Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Computer Architecture

ENCS4370

Project #2 report

Multicycle Processor Implementation

Prepared By:

1. Yazan AbuAlown 1210145
2. Noor Mouadi - 1211080
3. Hazar Michael - 1201838

Instructor : Ayman Hroub

Date : 29/1

Section : 1

Abstract

The aim of the project is to design a special multicycle computer processor that operates on 32-bit instructions. This processor has 16 general-purpose registers, a program counter, and a stack pointer, and uses two distinct memories for instructions and data. The focus was to enable the processor to execute instructions, from simple arithmetic operations like addition and subtraction to more complex tasks including decision-making processes.

This report involves a detailed layout of the creation of processor's Datapath and control signals, showing the collaboration between its components. Additionally, it will present the results of the design process and provide the Verilog code, demonstrating the functionality of the processor.

Table of Contents

Table of figures

1. Design Specifications

1.1 Motivation

This project is inspired by the 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. Each instruction in the set is 32 bits in length.
2. The design includes 16 32-bit general-purpose registers, labeled from R0 to R15.
3. Program Counter (PC) is a 32-bit special-purpose register.
4. stack pointer (SP) is 32-bit special purpose register.
5. four distinct instruction types – R-type, I-type, J-type, and S-type.
6. The processor is designed with two separate physical memory units: one for storing instructions and the other for data.
7. Arithmetic Logic Unit is also used to decide the outcomes of certain instructions, like whether to take a branch or not.

1.2 Instruction Formats

The instruction set has the following instruction formats:

1.1.1 R-Type

Opcode (6 bits)	Rd (4 bits)	Rs1 (4 bits)	Rs2 (4 bits)	Unused (14 bits)

Where:
Opcode: specifies the type of operation the processor should perform.
Rd: is the register where the result of an operation will be stored.
Rs1: is the first operand register
Rs2: is the second operand register

1.1.2 I-Type

Opcode (6 bits)	Rd (4 bits)	Rs1 (4 bit)	Immediate (16 bits)	Mode (2bits)

Where:
Opcode: specifies the type of operation the processor should perform.
Rd: is the register where the operation's result is stored.
Rs1: is the address of first operand register.

Immediate: Represents a numerical value, unsigned for logic and signed for other instructions.
Mode bits: Used in load/store instructions to control base register increment/decrement.

1.1.3 J-Type

J-type instructions come in three formats: Jump, Call, and Return. Both Jump and Call instructions follow the format bellow, whereas in the Return instruction, the offset section is not used. This is because the Return instruction retrieves the top element from the stack and places it into the Program Counter.

Opcode (6 bits)	Jump Offset (26 bits)

1.1.4 S-Type

The S-type category includes two instructions: Push and Pop. In the Push instruction, the value in Rd is placed at the top of the stack. Conversely, the Pop instruction removes the top element from the stack and stores it in Rd.

Opcode (6 bits)	Rd (4 bit)	Unused (22 bits)

1.3 Instruction Set

The table below lists the specific instructions that have been implemented in the processor design.

Table 1 : Instruction Set Implementation

R-Type		
AND	Reg(Rd) = Reg(Rs) & Reg(Rt)	000000
ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	000001
SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	000010
I-Type		
ANDI	Reg(Rd) = Reg(Rs1) & Imm	000011
ADDI	Reg(Rd) = Reg(Rs1) + Imm	000100
LW	Reg(Rd) = Mem(Reg(Rs1) + Imm)	000101
LW.POI	Reg(Rd) = Mem(Reg(Rs1) + Imm) Reg[Rs1] = Reg[Rs1] + 1	000110
SW	Mem(Reg(Rs1) + Imm) = Reg(Rd)	000111
BGT	if (Reg(Rd) > Reg(Rs1)) Next PC = PC + sign_extended(Imm) else PC = PC + 1	001000
BLT	if (Reg(Rd) < Reg(Rs1)) Next PC = PC + sign_extended(Imm) else PC = PC + 1	001001
BEQ	if (Reg(Rd) = = Reg(Rs1)) Next PC = PC + sign_extended(Imm) else PC = PC + 1	001010
BNE	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended(Imm) else PC = PC + 1	001011
J-Type		
JMP	Next PC = {PC[31:26], Immediate}	001100
CALL	Next PC = {PC[31:26], Immediate} PC + 1 is pushed on the stack	001101
RET	Next PC = top of the stack	001110
S-Type		
PUSH	Rd is pushed on the top of the stack	001111
POP	The top element of the stack is popped, and it is stored in the Rd register	010000

2. Components of Datapath

Following our examination of the instruction set, it was determined that the following components are necessary to complete the design:

2.1 Multiplexer

The multiplexer is a key part in a computer's data setup. It acts like a selector, picking one signal from four possible inputs, based on a control signal. This component is used in the data setup to handle where data making the system more practical and useful.

In our Datapath design, the multiplexer is utilized multiple times.

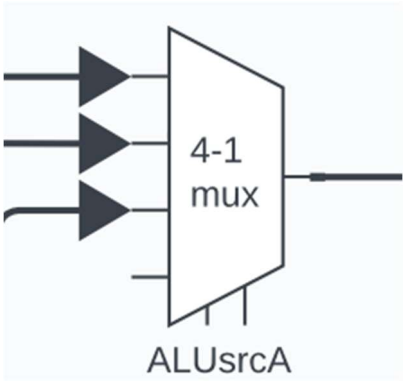


Figure 1 : Example of multiplexer use in the Datapath

For example, the logic of the multiplexer in figure 1 is as follows:

When ALUSrcA is set to 00, the input to the ALU is the Program Counter (PC) for operations like PC increment.

When ALUSrcA is 01, the ALU receives its input from the A register. This is for operations such as addition, subtraction, and bitwise AND, etc...

When ALUSrcA is 10, the input to the ALU is the top element of the stack, which is used in operations like stack increment or decrement.

2.2 Instruction Memory

In our processor implementation, memory has been divided into two distinct parts: Instruction Memory and Data Memory. This separation is designed to prevent conflicts that could arise from simultaneous actions like fetching instructions and loading or storing data.

The instruction memory is the part where instructions are stored, and it only needs to be able to read, not write, because the Datapath doesn't write instructions. It works like a simple logic for reading: receives a 32-bit address from the PC and outputs a 32-bit instruction, which corresponds to its design as word-addressable memory.

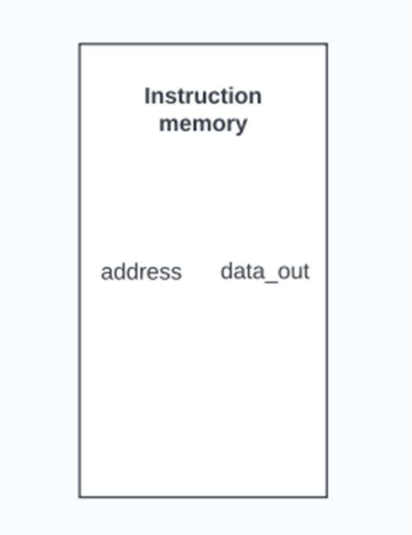


Figure 2 : Instruction memory

2.3 Data Memory

Figure 3 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 and two data lines: one to put data into storage 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.

The stack, an essential part of the data memory, is handled in the same way as the rest of the data memory, using these read and write signals.

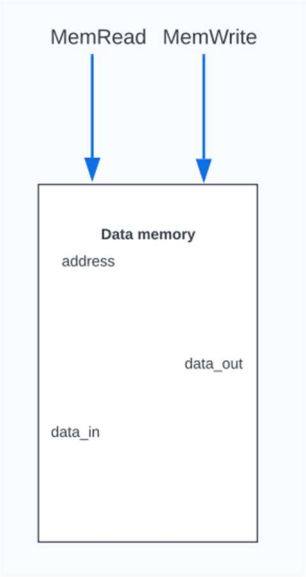


Figure 3 : Data Memory

2.4 Register file

The register file is a crucial component, including 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.

The register file includes a specific feature for the LW.POI instruction. This instruction requires the value in Rs1 to be incremented by one, in the same cycle where the register file is written. Therefore, when the control signal regWrite2 is set to 1, the data arriving on Bus2_W is written into Rd2, facilitating the required increment for the operation.

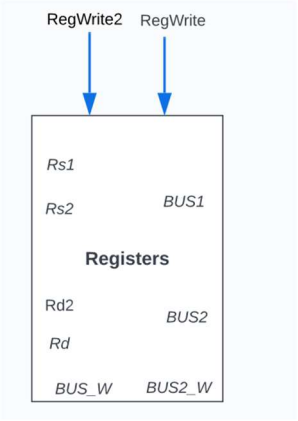


Figure 4 : Register File

2.5 Arithmetic Logic Unit (ALU)

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

In our Datapath, the ALU has two main inputs, where it gets two sets of 32 pieces of information, called bits. The ALU can do several things with these bits: it can combine them using a logic rule (AND), add them together, or subtract one from the other in two different ways. Then, based on a small 2-bit control signal, a 4-to-1 Mux picks which math or logic operation to use. The chosen result is then sent out of the ALU as the definitive answer.

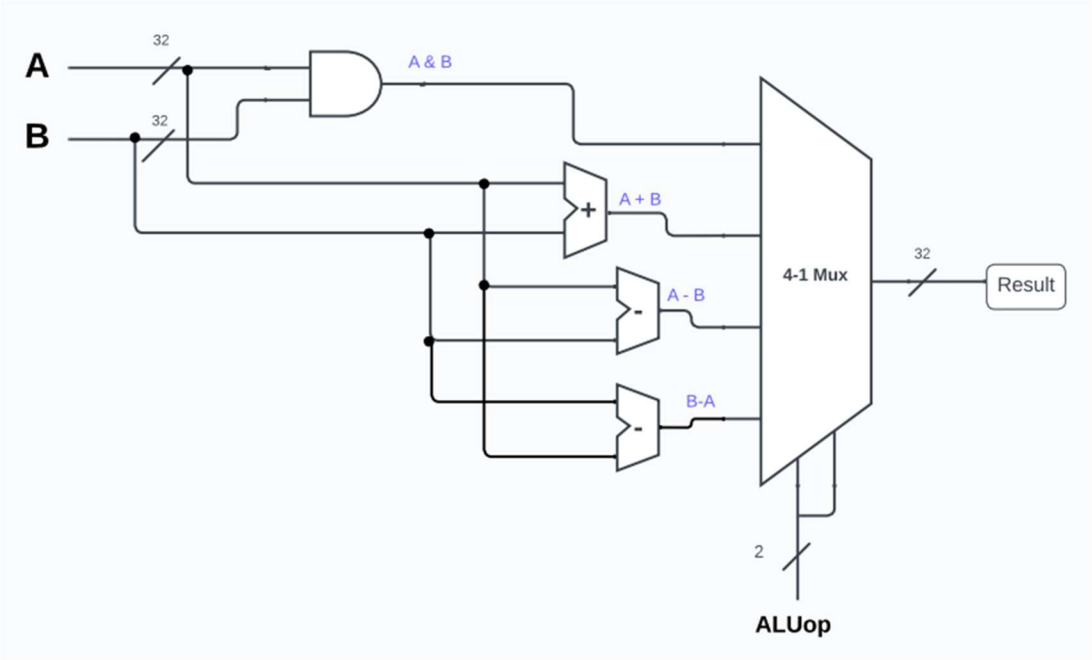


Figure 5: Internal structure of the 32-bit ALU

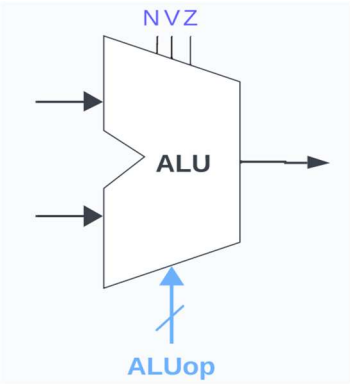


Figure 6 : ALU circuit symbol

3. Control Path

3.1 PC control unit

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, BLT, BEQ, and BNE). 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.

3.1.1 PC control Unit truth table

Table 2 : 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
BNE	0	X	X	1	X	0
BNE	0	X	X	0	7	1
BNE	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
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

3.1.2 Internal structure of the PC control unit

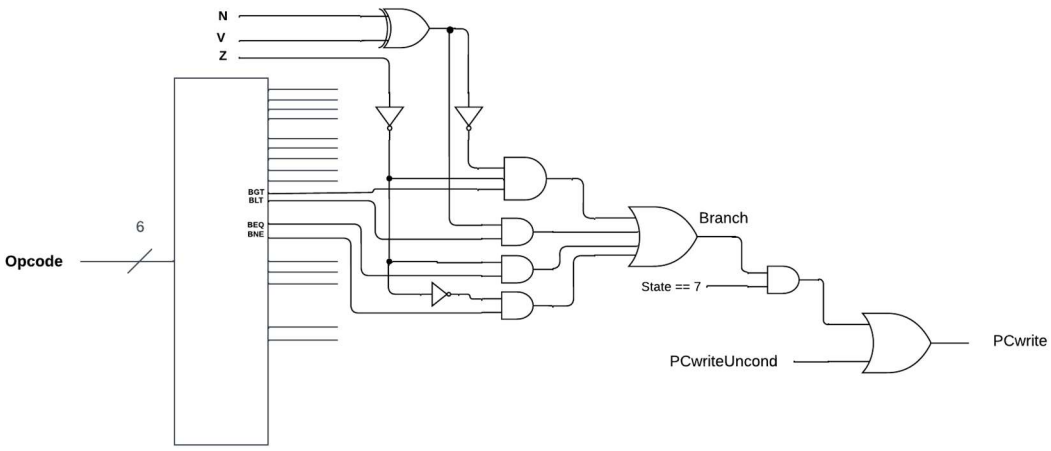


Figure 7 : PC control block diagram

3.1.3 Boolean Expression

The PC Control Unit determines program flow based on the mentioned flags. For 'Branch if Greater Than' (BGT), it needs Z=0 and N=V, indicating a positive result. 'Branch if Equal' (BEQ) requires Z=1, indicating equality. Conversely, 'Branch if Not Equal' (BNE) looks for Z=0, signaling inequality. 'Branch if Less Than' (BLT) checks N≠V, indicating a negative result. These conditions enable accurate branching decisions, yet it's very important to ensure that the state equals 7, indicating the Branch completion state, even if one of these conditions is met.

Presented below is the Boolean Expression:

$$Branch = (BEQ.z) + (BNE.z') + (BGT.z'.(N \oplus V)') + (BLT.(N \oplus V))$$

$$PCwrite = (state == 7).Branch + PCwriteUncond$$

Where ⊕ indicates *xor*

3.2 ALU control unit

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 Unit 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.

Table 3 : ALU control truth table

Inputs		**Output**
Opcode	**ALUCtrl (A_1A_0)**	**ALUop (F_1F_0)**
AND (000000)	00	00
ANDI (000011)	00	00
ADD (000001)	00	01
ADDI (000100)	00	01
SUB (000010)	00	10
X	01	01
X	10	10
X	11	11

The ALU Control Signals determine the operation to be executed by the ALU based on the opcode of the instruction.

Here's a summary of our ALU Control Signals and the resulting ALU operations:

- When ALUCtrl is '00',' the ALU operation is determined by the opcode of the instruction. For instance, when the opcode corresponds to 'ADDI' or 'ADD', the ALUop is set to '01,' which is reserved for addition. Similarly, 'ANDI' and 'AND' opcodes map to ALUop '00' for bitwise and operations, while 'SUB' corresponds to ALUop '10,' reserved for subtraction.

- ALUCtrl '01' indicates an addition operation.
- ALUCtrl '10' indicates a subtraction operation.
- ALUCtrl '11' indicates a reverse subtraction operation.

3.2.2 ALU control unit Boolean expression

$$F_1 = (A_1' A_0' . SUB) + A_1$$
$$F_0 = (A_1' A_0' . ADD) + (A_1' A_0' . ADDI) + A_0$$

3.3 Main control unit

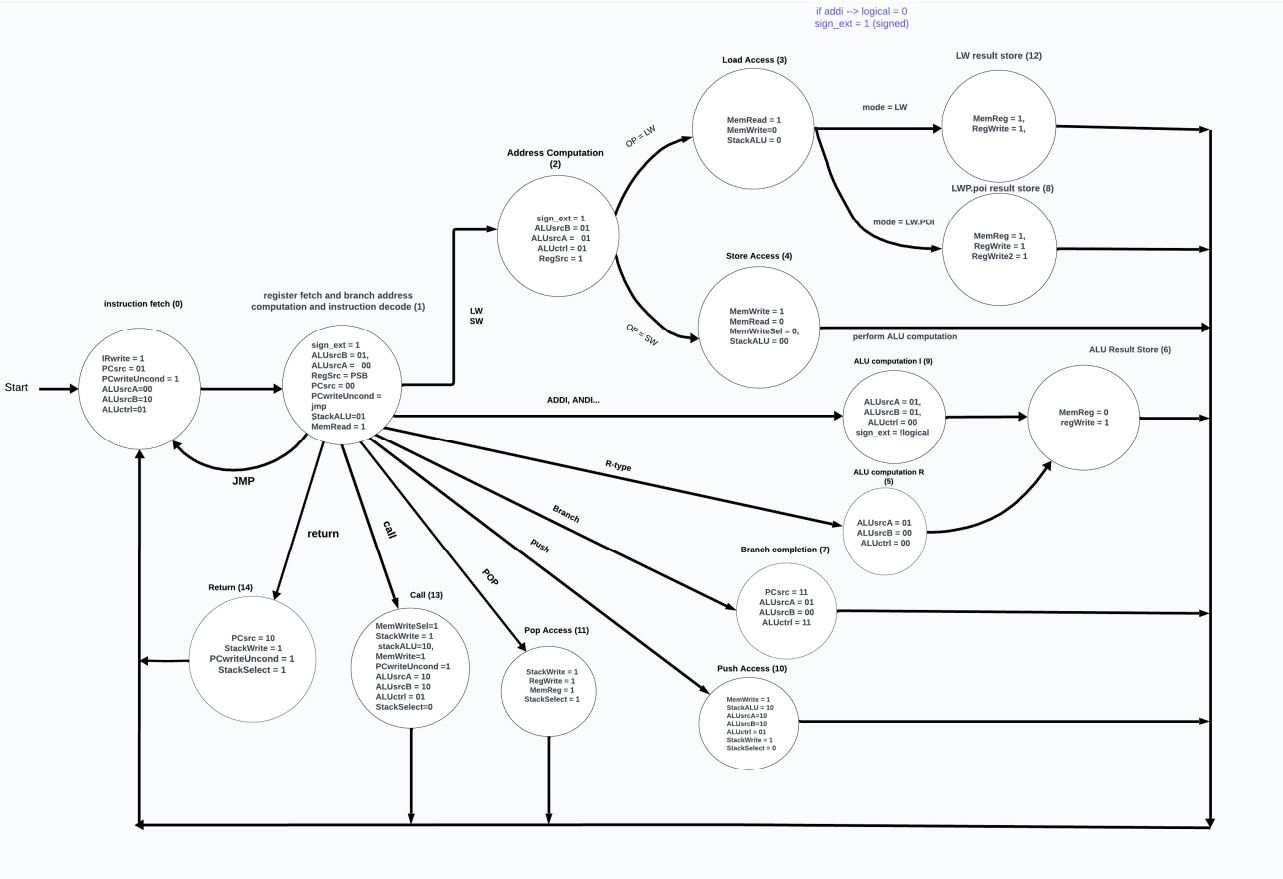
3.3.1 control signals

Signal name	Effect when de-asserted	Effect when asserted
signExt	The immediate operand is extended	The immediate operand is sign extended
MemWriteSel	Data written into the memory is the ALUout output	Data written into the memory is the PC+1 value
MemReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
RegSrc	The register operand is in rs2 field	The register operand is in rd field
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
RegWrite2	None	The general-purpose register specified by the Rd2 register number is written with the value on the BUS2_W data input.
MemReadData	No	Content of instruction memory at the location specified by the Address input is put on Memory data output.
MemWriteData	None.	Memory contents at the location specified by the Address input is replaced by the value on the Write data input.
IRWrite	None	The output of the memory is written into the IR.
PCWriteUncond	None	The PC is written; the source is controlled by PCSource.
branch	None	The PC is written if the operation is a branch is the flags satisfy the condition imposed

StackSelect	The input to the stack is the result of the ALU	The input to the stack is decremented SP (SP – 1)

Signal name	Value (binary)	Effect
ALUSrcA	00	The first ALU operand is the PC.
	01	The first ALU operand comes from the A register.
	10	The first ALU operand comes from the Stack pointer
ALUctrl	00	The ALU operation is determined by the opcode
	01	The ALU performs an add operation.
	10	The ALU performs a sub operation.
	11	The ALU performs a reverse SUB operation
ALUop	00	The ALU performs an AND operation
	01	The ALU performs an add operation
	10	The ALU performs a sub operation
	11	The ALU performs a reverse Sub operation
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the unsigned, lower 16 bits of the IR.
	10	The second input to the ALU is the constant 1.
	11	The second input to the ALU is the constant 0
PCSource	00	The jump target address is sent to the PC for writing
	01	The output of the ALU (PC+1) are sent to the PC for writing.
	10	The output of the memory is sent to the PC for writing
	11	The stored value in ALUout (For branch target) is sent to PC for writing
StackALU	00	The input to data memory is ALUou
	01	The input to the address of data memory is the decremented stack (SP - 1).
	10	The input to the address of data memory is the stack (SP).

3.3.2 Main control state diagram



The initial two states of any instruction cycle, the instruction fetch and instruction decode are constant.

During the instruction fetch phase, the processor prepares to read the next instruction from memory and increment the Program Counter (PC). This is achieved by asserting the IRWrite signal, enabling the fetched instruction to be loaded into the Instruction Register (IR). Simultaneously, the PCsrc is set to select the output of the ALU, which is configured to increment the PC by one. This is facilitated by setting the ALUSrcA to use the PC as the ALU's first operand and the ALUSrcB to provide a constant value of one to the ALU. The ALUctrl signal is also asserted to an add operation. The PCWriteUncond signal is activated, indicating that the PC should be updated unconditionally, preparing it for the next instruction fetch cycle.

In the instruction decode phase, the processor sets up for different instruction types. In this phase the immediate operand which is the lower 16 bits of the IR will be extended (sign_ext = 1) and uses it as a second source of the ALU (ALUsrcB = 01), PC will be the first ALU operand (ALUSrcA = 00), (ALUop = 01) which will lead to calculate the branch target address (this will save a cycle if the instruction was branch). This phase also pre fetches the required registers for the execution phase. The source operands for store, push, branch are Rd and Rs1, so RegSrc is set to 1 to fetch these. For the other instructions however, Rs1 and Rs2 are fetched, so RegSrc is set to 0 to fetch Rs2 instead of Rd. For jump and call instructions (PCsrc will be set to 00, so that the PC value will be updated to the jump target address which is ({PC[31:26], inst[25:0]}) and PCwriteUncond =1 in this case because the writing on the PC is unconditionaly in the case of Jump). The StackALU signal is set to access the memory and save the data in the MDR.

If the instruction was branch, the next state will be as follows: ALUsrcA is set to 01, selecting the A register as the first operand of the ALU, necessary for calculating the branch target address. ALUsrcB is set to 00, which means the second ALU operand comes from the B register. The ALUctrl signal is set to 11, instructing the ALU to perform a reverse subtraction. This reverse SUB operation is used to determine if the value in the destination register (rd) is greater than the source register (rs1). PCsrc is then set to 11, which routes the calculated branch target address from the ALU output buffer to the PC. However, the decision to actually write this address into the PC or write the incremented PC (PC + 1) is determined by the PC control unit, which assesses whether the branch conditions have been met.

If the instruction was R-type, the first ALU operand is selected as buffer A, achieved by setting ALUsrcA to 01. Simultaneously, the second ALU operand is selected as buffer B, which is accomplished by setting ALUsrcB to 00. The operation that the ALU performs is depending upon the opcode by setting ALUctrl to 00. Under this configuration, the ALU can execute one of three operations: ADD, AND, or SUBTRACT, with the specific operation chosen based on the given opcode.

If the instruction was ADDI, the first ALU operand is selected as buffer A, achieved by setting ALUsrcA to 01, while the second ALU operand is the sign extended lower 16 bit of the instruction, we achieve this by putting the sign_extend into 1 and ALUSrcB = 01 and the operation that the ALU performs depends on the opcode by setting ALUctrl to 00. The same steps are followed, if the instruction was ANDI except that the immediate is unsigned (the sign_extend = 0) in this case.

ALU result state is used to save the output from the ALU output buffer (by putting the MemReg = 0) to the Register file and specifically to the Rd register via BUS_W by activating the writing on the register file (RegWrite= 1).

If the instruction was push, Rd will be pushed on the top of the stack, to make this happen, the MemWrite will be 1 enabling the writing to the data memory since the stack is part of it, ALU is used to increment the Stack pointer by one , StackALU = 10 indicating that the coming address to the data memory is the Stack pointer, the fisrt ALU operand will be the Stack pointer by selecting ALUsrcA=10, the second operand to the ALU is the constant 1 (ALUsrcB=10), ALUctrl = 01 so the operation is ADD, also StackSelect = 0 indicating that the written value to the SP is coming from the ALU, StackWrite = 1 to enable writing the incremented address to the Stack pointer.

To execute a "pop" instruction, the process begins by decrementing the stack pointer using a special adder. This adjustment is important as it allows the pointer to reference the top element of the stack. Once the stack pointer is decremented, the StackWrite is set to 1, enabling the modification of the stack pointer to be written to the SP. Also the StackSelect is set to 1 indicating that the written value is the decremented value. Finally, this element will be stored in the Rd register so during this phase, RegWrite is set to 1 to allow writing to the register file, the value to be written is already in the MDR since it has been read at the decode state. Additionally, MemReg is set to 1, indicating that the data being written back to the register file originates from the data memory.

During the Computation state, the processor computes memory addresses for load and store instructions. This involves extending a 16-bit immediate value with zero-sign extension by setting the sign_ext signal to 0, ALUsrcA is set to 01 indicating RS1 as the first ALU source, ALUsrcB is set to 01 indicating the extended immediate value as the second ALU source, ALUctrl is set to 01 indicating an ADD operation, and he RegSrc signal is set to 1 to designate Rd as the second register source. This configuration is necessary for store operations where the content of Rd needs to be stored in memory at the address Rs1 + immediate. The next state depends on the opcode.

During the Load access state, the processor sets the MemRead signal to 1 to enable the memory for reading. Additionally, the ALUstack signal is set to 0 to ensure that the memory address from which the value will be loaded corresponds to the ALU result buffer the next state in the load instruction depends on the mode if it is 00 the next state will be (LW store result) and if it 01 next state will be (LW.POI store result).

During the LW store result state, the processor sets RegWrite to 1, enabling it to store the value comes from memory into the register file. Additionally, it sets MemReg signal to 1, directing the register to hold the data read from memory.

In LW.POI store result state, the processor sets RegWrite to 1, enabling it to store the value comes from memory into the register file. Additionally, it sets MemReg signal to 1, directing the register to hold the data read from memory and it sets RegWrite2 to 1, allowing for the increment of the address of RS1 by 1, which is performed using a separate adder.

During the store access state, the StackALU is configured to 00, specifying that the address originates from the ALUres buffer. Simultaneously, MemWriteSel is set to 0, allowing the data to be sourced from the register file. Furthermore, the MemWrite signal is activated to a value of 1, facilitating the writing of the stored value into memory.

During the call state, MemWriteSel is adjusted to 1 MemWriteSel is set to 1, indicating that data_in will correspond to pc+1. StackALU signal is set to 10, designating the stack pointer as the address for writing the pc. MemWrite is activated to write the value of PC+1 into memory. ALUsrcA is configured to 10, making the stack pointer the first ALU source, while ALUsrcB is set to 10, making the immediate value 1 the second source. ALUctrl is assigned the value 01, signifying an ADD operation. stackSelect is set to 0, indicating that the stack pointer will be the ALU result . StackWrite is enabled to update the stack pointer value to the ALU result (SP+1). Finally, PCwriteUncond is set to 1, updating the pc value to be the new address {PC[31:26],Instruction[25:0]}.

During the return state, StackSelect is adjusted to 1 to provide the stack pointer with its new value, which is SP-1. StackWrite is set to 1 to write the updated stack pointer value (SP=SP-1). PCsrc is set to 10 to designate the address to which the processor should return. This address is already stored in the MDR register during the decode state.

3.3.3 State Assignments :

State name	State abbreviation	State number
Instruction fetch	IF	0
Instruction decode	ID	1
Address computation	AC	2
Load access	LA	3
Store access	SA	4
ALU computation for R-type	R-type	5
Result store	RS	6
Branch completion	BC	7
Load word post increment	LW_POI	8
ALU computation for I-type	$ALUcomp_I$	9
Push access	PushA	10
Pop access	PopA	11
Result store to memory	$LW\_ResStore$	12
Call	Call	13
Return	Return	14

3.3.4 Control unit State table :

Inputs			Outputs								
State	Opcode	mode	Next state	StackSelect	MemWriteSel	PCWrite Uncond	IRwrite	MemReg	Reg Write	RegSrc	RegWrite2
IF	X	XX	ID	X	X	1	1	X	0	X	0
ID	JMP	XX	IF	X	X	1	0	X	0	0	0
ID	Push	XX	Push A	X	X	0	0	X	0	1	0
ID	SW	XX	AC	X	X	0	0	X	0	1	0
ID	LW	XX	AC	X	X	0	0	X	0	1	0
ID	R-type	XX	ALU-R	X	X	0	0	X	0	0	0
ID	B	XX	BC	X	X	0	0	X	0	1	0
ID	POP	XX	PopA	X	X	0	0	X	0	0	0
AC	LW	XX	LA	X	X	0	0	X	0	1	0
AC	SW	XX	SA	X	X	0	0	X	0	1	0
BC	XX	XX	IF	X	X	0	0	X	0	X	0

LA	X	LW	LW_result store	X	X	0	0	X	0	X	0
LA	X	LW_POI	LW_POI_RES	X	X	0	0	X	0	X	0
LW_result store	X	XX	IF	X	X	0	0	1	1	X	0
LW_poi_res	X	XX	IF	X	X	0	0	1	1	X	1
SA	X	XX	IF	X	0	0	0	X	0	1	0
ADDI	X	XX	RS	X	X	0	0	X	0	0	0
ANDI	X	XX	RS	X	X	0	0	X	0	0	0
RS	X	XX	IF	X	X	0	0	0	1	X	0
R-type	X	XX	RS	X	X	0	0	X	0	X	0
Push_A	X	XX	IF	0	0	0	0	1	1	X	0
PopA	X	XX	IF	1	X	0	0	1	0	X	0
Call	X	XX	IF	0	1	1	0	X	0	X	0
Return	X	XX	IF	1	X	1	0	0	0	X	0

Inputs				Outputs								
State	Opcode	mode	Next state	Stack ALU	PCsrc	StackWrite	Sign_ext	ALUctrl	ALUsrcA	ALUsrcB	MemWrite	MemRead
IF	X	XX	ID	XX	01	0	X	01	00	10	0	0
ID	JMP	XX	IF	01	00	0	1	01	00	01	0	1
ID	Push	XX	Push A	01	00	0	1	01	00	01	0	1
ID	SW	XX	AC	01	00	0	1	01	00	01	0	1
ID	LW	XX	AC	01	00	0	1	01	00	01	0	1
ID	R-type	XX	ALU-R	01	00	0	1	01	00	01	0	1
ID	B	XX	BC	01	00	0	1	01	00	01	0	1
ID	POP	XX	PopA	01	00	0	1	01	00	01	0	1
AC	LW	XX	LA	XX	XX	0	1	01	01	01	0	0
AC	SW	XX	SA	XX	XX	0	1	01	01	01	0	0
BC	XX	XX	IF	XX	11	0	X	11	01	00	0	0
LA	X	LW	LW_result store	00	XX	0	X	XX	XX	XX	0	1
LA	X	LW_POI	LW_POI_RES	00	XX	0	X	XX	XX	XX	0	1
LW_result store	X	XX	IF	XX	XX	0	X	XX	XX	XX	0	0
LW_poi_res	X	XX	IF	XX	XX	0	X	XX	XX	XX	0	0
SA	X	XX	IF	XX	XX	0	X	XX	XX	XX	1	0
ALU computation I	ADDI	XX	RS	XX	XX	0	1	00	01	01	0	0
ALU computation I	ANDI	XX	RS	XX	XX	0	0	00	01	01	0	0
RS	X	XX	IF	XX	XX	0	X	XX	XX	XX	0	0
R-type	X	XX	RS	XX	XX	0	X	00	01	00	0	0
Push_A	X	XX	IF	10	XX	1	X	01	10	10	1	0
Pop	X	XX	IF	XX	XX	1	X	XX	XX	XX	0	0
Call	X	XX	IF	10	00	1	X	01	10	10	1	0
Return	X	XX	IF	XX	10	1	X	XX	XX	XX	0	0

3.3.5 Boolean equations :

Below are the Boolean equations for each of the control signals.

$$StackSelect = PopA + Return$$

$$MemWrite = Call$$

$$PCwriteCond = IF + ID.Jmp + Call + Return$$

$$IRwrite = IF$$

$$MemReg = LW\_ResStore + LW\_POI + PushA + PopA$$

$$RegWrite = LW\_ResStore + LW\_POI + Rs + PushA$$

$$RegSrc = ID.Push + ID.SW + ID(BEQ + BNE + BLT + BGT) + SA$$

$$RegWrite2 = LW\_POI$$

$$StackALU_1 = PushA + Call$$

$$StackALU_0 = ID$$

$$PCsrc_1 = BC + Return$$

$$PCsrc_0 = IF + BC$$

$$StackWrite = PushA + Pop + Call + Return$$

$$SignExt = ID + AC + ALUcomp_I.ADDI$$

$$ALUctrl_1 = BC$$

$$ALUctrl_0 = IF + ID + AC + BC + PushA + Call$$

$$ALUsrcA_1 = PushA + Call$$

$$ALUsrcA_0 = AC + BC + ALUcomp_I + R - type$$

$$ALUsrcB_1 = IF + PushA + Call$$

$$ALUsrcB_0 = ID + AC + ALUcomp_I$$

$$MemWrite = SA + PushA + Call$$

$$MemRead = ID + LA$$

4. Full Datapath

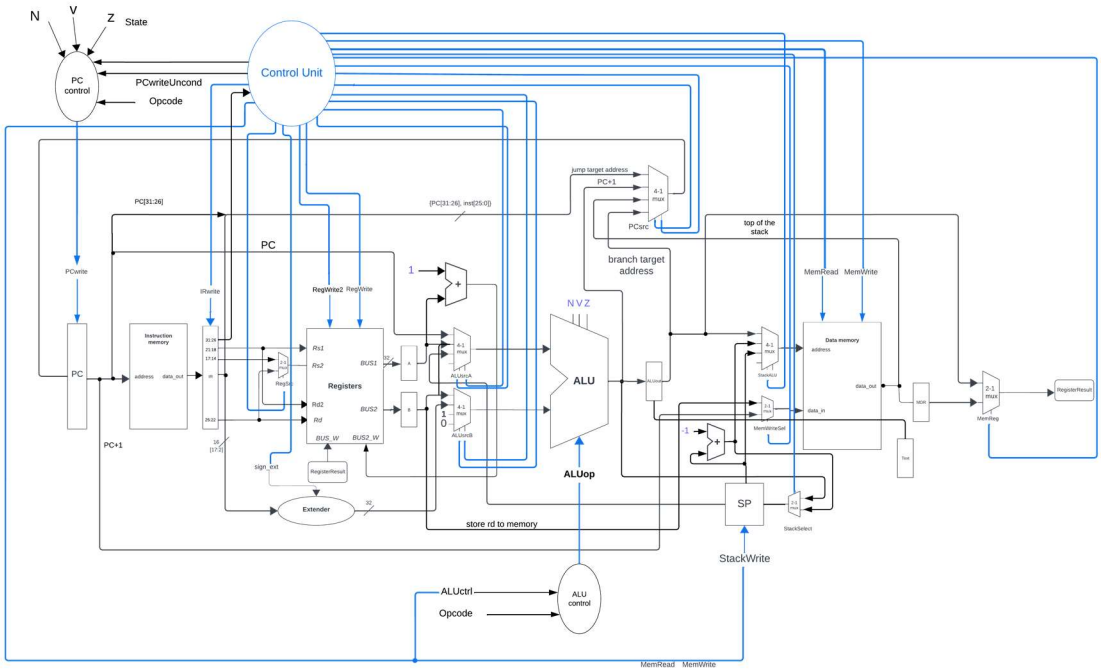


Figure 8 : Full Data path

5. Testing

5.1 Finding the maximum value:

```
addi r1, r0, 50
addi r2, r0, 123

call max
addi r10, r0, 999

; function that takes numbers in R1 and R2 and returns
; maximum in R3

max :

    bgt r1, r2, else
        addi r3, r2, 0
        jmp endif
    else :
        addi r3, r1, 0
    endif : ret

```

regArray	0, 50, 123, 123, ...
regArray[0]	0
regArray[1]	50
regArray[2]	123
regArray[3]	123
regArray[4]	7612
regArray[5]	6638
regArray[6]	10040
regArray[7]	3930
regArray[8]	4150
regArray[9]	6406
regArray[10]	999
regArray[11]	8572
regArray[12]	16324
regArray[13]	8840
regArray[14]	8258
regArray[15]	11228

```
reg [31:0] regArray [0:15] = '{0, 4198, 5596, 14426, 7612,
                               6638, 10040, 3930, 4150,
                               6406, 5400, 8572, 16324, 8840,
                               8258, 11228};
```

In this program, the initial values assigned are 50 for r1 and 123 for r2. Upon invoking the max function, once the function concludes its execution, r3 is set to hold the maximum value present in the registers. In this particular simulation, r3 retains the value 123, signifying the maximum value among the registers.

5.2 Testing S-type instructions:

```
1   addi r4, r0, 122
2   addi r5, r0, 1244
3   addi r6, r0, 12321
4   push r4
5   push r5
6   push r6
7   pop r6
8   pop r7
9   pop r8
10  add r10, r7, r8
11  addi r15, r0, 10
12  sw r10, r15, 0
```

mem	122, 1244, 1232...
mem[0]	122
mem[1]	1244
mem[2]	12321
mem[3]	?
mem[4]	?
mem[5]	?
mem[6]	?
mem[7]	?
mem[8]	?
mem[9]	?
mem[10]	1366

regArray	0, 4198, 5596, 1...
regArray[0]	0
regArray[1]	4198
regArray[2]	5596
regArray[3]	14426
regArray[4]	122
regArray[5]	1244
regArray[6]	12321
regArray[7]	1244
regArray[8]	122
regArray[9]	6406
regArray[10]	1366
regArray[11]	8572
regArray[12]	16324
regArray[13]	8840
regArray[14]	8258
regArray[15]	10
mem	122, 1244, 1232...

Initially, the registers hold the values: r4=122, r5=1244, and r6=12321. These values are sequentially pushed onto the stack and then popped into r6, r7, and r8, resulting in the assignments: r6=12321, r7=1244, and r8=122. The sum of r7 and r8 yields r10=1366, and r15 is set to 10. Using the sw instruction, the value in r10 is stored at the memory address 10, which corresponds to the content of r15. This value is reflected in the above simulation

5.3 Summing array elements:

```
;sum array elements in address 80 to 87

addi r12, r0, 80 ;address of the array
addi r14, r0, 87 ;address of the last eleme
addi r1, r0, 0 ;sum
loop :
        bgt r12, r14, endloop
        lw_poi r2, r12, 0
        add r1, r1, r2
        jmp loop

endloop : addi r3, r1, 0
```

regArray	0, 383, 12, 383, ..
regArray[0]	0
regArray[1]	383
regArray[2]	12
regArray[3]	383
regArray[4]	7612
regArray[5]	6638
regArray[6]	10040
regArray[7]	3930
regArray[8]	4150
regArray[9]	6406
regArray[10]	5400
regArray[11]	8572
regArray[12]	88
regArray[13]	8840
regArray[14]	87
regArray[15]	11228

mem[80]	8
mem[81]	44
mem[82]	432
mem[83]	-122
mem[84]	-200
mem[85]	-2
mem[86]	211
mem[87]	12

This program is designed to compute the sum of all array elements located between addresses 80 and 87, which are loaded into r13 and r14 respectively. The sum of this array is then loaded into r1. If we add the contents of the array, the resulting sum is 383. Upon completion, this sum is stored in r3, as depicted in the above simulation.

5.4 Implementing multiplication :

```
; implementing multiplication

addi r1, r0, 23
addi r2, r0, 12
call mul
jmp end

; function that takes args in r1 and r2 and
; returns multiplication in r3

mul :

    push r2 ; preserving operands

    addi r3, r0, 0 ;result

    loop : beq r2, r0, endloop
           addi r2, r2, -1
           add r3, r3, r1
           jmp loop

    endloop :
    pop r2
    ret

end : addi r8, r0, 999 ;indicates the end of the program
```

regArray	0, 23, 12, 276, 7...
regArray[0]	0
regArray[1]	23
regArray[2]	12
regArray[3]	276
regArray[4]	7612
regArray[5]	6638
regArray[6]	10040
regArray[7]	3930
regArray[8]	4150
regArray[9]	6406
regArray[10]	5400
regArray[11]	8572
regArray[12]	16324
regArray[13]	8840
regArray[14]	8258
regArray[15]	11228

mem	3, 12, ?, ?, ?, ?, ?...
mem[0]	3
mem[1]	12

The assembly code snippet illustrates a multiplication algorithm by repeated addition. It loads the values 23 and 12 into registers r1 and r2, respectively, then enters the mul subroutine which iteratively adds the value in r1 to r3 (initialized to 0) r2 times. The loop continues until r2 decrements to 0, at which point r3 holds the product of 23 and 12. The subroutine preserves the initial value of r2 by pushing it onto the stack before the loop and popping it off after. Finally, the program jumps to end, bypassing the subroutine to terminate.