Electrical and Computer Engineering Department

ENCS4370
COMPUTER ARCHITECTURE

Project #2
Multi-Cycle RISC processor

Prepared by:

Kareem Qutob	1211756	**section: 3**
Ahmad Qaimari	1210190	**section: 3**
Husain Abugosh	1210338	**section: 1**

Date: 2024/6/20

Abstract

The project aims to design and verify a simple multicycle RISC processor in Verilog, operating on 16-bit instructions. This processor features 8 general-purpose registers, a program counter, and separate memories for instructions and data. The design supports four instruction types: R-type, I-type, J-type, and S-type, and includes little-endian byte ordering.

The goal of the project is to provide the processor the ability to carry out a wide range of instructions, from simple arithmetic operations like addition and subtraction to more complex ones like memory access and branching. This report shows how the processor's components work together by giving a thorough explanation of the datapath and control signals. It also gives the Verilog code that illustrates the operation of the CPU and displays the outcomes of the design process. The processor's accuracy in executing the intended instruction set is confirmed by extensive testing and simulation.

Table of contents

Table of figures

1 Data-Path components

1.1 Instruction memory

An essential part of a processor is its instruction memory, which stores and provides the set of instructions that the processor uses to operate. 16-bit instructions are stored in the instruction memory of our multicycle RISC processor. It is divided from the data memory by a Harvard design, which improves efficiency by enabling simultaneous access to both. During the instruction cycle's fetch stage, the program counter (PC) contacts the instruction memory to get the subsequent instruction. The R-type, I-type, J-type, and S-type instruction types are all encoded and stored in a byte-addressable manner using little-endian ordering. The PC provides an address to the memory, retrieves the instruction, and places it in the instruction register for execution and decoding. This process is known as the instruction fetch cycle.

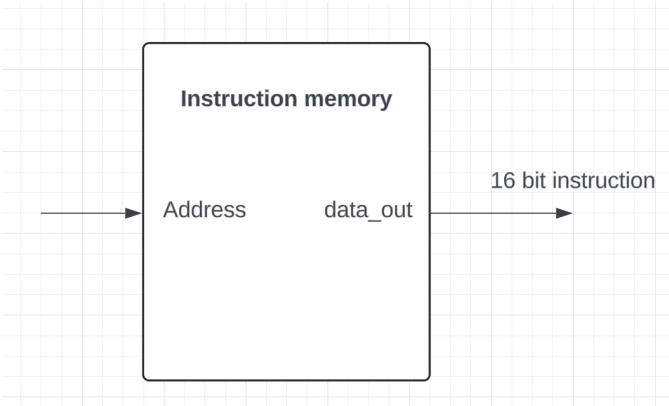

Figure 1:instructions memory block diagram

1.2 Data memory

A processor's data memory is an essential part that stores and retrieves data while a program is running. Our multicycle RISC processor has distinct operations for its data memory and instruction memory. Concurrent access to data and instructions is made possible by this division, which greatly increases efficiency. The data memory employs little-endian ordering and is byte-addressable, allowing access to specific bytes. It can handle a number of data operations, including

loading one byte at a time, putting register contents back into memory, and loading data into registers.

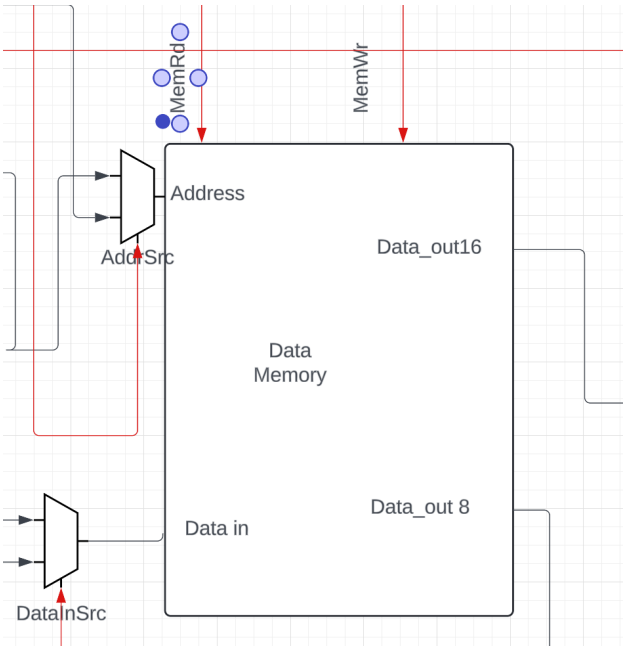

Figure 2: Data memory block diagram

Our design will load 16 and 8-bit (important for load byte operation) data in every use depending on what the instruction needs and it uses two control signals to manage read/write conflicts and hazards

1.3 Register file

A processor's register file is an essential part that acts as a quick and compact store space for data the processor uses often. The register file of our multicycle RISC processor is made up of eight general-purpose, 16-bit registers with the labels R0 through R7. However, R0 provides a consistent value for a variety of operations because it is hardwired to zero and cannot be changed. The register file makes it possible to manipulate data efficiently by providing instant access to operands required for logic, arithmetic, and other operations.

The source and destination register addresses are obtained at the instruction decode stage, and the relevant registers are read or written to in later stages. The architecture of the register file permits simultaneous read and write operations, guaranteeing that the processor may quickly retrieve operands and store outcomes. Fast data access and temporary storage are made possible by the register file, which greatly increases the speed and effectiveness of the processor while carrying out instructions.

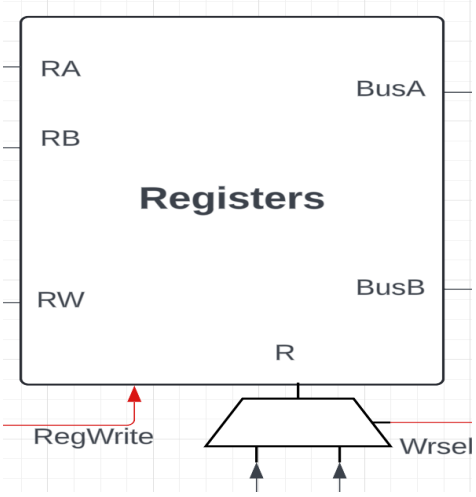

Figure 3: Register File block diagram

1.4 ALU

The Arithmetic Logic Unit (ALU) is a critical component in any processor, responsible for performing a variety of arithmetic and logical operations. In the provided diagram, the ALU is designed to handle five specific operations, including addition used in both arithmetic instructions and address calculation in load and store instructions, subtraction, incrementing by 2 for the pc counter, anding and reverse subtraction used to set flags in branch operations . The control signal, ALUop, is generated by the ALU control unit and determines which operation the ALU will execute based on the current instruction opcode , it contains 3 bits and can be used to encode 8 operations but in our alu, only 5 are used and the rest are unused.

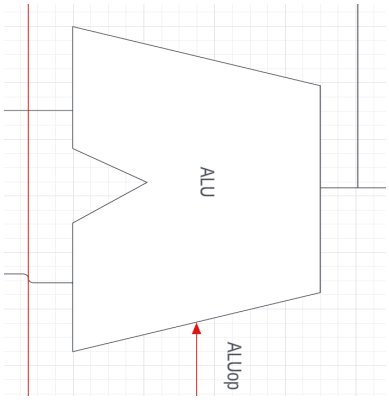

Figure 4:Alu block diagram

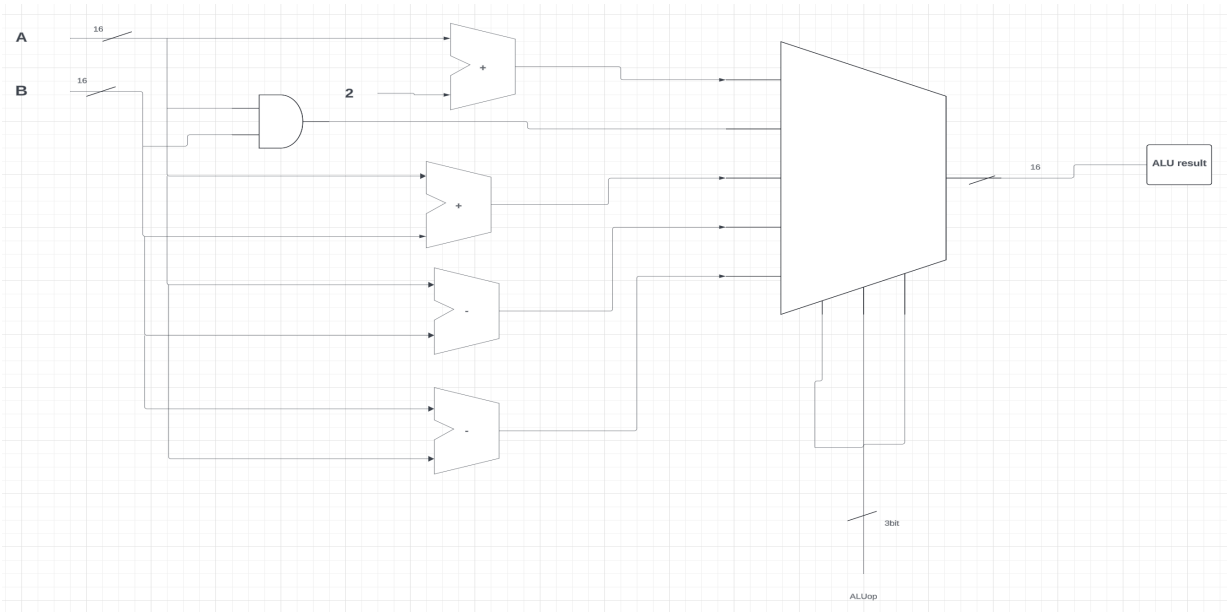

Figure 5: ALU Block diagram from the inside

1.5 Extender

In our multicycle RISC processor, we use two extenders to handle immediate values:

1. **I-Type Immediate Extender**: Extends a 5-bit immediate value to 16 bits, ensuring it fits the 16-bit data path for arithmetic operations.
2. **S-Type Immediate Extender**: Converts an 8-bit immediate value to 16 bits, used in store instructions for correct memory addressing.

These extenders ensure proper data handling and execution of instructions involving immediate values.

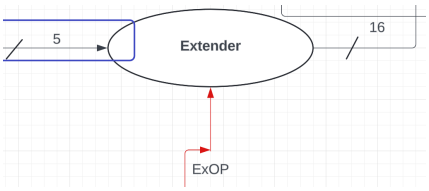

Figure 6: Extender block diagram

The extender will have a control signal to determine the type of extending (zero , signed).

2 Control units

2.1 Main control unit

2.1.1 control signals

Signal Name	Effect when de-asserted	Effect when asserted
PCForcedWrite	The PC is updated only if a branch condition is met	The PC is forced to update regardless of conditions
PCwrite	The PC is not updated	The PC is updated
IRwrite	The instruction register is not written	The instruction register is written
RBSrc	The RB source is the default value	The RB source is switched to an alternate value
RegWrite	The register file write is disabled	The register file write is enabled
Wrsel	The write selection is the default source	The write selection is switched to an alternate source
ExSrc	The extension source is the default value	The extension source is switched to an alternate value
DataInSrc	The data input source is the default	The data input source is switched to an alternate source
AddrSrc	The address source is the default	The address source is switched to an alternate source
MemRd	Memory read is disabled	Memory read is enabled
MemWr	Memory write is disabled	Memory write is enabled
MDRSrc	The MDR source is the default value	The MDR source is switched to an alternate value
WBdata	The write-back data is the default value	The write-back data is switched to an alternate value
RASrc	The RA source is the default value	The RA source is switched to an alternate value

RWSrc	The RW source is the default value	The RW source is switched to an alternate value

Signal Name	Value Binary	Effect
ALUSrcA	00	The First ALU operand is R0.
	01	The First ALU operand come from RegA
	10	The First ALU operand is PC.
ALUSrcB	00	The Second ALU operand come from RegB
	01	The Second ALU operand come from Extender
ALUctrl	000	The ALU performs Increment operation.
	001	The ALU performs an AND operation.
	010	The ALU performs an ADD operation.
	011	The ALU performs a SUB operation.
	100	The ALU performs a RSUB operation.
	111	Custom Operation based on the Opcode
ALUop	000	The ALU performs Increment operation.
	001	The ALU performs an AND operation.
	010	The ALU performs an ADD operation.
	011	The ALU performs a SUB operation.
	100	The ALU performs a RSUB operation.
	101	Unused
	111	Unused
PCSrc	00	the program counter (PC) is incremented by a fixed amount
	01	the PC is set to the jump target address provided by a jump instruction
	10	the PC is set to the branch target address
	11	the PC is set to a specific address (Call)
ExOP/ ExOP2	00	The immediate value remains unchanged.
	01	The immediate value is extended with zeros on the higher bits
	10	The immediate value is sign-extended
	11	Unused

2.1.2 Main control State Digram

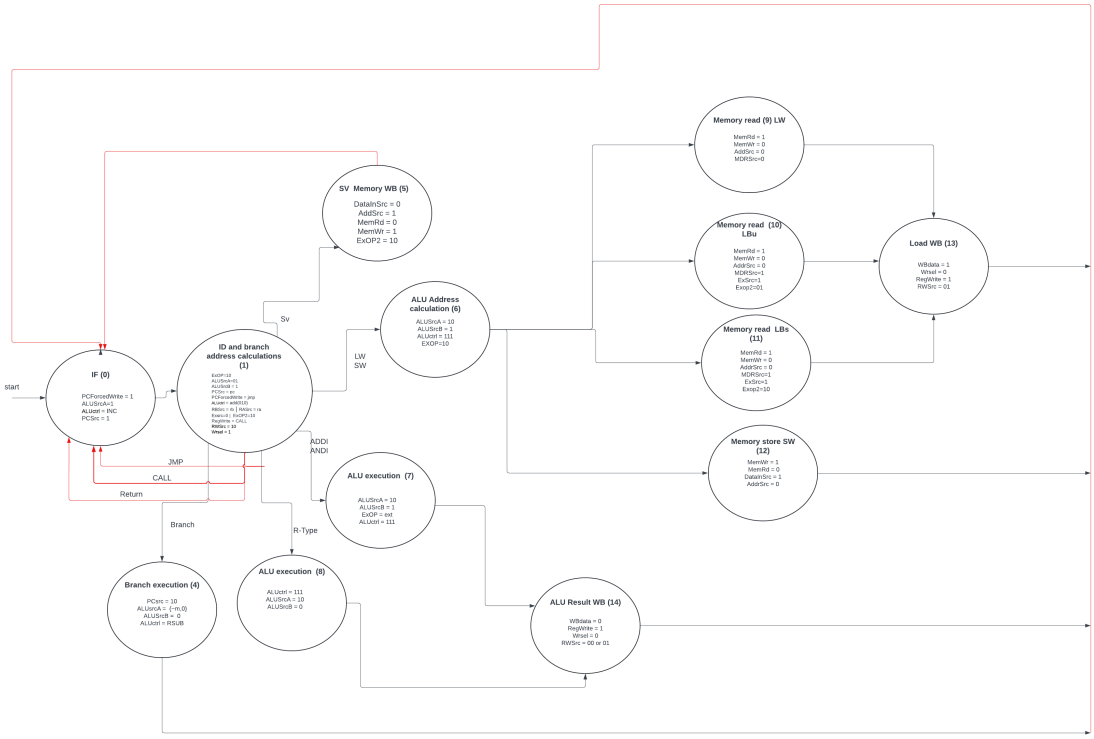

Figure 7: Main Control state Diagram

Instruction Fetch (IF) Phase

Operation: The processor prepares to read the next instruction from memory and increment the Program Counter (PC).
Control Signals:

- `PCForcedWrite = 1`: Forces the PC to update unconditionally.
- `ALUSrcA = 1`: Uses the PC as the ALU's first operand.
- `ALUctrl = 000`: Sets the ALU to perform an increment operation.
- `PCSrc = 01`: Selects the output of the ALU for the PC update.
- `IRwrite = 1`: Enables writing to the instruction register.

Instruction Decode (ID) Phase

Operation: The processor decodes the instruction and prepares for execution, Brach calaculations , R7 Write Back.
Control Signals:

- `ExOP = 10`: The immediate operand is sign-extended.
- `ALUSrcA = 01`: Uses the PC as the first ALU operand.
- `ALUSrcB = 1`: Uses the extended immediate as the second ALU operand.
- `PCSrc = pc`: Selects the branch target address.
- `PCForcedWrite = jmp`: Updates the PC unconditionally for jumps.
- `ALUctrl = 010`: Sets the ALU to perform an ADD operation.
- `RBSrc = rb`: Selects the register source for the next stage.
- `RASrc = ra`: Selects the register address for the next stage.
- `ExSrc = 0`: Selects the source for the extender.
- `ExOP2 = 10`: The second operand is sign-extended.
- `RegWrite = call`: Enables writing to the register file if it's a call instruction.
- `RWSrc = 10`: Selects the write register source as R7.
- `Wrsel = 1`: Selects the write data source.

Branch Execution (BRANCH_EXEC) Phase

Operation: Executes branch instructions by comparing registers and updating the PC if the condition is met.
Control Signals:

- `PCSrc = 10`: Selects the branch target address.
- `ALUctrl = 100`: Sets the ALU to perform a reverse subtraction.
- `ALUSrcA = {~m, 1'b0}`: Selects the A register as the first ALU operand.
- `ALUSrcB = 0`: Uses the B register as the second ALU operand.

Store Value Memory Write Back (SV_MEM_WB) Phase

Operation: Writes data to memory.
Control Signals:

- `DataInSrc = 0`: Selects the sExtended as data source.
- `ExOP2 = 10`: The second operand is sign-extended.
- `MemRd = 0`: Disables memory read.
- `MemWr = 1`: Enables memory write.

ALU Address Calculation (ALU_ADDR_CALC) Phase

Operation: Calculates the address for load and store instructions.
Control Signals:

- `ALUSrcA = 10`: Uses the A register as the first operand.
- `ALUSrcB = 1`: Uses the immediate value as the second operand.
- `ALUctrl = 111`: Sets the ALU to the operation based on the Opcode.
- `ExOP = 10`: The immediate operand is sign-extended.

ALU Execution (ANDI_ADDI) Phase

Operation: Executes ALU operations for ANDI and ADDI instructions.
Control Signals:

- `ALUSrcA = 10`: Uses the A register as the first operand.
- `ALUSrcB = 1`: Uses the immediate value as the second operand.
- `ALUctrl = 111`: Sets the ALU to perform the operation based on the Opcode.
- `ExOP = 10`: The immediate operand is sign-extended.

ALU Execution (R_TYPE) Phase

Operation: Executes ALU operations for R-type instructions.
Control Signals:

- `ALUctrl = 111`: Sets the ALU to perform the operation based on the Opcode.
- `ALUSrcA = 10`: Uses the A register as the first operand.
- `ALUSrcB = 0`: Uses the B register as the second operand.

Memory Read (LW) Phase

Operation: Reads data from memory for load instructions.
Control Signals:

- `MemRd = 1`: Enables memory read.
- `MemWr = 0`: Disables memory write.
- `AddrSrc = 0`: Selects the address source.
- `MDRSrc = 0`: Selects the data source.

Memory Read (LBU) Phase

Operation: Reads a byte from memory for load byte unsigned instructions.
Control Signals:

- `MemRd = 1`: Enables memory read.
- `MemWr = 0`: Disables memory write.
- `AddrSrc = 0`: Selects the ALU result as address source.
- `MDRSrc = 1`: Selects the extended HalfDataOut as data source.
- `ExSrc = 1`: Selects the HalfDataOut as extender source.
- `ExOP2 = 01`: Zero-extends the byte.

Memory Read (LBS) Phase

Operation: Reads a byte from memory for load byte signed instructions.
Control Signals:

- `MemRd = 1`: Enables memory read.
- `MemWr = 0`: Disables memory write.
- `AddrSrc = 0`: Selects the ALU result as address source.
- `MDRSrc = 1`: Selects the extended HalfDataOut as data source.
- `ExSrc = 1`: Selects the HalfDataOut as extender source.
- `ExOP2 = 10`: Sign-extends the byte.

Memory Write (SW) Phase

Operation: Writes data to memory for store instructions.
Control Signals:

- `MemRd = 0`: Disables memory read.
- `MemWr = 1`: Enables memory write.
- `DataInSrc = 1`: Selects the B register as data source.
- `AddrSrc = 0`: Selects the B register as address source.

Load Write Back (LOAD_WB) Phase

Operation: Writes loaded data back to the register file.
Control Signals:

- `WBdata = 1`: Selects the MDR as data source.
- `Wrsel = 0`: Selects the write data source.
- `RegWrite = 1`: Enables writing to the register file.
- `RWSrc = 01`: Selects the write register source.

ALU Result Write Back (ALU_RES_WB) Phase

Operation: Writes the ALU result back to the register file.
Control Signals:

- `WBdata = 0`: Selects the ALU result as the data source.
- `RegWrite = 1`: Enables writing to the register file.
- `Wrsel = 0`: Selects the write data source.
- `RWSrc = rw`: Selects the write register source.

2.1.3 State Assignments

State Name	State Code	State Number	Describtion
Instruction Fetch	IF	0	Fetches the instruction from memory and increments the Program Counter (PC).
Instruction Decode	ID	1	Decodes the instruction ,prepares the necessary control signals and decodes the call instruction and writes back to register R7.
Branch Execution	BRANCH_EXEC	4	Executes branch instructions and updates PC if the branch condition is met.
Store to Memory and Write Back	SV_MEM_WB	5	Executes store instructions and writes back to memory.
ALU Address Calculation	ALU_ADDR_CALC	6	Calculates the address for load/store instructions using the ALU.
ALU Execution for ANDI and ADDI	ALU_EXEC_ANDI_ADDI	7	Executes ADDI and ANDI instructions using the ALU.
ALU Execution for R-Type	ALU_EXEC_R_TYPE	8	Executes R-type instructions using the ALU.
Memory Read for Load Word	MEM_READ_LW	9	Reads a word from memory for load instructions.
Memory Read for Load Byte Unsigned	MEM_READ_LBU	10	Reads an unsigned byte from memory for load instructions.
Memory Read for Load Byte Signed	MEM_READ_LBS	11	Reads a signed byte from memory for load instructions.
Memory Write for Store Word	MEM_SW	12	Writes a word to memory for store instructions.
Load Write Back	LOAD_WB	13	Writes the loaded data back to the register file.
ALU Result Write Back	ALU_RES_WB	14	Writes the ALU result back to the register file.

2.1.4 Control Unit State Table

State	Opcode	mode	PCForcedWrite	IRwrite	RBSrc	RegWrite	Wrsel	ALUSrcA	ALUSrcB	ALUctrl	ExSrc	ExOP2	DataInSrc	AddrSrc	MemRd	MemWr	MDRSrc	WBdata	ExOP	RASrc	RWSrc	PCSrc
IF	X	X	1	1	X	0	0	0 1	1	0 0 0	X	0 0	0	0	0	0	0	0	10	0 0	0 0	0 1
ID	JMP	X	**1**	X	0	0	1	**0 1**	1	**0 1 0**	**0**	**1 0**	X	X	X	X	X	X	**10**	**0**	**10**	**0**
ID	CALL	X	**1**	X	0	1	1	**0 1**	1	**0 1 0**	**0**	**1 0**	X	X	X	X	X	X	**10**	**0**	**10**	**0 0**
ID	RET	X	**1**	X	0	1	1	**0 1**	1	**0 1 0**	**0**	**1 0**	X	X	X	X	X	X	**10**	**11**	**10**	**11**
ID	R-Type	X	**0**	X	1	1	1	**0 1**	**1**	**0 1 0**	**0**	**10**	X	X	X	X	X	X	**10**	**0 1**	**10**	**0 0**
ID	ADDI, ANDI	X	**0**	X	**0**	**0**	**1**	**0 1**	**1**	**0 1 0**	**0**	**10**	X	X	X	X	X	X	**10**	**10**	**10**	**0 0**
ID	Branch	X	**0**	**0**	**0**	**0**	**1**	**0 1**	1	**0 1 0**	**0**	**1 0**	X	X	X	X	X	X	10	10	10	0 0
ALU_ADDR_CALC	LW	X	X	X	X	X	X	10	0 1	1 1 1	X	X	X	X	X	X	X	X	10	X	X	X
ALU_ADDR_CALC	SW	X	X	X	X	X	X	10	0 1	1 1 1	X	X	X	X	X	X	X	X	10	X	X	X
ALU_ADDR_CALC	LBu	0	X	X	X	X	X	10	0 1	1 1 1	X	X	X	X	X	X	X	X	1	X	X	X
ALU_ADDR_CALC	LBu	1	X	X	X	X	X	10	0 1	1 1 1	X	X	X	X	X	X	X	X	10	X	X	X
ALU_EXEC_R_TYPE	R-Type	X	X	X	X	X	X	10	0	1 1 1	X	X	X	X	X	X	X	X	X	X	X	X
MEM_READ_LW	LW	X	X	X	X	X	X	X	X	X	X	X	X	**0**	**1**	**0**	**0**	X	X	X	X	X
MEM_READ_LBU	LBu	0	X	X	X	X	X	X	X	X	**1**	**0 1**	X	**0**	**1**	**0**	**1**	X	X	X	X	X
MEM_READ_LBS	LBS	1	X	X	X	X	X	X	X	X	**1**	**1 0**	X	**0**	**1**	**0**	**1**	X	X	X	X	X
MEM_SW	SW	X	X	X	X	X	X	X	X	X	X	X	**1**	X	**0**	**1**	X	X	X	X	**0 1**	X
LOAD_WB	LW, LBu, LBs	X	X	X	X	**1**	**0**	X	X	X	X	X	X	X	X	X	X	**1**	X	X	**0 1**	X
ALU_RES_WB	R-Type	X	X	X	X	**1**	**0**	X	X	X	X	X	X	X	X	X	X	**0**	X	X	**0 0**	X
BRANCH_EXEC	Branch	0	X	X	X	X	X	**10**	**0**	**100**	X	X	X	X	X	X	X	X	X	X	X	**0 1**

State	Opcode	mode	PCForcedWrite	IRwrite	RBSrc	RegWrite	Wrsel	ALUSrcA	ALUSrcB	ALUctrl	ExSrc	ExOP2	DataInSrc	AddrSrc	MemRd	MemWr	MDRSrc	WBdata	ExOP	RASrc	RWSrc	PCSrc
BRANCH_EXEC	Branch	1	X	X	X	X	X	**0 0**	**0**	**100**	X	X	X	X	X	X	X	X	X	X	X	**0 1**
SV_MEM_WB	Sv	X	X	X	X	X	X	X	X	X	X	**1 0**	**0**	**1**	**0**	**1**	X	X	X	X	X	X
ALU_EXEC_ANDI_ADDI	ADDI	X	X	X	X	X	X	**10**	**1**	**111**	X	X	X	X	X	X	X	X	**10**	X	X	X

2.1.5 Control Unit State Boolean Expression

PcforcedWrite = IF + ID JMP + ID RET + IDCALL
IRwrite = IF

RBSrc = ID -Rtype

RegWrite = IDCALL + ID RET

Wrsel = ID JMP + ID Call + ID RET + ID Rtype + ID Add,ANDI + ID brach

AluSrcA_0 = IF + ID JMP + ID Call + ID RET+ ID Rtype + ID Add,ANDI + ID brach

AluSrcA_1 = ALU ADDR_CALC LW+ ALU ADDR_CALC sw + ALU ADDR_CALC LBU0 + ALU ADDR_CALC LBU1 + ALU Excu Rtype + Bracnh_excution Brach + Alu_Excution_ANDI_ADDI

AluSrcB= IF + ID JMP + ID Call + ID RET++ ID Rtype + ID Add,ANDI + ID brach + Alu_Excution_ANDI_ADDI

ALUCTRL_0 = ALU ADDR_CALC LW+ ALU ADDR_CALC sw ++ ALU ADDR_CALC LBU0 + ALU ADDR_CALC LBU1 + ALU Excu Rtype + Alu_Excution_ANDI_ADDI

ALUCTRL_1 = ALU ADDR_CALC LW+ ALU ADDR_CALC sw ++ ALU ADDR_CALC LBU0 + ALU ADDR_CALC LBU1 + ALU Excu Rtype + Alu_Excution_ANDI_ADDI + ID JMP + ID Call + ID RET++ ID Rtype + ID Add,ANDI + ID brach

ALUCTRL_2 = ALU ADDR_CALC LW+ ALU ADDR_CALC sw ++ ALU ADDR_CALC LBU0 + ALU ADDR_CALC LBU1 + ALU Excu Rtype + Alu_Excution_ANDI_ADDI + Bracnh_excution Brach cu + Alu_Excution_ANDI_ADDI

ExSrc = Mem_rd_lbu + Mem_rd_lbs

EXOp2_0 = Mem_rd_lbu

EXOp2_1 = IF + ID JMP + ID Call + ID RET++ ID Rtype + ID Add,ANDI + ID brach + + Mem_rd_lbs +sv_mem_wb sv

DataInSrc = MEM_SW . SW

AddrSrc = SV_MEM_WB . Sv

MemRd= MEM_READ_LW . LW + MEM_READ_LBU . LBU + MEM_READ_LBS. LBS

MemWr = MEM_SW . SW + SV_MEM_WB . Sv

MDRSrc = MEM_READ_LBU . LBU + MEM_READ_LBS. LBS

WBdata = Load_Wb . LW . LBU . LBS

ExOP 0 = ADDR_calc.LBSu.0

ExOP 1 = ID + IF + ADDR_calc.LW + ADDR_calc.LBSu.1 + ADDR_calc.SW.1 + ALUex.addi

RASrc 0 = ID.RET + ID.Rtype
RASrc 1 = ID.ret + ID.ANDI.ADDI + ID. BRANCH

RWSrc 0 = MEM_WB.SW + LW_WB.LW.LBU.LBS
RWSrc 1 = ID

PCSrc 0 = ID.RET + IF + BRANCH_EX.BRANCH
PCSrc 1 = ID.RET

2.2 Pc control unit

The PC Control Unit (Program Counter Control Unit) is a critical component of the CPU that determines when and how the program counter (PC) should be updated. It ensures the correct flow of instruction execution by managing branching and jumping to different parts of the program based on specific conditions. Using flags (Negative (N), Zero (Z), and Overflow (V)) for instructions like BGT, BLT, BEQ, and BNE, the PC Control Unit sets PCWrite to 1 if conditions are met for branching, and employs a special signal, PCwriteUncond, for unconditional jumps, ensuring the processor either proceeds step-by-step or branches as required.

2.2.1 Pc control truth table

Opcode	PCForcedWrite	N	V	Z	State	PCwrite
X	1	X	X	X	X	1
BEQ	0	X	X	1	4	1
BEQ	0	X	X	1	Not 4	0
BEQ	0	X	X	0	X	0
BNE	0	X	X	1	X	0
BNE	0	X	X	0	4	1
BNE	0	X	X	0	Not 4	0
BGT	0	0	0	0	4	1
BGT	0	0	0	0	Not 4	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	4	1
BGT	0	1	1	0	Not 4	0
BGT	0	1	1	1	X	0
BLT	0	0	0	X	X	0
BLT	0	0	1	X	4	1
BLT	0	0	1	X	Not 4	0
BLT	0	1	0	X	4	1
BLT	0	1	0	X	Not 4	0
BLT	0	1	1	X	X	0

2.2.2 Internal Structure of the PC control unit

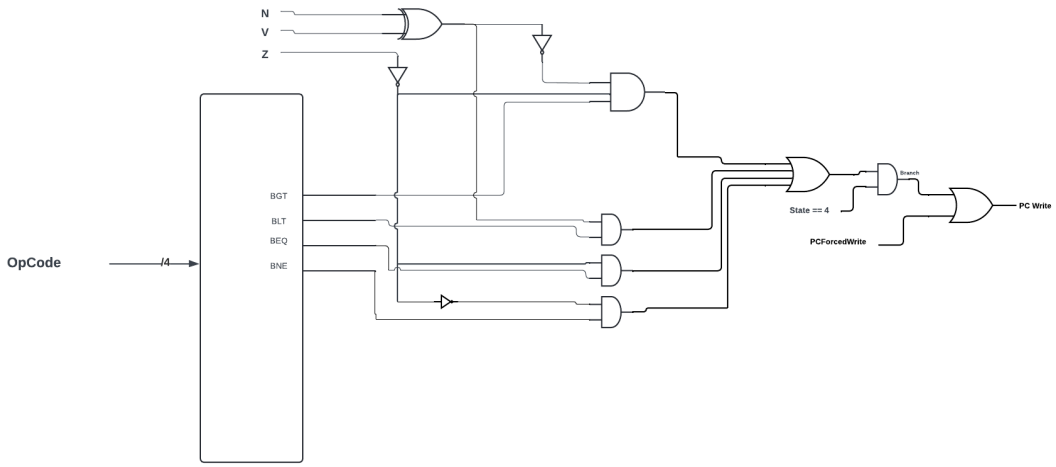

Figure8: Pc control block diagram

2.2.3 Pc Boolean Expression

The PC Control Unit determines program flow based on specific conditions set by the flags. It updates the program counter (PC) based on the following branching instructions:

- **BGT (Branch if Greater Than)**: Requires `Z=0` and `N=V`, indicating a positive result.
- **BEQ (Branch if Equal)**: Requires `Z=1`, indicating equality.
- **BNE (Branch if Not Equal)**: Requires `Z=0`, signaling inequality.
- **BLT (Branch if Less Than)**: Requires `N≠V`, indicating a negative result.

These conditions are evaluated to make accurate branching decisions. Additionally, the state must equal `4`, indicating the branch completion state. If `PCForcedWrite` is set, the PC is updated unconditionally.

Presented below is the Boolean Expression:

$Branch = (BEQ.z) + (BNE.z^{\square}) + (BGT.z'.(N \oplus V)^{\square}) + (BLT.(N \oplus V))$
$PCwrite = (\ state == 4)\ .\ Branch$ + `PCForcedWrite`

Where $\oplus$ indicates ***xor***

2.3 ALU control unit

The job of the ALU Control Unit is to produce precise control signals (ALUop) that tell the Arithmetic Logic Unit (ALU) what to do in response to the current instruction. When it receives inputs from the main control unit, it decodes the opcode and function code of the instruction to find the necessary operation, which could be addition, subtraction, or logical operations, pc increment. By deciphering the instruction specifics and establishing the proper control signals, the ALU Control Unit guarantees the proper execution of different instruction kinds (R-type, I-type, etc.). In order to ensure that the ALU can accurately perform the required operations and that the processor operates correctly and effectively, this step is essential.

2.3.1 ALU control truth table

Opcode (O3 O2 O1 O0)	ALUCtrl (A2 A1 A0)	ALUop (F2 F1 F0)	Description
X	000	000	INC
X	001	UNDEFINED	UNDEFINED
X	010	010	ADD
X	011	UNDEFINED	UNDEFINED
X	100	100	RSUB
X	101	UNDEFINED	UNDEFINED
X	110	UNDEFINED	UNDEFINED
0000	111	001	AND
0001	111	010	ADD
0010	111	011	SUB
0011	111	010	ADD
0100	111	001	AND
0101	111	010	ADD

0110	111	010	ADD
0111	111	010	ADD
1000	111	UNDEFINED	UNDEFINED
1001	111	UNDEFINED	UNDEFINED
1010	111	UNDEFINED	UNDEFINED
1011	111	UNDEFINED	UNDEFINED
1100	111	UNDEFINED	UNDEFINED
1101	111	UNDEFINED	UNDEFINED
1110	111	UNDEFINED	UNDEFINED
1111	111	UNDEFINED	UNDEFINED

2.3.2 ALU control signals Boolean expression

F2= (A2·∼A1·A0) + (∼A2·A1·∼A0) + (A2·A1·A0·O2·∼O1·∼O0)

F1 = (A2·∼A1·A0) + (A2·A1·∼A0) + (A2·A1·A0·(O2+O1)·∼O0)

F0 = (∼A2·∼A1·∼A0) + (A2·A1·A0·((∼O3·∼O2·∼O1·∼O0) + (∼O3·O2·∼O1·∼O0)))

3 Data-Path components

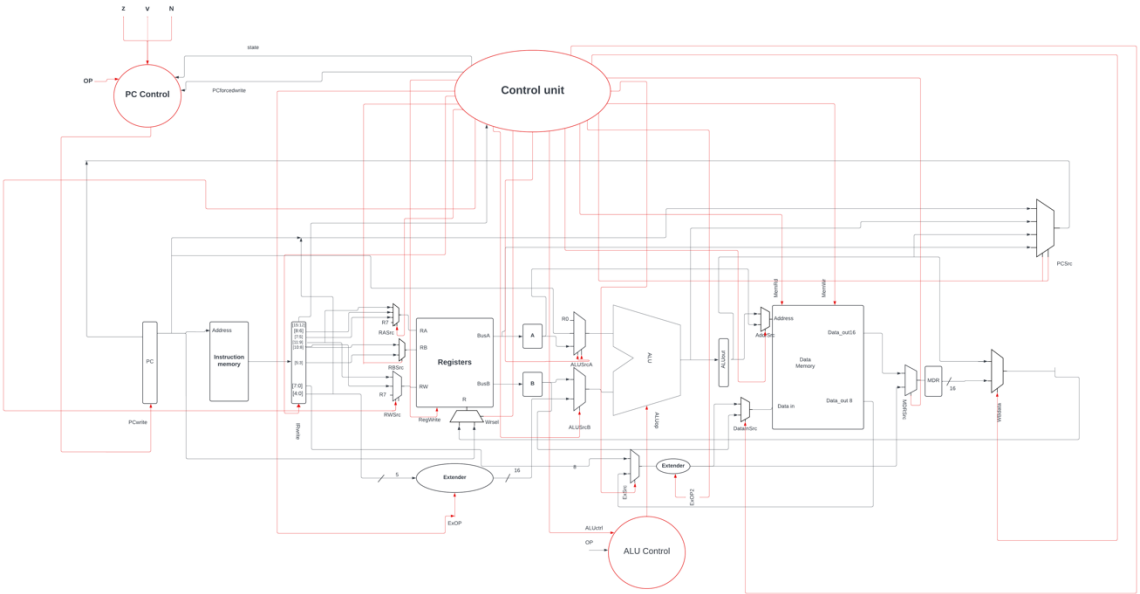

Figure9: Full Data Path

4 Testing

4.1 Test Case 1

```
// number summation

addi r1 ,r0 , 10 // summation from 0 to 10

call sum
jmp end

sum:
    add r2 ,r0,r0 // start with zero
    add r3 ,r0,r0 // for summation
    loop :
    addi r2,r2,1
    add r3,r3,r2
    beq  r2,r1 ,sumEND
    jmp loop

    sumEnd:
    ret
end:
```

Figure10: Test case1

regarr	0000, 000A, 000A, 0037,
regarr[0]	0000
regarr[1]	000A
regarr[2]	000A
regarr[3]	0037

Figure11: Test case Result

This code will calculate the sumation of the number in regiseter R1and will save the result in Register R3 as shown in the Figure_11, the value of the R3 is equal to 37 in hexa which equal 55 in decimal which is the sumation of nbumer 10.

4.2 Test Case 2

```
LW R1, R0, 8
LBU R2, R0, 8
LBS R3, R0, 8

SV R0, 10
SW R2, R0, 5
```

Figure12: Test case 2

mem	0A, 00, 00, 2C, 01, FF, 00, 86, FF, 38, FF, FE, 00, D3, (
mem[0]	0A
mem[1]	00
mem[2]	00
mem[3]	2C
mem[4]	01
mem[5]	FF
mem[6]	00
mem[7]	86
mem[8]	FF
mem[9]	38
mem[10]	FF
mem[11]	FE

regarr	0000, 38FF, 00FF, FFFF,
regarr[0]	0000
regarr[1]	38FF
regarr[2]	00FF
regarr[3]	FFFF

Figure13: Test case 2 Results

First, the program loads data from memory into registers using three different instructions. The `LW` instruction loads a full word (typically 16 bits) from memory into the register `R1`. The `LBU` instruction loads an unsigned byte (8 bits) from memory into the register `R2`, zero-extending the byte. The `LBS` instruction loads a signed byte from memory into the register `R3`, sign-extending the byte. In all these load operations, the memory address is computed by adding `8` to the value in `R0`.

Next, the program stores values into memory. The `SV` instruction, stores the value `10` in memory at the address specified by `R0`. Finally, the `SW` instruction stores the word contained in `R2` into memory at the address computed by adding `5` to the value in `R0`.

4.3 Test Case 3

```
AND R1 , R1 , R0

BLTZ R1 , negative
BEQZ R1 , zero

negative:
    ADDI R1, R1, 2
    JMP END
zero:
    ADDI R1, R1, 4
    JMP END
END:
```

Figure14: Test case 3

⊟ regarr	0000, 0004, 0001, 0012,
⊞ regarr[0]	0000
⊞ regarr[1]	0004
⊞ regarr[2]	0001
⊞ regarr[3]	0012

Figure16: Test case15 Result

This code will AND the value of R1 with Zerothen it will go to 2 different Brach instruction BLTZ which is not taken then it will go to instrctuion branch equal 0, which will be taken and will add 4 to R0 to show its taken as shown in figure above.