Faculty of Engineering & Technology

Electrical & Computer Engineering Department

**Computer Architecture**

**ENCS4370**

**Project #2 report**

# Multi cycle Processor Implementation

**Prepared By**:

1. Diana Naseer-1210363 Sec:1

2. Lana Musaffer-1210455 Sec:2

3. Roaa Sroor-1221727 Sec:3

**Instructor:** Dr. Ayman Hroub

**Date** : 1/7/2025.

## Abstract

The goal of this project was to build a Multi cycle computer processor designed to handle 16-bit instructions. Equipped with eight general-purpose registers, a program counter to keep track of operations, and a separate return register for functions, this processor uses two different types of memory—one for storing instructions and another for data. Our focus was to make sure the processor could carry out a wide range of tasks, from basic arithmetic like adding and subtracting to more complex decisions that require multiple steps.

This report lays out how we created the processor's pathways and control signals, showing how each part works together. We also share the results of our design efforts and provide the actual Verilog code, which illustrates how the processor functions.

## Contents

## Table of figures:

# List of Tables

# 1 – Design and Implementation:

## 1.1 Motivation:

This project develops a multicycle processor based on RISC architecture, tailored to manage 16-bit instructions efficiently. Key features **include**:

1. **Instruction Design**: The instruction size and the word size is 16 bits.
2. **Registers**: Includes eight 16-bit registers for flexible data handling.
3. **Control Units**: Features a 16-bit Program Counter (PC) and a 16-bit RR (Return Register) to store the return address during function calls.
4. **Instruction Types**: Supports R-type, I-type, and J-type instructions, facilitating a wide range of operations.
5. **Dual Memory**: Utilizes separate memories for instructions and data, enhancing execution efficiency.
6. **ALU**: The Arithmetic Logic Unit performs both basic and complex operations, influencing decision-making processes.

## 1.2 Instruction Types and Formats:

This ISA has three instruction types. These instruction types have the following formats:

### 1.2.1: R-Type:

| | | | | |
|---|---|---|---|---|
| Opcode (4 bits) | Rd (3 bits) | Rs (3 bits) | Rt (3 bits) | Function (3 bits) |

Where:

- **4-bit opcode**: opcode.
- The opcode is zero for all R-Type instructions.
- **3-bit Rd:** destination register
- **3-bit Rs**: first source register
- **3-bit Rt**: second source register

- **3-bit**: Function. This field determines the specific operation of the instruction.

### 1.2.2: I-Type:

| Opcode (4 bits) | Rs (3 bits) | Rt (3 bits) | Function (16 bits) |
|---|---|---|---|

Where:

- **4-bit opcode:** opcode.  3-bit Rs: first source register
- **3-bit Rt**: destination register
- The immediate value is zero-extended for logical instructions and sign-extended for all other instructions.
- In the case of BEQ and BNE instructions, which are I-type instructions, the branch target is calculated as follows:

  Branch target = Current PC + sign extended immediate

### 1.2.3: J-Type:

| Opcode (4 bits) | 9-bit offset | Function (3 bits) |
|---|---|---|

Where:

- **4-bit opcode:** opcode.
- The opcode is 1 for all J-Type instructions.
- **3-bit**: Function. This field determines the specific operation of the instruction.  The jump target address is calculated by concatenating these two fields: PC[15:9], 9-bits offset.

## 1.3 Instructions' Encoding:

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

*Table 1: Instructions on Set Implementation*

| Instruction | Meaning | Opcode Value | Function Value |
|---|---|---|---|
| **R-Type Instructions** | | | |
| AND Rd, Rs, Rt | Reg(Rd) = Reg(Rs) & Reg(Rt) | 0000 | 000 |
| ADD Rd, Rs, Rt | Reg(Rd) = Reg(Rs) + Reg(Rt) | 0000 | 001 |
| SUB Rd, Rs, Rt | Reg(Rd) = Reg(Rs) - Reg(Rt) | 0000 | 010 |
| SLL Rd, Rs, Rt | Reg(Rd) = Reg(Rs) << Reg(Rt) | 0000 | 011 |
| SRL Rd, Rs, Rt | Reg(Rd) = Reg(Rs) >> Reg(Rt) | 0000 | 100 |
| **I-Type Instructions** | | | |
| ANDI Rt, Rs, Imm | Reg(Rt) = Reg(Rs) & Imm | 0010 | NA |
| ADDI Rt, Rs, Imm | Reg(Rt) = Reg(Rs) + Imm | 0011 | NA |
| LW Rt, Imm(Rs) | Reg(Rt) = Mem(Reg(Rs) + Imm) | 0100 | NA |
| SW Rt, Imm(Rs) | Mem(Reg(Rs) + Imm) = Reg(Rt) | 0101 | NA |
| BEQ Rs, Rt, Imm | if (Reg(Rs) == Reg(Rt)) Next PC = branch target else Next PC = PC + 1 | 0110 | NA |
| BNE Rs, Rt, Imm | if (Reg(Rs) != Reg(Rt)) Next PC = branch target else Next PC = PC + 1 | 0111 | NA |

| | | | |
|---|---|---|---|
| FOR Rs, Rt | · Rs stores the loop target address, i.e., the address of the first instruction in the loop block<br>· Rt stores the initial number of the loop iterations, i.e., the initial value of the loop counter<br><br>· The Rt register is decremented at the end of each iteration. The loop exits when the content of the Rt register becomes zero<br><br>· The immediate field is ignored in this instruction | 1000 | NA |
| **J-Type Instructions** | | | |
| JMP Offset | Next PC = jump target | 0001 | 000 |
| CALL Offset | Next PC = jump target<br><br>PC + 1 is stored on the RR | 0001 | 001 |
| RET | Next PC = value of the RR<br><br>The 9-bit field is ignored in this instruction | 0001 | 010 |

# 2- Datapath Components:

## 2.1: Instruction Memory

In this processor design, the memory system is strategically partitioned into two distinct sections: Instruction Memory and Data Memory. This division is crucial for mitigating potential conflicts that could occur from concurrent operations such as instruction fetching and data manipulation.

The Instruction Memory is solely dedicated to storing the processor's instructions. Its primary function is to facilitate read operations—it does not support writing since Datapath is not designed to modify instructions once they are stored. This component functions by receiving a 16-bit address from the Program Counter (PC) and, in return, it outputs a 16-bit instruction corresponding to its design as a word-addressable memory system.

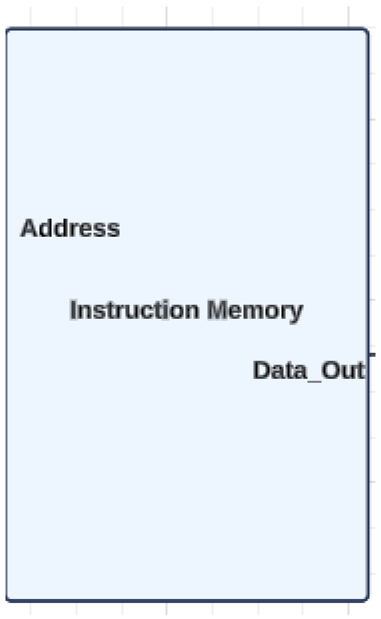


*Figure 1: Instruction Memory.*

## 2.2: Data Memory

In our processor architecture, Data Memory is critical for managing the data used and generated by the system's operations. It supports both reading and writing functions:

- **Reading**: Activating the MemRead signal enables the retrieval of data from a specified address, which is then sent out via the Data_Out line.
- **Writing**: When MemWrite is active, Data Memory accepts input through the Data_In line and stores it at the designated address.

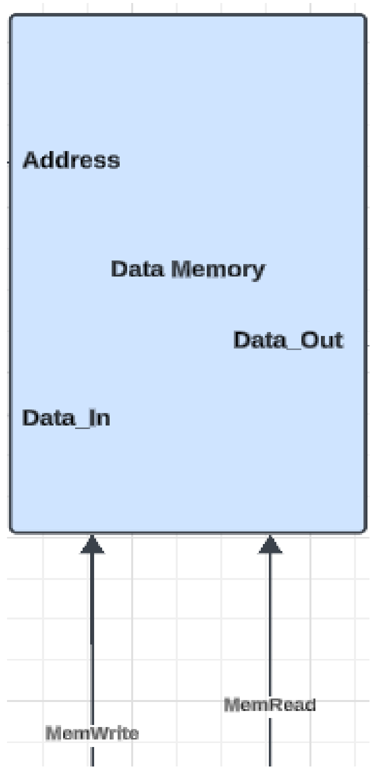


*Figure 2:Data Memory.*

## 2.3: Register File:

The Register File is an essential component of our processor, consisting of multiple registers that facilitate temporary data storage during operations. It supports simultaneous reading from two source registers and writing to a destination register:

- **Source Registers (Rs1 and Rs2)**: The contents of Rs1 are routed to **BusA**, while the contents of Rs2 are routed to **BusB**, enabling parallel data retrieval for processing.
- **Destination Register (Rd)**: Data is written to Rd when the regWrite control signal is enabled, allowing new values to be stored from **Bus_W**.

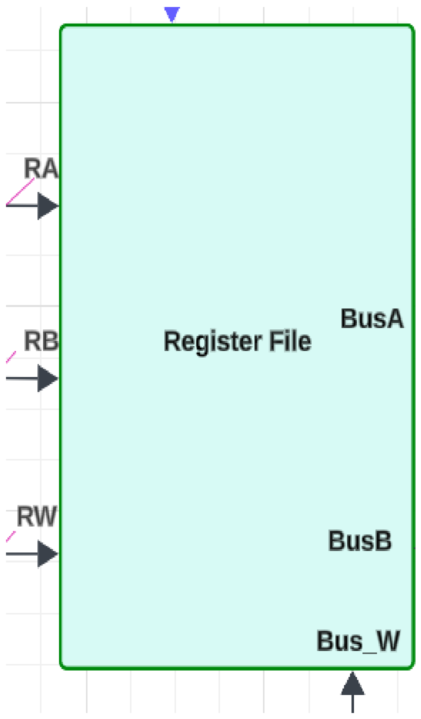


*Figure 3:Register File.*

## 2.4: Multiplexer:

The multiplexer plays a crucial role in a computer's data setup, functioning as a selector that chooses one signal from multiple possible inputs based on a control signal. This component is essential in efficiently managing data flow, making the system more practical and versatile.

In our Datapath design, the multiplexer is employed multiple times to streamline data handling and ensure proper routing of signals, highlighting its importance in building a functional and efficient system architecture.

Like The **PC_MUX** in the datapath is an 8-to-1 multiplexer responsible for selecting one of eight possible inputs to determine the next value of the Program Counter (PC). The selection process is governed by the 3-bit control signal `PC_Src`, composed of selectors `s1`, `s2`, and `s3`. Each input represents a distinct source for the PC value, such as:

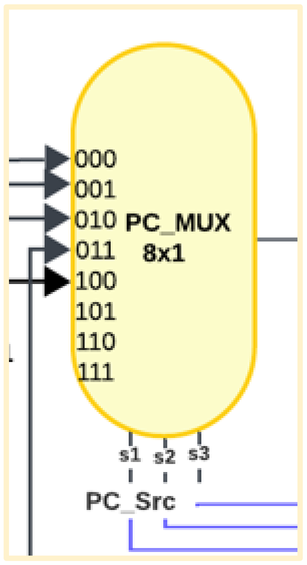


*Figure 4:MUX EXP.*

I. **000**: The incremented PC value (PC+1).
II. **001**: A branch target address.
III. **010**: A jump target address.
VI. **011**: A return address from a subroutine.

## 2.5: Arithmetic logic unit (ALU):

The **Arithmetic Logic Unit (ALU)** is a critical component of the CPU responsible for performing arithmetic and logical operations. It is the computational engine of the processor and enables the execution of instructions by processing data based on control signals

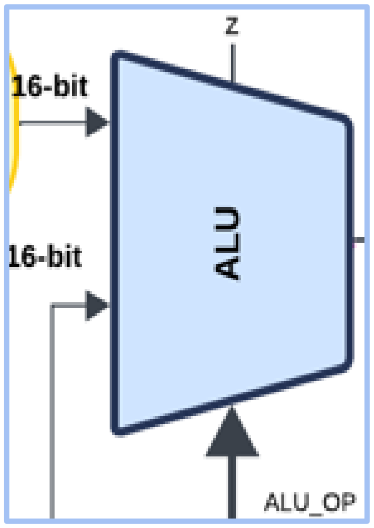


*Figure 5:Arithmetic logic unit (ALU).*

In the datapath, the ALU (Arithmetic Logic Unit) features two primary inputs, **A** and **B**, which are fed through multiplexers (**ALUSrc1** and **ALUSrc2**). These multiplexers allow the selection of input data from various sources, such as registers, immediate values, or other data paths, depending on control signals. The **ALU_OP** signal, provided by the control unit, determines the specific operation the ALU performs, including arithmetic operations (e.g., addition, subtraction) and logical operations (e.g., AND, OR).

The result of the ALU operation is directed to **ALU_OUT** for subsequent processing, such as writing back to a register or serving as an address for memory operations. This design ensures flexibility, enabling the ALU to support diverse instruction types. By efficiently managing computations, memory addressing, and control flow, this configuration allows the datapath to execute a wide range of tasks effectively and adaptively.

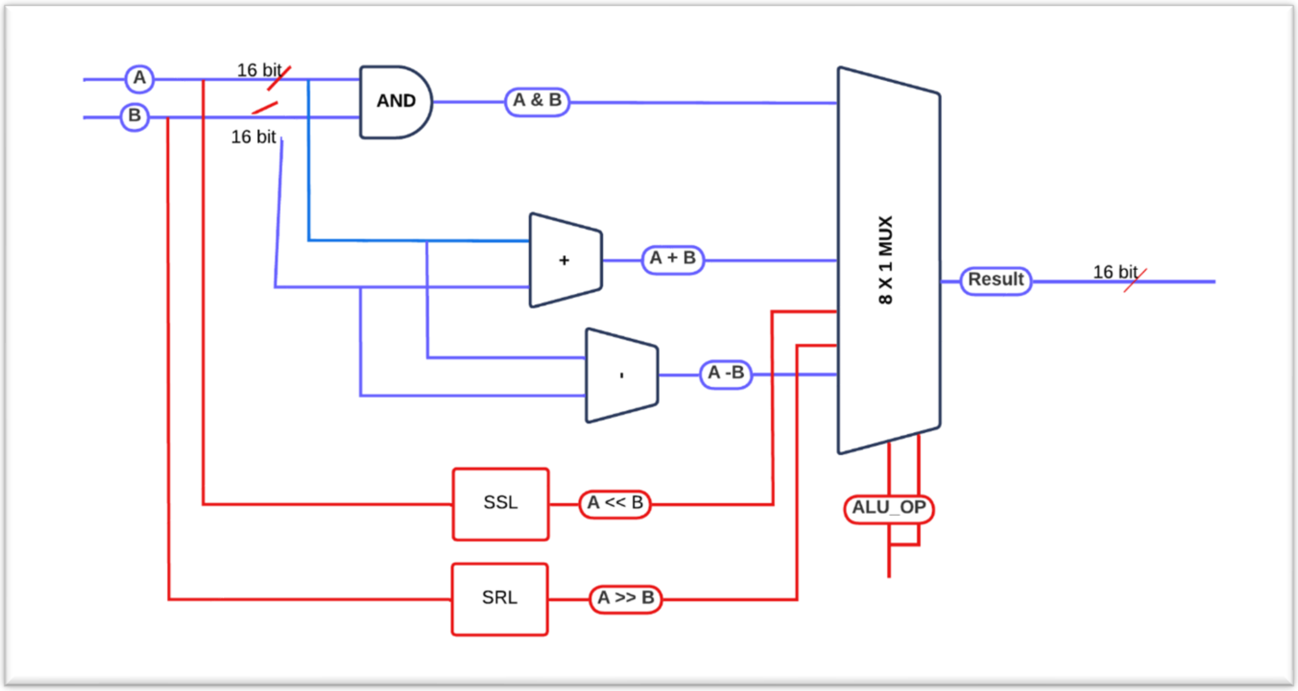


*Figure 6:ALU Internal structure*

ALU supports arithmetic operations (addition, subtraction), logical operations (AND), and bit-level manipulations (shift left, shift right). The **ALU_OP control signal** selects the desired operation, with an 8-to-1 multiplexer routing the result to the output. This design ensures flexibility and efficient execution of various computational tasks.

# 3: Control Path

## 3.1: PC Control Unit.

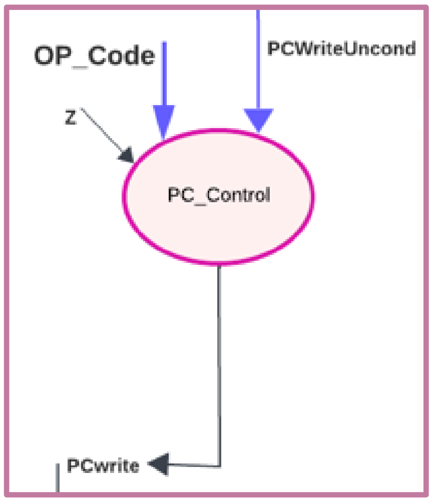


*Figure 7: PC control.*

The **PC Control Unit** updates the Program Counter (PC) to manage instruction flow. Using the **PC MUX**, it selects the next PC value from options like sequential increments (PC + 1), branch/jump addresses, subroutine return addresses, or interrupt handlers. Guided by control signals like PCSrc and branch conditions (e.g., ALU Zero Flag), it ensures smooth execution for sequential instructions, branching, jumping, subroutines, and interrupts, maintaining efficient control flow in the Datapath.

**1-PC_write**: determine if the instruction is allowed to write in the PC register or not. (0: disabled written in PC. / 1: enable writing on PC.)

### 3.1.1: PC Control Truth Table.

*Table 2:PC Control truth table.*

| OP Code | Zero flag | PCSrc |
|---|---|---|
| **BEQ** | **Z==1** | **011** |
| **BNE** | **Z==0** | **011** |
| **FOR** | **Z==0** | **100** |
| **JUMP** | **--------** | **001** |
| **CALL** | **--------** | **001** |
| **RET** | **--------** | **000** |

*Table 3:State Assignments.*

| State name | State number | description |
|---|---|---|
| InstructionFetch | 0 | Fetch retrieves the instruction from memory and updates the **Program Counter (PC)** for the next instruction. |
| InstructionDecode | 1 | The decode stage interprets the instruction, generates control signals, and writes the return address for call instructions. |
| AddressComputation | 2 | Executes branch instructions and updates PC if the branch condition is met. |
| LoadAccess | 3 | Execute store instructions and write back to memory. |
| Load Completion | 4 | Calculates the address for load/store instructions using the ALU. |
| Store Access | 5 | Executes ADDI and ANDI instructions using the ALU. |
| R_TypeALU | 6 | Executes R-type instructions using the ALU. |
| R_Type_ALU_Completion | 7 | Reads The word from memory for the load instructions. |
| logicalALU | 8 | Reads an unsigned byte from |
| logical_ALU_Completion | 9 | Reads a signed byte from memory for load instructions. |
| Branch | 10 | Writes a word to memory for store instructions. |
| ForState | 11 | Write the loaded data back |
| ForCompletion | 12 | Write the ALU result back to the register file. |

## 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 operations. This role allows the processor to perform a wide array of computations and comparisons.

### 3.2.1 ALU control Unit truth table

| Instruction | WB_Data | RegWrite | PCWriteUncond | ALU_Ctr | RsSrc | RtSrc | Reg_Des |
|---|---|---|---|---|---|---|---|
| AND | 0 | 1 | 1 | 000 | 0 | 1 | 1 |
| ADD | 0 | 1 | 1 | 000 | 0 | 1 | 1 |
| SUB | 0 | 1 | 1 | 000 | 0 | 1 | 1 |
| SLL | 0 | 1 | 1 | 000 | 0 | 1 | 1 |
| SRL | 0 | 1 | 1 | 000 | 0 | 1 | 1 |
| ANDI | 0 | 1 | 1 | 010 | 0 | x | 1 |
| ADDI | 0 | 1 | 1 | 010 | 0 | x | 1 |
| LW | 1 | 1 | 1 | 010 | 0 | x | 1 |
| SW | x | 0 | 1 | 010 | 0 | x | x |
| BEQ | x | 0 | 0 | 010 | 1 | 0 | x |
| BNE | x | 0 | 0 | 010 | 1 | 0 | x |
| FOR | 0 | 1 | 1 | 010 | 1 | 0 | 0 |
| JUMP | x | x | 1 | x | x | x | x |
| CALL | x | x | 1 | x | x | x | x |
| RET | x | x | 1 | x | x | x | x |

## 3.3: Main Control.

### 3.3.1: Main Control description.

**1- PCWriteUncond:** determine if the PC(Program counter) will updated unconditionally

0: the PC will be updated without any condition (R-type , Logical I-type , J-type).

1: The PC will be updated just if the condition is correct (Branch:BEQ,BNE).

==========

**2- IRwrite :** determine if the fetched instruction from the instruction memory will be written in the IR(instruction Register ).

0: the fetched instruction will not be written in the IR , the current value of IR is the current instruction

1: the fetched instruction from the instruction memory will be written in the IR(in the instruction fetch state).

==========

**3- RsSrc :** determine which bits from the instruction will be taken as the number of the register for Rs.

0: [11-9 bits ] Rs for R-type.

1: [8-6 bits ] Rs for I-type.

==========

**4- RtSrc :** determine which bits from the instruction will be taken as the number of the register for Rt.

0: [8-6 bits ] Rt for I-type.

1: [5-3 bits ] Rt for R-type.

==========

**5- Reg_Des :** determine which bits from the instruction will be taken as the number of the register for the destination .

0: [8-6 bits ] Rd for I-type.

1: [11-9 bits ] Rd for R-type.

==========

**6- SignedSel** : determine if the extension for the immediate bits is signed or unsigned extension.

0: unsigned extension.

1: signed extension.

==========

**7- ALU_Cntr :** determine the ALU control will decide the ALU operation by the Opcode, function , or just ADD to calculate the next PC .

000: Determine the ALU operation by the function [2:0].

001: the ALU operation will be added to calculate the next PC , and the branch target.

010:  Determine the ALU operation by the opcode [15:12].

==========

**8- RegWrite :** determine whether the instruction will be written in the register file or not .

0: disabled written in the register file.

1: enable written in the register file.

==========

**9- RRWrite :** determine if the next PC will be written in the return register (RR).

0:disabled written in the return register(RR).

1:enable written in the return register(RR).

==========

**10- ALUSrc1:** determine the first operand for the ALU.

00: current PC to calculate the next PC or branch target.

01: the value of  Rs from the register file.

10:the value of  Rt from the register file.

==========

**11- ALUSrc2**

00:the value of  Rt from the register file.

01: extended immediate.

10: constant one .

==========

**12- PC_Src :** determine the source of the next PC .

000: The next PC is the PC stored in the RR(Return register).

001:jump target Pc

010: next Pc ,(PC+1).

011: jump target.

100:   Rs ,for FOR instruction.

==========

**13- MemoWrite :** determine if the instruction is allowed to write in the Memory or not.

0:disable reading from the memory.

1:enable reading from the memory.

==========

**14- MemoRead:**determine if the instruction is allowed to read from the memory or not

0:disable writing from the memory.

1:enable writing from the memory.

==========

**15- WB_Data :** determine the source of the data that will be written back in the register file .

0: The source of the data is ALU .

1: the source of the data is Memory

==========

### 3.3.2: Main Control Truth Table1.

*Table 4:truth table for the main control signal's part one.*

| Inst. | SignedSel | IRwrite | ALUSrc1 | ALUSrc2 | RRWrite | PC_Src | MemWrite | MemRd |
|---|---|---|---|---|---|---|---|---|
| AND | x | 1 | 01 | 0 | 0 | 010 | 0 | 0 |
| ADD | x | 1 | 01 | 0 | 0 | 010 | 0 | 0 |
| SUB | x | 1 | 01 | 0 | 0 | 010 | 0 | 0 |
| SLL | x | 1 | 01 | 0 | 0 | 010 | 0 | 0 |
| SRL | x | 1 | 01 | 0 | 0 | 010 | 0 | 0 |
| ANDI | 0 | 1 | 01 | 01 | 0 | 010 | 0 | 0 |
| ADDI | 0 | 1 | 01 | 01 | 0 | 010 | 0 | 0 |
| LW | 1 | 1 | 01 | 01 | 0 | 010 | 0 | 1 |
| SW | 1 | 1 | 01 | 01 | 0 | 010 | 1 | 0 |
| BEQ | 1 | 1 | 01 | 00 | 0 | 011 | 0 | 0 |
| BNE | 1 | 1 | 01 | 00 | 0 | 011 | 0 | 0 |
| FOR | x | 1 | 11 | 10 | 0 | 100 | 0 | 0 |
| JUMP | x | 1 | x | x | 0 | 001 | 0 | 0 |
| CALL | x | 1 | x | x | 1 | 001 | 0 | 0 |
| RET | x | 1 | x | x | 0 | 000 | 0 | 0 |

### 3.3.3: Main Control Truth Table2.

*Table 5:truth table for the main control signal's part two.*

| Inst. | WB_Data | RegWrite | PCWriteUncond | ALU_Ctr | RsSrc | RtSrc | Reg_Des |
|---|---|---|---|---|---|---|---|
| AND | 0 | 1 | 1 | 000 | 0 | 1 | 1 |
| ADD | 0 | 1 | 1 | 000 | 0 | 1 | 1 |
| SUB | 0 | 1 | 1 | 000 | 0 | 1 | 1 |
| SLL | 0 | 1 | 1 | 000 | 0 | 1 | 1 |
| SRL | 0 | 1 | 1 | 000 | 0 | 1 | 1 |
| ANDI | 0 | 1 | 1 | 010 | 0 | x | 1 |
| ADDI | 0 | 1 | 1 | 010 | 0 | x | 1 |
| LW | 1 | 1 | 1 | 010 | 0 | x | 1 |
| SW | x | 0 | 1 | 010 | 0 | x | x |
| BEQ | x | 0 | 0 | 010 | 1 | 0 | x |
| BNE | x | 0 | 0 | 010 | 1 | 0 | x |
| FOR | 0 | 1 | 1 | 010 | 1 | 0 | 0 |
| JUMP | x | x | 1 | x | x | x | x |
| CALL | x | x | 1 | x | x | x | x |
| RET | x | x | 1 | x | x | x | x |

### 3.3.4 Main control state diagram

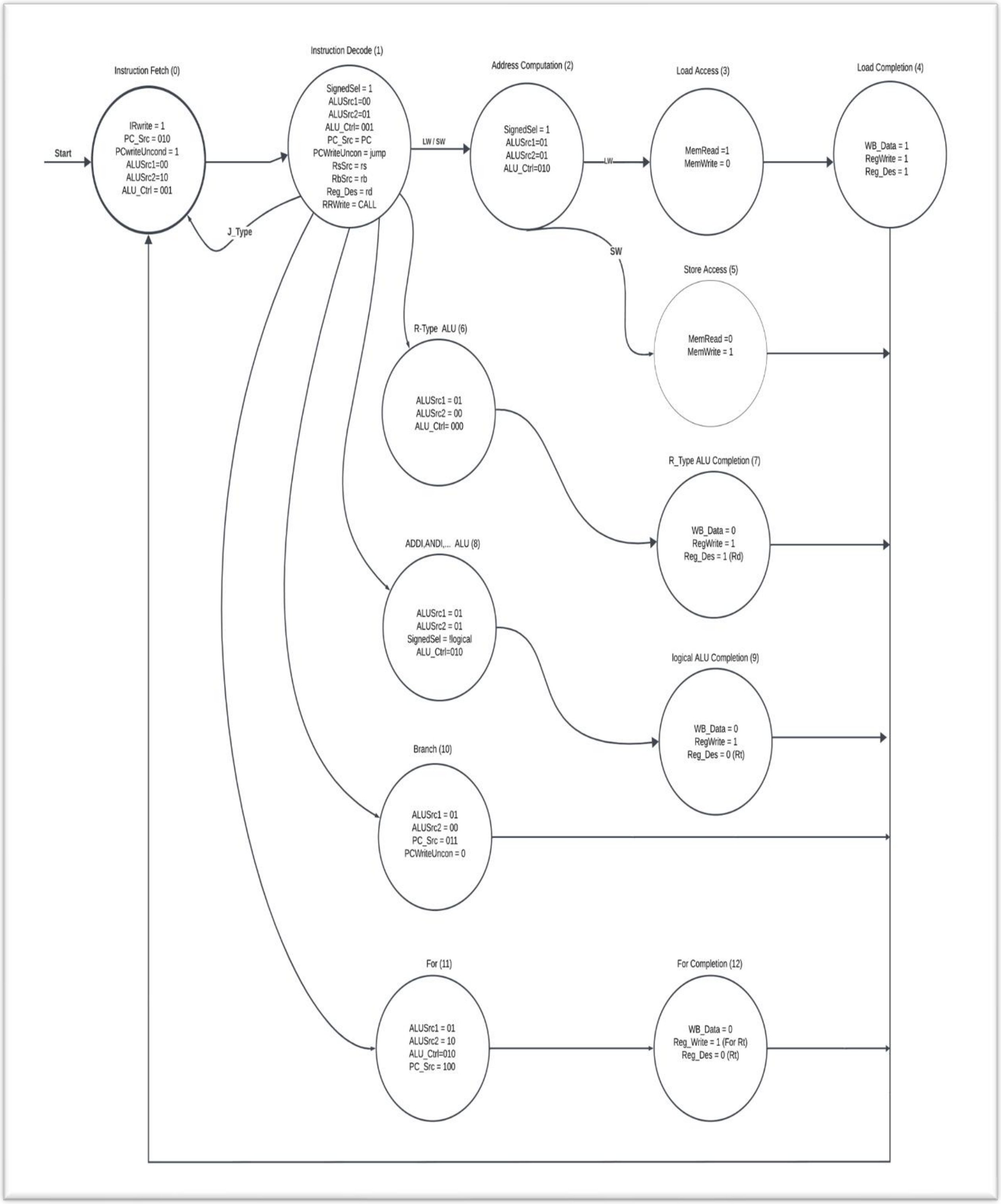


*Figure 8:Main control state diagram*

### 3.3.5 State Assignments:

*Table 6:State assignments table.*

| State Name | State Abbreviation | State Number |
|---|---|---|
| Instruction Fetch | **IF** | **0** |
| Instruction Decode | **ID** | **1** |
| Address Computation | **AC** | **2** |
| Load Access | **LA** | **3** |
| Load Completion | **LC** | **4** |
| Store Access | **SA** | **5** |
| R Type ALU | **R_Type** | **6** |
| R Type ALU Completion | **R_TypeComp** | **7** |
| Logical ALU | **LogicalALU** | **8** |
| Logical Completion | **LogicalComp** | **9** |
| Branch | **Branch** | **10** |
| For | **For** | **11** |
| For Completion | **ForComp** | **12** |

## 3.3.6 Control Unit State Table:

*Table 7:Control unit state table.*

| State | Opcode | SignedSel | IRwrite | Reg_Des | ALUSrc1 | ALUSrc2 | RRWrite | PC_Src | Mem Write | Mem Read | WB_Data | RegWrite | PCWriteUncond | ALU_Ctr | RsSrc | RtSrc |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IF | X | X | 1 | X | 00 | 10 | X | 010 | 0 | 0 | X | 0 | 1 | 001 | X | X |
| ID | JMP | 1 | 0 | X | 00 | 01 | X | 001 | 0 | 1 | X | 0 | 1 | 001 | X | X |
| ID | CALL | 1 | 0 | X | 00 | 01 | 1 | 001 | 0 | 1 | X | 0 | 1 | 001 | X | X |
| ID | RET | 1 | 0 | X | 00 | 01 | X | 000 | 0 | 1 | X | 0 | 1 | 001 | X | X |
| ID | SW | 1 | 0 | X | 00 | 01 | X | xxx | 0 | 1 | X | 0 | 0 | 001 | 0 | X |
| ID | LW | 1 | 0 | X | 00 | 01 | X | xxx | 0 | 1 | X | 0 | 0 | 001 | 0 | X |
| ID | R_Type | 1 | 0 | X | 00 | 01 | X | xxx | 0 | 1 | X | 0 | 0 | 001 | 0 | 1 |
| ID | Logical | 1 | 0 | X | 00 | 01 | X | xxx | 0 | 1 | X | 0 | 0 | 001 | 0 | X |
| ID | Branch | 1 | 0 | X | 00 | 01 | X | xxx | 0 | 1 | X | 0 | 0 | 001 | 1 | 0 |
| ID | For | 1 | 0 | X | 00 | 01 | X | xxx | 0 | 1 | X | 0 | 0 | 001 | 1 | 0 |
| AC | LW | 1 | 0 | X | 01 | 01 | X | xxx | 0 | 0 | X | 0 | 0 | 010 | 0 | X |
| AC | SW | 1 | 0 | X | 01 | 01 | x | xxx | 0 | 0 | X | 0 | 0 | 010 | 0 | X |
| LA | X | X | 0 | X | xx | xx | X | xxx | 0 | 1 | X | 0 | 0 | xxx | X | X |
| LC | X | X | 0 | 1 | xx | xx | X | xxx | 0 | 0 | 1 | 1 | 0 | xxx | X | X |
| SA | x | X | 0 | X | xx | xx | X | xxx | 1 | 0 | X | 0 | 0 | xxx | 0 | X |
| R_Type | X | X | 0 | X | 01 | 00 | X | xxx | 0 | 0 | X | 0 | 0 | 000 | X | X |
| R_TypeComp | X | X | 0 | 1 | X | X | X | xxx | 0 | 0 | 0 | 1 | 0 | xxx | X | X |
| Logical ALU | ADDI | 1 | 0 | X | 01 | 01 | X | xxx | 0 | 0 | X | 0 | 0 | 010 | 0 | X |
| Logical ALU | ANDI | 0 | 0 | X | 01 | 01 | X | xxx | 0 | 0 | X | 0 | 0 | 010 | 0 | X |
| Logical Comp | X | X | 0 | 0 | X | X | X | xxx | 0 | 0 | 0 | 1 | 0 | xxx | X | X |
| Branch | X | X | 0 | X | 01 | 00 | X | 011 | 0 | 0 | X | 0 | 0 | 010 | X | X |
| For | For | X | 0 | X | 01 | 10 | X | 100 | 0 | 0 | X | 0 | 0 | 010 | X | X |
| ForComp | X | X | 0 | 0 | X | X | X | xxx | 0 | 0 | 0 | 1 | 0 | xxx | X | X |

## 3.3.7 Boolean equations for each of the control signals:

- SignedSel = ID + AC + Logical_ALU.ADDI  or SignedSel = (Logical_ALU.ANDI)`
- IRwrite = IF
- Reg_Des = LC + R_TypeComp or Reg_Des = (LogicalComp + ForComp) `
- ALUSrc1$_1$ = 0
- ALUSrc1$_0$ = (IF + ID) `
- ALUSrc2$_1$ = IF + For
- ALUSrc2$_0$ = (IF + R_Type + Branch + For) `
- RRWrite = ID. CALL
- PC_Src$_2$ = For
- PC_Src$_1$ = IF + Branch
- PC_Src$_0$ = ID.JMP + ID. CALL + Branch
- MemWrite = SA
- MemRead = ID + LA
- WB_Data = LC
- RegWrite = LC + R_TypeComp + LogicalComp + ForComp
- PCWriteUncond = IF + ID.JMP + ID. CALL + ID.RET
- ALU_Ctr$_2$ = 0
- ALU_Ctr$_1$ = (IF + ID + R_Type) `
- ALU_Ctr$_0$ = IF + ID
- RsSrc = Branch + For
- RtSrc = R_Type

## 4: Data-Path.

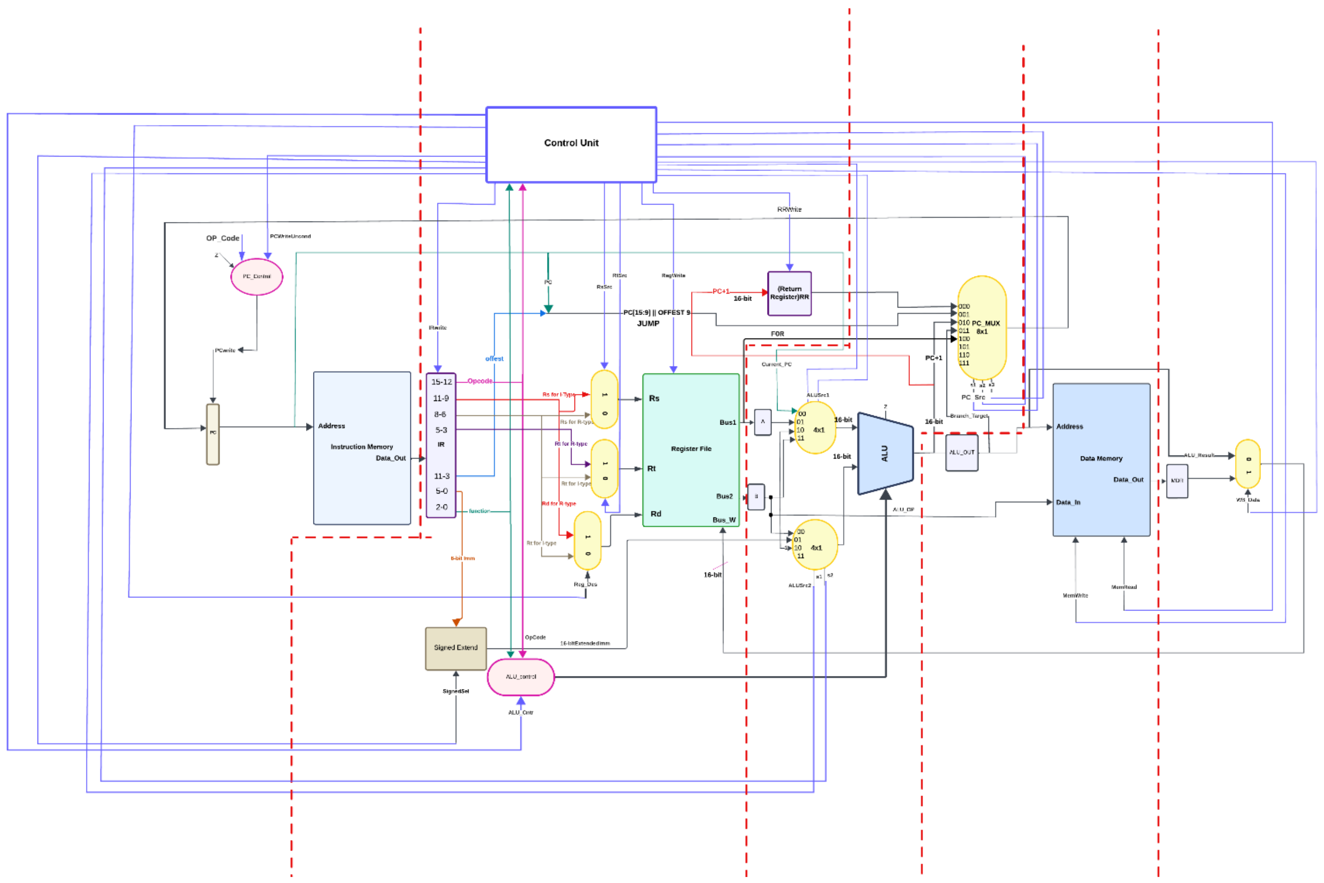


*Figure 9:Full Data Path.*

# 5- Testing.

## 5-1 Computer Test bench.

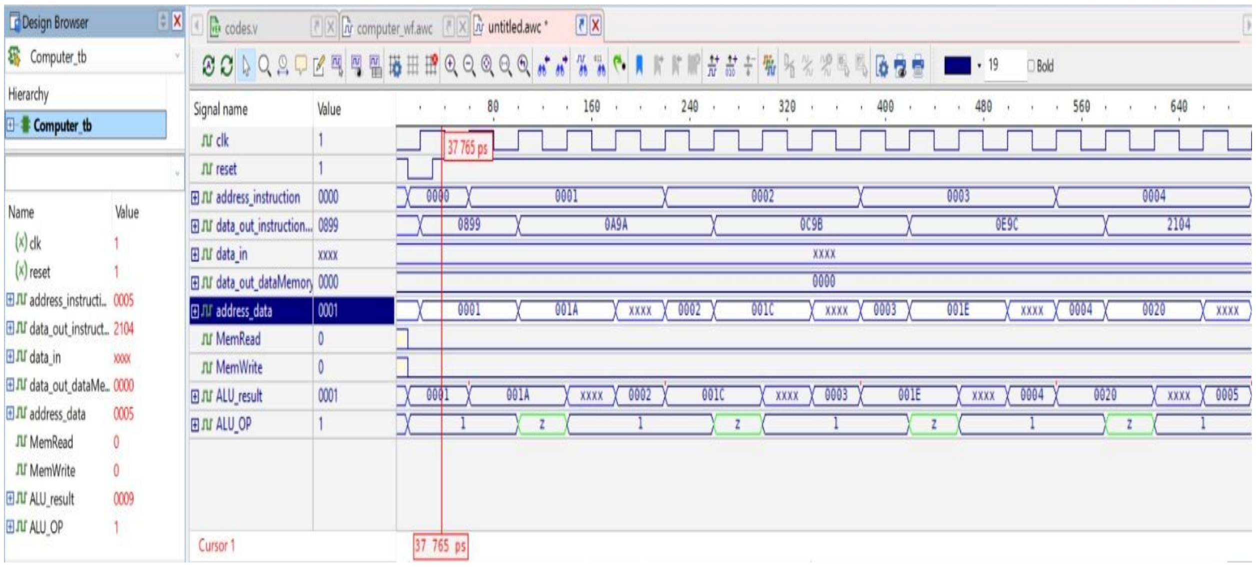
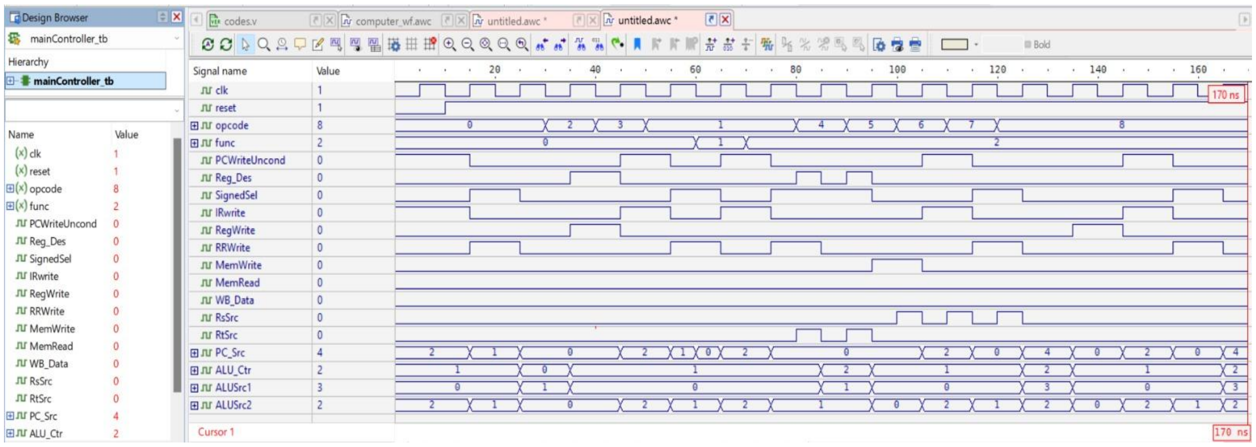
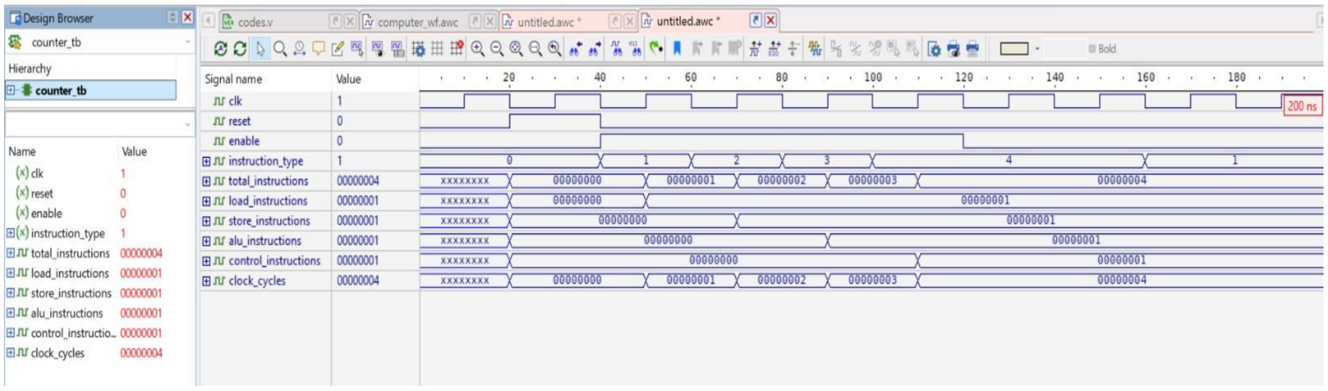


*Figure 10:Test bench.*

## 5-2 Main Controller Test Bench.



## 5-3 Counter Test bench.



| Name | Value |
|---|---|
| clk | 1 |
| reset | 0 |
| enable | 0 |
| instruction_type | 1 |
| total_instructions | 00000004 |
| load_instructions | 00000001 |
| store_instructions | 00000001 |
| alu_instructions | 00000001 |
| control_instructio... | 00000001 |
| clock_cycles | 00000004 |

| Signal name | Value |
|---|---|
| clk | 1 |
| reset | 0 |
| enable | 0 |
| instruction_type | 1 |
| total_instructions | 00000004 |
| load_instructions | 00000001 |
| store_instructions | 00000001 |
| alu_instructions | 00000001 |
| control_instructions | 00000001 |
| clock_cycles | 00000004 |

```
instruction_type      0         1    2    3    4                1
total_instructions    XXXXXXXX  00000000  00000001  00000002  00000003  00000004
load_instructions     XXXXXXXX  00000000  00000001
store_instructions    XXXXXXXX  00000000  00000001
alu_instructions      XXXXXXXX  00000000  00000001
control_instructions  XXXXXXXX  00000000  00000001
clock_cycles          XXXXXXXX  00000000  00000001  00000002  00000003  00000004
```

*Figure 11:Counter Test bench.*