture this design is part of the RISC (Reduced Instruction Set Computer) architecture family and is specifically based on the MIPS32. Our goal is to create a set of instructions for a Multicycle processor. The key features of this instruction set include:

1. The instruction size and the word size is 16 bits
2. 8 16-bit general-purpose registers: from R0 to R7.
3. R0 is hardwired to zero. Any attempt to write it will be discarded.
4. 16-bit special purpose register for the program counter (PC)
5. Four instruction types (R-type, I-type, J-type, and S-type).
6. Separate data and instruction memories
7. Byte addressable memory
8. Little endian byte ordering
9. You need to generate the required signals from the ALU to calculate the condition branch outcome (taken/ not taken). These signals might include zero, carry, overflow, etc.

1.2.Instruction formats

The Instruction Set Architecture (ISA) has four instruction types, namely, R-type, I-type, J-type, and S-type. These four types have a common **opcode** field, which determines the specific operation of the instruction.

- **R-type**
  For register type instructions, it follows the format below:

Opcode - 4	Rd - 3	Rs1 - 3	Rs2 - 3	Unused - 3

  - 3-bit Rd: destination register
  - 3-bit Rs1: first source register
  - 3-bit Rs2: second source register
  - 3-bit unused

- **I-type**
  For immediate type instructions, it follows the format below:

Opcode - 4	Rd - 3	Rs1 - 3	M - 1	Immediate - 5

- 3-bit Rd: destination register
- 3-bit Rs1: first source register
- 5-bit immediate: unsigned for logic instructions, and signed otherwise.
- 1-bit mode: this is used with load and branch instructions, such that:
  For the load:
  0: LBs load byte with zero extension
  1: LBu load byte with sign extension
  For the branch:
  0: compare Rd with Rs1
  1: compare Rd with R0

  Note: Dr.Aziz allowed us to change the location for the mode bit, so it would make the implementation of the datapath simpler (less selection lines on the muxes of the register file inputs).

- **J-type**
  For jump type instructions, it includes the following instruction format:
  jmp L # Unconditional jump to the target L.
  call F # Call the function F. F is a label.
  # The return address is pushed on r7.

  The target address is calculated by concatenating the most significant 7-bit of the current PC with the 12-bit offset after multiplying offset by 2.

Opcode - 4	Jump Offset - 12

ret # return from a function.
# the next PC will be the value stored in r7

Opcode - 4	Unused - 12

- **S-type**
  For store instructions. This format supports only one instruction
  Sv rs, imm # M[rs] = imm

Opcode - 4	Rs - 3	Immediate - 9

1.3.Instruction set

The table below lists the instruction set in our processor design.

R-type			
Instruction	Meaning	Opcode	Mode
AND	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	0000	
ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	0001	
SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	0010	
I-type			
Instruction	Meaning	Opcode	Mode
ADDI	Reg(Rd) = Reg(Rs1) + Imm	0011	
ANDI	Reg(Rd) = Reg(Rs1) + Imm	0100	
LW	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0101	
LBu	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0110	0
LBs	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0110	1
SW	Mem(Reg(Rs1) + Imm) = Reg(Rd)	0111	
BGT	if (Reg(Rd) > Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1000	0
BGTZ	if (Reg(Rd) > Reg(0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1000	1
BLT	if (Reg(Rd) < Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1001	0
BLTZ	if (Reg(Rd) < Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1001	1
BEQ	if (Reg(Rd) == Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1010	0
BEQZ	if (Reg(Rd) == Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1010	1
BNE	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1011	0
BNEZ	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1011	1
J-type			
Instruction	Meaning	Opcode	Mode
JMP	Next PC = {PC[15:10], Immediate}	1100	
CALL	Next PC = {PC[15:10], Immediate} PC + 4 is saved on r7	1101	
RET	Next PC = r7	1110	
S-type			
Instruction	Meaning	Opcode	Mode
Sv	M[rs] = imm	1111	

Table 1: Instruction set table

2. Datapath Stages and Components

Through our examination to the datapath, we determined the necessary components for each of the 5 stages, which are used to complete the full design of the processor datapath.

2.1.IF (Instruction Fetch)

In the multicycle approach, the IF stage is responsible for retrieving the next instruction to executed. It begins with the PC supplying the instruction memory of the address of the instruction, which is stored in the instruction register to be used later in the decoding stage. The PC will be updated during this stage to have the address of the next instruction to be executed, or target address for jump and branch operations. Below, a detailed explanation of each component used in this stage.

- **4x1 Mux**
  A multiplexer is a crucial component in the computer datapath, responsible for selecting signals based on control inputs. During the fetch stage, a 4x1 multiplexer is used for Program Counter (PC) operations to determine the next address. Depending on the value of the selection line **PC_Selection**, the multiplexer selects one of four possible addresses: if **PC_Selection** is **00**, the PC is incremented by 1 (PC + 1); if **PC_Selection** is **01**, the PC is formed by concatenating the most significant 4 bits of the current PC with a 12-bit immediate value; if **PC_Selection** is **10**, the PC is set to the Branch Target Address (BTA), which is the sum of the current PC and the sign-extended immediate value; and if **PC_Selection** is **11**, the PC is set to the value stored in register R7.

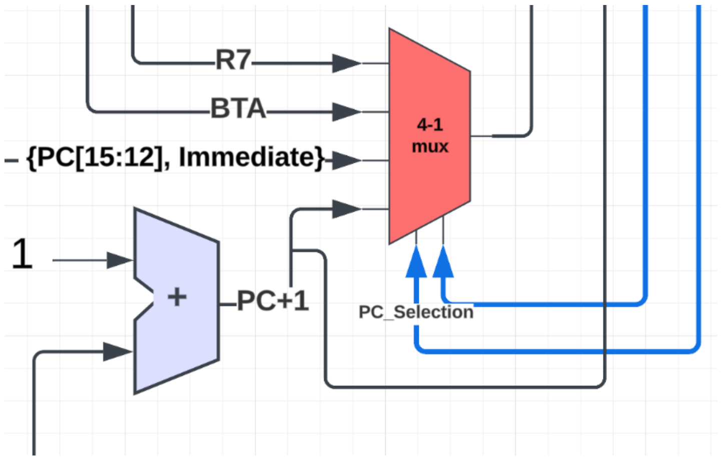


Figure 1: PC selection multiplexer

- **PC**
  The Program Counter (PC) is a fundamental register in the CPU's datapath, it keeps track of the address of the next instruction to be executed. It is essential for ensuring the sequential execution of instructions by the processor. At each clock cycle, the PC is updated to point to the next instruction, enabling the CPU to fetch, decode, and execute instructions in a coordinated manner. In cases of jumps, branches, or subroutine calls, the PC is updated to the appropriate address, allowing for non-sequential execution when necessary. Without the PC, the CPU would be unable to systematically progress through the instruction sequence, rendering the execution process chaotic and unmanageable.

- **Adder**
  The adder, as shown in figure 1, is used for adding the PC value to 1, in order to calculate the address of the next instruction.

- **Instruction Memory**
  In our processor design, memory is divided into **two** distinct segments: Instruction Memory and Data Memory. This separation is intentional to avoid conflicts that could occur from concurrent operations, such as fetching instructions while loading or storing data. Instruction Memory is dedicated to storing instructions and is read-only, as the Datapath does not require writing instructions. It operates on a straightforward logic for reading: it receives a **16-bit address** from the Program Counter (PC) and outputs a corresponding **16-bit instruction**, adhering to its word-addressable memory design. This ensures that the processor always knows what tasks to perform, allowing it to execute complex computations and tasks as specified by the user or program.

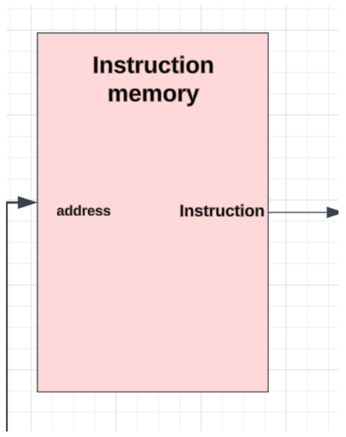


Figure 2: Instruction Memory

- **IR**
  The instruction register, which is used to store the instructed that has been fetched to be executed, in order to be used (decoded) in the decode stage.

2.2.ID (Instruction Decode)

The instruction decode is the second stage after the instruction fetch in the datapath. Its main job is to determine the operations to be done for the instruction fetched previously. The processor defines the type of the operation and needed operands for it, and extracts them from the register file, or takes the immediate values and extends them for I-type, then stores needed operands in temporary registers to be used later in the execution stage. This stage also generates some needed control signals.

- **Register file**

  The register file is a crucial component in the datapath, it contains several registers that serve as temporary storage for data being transferred between the processor's memory and its operational units. To read from the registers, the processor uses two designated registers, Rs1 and Rs2. The data from Rs1 is sent to Bus1, and data from Rs2 is sent to Bus2. The register labeled Rd in the register file is the designated location for writing data; when the control signal regWrite is set to 1, it indicates that data arriving on Bus_W would be stored into Rd.

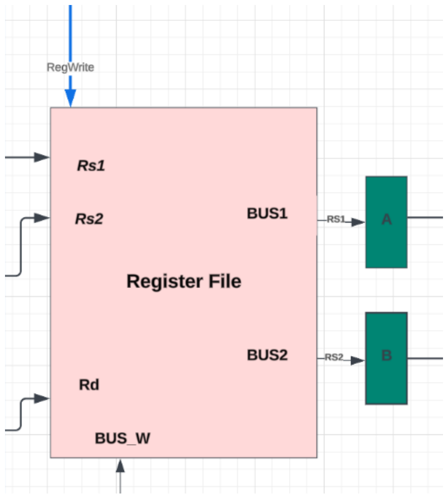


Figure 3: Register File

- **Extender**

  In the datapath, the immediate extender is a dedicated unit responsible for extending immediate values extracted from instructions. It takes a 5-bit (I-type) or 9-bit (S-type) immediate value (based on **select_type** control signal) and extends it to 16 bits. This extended immediate value is then available for arithmetic and logical operations within the datapath. The immediate extender ensures that immediate values are properly aligned and zero/sign-extended as needed for accurate computation and data manipulation in the processor.

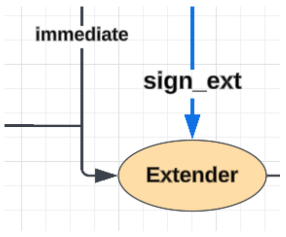


Figure 4: Immediate Extender

- **4x1 Mux**
  This mux is used for the address input of the first source register of the register file, it selects between Rs1, R0, and R7.

- **2x1 Mux**
  This mux is used twice for the address input of the register file, firstly for the second source register, it selects between Rs2, and Rd. It is also used for the destination register, it selects between Rd and R7.

- **A and B buffers**
  They are used and temporary buffers to store the outputs (Buses) of the register file, to be used later in the execution stage. (shown in figure 4).

2.3.EX (Execution)

In the execution stage of the datapath, the processor performs the specified operation on the operands prepared during the decode stage. This can involve arithmetic or logical operations executed by the Arithmetic Logic Unit (ALU), such as addition, subtraction, and bitwise operations. The results of these operations are then used to update registers, memory, or to determine the flow of control in the program. The execution stage is crucial for carrying out the core computations dictated by the instruction set, effectively transforming inputs into desired outputs.

- **ALU**

  The ALU is a crucial component of the CPU that handles arithmetic and logic operations, such as addition, subtraction, and logical comparisons.

  In our Datapath, the ALU has **two** main **16-bit** length inputs, the first input is taken from the A bus (resulting from the register file), and the second input could be either taken from bus B or the extended immediate.

  The ALU can do **4 operations** with these inputs, AND (a&b), ADD (a+b), SUB (a-b), RSUB (Reversed subtraction: b-a), and it results **3 flags** (N: negative, V: overflow, Z: zero), that are used later for comparison. An ALUop signal that is generated from the ALU control unit is used to determine the operation done in the ALU.

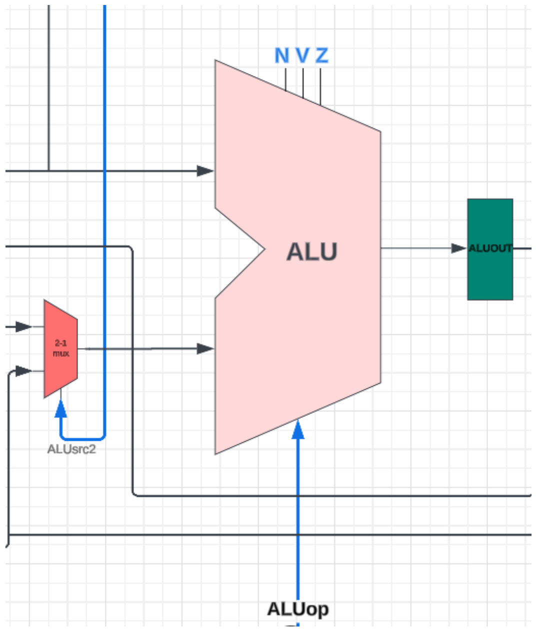


Figure 5: ALU

- **2x1 Mux**
  It has been used to select between the extended immediate and Bus B for which to be used as the ALU’s second source as shown in figure 5.

- **ALUOUT**
  A temporary buffer that is used to store the ALU result (shown in figure 5), in order to be used later in the MEM and WB stages.

2.4.MEM (Memory Access)

In the memory access stage of the datapath, the processor interacts with Data Memory to either read data from or write data to memory locations. If the instruction specifies a load operation, the processor reads the required data from memory and stores it in a register. Conversely, if a store operation is specified, the processor writes data from a register into the specified memory address. This stage ensures that the processor can access and modify data in memory, enabling dynamic data handling and manipulation essential for program execution.

- **Data Memory**

  Figure 6 illustrates the data memory, which acts as the storage space for the data the processor uses. This data memory is connected with the rest of the Datapath through an **address input** (could be calculated form the ALU for I-type, or taken form Rs1 for S-type), and **two data lines**: one to put **data into** storage (could be the extended immediate for S-type, or the value stored in bus B (Rd) for I-type), and another to get **data out**.
  Here's how it works:
  When the **MemRead** signal is **1**, the data memory finds the data at the specified address and sends it out. This is for reading data. When the **MemWrite** signal is **1**, the data memory takes in data from the line and keeps it at the address given. This is for writing data.

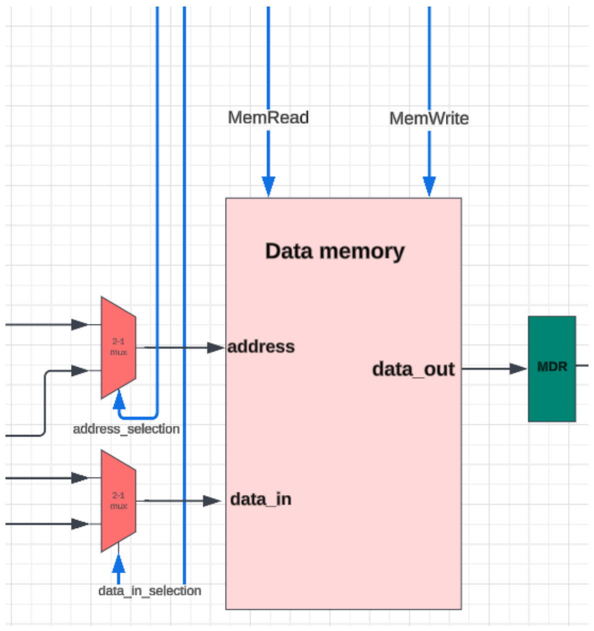


Figure 6: Data Memory

- **2x1 Mux**
  It was used twice in our design (shown in figure 6), one for address selection (either the ALU result for I-type, or the value on bus A – Rs1 for S-type), and the other for data_in selection (either the extended immediate for S-type, or the value on bus B – Rd for I-type)

- **MDR**
  A temporary buffer that is used to store the data have been read from memory, to be written back to the register file. (shown in figure 6).

- **Extender**
  This extender is used to determine what type of load to perform (**load byte or load word**), it has **two** control signals (**enable and selection**), the enable signal is used to enable extension, or to determine when extension should be performed (extend for LBu and LBs, do not for LW), and the selection control determines the type of extension (**zero or signed**).

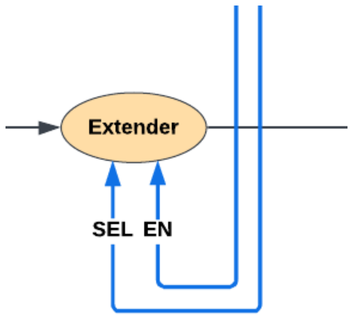


Figure 7: Second Extender

2.5.WB (Write Back)

In the write-back stage of the datapath, the processor updates the register file with the results of the executed instruction. This stage ensures that the outcome of arithmetic operations, memory reads, or any other data-generating instruction is written back to the appropriate register. This step is crucial for maintaining accurate and up-to-date data within the processor, allowing subsequent instructions to utilize the latest computed values, thereby ensuring correct and efficient program execution.

- **4x1 Mux**
  The mux is used to select which data to be written back to the register file, it is controlled by a signal generated from the control unit **(MemReg).** It selects between **3** options, either the **extender result** from data memory for load operations in **I-type (on Rd)**, or the **ALU result for R-type (on Rd)**, or the value of **PC + 1 for J-type (CALL) (on R7)**.

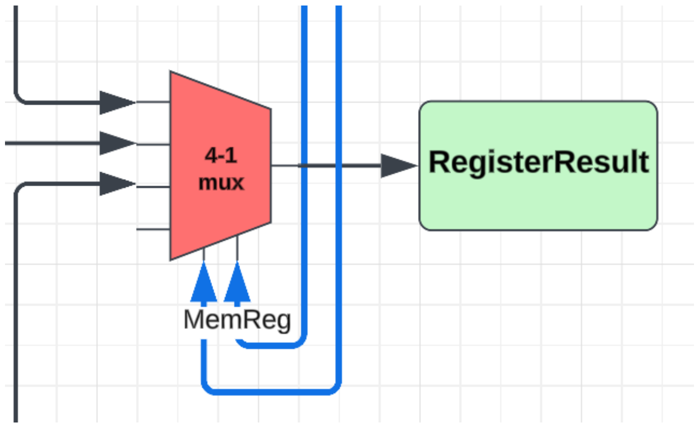


Figure 8: Write Back stage

Note: **the RegisterResult is a label for a wire to write back to the register file, we used a label instead of the wire to enhance readability of the datapath.**

3. Control path

3.1.Control Unit

The main control unit is an essential part in our datapath, it generates most of the control signals needed during the instruction's journey in the datapath stages.

3.1.1. Control signals and their effect

Signal Name	Value	Effect
IRWrite	0	None
	1	The output of memory is written to the IR
sign_ext	0	The immediate is zero-extended
	1	The immediate is sign-extended
select_type	0	Extension for I-type – by 11 bits
	1	Extension for S-type – by 7 bits
Regwrite	0	None
	1	The general purpose register selected by the write register number is written with the value of the write data input
PC_selection	00	The value of R7 (for RET) is chosen to be the next PC
	01	The value of the branch target address (extended immediate + PC value) is chosen to be the next PC
	10	The value of the concatenation of the most significant 4 bits of the PC with the immediate value (for jumps) is chosen to be the next PC
	11	The next PC is PC + 1
RegSrc1	00	The register first operand is in Rs1 field
	01	The register first operand is in R0 field
	10	The register first operand is in R7 field
RegSrc2	0	The register second operand is in Rd field
	1	The register second operand is in Rs2 field
RegDest	0	The destination register is Rd
	1	The destination register is R7
ALUSrc2	0	The second ALU input comes from register B
	1	The second ALU input is the extended immediate
address_selection	0	The input to the address of the data memory comes from register A
	1	The input to the address of the data memory is ALU result (comes from the ALUOUT register)
data_in_selection	0	Data written into the memory comes from register B
	1	Data written into the memory is the extended immediate
MemRead	0	Reading from memory is disabled
	1	Reading from memory is enabled for load operations
MemWrite	0	Writing to memory is disabled
	1	Writing to memory is enabled for store operations
EN	0	disable extension for second extender when the instruction is LW
	1	Enable extension for second extender when the instruction is LBu or LBs

SEL	0	Zero extension
	1	Sign extension
MemReg	00	Data to be written back to the register file is PC+1
	01	Data to be written back to the register file is the data out from memory (comes from the MDR register)
	10	Data to be written back to the register file is ALU result (comes from the ALUOUT register)

Table 2: Control Signals effect

3.1.2. State Diagram

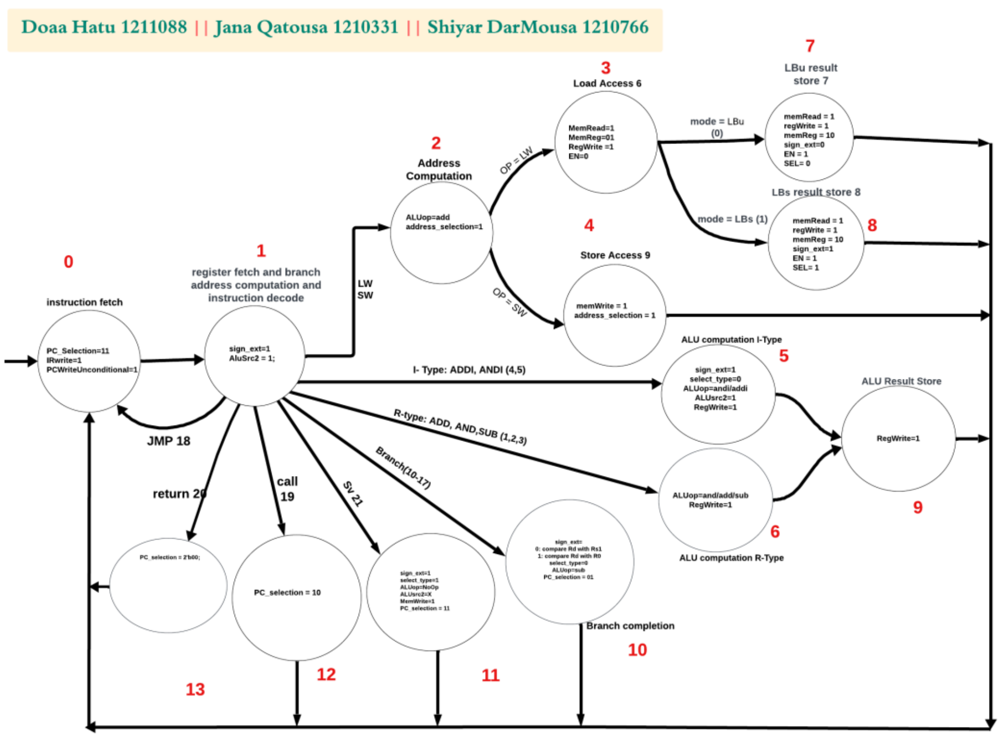


Figure 9: Conrtol Unit State Diagram

Explanation:

The first two states of any instruction cycle, the instruction fetch and instruction decode, are the same.

During the instruction fetch part, the processor makes ready to read the next instruction from memory and increment the Program Counter (PC). This is done by asserting the IRwrite signal, allowing the fetched instruction to be put into the Instruct Register (IR).

All together, the PC_Selection is set to '11' to pick PC+1, which is set to increment the PC by one. The PCwrite signal is turned on, showing that the PC should be changed always (updated unconditionally), making it ready for the next instruction fetch cycle.

In the register get and branch address computation part, which also covers the instruction decode step, the sign_ext signal is asserted to extend the sign of the immediate value. The select_type signal is set to 'X' (don't mind), telling that it is not important in this part. The PCwrite signal is turned on for jumps, and the PC_Selection is set to '10' to choose the branch target address for possible branching.

For load and save instructions, special control signals signs are turned on to make easy the memory get actions.

During the load instructions part, the RegSrc1 and RegSrc2 signals are set to '0', picking the right register sources: RS1 as source 1, RD as source 2. The ALUsrc2 signal is set to '1', telling the ALU to use the second number from an immediate value. The ALUop signal is set to act an add action to find the real address. The address_selection signal is set to '1' to pick the calculated address in ALU for memory actions.

For load instructions particularly, the data_in_selection signal is 'X' (don't mind) since data is being read, not written. The MemRead signal is said to '1' to allow reading from memory, and the MemWrite signal is set to '0' to stop writing. The MemReg signal is set to '1' to show that data should be put from memory into a register. The RegDest signal is set to '0', naming the result register RD for the got data.

If the way is 'LBu' (load byte unsigned), additional signals are turned on as follows: the sign_ext signal is set to '0' to load byte with zero extension, and the select_type signal is set to '0' to determine that it is an I-Type.

If the way is 'LBs' (load byte signed), additional signals are turned on differently: the sign_ext signal is set to '1' to load byte with sign extension, and the select_type signal is set to '0' to determine that it is an I-Type.

For store instructions, the RegSrc1 and RegSrc2 signals are set to '0', picking the right register sources: RS1 as source 1, RD as source 2. The ALUsrc2 signal is set to '1', telling the ALU to use the second number from an immediate value.

The ALUop signal is set to act an add action to find the real address. The address_selection signal is set to '1' to pick the found address for memory actions. The data_in_selection signal is set to '0', telling the data source RD for storing.

The MemRead signal is set to '0', stopping reading from memory, and the MemWrite signal is said to '1' to allow writing to memory. The MemReg signal is 'X' (don't mind) since data is being written on memory, not on Register File. The sign_ext signal is set to '1' to allow sign extension, and the select_type signal is set to '0' to determine that it is an S-Type.

For I-type instructions like ADDI and ANDI, the control signals are set to help immediate arithmetic or logical operations. The 'RegSrc1' signal is set to '00' to pick the right register source, while 'RegSrc2' is 'X' (don't mind). The 'RegDest' signal is set to '0', naming the result register RD for the calculated result. The 'sign_ext' signal is set to '1' to allow sign extension of the immediate value, and the 'select_type' signal is set to '0'. The 'ALUop' signal is set to the specific action (andi/addi) to act the wanted arithmetic or logical action. The 'ALUsrc2' signal is set to '1', telling the ALU to use the second number from an immediate value. The 'RegWrite' signal is said to '1' to allow writing the result to the result register RD. The 'address_selection', 'data_in_selection', 'MemRead' and 'MemWrite' signals are set to 'X', 'X', '0' and '0' respectively, as they are not important for this type of order. 'MemReg' is set to'10' so that the result of ALU will be written on RD.

For branch instructions (10-17), the control signals are set to help conditional branching based on register comparisons. The 'RegSrc1' signal is set to pick either 'RS1' or 'R0', depending on the specific branch case. The 'RegSrc2' signal is set to '0' to pick the RD as second source register. The 'RegDest' signal is 'X' (don't mind) as the branch orders do not write to a result register. The 'sign_ext' signal is set based on the match: '0' to match 'Rd' with 'Rs1', or '1' to compare 'Rd' with

'R0'. The 'select_type' signal is set to '0', showing that the instruction type is I-type. The 'ALUop' signal is set to 'sub' to act a subtract action for the comparison. The 'ALUsrc2' signal is set to '0', telling the ALU to use the second number from a register. The 'RegWrite' signal is set to '0' as no register write action is done in branch instructions. The 'address_selection', 'data_in_selection', 'MemRead', 'MemWrite', and 'MemReg' signals are set to 'X', 'X', '0', '0', and 'X' separately, as they are not important for this type of instructions.

For Sv instruction, the control signals are set as follows to indicate the wanted action. The 'RegSrc1' signal is set to '00' to pick the right first register source, while 'RegSrc2' is 'X' (don't mind). The 'RegDest' signal is 'X' (don't mind) as this operation does not write to a result register. The 'sign_ext' signal is set to '1' to allow sign extension. The 'select_type' signal is set to '1', showing that the instruction type is S-Type. The 'ALUop' signal is set to 'NoOp' to show no action is to be done by the ALU. The 'ALUsrc2' signal is 'X' (don't mind) as the ALU action is not important here. The 'RegWrite' signal is set to '0' as no register write action is done. The 'address_selection' signal is set to '0' to pick the right address which is RS1. The 'data_in_selection' signal is set to '1' to show that the data input source is the extended immediate. The 'MemRead' signal is set to '0' to stop reading from memory, and the 'MemWrite' signal is said to '1' to allow writing to memory. The 'MemReg' signal is 'X' (don't mind) as it is not important for this action.

For the call instruction, the control signals are set as follows to help the wanted action. The 'RegSrc1' and 'RegSrc2' signals are set to 'X' (don't mind) as the action does not involve particular register sources. The 'RegDest' signal is set to '1' to name the result register (R7). The 'sign_ext', 'select_type', and 'ALUsrc2' signals are also set to 'X' (don't mind) as they are not important for this instruction. The 'ALUop' signal is set to 'NoOp' to show that no action is to be done by the ALU. The 'RegWrite' signal is said to '1' to allow writing the result to R7. The 'address_selection', 'data_in_selection', 'MemRead', 'MemWrite', and 'MemReg' signals are set to 'X', 'X', '0', '0', and '0' separately, as they are not important for this type of instructions. Additionally, the 'PC_Selection' signal is set to '10' to pick {PC[15:12], Immediate}.

For the return instruction, the control signals are set as follows to help the wanted action. The 'RegSrc1' signal is set to '10' to pick R7 as a first register source, while 'RegSrc2' is 'X' (don't mind). The 'RegDest' signal is 'X' (don't mind) as this operation does not write to a result register. The 'sign_ext', 'select_type', and 'ALUsrc2' signals are also set to 'X' (don't mind) as they are not important for this instruction. The 'ALUop' signal is set to 'NoOp' to show that no action is to be done by the ALU. The 'RegWrite' signal is said to '0' as no register write action is done. The 'address_selection', 'data_in_selection', 'MemRead', 'MemWrite', and 'MemReg' signals are set to 'X', 'X', '0', '0', and '0' separately, as they are not important for this type of instructions. Additionally, the 'PC_Selection' signal is set to '00' so that the Next PC value is the value of R7.

3.1.3. State Assignment

State Name	Abbreviation	State Number
InstructionFetch	**IF**	**0**
InstructionDecode	**ID**	**1**
AddressComputation	**AC**	**2**
LoadAccess	**LA**	**3**
StoreAccess	**SA**	**4**
ALUcomputationI	**I-type**	**5**
ALUcomputationR	**R-type**	**6**
LBuResultStore	**LBu_resStore**	**7**
LBsResultStore	**LBs_resStore**	**8**
ALUResultStore	**ALU_resStore**	**9**
BranchCompletion	**BC**	**10**
SetValue	**SV**	**11**
Call	**C**	**12**
Return	**RET**	**13**

Table 3: State assignment table

3.1.4. Control Unit State Table

Inputs			Outputs									
State	**Opcode**	**mode**	**Next state**	**SEL**	**EN**	**PCWrite Uncond**	**IRwrite**	**PC_selection**	**RegSrc 1**	**RegSrc 2**	**RegDest**	**Sign_ext**
IF	X	X	ID	X	X	1	1	11	XX	X	X	X
ID	JMP	X	IF	X	X	1	0	10	XX	X	X	1
ID	LW	X	AC	X	X	0	0	11	00	X	0	1
ID	SW	X	AC	X	X	0	0	11	00	0	X	1
ID	R-type	X	R-type	X	X	0	0	11	00	1	0	X
ID	Branch	0	BC	X	X	0	0	01	00	0	X	1
ID	Branch	1	BC	X	X	0	0	01	01	0	X	1
ID	SV	X	SV	X	X	0	0	11	00	X	X	1
ID	I-type	X	I-type	X	X	0	0	11	00	X	0	1
AC	LW	X	LA	X	0	0	0	XX	00	X	0	1
AC	SW	X	SA	X	X	0	0	XX	00	0	X	1
LA	X	0	LBu_resStore	0	1	0	0	XX	00	X	0	0
LA	X	1	LBs_resStore	1	1	0	0	XX	00	X	0	1
SA	X	X	IF	X	X	0	0	XX	00	0	X	1
LBu_resStore	X	X	IF	X	X	0	0	XX	XX	X	X	X
LBs_resStore	X	X	IF	X	X	0	0	XX	XX	X	X	X
ALU_resSTORE	X	X	IF	X	X	0	0	XX	XX	X	X	X
BC	X	X	IF	X	X	0	0	01	XX	X	X	X
SV	X	X	IF	X	X	0	0	11	00	X	X	1
C	X	X	IF	X	X	1	0	10	XX	X	1	1
RET	X	X	IF	X	X	1	0	00	XX	X	X	X
R-type	X	X	ALU_resStore	X	X	0	0	XX	XX	X	X	X
I-type	X	X	ALU_resStore	X	X	0	0	XX	XX	X	X	X

Table 4: Control Unit State table 1

Inputs			**Outputs**									
State	**Opcode**	**mode**	**Next state**	**Select Typ**	**RegWrite**	**ALUctrl**	**ALUsrc2**	**Address_ selection**	**Data_in_ selection**	**MemRead**	**MemWrite**	**MemReg**
IF	X	X	ID	X	X	XX	X	X	X	X	X	XX
ID	JMP	X	IF	X	X	XX	X	X	X	X	X	XX
ID	LW	X	AC	X	1	01	1	1	X	1	0	01
ID	SW	X	AC	X	0	01	1	1	0	0	1	XX
ID	R-type	X	R-type	X	1	00/01/10	0	X	X	0	0	10
ID	Branch	0	BC	0	0	10	0	X	X	0	0	00
ID	Branch	1	BC	0	0	10	0	X	X	0	0	00
ID	SV	X	SV	1	0	XX	X	0	1	0	1	XX
ID	I-type	X	I-type	0	1	00/01	1	X	X	0	0	10
AC	LW	X	LA	X	1	XX	1	1	X	1	0	01
AC	SW	X	SA	X	0	XX	1	1	0	0	1	XX
LA	X	0	LBu_resStore	X	1	XX	1	1	X	1	0	01
LA	X	1	LBs_resStore	X	1	XX	1	1	X	1	0	01
SA	X	X	IF	X	0	XX	1	1	0	0	1	XX
LBu_res Store	X	X	IF	X	1	XX	X	X	X	1	0	01
LBs_res Store	X	X	IF	X	1	XX	X	X	X	1	0	01
ALU_res STORE	X	X	IF	X	1	XX	X	X	X	0	0	10
BC	X	X	IF	X	0	10	0	X	X	0	0	XX
SV	X	X	IF	X	0							
C	X	X	IF	X	1	XX	X	X	X	0	0	00
RET	X	X	IF	X	1	XX	X	X	X	0	0	XX
R-type	X	X	ALU_R-type	X	0	00	0	X	X	0	0	XX
I-type	X	X	ALU_I-type	X	0	00	1	X	X	0	0	XX

Table 5: Control Unit State table 2

3.1.5. Boolean Expressions

SEL = LA . mode

EN = LA
PCWrite Uncond = IF + ID.JMP + C + RET

IRwrite = IF

PC_selection0 = IF + ID.(LW + SW + R-type + Branch + SV + I-type)+ BC + SV
PC_selection1 = IF + ID.(JMP + LW + SW + R-type + SV + I-type)+ SV + C

RegSrc10 = ID.Branch.mode
RegSrc11 = ID.(LW + SW + R-type + (Branch.mode') + SV + I-type) + AC.(LW + SW) + LA + R-type + I-type

RegSrc2 = ID.R-type

RegDest = C

Signet = ID.(JMP + LW + SW +Branch + SV + I-type) + AC + LA.mode + SA + SV + C

SelectType = ID

RegWrite =ID. (R-type + I-type + JMP) + AC + LA. (LBu_resStore + LBs_resStore)+LBu_resStore +LBs_resStore +ALU_resSTORE +C +RET

ALUctrl0 = ID
ALUctrl 1= ID. (I-type + R-type)

ALUsrc2 = ID. LW + ID.SW + ID. I-type + AC. (SW+ LW) + LA+ SA + I-type

Address_selection =ID. (LW + SW)+ AC. (SW+ LW) + SA+ LA

Data_in_selection =ID

MemRead =ID+ AC+LA+LBu_resStore+LBs_resStore

MemWrite =ID. (SW+SV) + AC+SA

MemReg 0= ID. (R-type +I-type)+ALU_resSTORE

MemReg1= ID + AC + LA + LBs_resStore + LBu_resStore

3.2.ALU Control

The ALU Control Unit’s primary task is to generate control signals that instruct the ALU to execute specific arithmetic or logical operation. This role allows the processor to perform a wide array of computations and comparisons.

3.2.1. ALU Control Truth Table

The truth table illustrates how specific opcodes from our instruction set map to these ALU Control Signals and, in turn, to the corresponding ALU operations.

Inputs		Output
Opcode	**ALUctrl (A1 A0)**	**ALUop (F1 F0)**
AND 0000	00	00
ADD 0001	00	01
SUB 0010	00	10
ADDI 0011	00	01
ANDI 0100	00	00
X	01	01
X	10	10
X	11	11

Table 6: ALU Control truth table

The ALU Control Signals dictate the operation the ALU performs based on the instruction's opcode. Here is a summary of the ALU Control Signals and their corresponding ALU operations:

- When ALUctrl is **'00'**, the ALU operation is determined by the instruction's **opcode**. For example:
  - ➢ Opcodes for 'ADDI' or 'ADD' set the ALUop to **'01'**, which signifies an **addition** operation.
  - ➢ Opcodes for 'ANDI' or 'AND' set the ALUop to **'00'**, which signifies a bitwise **AND** operation.
  - ➢ The 'SUB' opcode sets the ALUop to **'10'**, which signifies a **subtraction** operation.

- ALUctrl '**01**' indicates an **addition** operation.

- ALUctrl '**10**' indicates a **subtraction** operation.

- ALUctrl '**11**' indicates a reverse **subtraction** operation.

3.2.2. ALU control Boolean Expressions

- $F1 = (A1`\ A0`.SUB) + A1$
- $F0 = (A1`\ A0`.ADD) + (A1`\ A0`.ADDI) + A0$

These two Boolean expressions were obtained from the truth table of the ALU Control Unit.

3.3.PC Control

The PC Control Unit decides when to jump to different parts of a program. It uses flags (Negative (N), Zero (Z), and Overflow (V)) to decide when to jump to a different part of the program (instructions like BGT, BGTZ, BLT, BLTZ, BEQ, BEQZ, BNE, BNEZ). If the proper conditions are met, the PCWrite is set to 1, indicating that the processor will branch. There's also a special signal called PCwriteUncond. When this signal is set, the computer jumps to a new place without checking anything else. This setup leads the processor to either go step-by-step through the program or jump around/ branch when needed.

3.3.1. PC Control Truth Table

Inputs						Output
Opcode	**PCwriteUncond**	**N**	**V**	**Z**	**State**	**PCWrite**
X	1	X	X	X	X	1
BEQ	0	X	X	1	7	1
BEQ	0	X	X	1	Not 7	0
BEQ	0	X	X	0	X	0
BEQZ	0	X	X	1	7	1
BEQZ	0	X	X	1	Not 7	0
BEQZ	0	X	X	0	X	0
BNE	0	X	X	1	X	0
BNE	0	X	X	0	7	1
BNE	0	X	X	0	Not 7	0
BNEZ	0	X	X	1	X	0
BNEZ	0	X	X	0	7	1
BNEZ	0	X	X	0	Not 7	0
BGT	0	0	0	0	7	1
BGT	0	0	0	0	Not 7	0
BGT	0	0	0	1	X	0
BGT	0	0	1	0	X	0
BGT	0	0	1	1	X	0

BGT	0	1	0	0	X	0
BGT	0	1	0	1	X	0
BGT	0	1	1	0	7	1
BGT	0	1	1	0	Not 7	0
BGT	0	1	1	1	X	0
BGTZ	0	0	0	0	7	1
BGTZ	0	0	0	0	Not 7	0
BGTZ	0	0	0	1	X	0
BGTZ	0	0	1	0	X	0
BGTZ	0	0	1	1	X	0
BGTZ	0	1	0	0	X	0
BGTZ	0	1	0	1	X	0
BGTZ	0	1	1	0	7	1
BGTZ	0	1	1	0	Not 7	0
BGTZ	0	1	1	1	X	0
BLT	0	0	0	X	X	0
BLT	0	0	1	X	7	1
BLT	0	0	1	X	Not 7	0
BLT	0	1	0	X	7	1
BLT	0	1	0	X	Not 7	0
BLT	0	1	1	X	X	0
BLTZ	0	0	0	X	X	0
BLTZ	0	0	1	X	7	1
BLTZ	0	0	1	X	Not 7	0
BLTZ	0	1	0	X	7	1
BLTZ	0	1	0	X	Not 7	0
BLTZ	0	1	1	X	X	0

Table 7: PC Control table

3.3.2. PC Control Boolean Expression

- $PCWrite = (state == 7).Branch + PCWriteUncond$
- $Branch = (BEQ.Z) + (BEQZ.Z) + (BNE.Z`) + (BNEZ.Z`) + (BGT.Z`.(N \oplus V)`) + (BGTZ.Z`.(N \oplus V)`) + \big(BLT.(N \oplus V)\big) + \big(BLTZ.(N \oplus V)\big)$

* We took advantage from the previous semester report to know how to extract these two expressions, and to know how the pc is connected internally.

4. Full Datapath

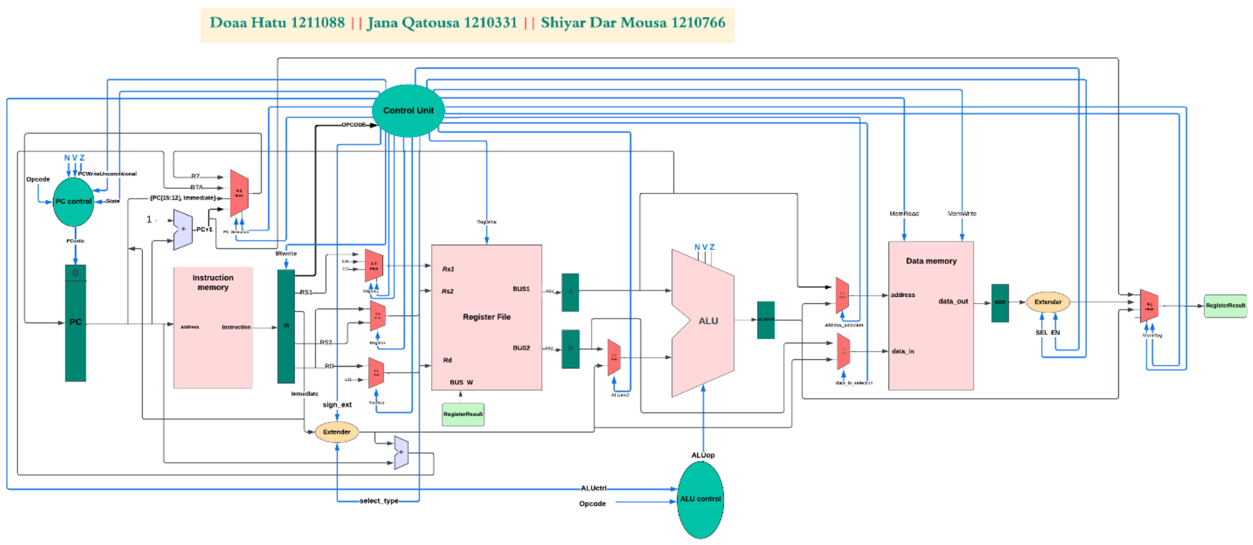


Figure 10: Full Datapath

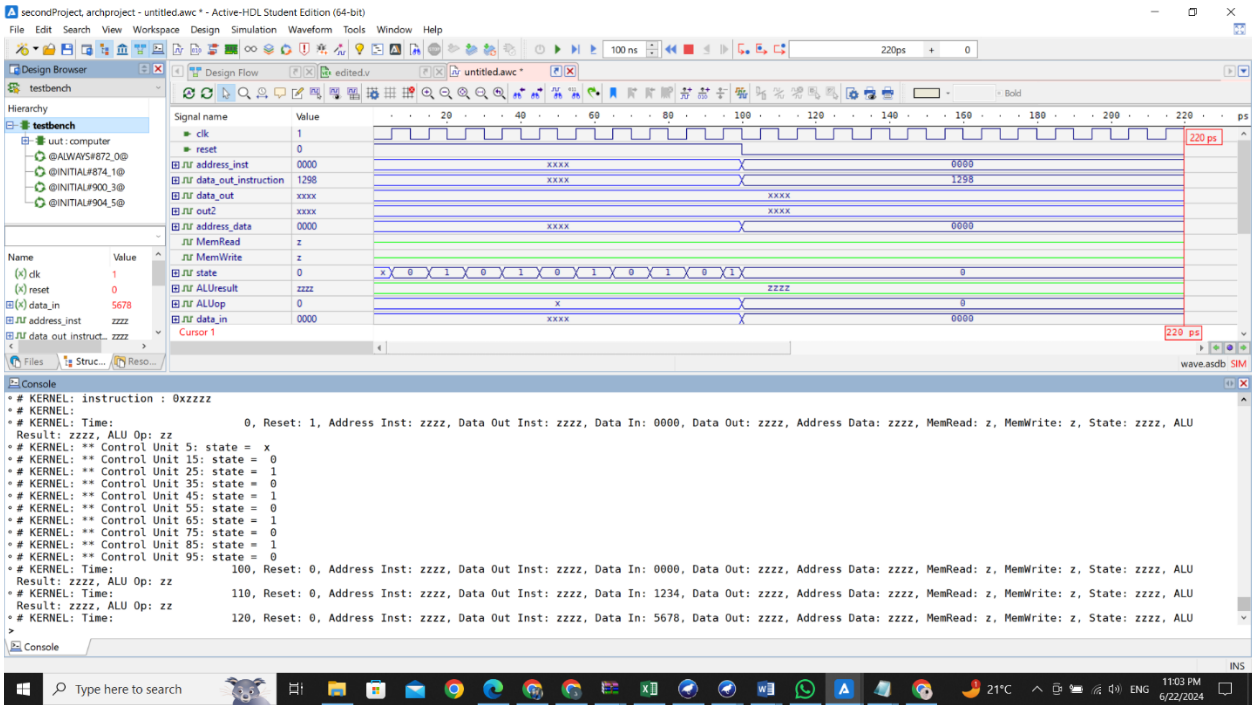
Simulation and Testing

First test bench

Code

```
module testbench;

    // Inputs
    reg clk;
    reg reset;
    reg [15:0] data_in;

    // Outputs
    wire [15:0] address_inst;
    wire [15:0] data_out_instruction;
    wire [15:0] data_out;
    wire [15:0] address_data;
    wire MemRead;
    wire MemWrite;
    wire [3:0] state;
    wire [15:0] ALUresult;
    wire [1:0] ALUop;

    // Instantiate the computer module
    computer uut (
        .clk(clk),
        .reset(reset)
    );

    // Clock generation
    always
        #5 clk = ~clk;
                    initial begin
        // Initialize Inputs
        clk = 0;
        reset = 1;
        data_in = 16'h0000;

        // Wait 100 ns for global reset to finish
        #100;

                    // Deassert reset
        reset = 0;
        // Add stimulus here
        #10;
        data_in = 16'h1234; // Example data input
        #10;
        data_in = 16'h5678; // Another example data input

        // Wait for a while to observe the behavior
        #100;

        // Finish the simulation
        $finish;
    end

            initial begin
            $monitor("instruction : 0x%x\n", data_out_instruction);
            end

    initial begin
        // Monitor the signals
        $monitor("Time: %d, Reset: %b, Address Inst: %h, Data Out Inst: %h, Data In: %h,
Data Out: %h, Address Data: %h, MemRead: %b, MemWrite: %b, State: %b, ALU
Result: %h, ALU Op: %b",
            $time, reset, address_inst, data_out_instruction, data_in, data_out, address_data,
MemRead, MemWrite, state, ALUresult, ALUop);
    end

endmodule
```

Results:



```
• # KERNEL: instruction : 0xzzzz
• # KERNEL:
• # KERNEL: Time:                      0, Reset: 1, Address Inst: zzzz, Data Out Inst: zzzz, Data In: 0000, Data Out: zzzz, Address Data: zzzz, MemRead: z, MemWrite: z, State: zzzz, ALU
  Result: zzzz, ALU Op: zz
• # KERNEL: ** Control Unit 5: state =  x
• # KERNEL: ** Control Unit 15: state =  0
• # KERNEL: ** Control Unit 25: state =  1
• # KERNEL: ** Control Unit 35: state =  0
• # KERNEL: ** Control Unit 45: state =  1
• # KERNEL: ** Control Unit 55: state =  0
• # KERNEL: ** Control Unit 65: state =  1
• # KERNEL: ** Control Unit 75: state =  0
• # KERNEL: ** Control Unit 85: state =  1
• # KERNEL: ** Control Unit 95: state =  0
• # KERNEL: Time:                    100, Reset: 0, Address Inst: zzzz, Data Out Inst: zzzz, Data In: 0000, Data Out: zzzz, Address Data: zzzz, MemRead: z, MemWrite: z, State: zzzz, ALU
  Result: zzzz, ALU Op: zz
• # KERNEL: Time:                    110, Reset: 0, Address Inst: zzzz, Data Out Inst: zzzz, Data In: 1234, Data Out: zzzz, Address Data: zzzz, MemRead: z, MemWrite: z, State: zzzz, ALU
  Result: zzzz, ALU Op: zz
• # KERNEL: Time:                    120, Reset: 0, Address Inst: zzzz, Data Out Inst: zzzz, Data In: 5678, Data Out: zzzz, Address Data: zzzz, MemRead: z, MemWrite: z, State: zzzz, ALU
>
```

Figure 11: testbench1

Explanation:

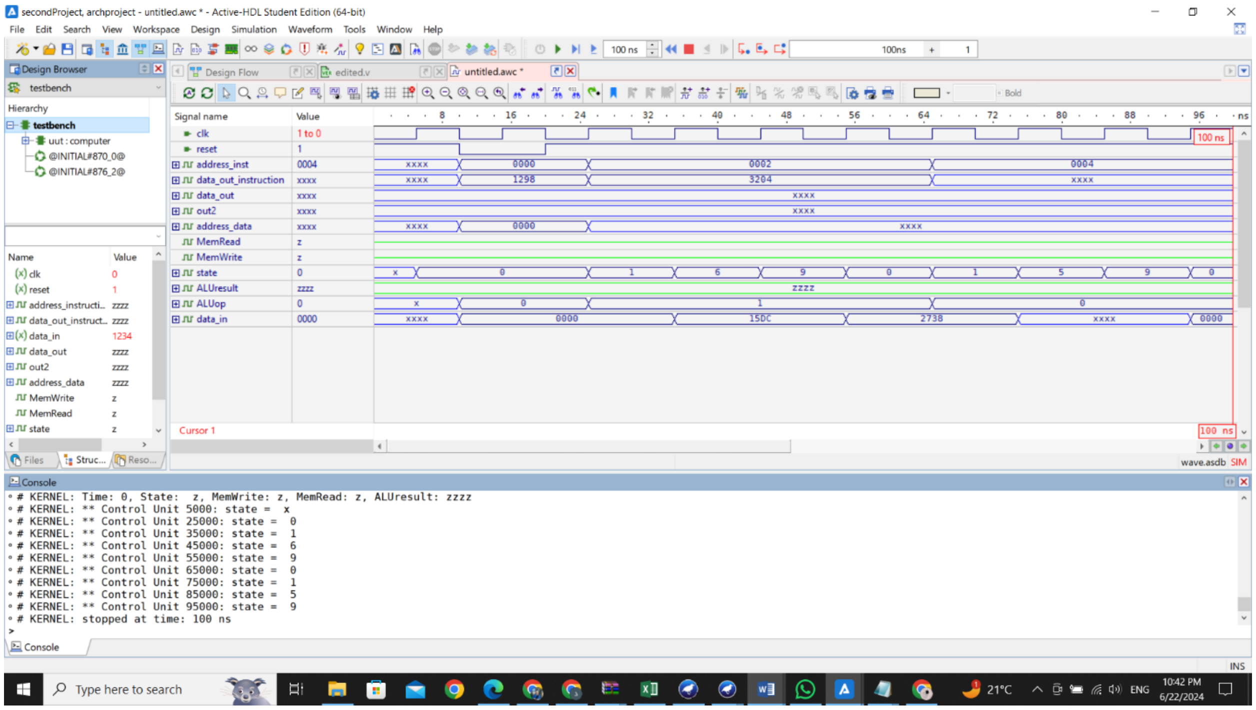
The test bench aims to show the values of the main control signals we have, as shown, it was not successful unfortunately, this may be due to some connection issues, or logical issues inside the code. We made sure that everything's implemented correctly, but we might have done a mistake somewhere.

Second test bench

Code

```
`timescale 1ns / 1ps


module testbench;
    reg clk;
    reg reset;
    wire [15:0] address_instruction;
    wire [15:0] data_out_instruction;
    reg [15:0] data_in;
    wire [15:0] data_out, out2;
    wire [15:0] address_data;
    wire MemWrite;
    wire MemRead;
    wire [3:0] state;
    wire [15:0] ALUresult;
    wire [1:0] ALUop;

    // Instantiate the computer module
    computer uut (
        .clk(clk),
        .reset(reset)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end


    // Initial block to apply stimulus
    initial begin
        // Monitor signals for debugging
        $monitor("Time: %0t, State: %d, MemWrite: %b, MemRead: %b, ALUresult: %h", $time,
state, MemWrite, MemRead, ALUresult);


        // Apply reset
        reset = 1;
        #10 reset = 0;
        #10 reset = 1;


        // Provide some input data
        data_in = 16'h1234;


        // Run the simulation for some time
        #500 $finish;
    end
endmodule
```

Results



```
• # KERNEL: Time: 0, State:  z, MemWrite: z, MemRead: z, ALUresult: zzzz
• # KERNEL: ** Control Unit 5000: state =  x
• # KERNEL: ** Control Unit 25000: state =  0
• # KERNEL: ** Control Unit 35000: state =  1
• # KERNEL: ** Control Unit 45000: state =  6
• # KERNEL: ** Control Unit 55000: state =  9
• # KERNEL: ** Control Unit 65000: state =  0
• # KERNEL: ** Control Unit 75000: state =  1
• # KERNEL: ** Control Unit 85000: state =  5
• # KERNEL: ** Control Unit 95000: state =  9
• # KERNEL: stopped at time: 100 ns
>
```

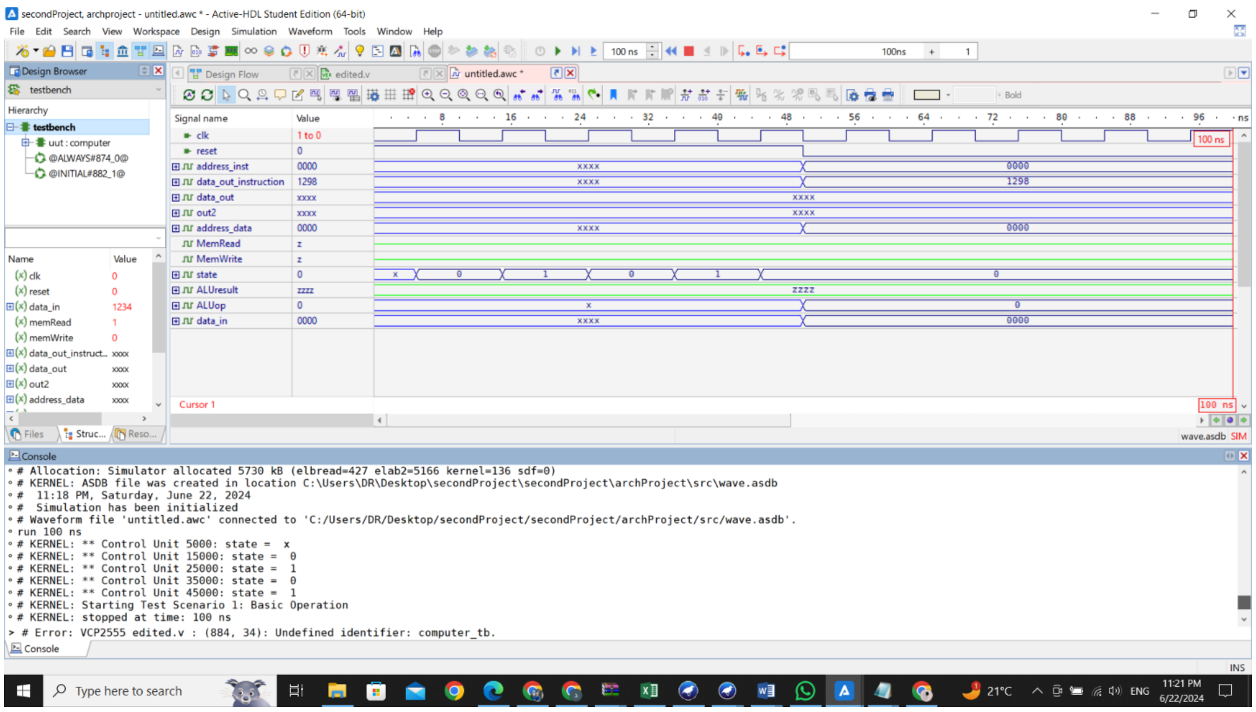
Figure 12: testbench2

Third test bench
Code
```
`timescale 1ns / 1ps


module testbench;


   // Parameters

   parameter CLK_PERIOD = 10; // Clock period in ns

   parameter SIM_TIME = 500; // Simulation time in ns


   // Signals for the test bench

   reg clk;

   reg reset;

   reg [15:0] data_in;

   reg memRead, memWrite;

   reg [15:0] data_out_instruction;

   reg [15:0] data_out, out2;

   reg [15:0] address_data;

   reg [3:0] state;

   reg [15:0] ALUresult;

   reg [1:0] ALUop;


   // Instantiate the computer module

    computer uut (

      .clk(clk),

      .reset(reset)

   );


   // Clock generation
```

```
always begin
    clk = 0;
    #((CLK_PERIOD / 2)); // Adjusted delay calculation
    clk = 1;
    #((CLK_PERIOD / 2)); // Adjusted delay calculation
end

// Reset initialization
initial begin
    $dumpfile("computer_tb.vcd");
    $dumpvars(0, testbench);

    reset = 1;
    memRead = 0;
    memWrite = 0;
    data_in = 16'h0000;
    #50;
    reset = 0; // Release reset

    // Test scenario 1: Basic operation
    $display("Starting Test Scenario 1: Basic Operation");
    data_in = 16'h1234;
    memRead = 1;
    #100;

    // Test scenario 2: Memory write operation
    $display("Starting Test Scenario 2: Memory Write");
    data_in = 16'h5678;
```

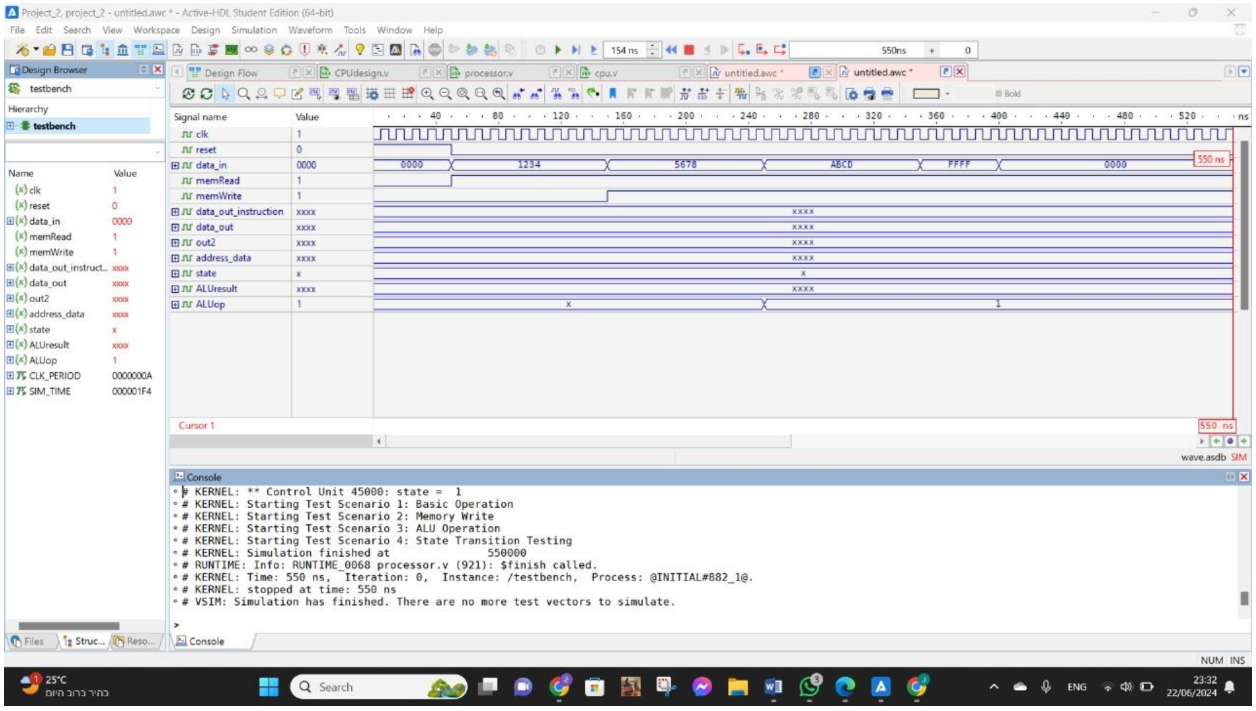
```
        memWrite = 1;
        #100;

        // Test scenario 3: ALU operation
        $display("Starting Test Scenario 3: ALU Operation");
        data_in = 16'hABCD;
        ALUop = 2'b01; // Example ALU operation code
        #100;

        // Test scenario 4: State transition testing
        $display("Starting Test Scenario 4: State Transition Testing");
        data_in = 16'hFFFF;
        #50;
        data_in = 16'h0000;
        #50;

        // End simulation
        #100;
        $display("Simulation finished at %t", $time);
        $finish;
    end

endmodule
```

Results



```
• # Allocation: Simulator allocated 5730 kB (elbread=427 elab2=5166 kernel=136 sdf=0)
• # KERNEL: ASDB file was created in location C:\Users\DR\Desktop\secondProject\secondProject\archProject\src\wave.asdb
• #  11:18 PM, Saturday, June 22, 2024
• #  Simulation has been initialized
• # Waveform file 'untitled.awc' connected to 'C:/Users/DR/Desktop/secondProject/secondProject/archProject/src/wave.asdb'.
• run 100 ns
• # KERNEL: ** Control Unit 5000: state =  x
• # KERNEL: ** Control Unit 15000: state =  0
• # KERNEL: ** Control Unit 25000: state =  1
• # KERNEL: ** Control Unit 35000: state =  0
• # KERNEL: ** Control Unit 45000: state =  1
• # KERNEL: Starting Test Scenario 1: Basic Operation
• # KERNEL: stopped at time: 100 ns
> # Error: VCP2555 edited.v : (884, 34): Undefined identifier: computer_tb.
```

Figure 13: testbench3



```
• # KERNEL: ** Control Unit 45000: state =  1
• # KERNEL: Starting Test Scenario 1: Basic Operation
• # KERNEL: Starting Test Scenario 2: Memory Write
• # KERNEL: Starting Test Scenario 3: ALU Operation
• # KERNEL: Starting Test Scenario 4: State Transition Testing
• # KERNEL: Simulation finished at                550000
• # RUNTIME: Info: RUNTIME_0068 processor.v (921): $finish called.
• # KERNEL: Time: 550 ns,  Iteration: 0,  Instance: /testbench,  Process: @INITIAL#882_1@.
• # KERNEL: stopped at time: 550 ns
• # VSIM: Simulation has finished. There are no more test vectors to simulate.
>
```

Explanation:
As shown in the figure, it seems like the program is getting stuck in the first instruction, and cannot operate after it, the problem could be in the PC, or in fetching.

When trying from another pc, the program gives different results (surprisingly), but the same issues exist..

Conclusion

In conclusion, our project successfully developed a detailed processor design, incorporating distinct segments for Instruction Memory and Data Memory to avoid conflicts during concurrent operations. We meticulously implemented and integrated all crucial stages of the processor's datapath, including instruction fetch, decode, execution, memory access, and write back. Throughout the process, we adhered to a systematic approach to ensure each component functioned correctly and cohesively within the overall architecture. However, despite our rigorous efforts and application of the best available methodologies, the test bench did not yield the anticipated results due to unforeseen issues that we were unable to fully diagnose and resolve. These challenges highlighted areas for improvement in our design and testing processes. Nonetheless, the experience provided invaluable insights into processor architecture and the complexities involved in its implementation. The lessons learned from these difficulties will be instrumental in guiding future enhancements and ensuring greater success in subsequent iterations of the project.