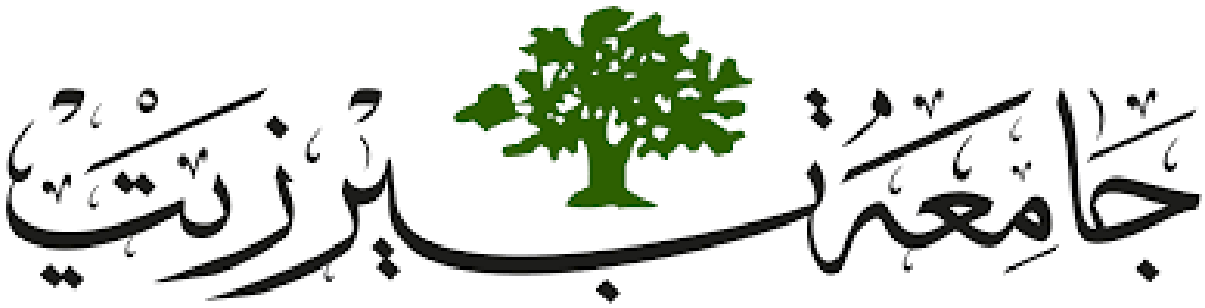



BIRZEIT UNIVERSITY

**Faculty of Engineering & Technology**
**Department of Electrical & Computer Engineering**
**ENCS4370-COMPUTER ARCHITECTURE**
**Project #2 Multicycle Processor Implementation**
**Report**

---

**Prepared by:**

| | |
|---|---|
| Saja Asfour | 1210737 |
| Yara Khattab | 1210520 |
| Masa Jalamneh | 1212145 |

---

**Instructor:**
Dr. Aziz Qaroush
**Section:**
Sec1

---

**Date:**
15/6/2024

## Abstract:

This project focuses on the design and verification of a special multi-cycle computer processor implemented in Verilog. The processor is configured to handle 16-bit instructions and features 8 general-purpose registers, a program counter, and separate memories for instructions and data. It supports four instruction type R-type, I-type, J-type, and S-type and employs little-endian byte ordering. The primary goal is to achieve a processor capable of performing various operations, from basic arithmetic to conditional branching.

The report details the design process, including the creation of the Datapath and control signals, and outlines the assembly of these components. It also covers the verification phase, which involves writing test programs in the given ISA and demonstrating the processor's execution of these programs through simulation.

# Contents

## Table of Figures:

## Table of tables:

## 1. Design Specifications:

Our datapath operates in a multi-cycle manner. To construct it, we start with a single-cycle datapath and then introduce register buffers between stages. These buffers store the output from one stage, allowing the next stage to read and utilize this output. This way, data is preserved and passed along sequentially through each stage, preventing data loss when a new clock cycle begins. It's important to note that reading from and writing to these buffers takes time, so the duration of each stage in a multi-cycle processor isn't exactly one-fifth of a single-cycle processor's cycle time. However, the multi-cycle approach is generally more efficient than the single-cycle method. While single-cycle execution may perform better for specific instructions like load operations, evaluating overall performance across various scenarios demonstrates the advantages of the multi-cycle design.

### 1.1 The key features of this instruction set include:

1. each instruction in the set contains of 16 bit in length .
2. the design include 8 general-purpose register R0-R7 each of them is 16 bit in length which R0 is hardwired to zero so any attempt to write it will be discarded.
3. program counter(PC)  which is special purpose register have 16-bit .
4. the design have four instruction type ( R-type ,I-type , J-type , and S-type).
5. the instruction memory and data memory are separated.
6. the memory is byte addressable.
7. the byte ordering  is little endian.
8. the required signals from ALU is generated to calculate the condition branch outcome (taken/ not taken).

### 1.2 Instruction Formats :

#### *1.2.1 R-Type (Register-Type):*

| Opcode (4 bit) | Rd (3 bit) | Rs1 (3 bit) | Rs2 (3 bit) | Unused (3 bit) |
|---|---|---|---|---|

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.

### *1.2.2 I-Type (Immediate Type):*

| Opcode (4 bit) | m (1 bit) | Rd (3 bit) | Rs1 (3 bit) | Immediate (5bit) |
|---|---|---|---|---|

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.
Immediate: Represents a numerical value, unsigned for logic and signed for other instructions.
Mode bits: Used in load/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

### *1.2.3 J-Type (Jump Type):*

J-type instructions come in three formats: Jump (jmp) , Call, and Return (ret) . Both Jump and Call instructions follow the format bellow.

| Opcode (4 bit) | Jump Offset (12 bit) |
|---|---|

The target address is calculated by multiplying the 12-bit offset by 2 then concatenating the most 3-bit of the current Pc . (this because when we multiply by 2 the bit increased by 1 so it be 13 bit and the address is 16 bit so we concatenating the remaining bit from the most significant of the current Pc which is 3 bit).

whereas in the Return instruction (ret) , the offset section is not used like the format below. This is because the Return instruction retrieves the next Pc and places it into R7.

| Opcode (4 bit) | Unused (12 bit) |
|---|---|

### *1.2.4 S-Type (Store):*

This format support only one instruction (Sv instruction)

| Opcode (4 bit) | Rs (3 bit) | Immediate (8 bit) |
|---|---|---|

Sv rs, imm # M[rs] = imm

### 1.3 Instruction Set:

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

| NO. | Instruction | Format | Meaning | Opcode Value | M |
|---|---|---|---|---|---|
| 1. | AND | R-Type | Reg(Rd) = Reg(Rs1) & Reg(Rs2) | 0000 | |
| 2. | ADD | R-Type | Reg(Rd) = Reg(Rs1) + Reg(Rs2) | 0001 | |
| 3. | SUB | R-Type | Reg(Rd) = Reg(Rs1) - Reg(Rs2) | 0010 | |
| 4. | ADDI | I-Type | Reg(Rd) = Reg(Rs1) + Imm | 0011 | |
| 5. | ANDI | I-Type | Reg(Rd) = Reg(Rs1) & Imm | 0100 | |
| 6. | LW | I-Type | Reg(Rd) = Mem(Reg(Rs1) + Imm) | 0101 | |
| 7. | LBu | I-Type | Reg(Rd) = Mem(Reg(Rs1) + Imm) | 0110 | 0 |
| 8. | LBs | I-Type | Reg(Rd) = Mem(Reg(Rs1) + Imm) | 0110 | 1 |
| 9. | SW | I-Type | Mem(Reg(Rs1) + Imm) = Reg(Rd) | 0111 | |
| 10. | BGT | I-Type | if (Reg(Rd) > Reg(Rs1))<br>Next PC = PC + sign_extended (Imm)<br>else PC = PC + 2 | 1000 | 0 |
| 11. | BGTZ | I-Type | if (Reg(Rd) > Reg(0))<br>Next PC = PC + sign_extended (Imm)<br>else PC = PC + 2 | 1000 | 1 |
| 12. | BLT | I-Type | if (Reg(Rd) < Reg(Rs1))<br>Next PC = PC + sign_extended (Imm)<br>else PC = PC + 2 | 1001 | 0 |
| 13. | BLTZ | I-Type | if (Reg(Rd) < Reg(0))<br>Next PC = PC + sign_extended (Imm)<br>else PC = PC + 2 | 1001 | 1 |
| 14. | BEQ | I-Type | if (Reg(Rd) = = Reg(Rs1))<br>Next PC = PC + sign_extended (Imm)<br>else PC = PC + 2 | 1010 | 0 |
| 15. | BEQZ | I-Type | if (Reg(Rd) = =Reg(0))<br>Next PC = PC + sign_extended (Imm)<br>else PC = PC + 2 | 1010 | 1 |
| 16. | BNE | I-Type | if (Reg(Rd) != Reg(Rs1))<br>Next PC = PC + sign_extended (Imm)<br>else PC = PC + 2 | 1011 | 0 |
| 17. | BNEZ | I-Type | if (Reg(Rd) != Reg(0))<br>Next PC = PC + sign_extended (Imm)<br>else PC = PC + 2 | 1011 | 1 |
| 18. | JMP | J-Type | Next PC = {PC[15:10], Immediate} | 1100 | |
| 19. | CALL | J-Type | Next PC = {PC[15:10], Immediate}<br>Pc+2 is saved on r7 | 1101 | |
| 20. | RET | J-Type | Next PC=r7 | 1110 | |
| 21 | Sv | S-Type | M[rs]= imm | 1111 | |

## 2. Component of DataPath:

Based on the analysis of the instruction set, we identified the essential components needed to complete the design:

### 2.1 Multiplexer:

The multiplexer plays a crucial role in a computer's data configuration. It functions as a selector, choosing one signal from four potential inputs based on a control signal. This component is essential in the data setup for directing data flow, enhancing the system's practicality and efficiency.

In our datapath design, the multiplexer is employed in several instances which is:

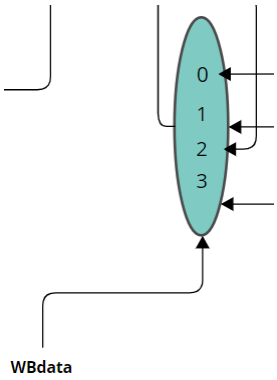



**Figure 1:Mux used in Write back data in Register File**

This mux is 4x1 and the selection line (WBdata) have 2 bit.

- When WBdata assign to 00 then we write bake to the register file the result from LB.
- When WBdata assign to 01 then we write bake to the register file the result from LW.
- When WBdata assign to 10 then we write bake to the register file the value of Pc+2 (next address).
- When WBdata assign to 00 then we write bake to the register file the ALU result.

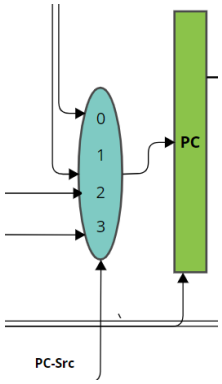



**Figure 2:mux that used for PC**

This mux is 4x1 and the selection line which is PC-Src is have 2 bit

- When Pc-Src is set to 00 , then the next Pc is Pc+2.
- When Pc-Src is set to 01 , then the next Pc is the target address for branch.
- When Pc-Src is set to 10 , then the next Pc is the target address for Jmp.
- When Pc-Src is set to 11 , then the next Pc is the target address for Ret.

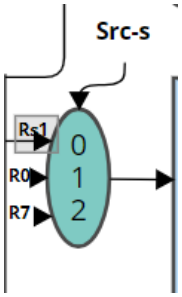



Figure 3: mux that use to select what source enter in Rs in RF

This mux is 4x1 and selection (Src-s) have 2 bit:

- When Src-s is set to 00, then RA=Rs1.
- When Src-s is set to 01, then RA=R0 ( this is for branch condition instruction(with z)).
- When Src-s is set to 10, then RA=R7 ( this is for Ret instruction).
- When Src-s is set to 11 this case unesed.

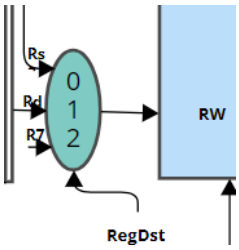



Figure 4: mux that use to select the destination register in RF

This mux is 4x1 with 2 bit selection line (RegDst) :

- When RegDst is set to 00, then the destination register is Rs (RW=Rs).
- When RegDst is set to 01, then the destination register is Rd (RW=Rd).
- When RegDst is set to 10, then the destination register is R7( this for call instruction)(RW=R7).
- When RegDst is set to 11 this case unused.

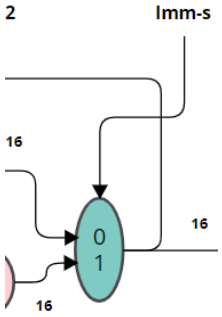



Figure 5: mux that use to select Extened immediate that enter ALU

This mux is 2x1 with 1 bit selection line ( Imm-s):

- When Imm-s is set to 0 , then the immediate that enter the ALU is the immediate that extend from 8bit to 16 bit ( in S-Type)
- When Imm-s is set to 1 , then the immediate that enter the ALU is the immediate that extend from 5bit to 16 bit ( in I-Type)

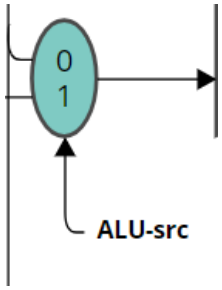



Figure 6:mux to select the input of ALU

This mux is 2x1 with 1 bit selection line (ALU-src):

- When ALU-src is set to 0, then the ALU input is the immediate extened(for I-Type).
- When ALU-src is set to 1, then the ALU input is the output from register file (BusB) (for R-Type).

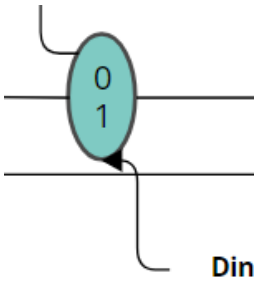



Figure 7:mux to select the data-in in data memory

This mux is 2x1 with 1 bit selection line (Din):

- When Din set to 0 , then the Data-in in data memory is the extended immediate ( for Sv instruction).
- When Din set to 1 , then the Data-in in data memory is the output of RF BusB ( for SW instruction).

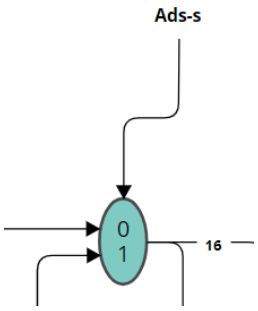



Figure 8:mux to select the address of data memory

This mux is 2x1 with 1 bit selection line(Ads-s):

- When Ads-s set to 0 , then the address of data memory is output of RF BusA (this in Sv instruction)
- When Ads-s set to 1 , then the address of data memory is the ALU-Result (this in Store and load instruction which is Lw ,LBu, LBs, and Sw).

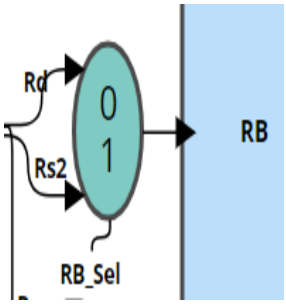



Figure 9:mux to select the second source

This mux is 2x1 with 1 bit selection line(RB_Sel):

- When RB_Sel set to 0 , then the second operand is Rd (RB=Rd) (this in Sw and Branch's instruction)
- When RB_Sel set to 1 , then the second operand is Rs2 (RB=Rs2) (this in R-Type instruction)

## 2.2 Instruction memory:

In our processor implementation, memory is divided into two separate sections: Instruction Memory and Data Memory. This separation is intended to avoid conflicts that could occur from simultaneous operations, such as fetching instructions while loading or storing data. The instruction memory stores the instructions and only needs read capability, as the datapath does not write instructions. It functions with straightforward read logic.

Because the memory is byte-addressable so it stored instruction which is 16 bit by dived it to be each 8bit together , so when we want to read any instruction we should read from two address one after the other so we design the instruction memory to read from 2 address (address from Pc and address +1) , and this cause of two instruction each of them 8 bit ( we combine it later before using it ).

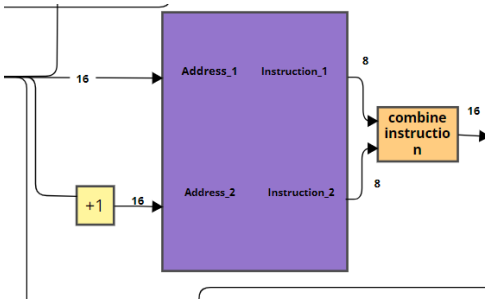



Figure 10:Instruction memory

In Figure9 above:

- Address_1 → the 16bit comes from PC.
- Address_2 → the 16bit comes from PC+1 (which is the next address).
- Instruction_1 → the 8 bit instruction that read from memory which is the instruction[0:7] that comes from reading Address_1.
- Instruction_2 → the 8 bit instruction that read from memory which is the instruction[8:15] that comes from reading Address_2.

Then we combine this two instruction into one instruction with 16 bit to use it later in datapath.

## 2.3 Data Memory:

The data memory serves as a repository where the processor stores and retrieves data. Like the instruction memory the data memory is also byte-addressable so it store 8 bit instruction inside it so we also have 2 address for it one of them comes from the extended immediate and the address after it . also we dived the data that want to enter data memory into two data-in each of them is 8 bit . additionally , we have two signal for data memory which is MemRd (be 1 if we want to read , other 0) and MemWr ( be 1 if we want to write , other 0). Finally, the output of this component is Data-out1 and Data-out2 which is used when we read instruction that is in fact is 16 bit but stored in memory as 8 bit , so each of them is 8 bit and we combine the two data-out to be 16 bit .

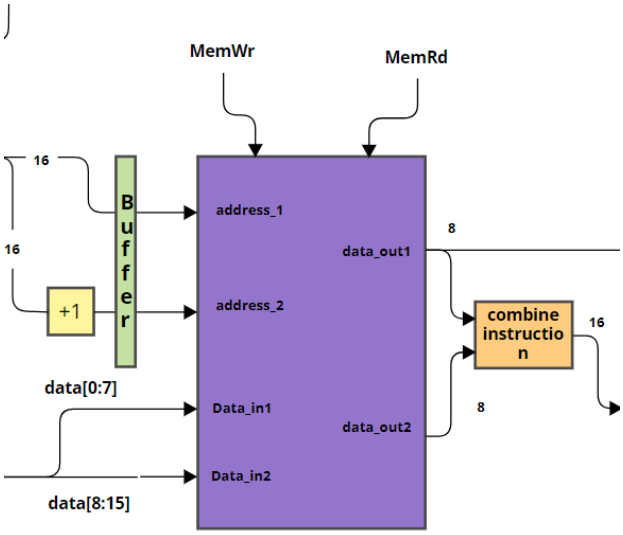



Figure11 :Data memory

In figure 10 above:

- MemWr → 1 if we write to the memory , 0 otherwise.
- MemRd → 1 if we read from memory , 0 otherwise.
- address_1 → 16 bit from Extended immediate , and it's the address that Data_in1 write on it.
- address_2 → 16 bit from Extended immediate+1 , and it's the address that Data_in2 write on it.
- Data_in1 → first part from data that we want to write on address_1 in data memory which is data[0:7].
- Data_in2 → second part from data that we want to write on address_2 in data memory which is data[8:15].
- data_out1 → the first part of instruction that we read from address_1 .
- data_out2 → the second part of instruction that we read from address_2 .

then we combine data_out1 and data_out2 into one 16 bit instruction.

## 2.4 Register File:

The register file, a vital component, houses multiple registers used for temporarily storing data moved between the processor's memory and operational units. To access these registers, the processor employs two specific ones known as RA and RB. RA supplies data to BusA, while RB sends data to BusB. Rd, another register in the file, serves as the designated location for data writing. When the control signal RegWr is set to 1, it signifies that incoming data on BusW will be stored into Rd.

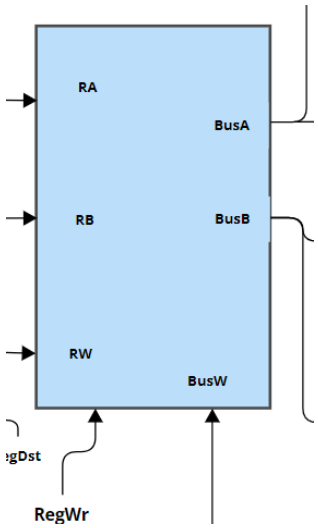



Figure 12:Register File

## 2.5 Arithmetic and Logic unit (ALU):

The Arithmetic Logic Unit (ALU) within the CPU is responsible for executing arithmetic and logical operations, such as addition, subtraction, and comparisons.

In our Datapath design, the ALU features two primary inputs, each receiving 16 bits of data. These inputs undergo various operations within the ALU: they can be combined using a logical operation (like AND), added together, or subtracted in two distinct manners. A small 2-bit control signal (ALU-Op) determines which specific arithmetic or logical operation is selected. The resulting computation is then outputted from the ALU as the final result.

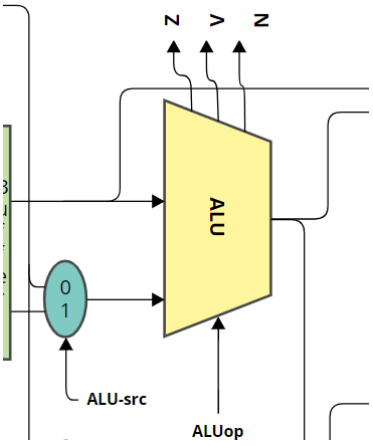



Figure 13:ALU

Also we use a small adder in our Datapath which we used it to find the target address for branch case.

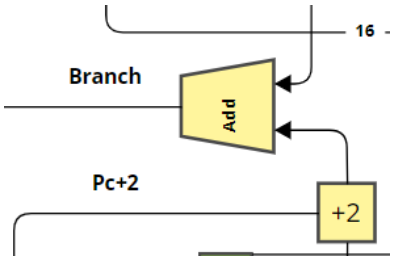



Figure 14:Adder

## 2.6 Extender:

The main purpose of an extender is to ensure consistency in data format across different parts of the datapath, particularly when dealing with operations that require operands of equal length. For example, in many processor architectures, arithmetic operations like addition or subtraction often require that both operands (inputs) have the same number of bits.
We use a lot of Extender in datapath which is :

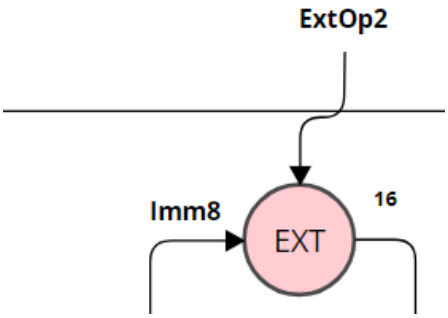



Figure 15:Extender immediate from 8 bit to 16 bit

The extender in Figure 14 is used to make a signed or unsigned extend according to ExtOp2 (0 for unsigned , 1 for signed ) for immediate from 8 bit to 16 bit and this case happen if we have S-Type.

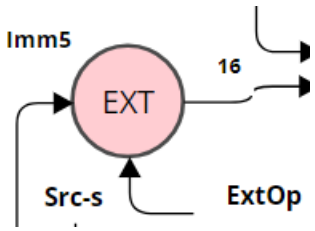



Figure 16:Extender immediate from 5 bit to 16 bit

The extender in Figure 15 is used to make a signed or unsigned extend according to ExtOp (0 for unsigned , 1 for signed ) for immediate from 5 bit to 16 bit and this case happen if we have I-Type.

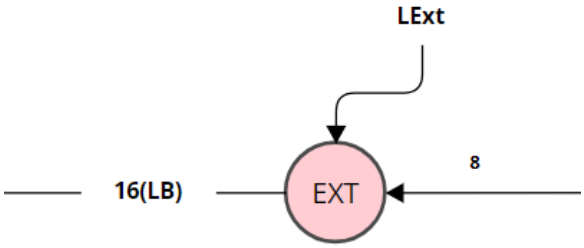



Figure 17:Extender the Byte we load it from memory to be 16 bit

The extender in Figure 16 is used to make a signed or unsigned extend according to LExt (0 for LBs to zero extension, 1 for LBu to signed extension) for the byte we load from memory to be 16 bit.

## 2.7 Buffers:

A buffer is a fundamental component used for temporarily storing data as it moves between different stages or components of the processor. And we use it to have a multi-cycle design. buffers help synchronize the transfer of data between components that may operate at different speeds or on different clock cycles. For example, data fetched from memory may need to be temporarily stored in a buffer before being processed by the ALU (Arithmetic Logic Unit).

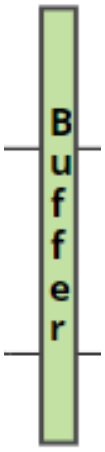



Figure 18:Buffer

when an instruction is fetched from memory, it stored temporarily in a buffer before being decoded and executed. Similarly, after execution, the results of the operation stored in another buffer before being written back to memory or registers. These buffers ensure that each stage of the datapath operates independently and can process data at its own pace without causing delays or conflicts which make contribute to the overall performance and reliability of the datapath design.

# 3. Control Path:

## 3.1 PC control:

Its control the address that loading into the Pc register, its generated during the fetch stage, it has four input the 4-bit opcode, ALU zero,overflow and negative flags, and 2 bit output PC-Src (PC-Src0 and Pc-Src1)which is the selection lines for the mux that decide the address that will stores in the PC register.

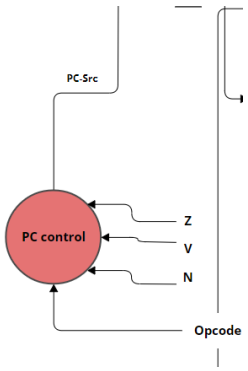



Figure 19:PC control

### *3.1.1 PC control truth table:*

Table 1:PC control truth table

| opcode | Z | N | V | Pc- src |
|---|---|---|---|---|
| R-type | X | X | X | 00=incr pc |
| JMP | X | X | X | 10=jump target Ads |
| RET | X | X | X | 11=next pc |
| BGT(taken) | 0 | 0 | 0 | 01=branch target Ads |
| BGT(taken) | 0 | 1 | 1 | 01=branch target Ads |
| BGT(not taken) | 0 | 0 | 1 | 00=incr pc |
| BGT(not taken) | 0 | 1 | 0 | 00=incr pc |
| BGTZ(taken) | 0 | 0 | 0 | 01=branch target Ads |
| BGTZ(taken) | 0 | 1 | 1 | 01=branch target Ads |
| BGTZ(not taken) | 0 | 0 | 1 | 00=incr pc |
| BGTZ(not taken) | 0 | 1 | 0 | 00=incr pc |
| BLT(taken) | X | 0 | 1 | 01=branch target Ads |
| BLT(taken) | X | 1 | 0 | 01=branch target Ads |
| BLT(not taken) | X | 0 | 0 | 00=incr pc |
| BLT(not taken) | X | 1 | 1 | 00=incr pc |
| BLTZ(taken) | X | 0 | 1 | 01=branch target Ads |
| BLTZ(taken) | X | 1 | 0 | 01=branch target Ads |
| BLTZ(not taken) | X | 0 | 0 | 00=incr pc |
| BLTZ(not taken) | X | 1 | 1 | 00=incr pc |
| BEQ(taken) | 1 | X | X | 01=branch target Ads |
| BEQ(not taken) | 0 | X | X | 00=incr pc |
| BEQZ(taken) | 1 | X | X | 01=branch target Ads |
| BEQZ(not taken) | 0 | X | X | 00=incr pc |
| BNE(taken) | 0 | X | X | 01=branch target Ads |
| BNE(not taken) | 1 | X | X | 00=incr pc |
| BNEZ(taken) | 0 | X | X | 01=branch target Ads |
| BNEZ(not taken) | 1 | X | X | 00=incr pc |
| Others | X | X | X | 00=incr pc |

### *3.1.2 PC control Boolean equation:*

if(opcode= =jmp) Pc- src=10
else if( opcode= =RET) Pc- src=11
else if (opcode= = BGT &&V= =N || opcode= = BGTZ && V= =N || opcode== BLT && V!=N || opcode== BLTZ && V!=N || opcode== BEQ && Z= =1 || opcode== BEQZ && Z= =1 || opcode== BNE && Z= =0 || opcode== BNEZ && Z= =0) Pc- src=01
else pc-src =00

Pc- src1=JMP+RET
Pc- src0= RET+ BGT(taken)+ BGTZ(taken)+ BLT(taken)+ BLTZ(taken)+ BEQ(taken)+ BEQZ(taken)+ BNE(taken)+ BNEZ(taken)

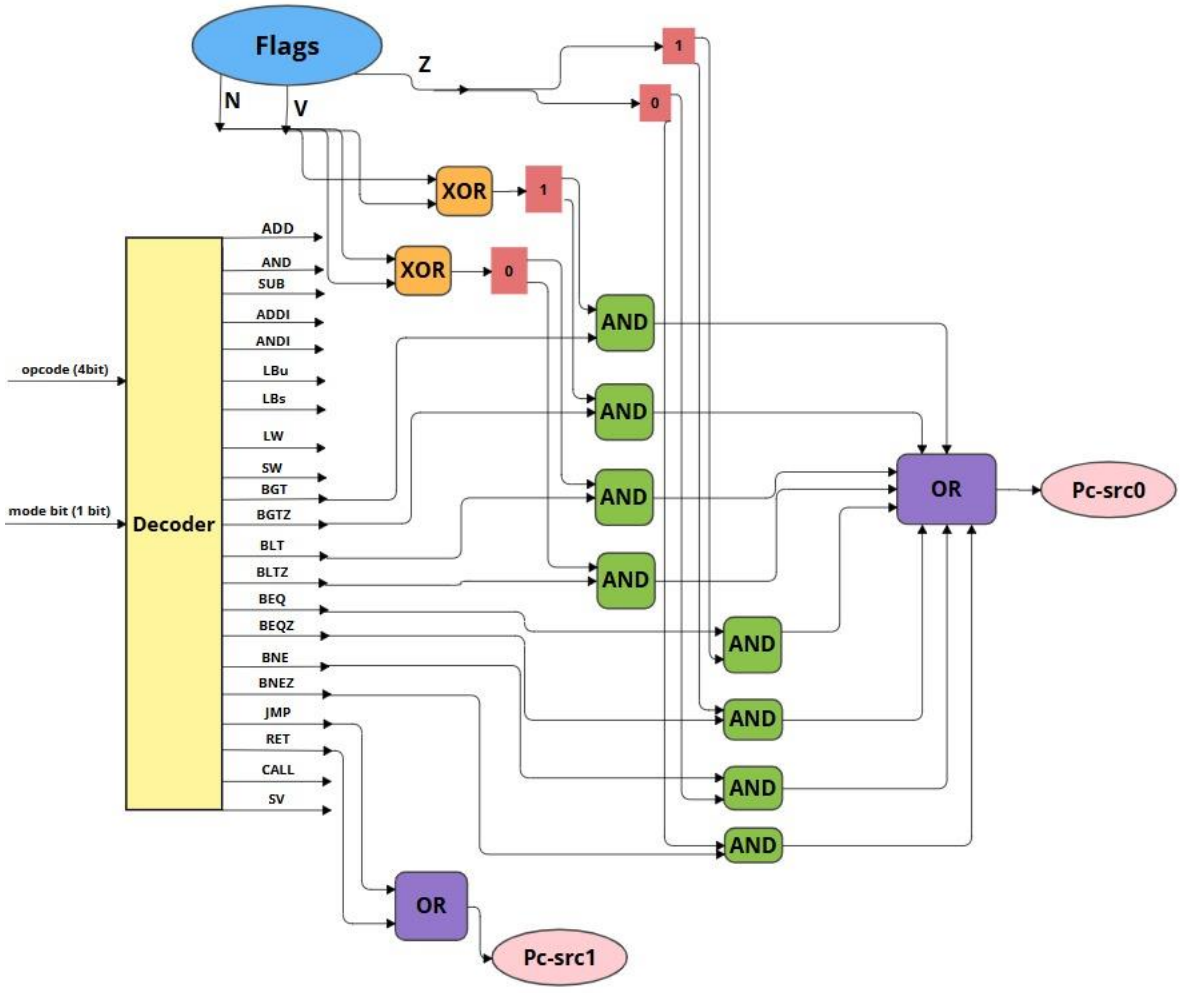



**Figure 20:Internal structure of the PC control unit**

In figure 19 we use Decoder 5x32

### 3.2 Main Control:

It is responsible for managing overall operations by generating signals based on a 4-bit opcode input.

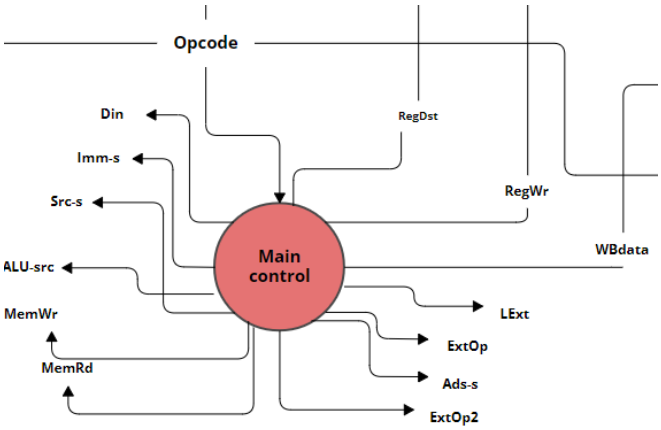



**Figure 21:Main control**

It produces multiple output signals:

- Din → Decide which data be the Data_in in data memory.
- Imm-S → Decide which extend immediate enter ALU.
- Src-s →Decide which register is the source register (RA) in register file.
- ALU-src → to select the second input for ALU.
- MemWr → to decide we want to write to data memory or not.
- MemRd → to decide we want to read from data memory or not.
- Extop2 → to select from signed or unsigned extension for imm8 in S-Type.
- ExtOp → to select from signed or unsigned extension for imm5 in I-Type.
- Ads-s → to select the address that we want to read or write data in data memory.
- LExt → to select from signed or unsigned extension for byte instruction that we load.
- WBdata →to select the data that we want to write on register file.
- RegWr → to decide that we want to write to register file or not.
- RegDst → to choose the destination register (RW) in the register file.

And the details of it are described in the diagrams below ( and we attached it separate to be more clear):

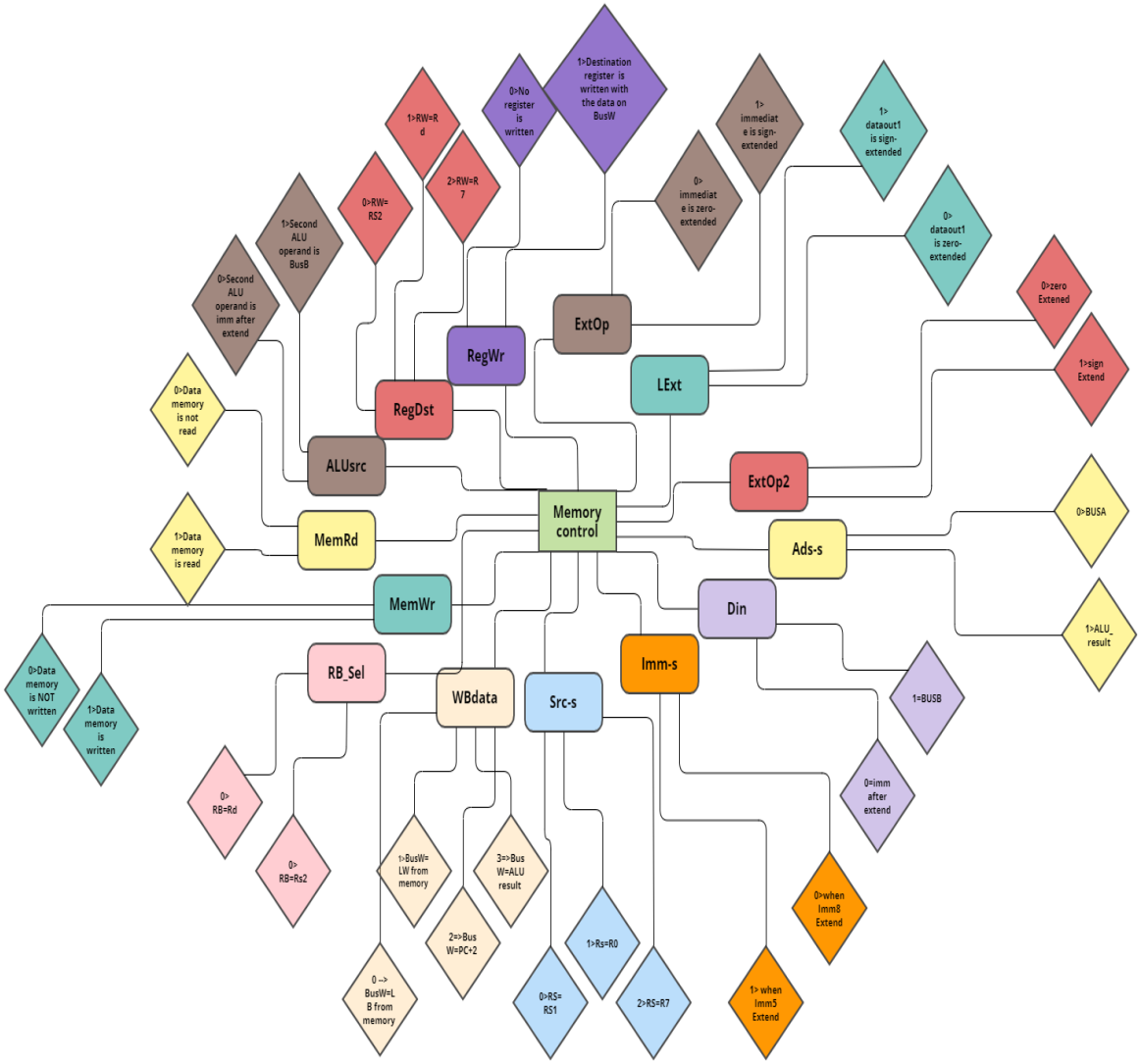



**Figure 22:Main control signal description**

This unit plays a critical role in coordinating and directing the flow of operations within the processor, ensuring that each instruction is executed correctly based on its opcode and the associated signals generated.

### 3.2.1 Main control truth table:

**Table2 :Main control truth table**

| Instr | Type | op | M | Reg Dst | RegWr | ExtOp | ExtOp2 | Imm-s | ALUSrc | MemRd | MemWr | WBdata | Src-S | Ads-S | LExt | Din | RB_ Sel |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AND | R-Type | 0000 | | 01 | 1 | X | X | X | 1 | 0 | 0 | 11 | 00 | X | X | X | 1 |
| ADD | R-Type | 0001 | | 01 | 1 | X | X | X | 1 | 0 | 0 | 11 | 00 | X | X | X | 1 |
| SUB | R-Type | 0010 | | 01 | 1 | X | X | X | 1 | 0 | 0 | 11 | 00 | X | X | X | 1 |
| ADDI | I-Type | 0011 | | 00 | 1 | 1 | X | 1 | 0 | 0 | 0 | 11 | 00 | X | X | X | X |
| ANDI | I-Type | 0100 | | 00 | 1 | 0 | X | 1 | 0 | 0 | 0 | 11 | 00 | X | X | X | X |
| LW | I-Type | 0101 | | 00 | 1 | 1 | X | 1 | 0 | 1 | 0 | 01 | 00 | 1 | X | X | X |
| LBu | I-Type | 0110 | 0 | 00 | 1 | 1 | X | 1 | 0 | 1 | 0 | 00 | 00 | 1 | 0 | X | X |
| LBs | I-Type | 0110 | 1 | 00 | 1 | 1 | X | 1 | 0 | 1 | 0 | 00 | 00 | 1 | 1 | X | X |
| SW | I-Type | 0111 | | XX | 0 | 1 | X | 1 | 0 | 0 | 1 | XX | 00 | 1 | X | 1 | 0 |
| BGT | I-Type | 1000 | 0 | XX | 0 | 1 | X | 1 | 1 | 0 | 0 | XX | 00 | X | X | X | 0 |
| BGTZ | I-Type | 1000 | 1 | XX | 0 | 1 | X | 1 | 1 | 0 | 0 | XX | 01 | X | X | X | 0 |
| BLT | I-Type | 1001 | 0 | XX | 0 | 1 | X | 1 | 1 | 0 | 0 | XX | 00 | X | X | X | 0 |
| BLTZ | I-Type | 1001 | 1 | XX | 0 | 1 | X | 1 | 1 | 0 | 0 | XX | 01 | X | X | X | 0 |
| BEQ | I-Type | 1010 | 0 | XX | 0 | 1 | X | 1 | 1 | 0 | 0 | XX | 00 | X | X | X | 0 |
| BEQZ | I-Type | 1010 | 1 | XX | 0 | 1 | X | 1 | 1 | 0 | 0 | XX | 01 | X | X | X | 0 |
| BNE | I-Type | 1011 | 0 | XX | 0 | 1 | X | 1 | 1 | 0 | 0 | XX | 00 | X | X | X | 0 |
| BNEZ | I-Type | 1011 | 1 | XX | 0 | 1 | X | 1 | 1 | 0 | 0 | XX | 01 | X | X | X | 0 |
| JMP | J-Type | 1100 | | XX | 0 | X | X | 1 | X | 0 | 0 | XX | XX | X | X | X | X |
| CALL | J-Type | 1101 | | 10 | 1 | X | X | 1 | X | 0 | 0 | 10 | XX | X | X | X | X |
| RET | J-Type | 1110 | | XX | 0 | X | X | 1 | X | 0 | 0 | XX | 10 | X | X | X | X |
| Sv | S-Type | 1111 | | XX | 0 | X | 1 | 0 | X | 0 | 1 | XX | 00 | 0 | X | 0 | X |

### 3.2.2 Main control Boolean equation:

- RegDet1 =CALL
- RegDet0= AND+ADD+SUB
- RegWr= AND+ADD+SUB+ADDI+ANDI+LW+LBu+LBs+CALL
- ExtOp=(ANDI)’
- ExtOp2=Sv
- Imm-s=(Sv)’
- ALUSrc= (ADDI+ANDI+LW+LBu+LBs+SW)’
- MemRd= LW+LBu+LBs
- MemWr=SW+Sv
- WBdata1=(LBu+LBs+LW)’
- WBdata0=(LBu+LBs+CALL)’
- Src-S0= BGTZ+BLTZ+BEQZ+BNEZ
- Src-S1=RET
- Ads=(sv)’
- LExt=LBs
- Din=SW

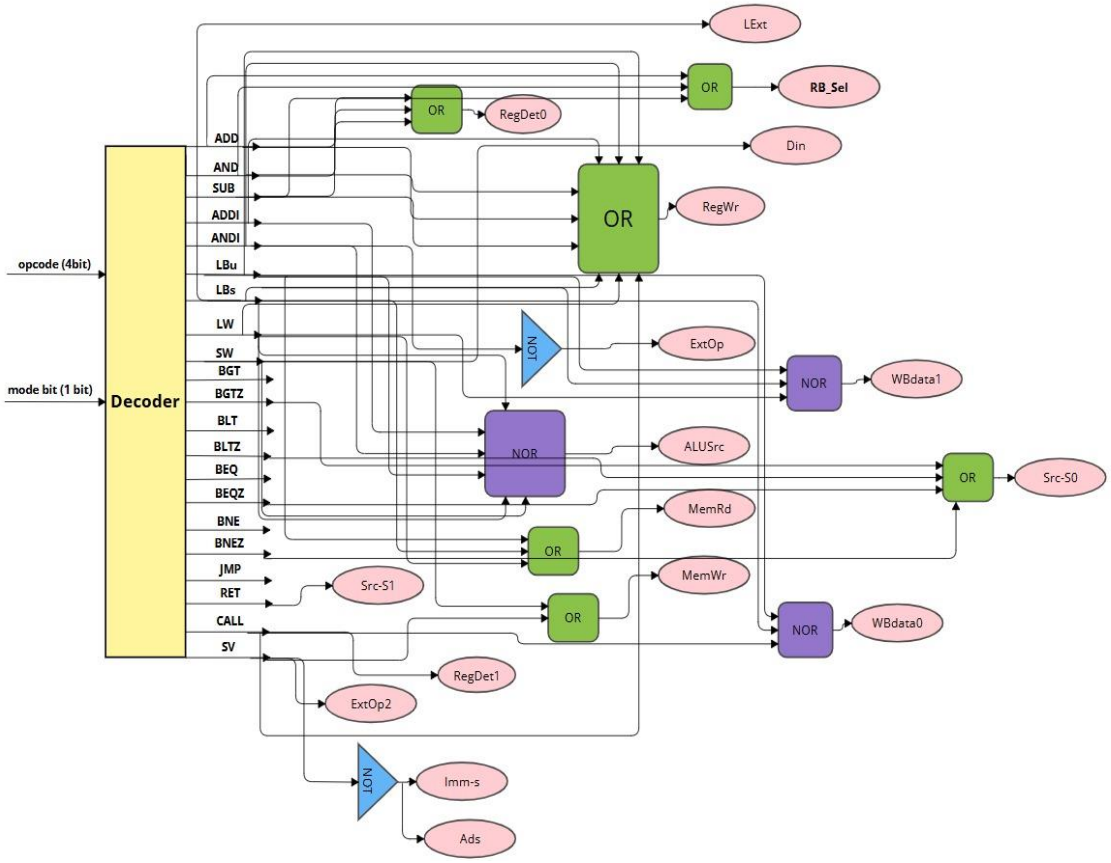



**Figure 23:Internal structure of the Main control unit**

In figure 22 we use Decoder 5x32

### *3.3 ALU control :*

Its responsible for choosing the operation inside the ALU, it take 4-bit opcode as input and ALUop as output it control the operation inside the ALU for example for the load the operation is add ,and the branch the operation is sub,etc.

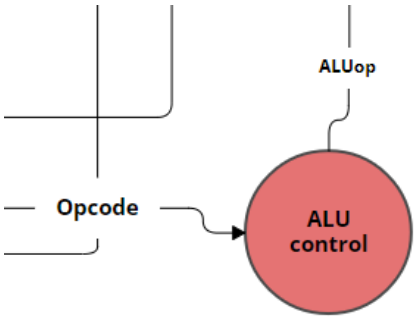



**Figure 24:ALU control unit**

### *3.3.1 ALU control truth table:*

**Table 3: ALU control truth table**

| Instruction | opcode | ALUOp | Type of operation (Sel) |
|---|---|---|---|
| AND | 0000 | AND | 01 |
| ADD | 0001 | ADD | 10 |
| SUB | 0010 | SUB | 11 |
| ADDI | 0011 | ADD | 10 |
| ANDI | 0100 | AND | 01 |
| LW | 0101 | ADD | 10 |
| LBu | 0110 | ADD | 10 |
| LBs | 0110 | ADD | 10 |
| SW | 0111 | ADD | 10 |
| BGT | 1000 | SUB | 11 |
| BGTZ | 1000 | SUB | 11 |
| BLT | 1001 | SUB | 11 |
| BLTZ | 1001 | SUB | 11 |
| BEQ | 1010 | SUB | 11 |
| BEQZ | 1010 | SUB | 11 |
| BNE | 1011 | SUB | 11 |
| BNEZ | 1011 | SUB | 11 |
| JMP | 1100 | X | 00 |
| CALL | 1101 | X | 00 |
| RET | 1110 | X | 00 |
| Sv | 1111 | X | 00 |

### 3.3.2 ALU control Boolean equation:

If (opcode = = 0000 || opcode = = 0100 ) Sel=01
Else if (opcode = = 0001 || opcode = = 0011 || opcode = = 0101 || opcode = = 0110 || opcode = = 0111) sel =10
Else if (opcode = = 0010 || opcode = = 1000 || opcode = = 1001 || opcode = =1010 || opcode == 1011) sel =11
Else sel =00

sel1= (AND+ANDI)'
sel0= (ADD+ADDI+LW+LBs+LBu+SW)'

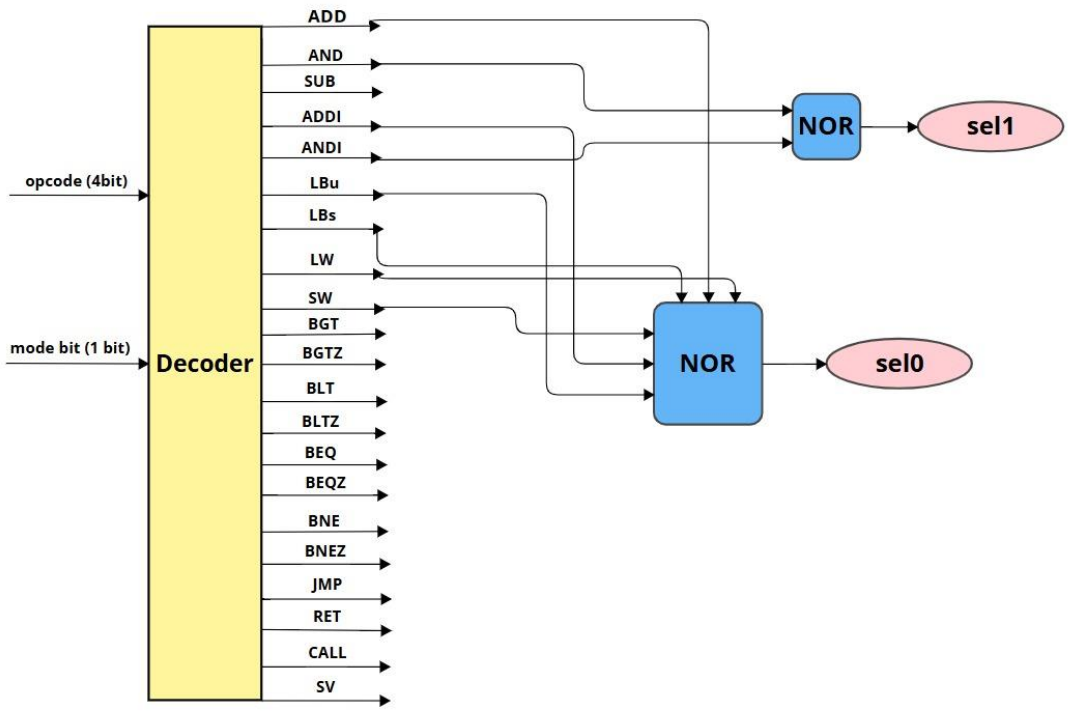



**Figure 25:Internal structure of the ALU control unit**

In figure 24 we use Decoder 5x32

## 4. Data Path stages:

The datapath operates through five sequential stages, each completed within one clock cycle:

1. Fetch Stage: The instruction is retrieved from the instruction cache memory. The CPU's control unit supplies the memory address to fetch the instruction into the CPU. This stage ensures that the instruction is ready for subsequent processing.

2. Decode Stage: Here, the opcode of the instruction determines its type, and the necessary registers are identified from the Instruction Register (IR). Each instruction type accesses specific registers, whose values are then routed via buses to the Arithmetic Logic Unit (ALU) for processing.

3. Execution Stage: Depending on the instruction type, this stage accepts inputs from registers or combines registers with immediate values. The ALU performs the specified operation, potentially altering flags based on the outcome. The result then proceeds as input to the next stage, Memory Access.

4. Memory Access: This stage interacts with the data memory portion of the cache memory. Instructions such as store and load directly access this memory: store writes data to memory, while load retrieves data from memory. This stage is pivotal for managing data storage and retrieval operations.

5. Write Back Stage: Not all instructions progress to this final stage; only those needing to update registers proceed here. The stage retrieves the ALU's output and writes it back to the appropriate registers. For load instructions, the data fetched from memory in the previous stage is written back to the registers.

Each stage operates in synchronization with the system clock, ensuring precise timing and coordinated data flow throughout the datapath. This structured approach enables efficient execution of instructions within the CPU, from initial instruction retrieval to final result storage or retrieval.

### 3.1 Stage for each instruction:

- The R-type instructions ( ADD, AND, SUB) need 4 stages which is: Fetch, Decode, Execution (ALU), Write Back.
- The ADDI, ANDI instruction from I-type need 4 stages which is: Fetch, Decode, Execution (ALU), Write Back.
- The Load instruction (LW,LBs,LBu) from I-type need 5 stages which is :Fetch, Decode, Execution (ALU), Memory, Write Back.
- The SW (Store instruction) from I-type need 4 stages which is : Fetch, Decode, Execution (ALU), Memory.
- The all Branch instructions from I-type need 3 stages which is: Fetch, Decode, Execution (ALU).
- The Jump and Return instruction from J-type need 2 stages which is: Fetch, Decode.

- The Call instructions from J-type need 3 stages which is : Fetch, Decode, Write Back.
- The Sv instruction from S-Type need 3 stages which is : Fetch ,Decode, Memory.

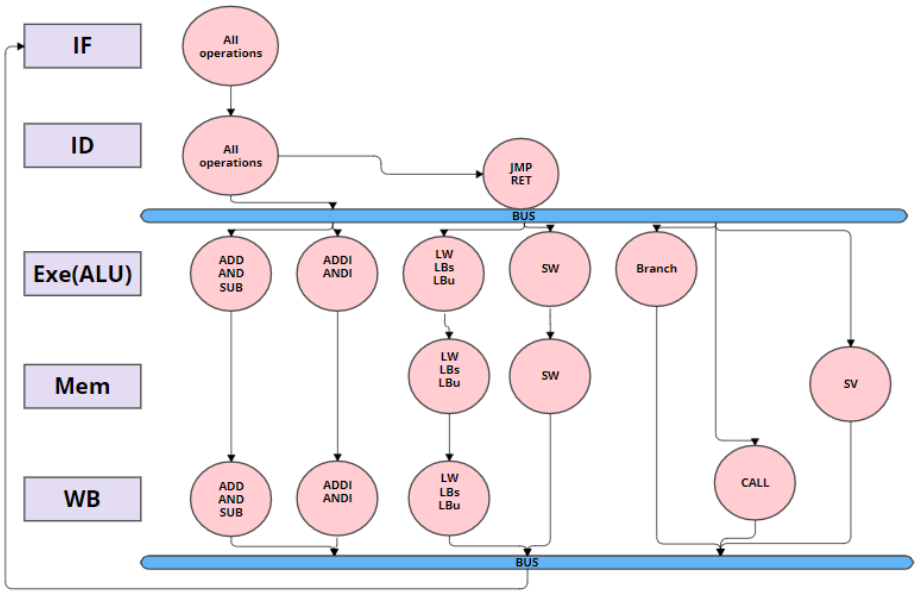



**Figure 26:State machine for operations**

## 5. RTL Design:

- **R-type (AND, ADD, SUB)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch operands:

- BusA ← Reg(RS1), BusB ← Reg(RS2)

Execute operation:

- ALU_result ← func(BusA, BusB)

Write ALU result:

- Reg(RD) ← ALU_result

Next PC address:

- PC ← PC + 2

- **I-TYPE (ADDI & ANDI)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch operands:

- BusA ← Reg(RS1), BusB ← Extend(imm5)

Execute operation:

- ALU_result ← op(BusA, BusB)

Write ALU result:

- Reg(RD) ← ALU_result

Next PC address:

- PC ← PC + 2

- **LBu (Load From I-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch base register:

- base ← Reg(RS1)

Calculate address:

- address_1 ← base + sign_extend(imm5)

Read memory:

- data_out_1 ← MEM[address_1]
- address_2 ←address_1 + 1
- data_out_2 ← MEM[address_2]

Zero Extend:

- data ← zero_extend(data_out_1)

Write register :

- Reg(RD) ← data

Next PC address:

- PC ← PC + 2

- **LBs (Load From I-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch base register:

- base ← Reg(RS1)

Calculate address:

- address_1 ← base + sign_extend(imm5)

Read memory:

- data_out_1 ← MEM[address_1]
- address_2 ←address_1 + 1
- data_out_2 ← MEM[address_2]

Sign Extend:

- data ← sign_extend(data_out_1)

Write register :

- Reg(RD) ← data

Next PC address:

- PC ← PC + 2

- **Lw (Load From I-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch base register:

- base ← Reg(RS1)

Calculate address:

- address_1 ← base + sign_extend(imm5)

Read memory:

- data_out_1 ← MEM[address_1]
- data[7:0] ← data_out_1
- address_2 ←address_1 + 1
- data_out_2 ← MEM[address_2]
- data[15:8] ← data_out_2

Write register :

- Reg(RD) ← data

Next PC address:

- PC ← PC + 2

- **SW (Store From I-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch registers:

- base ← Reg(RS1) , data← Reg(RD)

Calculate address:

- address_1 ← base + sign_extend(imm5)

Write memory:

- Data_in1← data[7:0]
- MEM[address_1] ← Data_in1
- Data_in2← data[15:8]
- address_2←address_1 +1
- MEM[address_2]← Data_in2

Next PC address:

- PC ← PC + 2

- **BGT (From I-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1]
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch operands:

- data1 ← Reg(RS1), data2 ← Reg(RD)

Greater:

- V,Z,N ← subtract(data1, data2)

Branch:

- if (Z==0 && N==V) PC ← PC + 2 + sign_ext(imm5)
- else PC ← PC + 2

- **BGTZ (From I-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch operands:

- data1 ← Reg(R0), data2 ← Reg(RD)

Greater:

- V,Z,N ← subtract(data1, data2)

Branch:

- if (Z= =0 && N= =V) PC ← PC + 2 + sign_ext(imm5)
- else PC ← PC + 2

- **BLT (From I-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch operands:

- data1 ← Reg(RS1), data2 ← Reg(RD)

Greater:

- V,N ← subtract(data1, data2)

Branch:

- if (N!=V) PC ← PC + 2 + sign_ext(imm5)
- else PC ← PC + 2

- **BLTZ (From I-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch operands:

- data1 ← Reg(R0), data2 ← Reg(RD)

Greater:

- V,N ← subtract(data1, data2)

Branch:

- if (N!=V) PC ← PC + 2 + sign_ext(imm5)
- else PC ← PC + 2

- **BEQ (From I-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch operands:

- data1 ← Reg(RS1), data2 ← Reg(RD)

Equality:

- zero ← subtract(data1, data2)

Branch:

- if (zero) PC ← PC + 2 + sign_ext(imm5)
- else PC ← PC + 2

- **BEQZ (From I-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch operands:

- data1 ← Reg(R0), data2 ← Reg(RD)

Equality:

- zero ← subtract(data1, data2)

Branch:

- if (zero) PC ← PC + 2 + sign_ext(imm5)
- else PC ← PC + 2

- **BNE (From I-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch operands:

- data1 ← Reg(RS1), data2 ← Reg(RD)

Equality:

- zero ← subtract(data1, data2)

Branch:

- if (zero==0) PC ← PC + 2 + sign_ext(imm5)
- else PC ← PC + 2

- **BNEZ (From I-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch operands:

- data1 ← Reg(R0), data2 ← Reg(RD)

Equality:

- zero ← subtract(data1, data2)

Branch:

- if (zero==0) PC ← PC + 2 + sign_ext(imm5)
- else PC ← PC + 2

- **JMP (From J-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Target PC address:

- target ← PC[15:13] || multiplication(jump offset12,2)

Jump:

- PC ← target

- **Call (From J-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Target PC address:

- target ← PC[15:13] || multiplication(jump offset12,2)
- data←PC+2

Jump:

- PC ← target

Write to register:

- Reg(R7)←data

- **Ret (from J-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Read Register:

- data ← Reg(R7)

return:

- PC ← data

- **sv(from s-type)**

Fetch instruction:

- Address_1 ← PC
- Address_2 ← PC+1
- Instruction_1 ← MEM[Address_1]
- Instruction [7:0] ← Instruction_1
- Instruction_2 ← MEM[Address_2]
- Instruction [15:8] ← Instruction_2

Fetch operands :

- address_1 ← Reg(RS1) , data← sign_extend(imm8)

Write memory:

- Data_in1← data[7:0]
- MEM[address_1] ← Data_in1
- Data_in2← data[15:8]
- address_2←address_1 +1
- MEM[address_2]← Data_in2

Next PC address:

- PC ← PC + 2

## 6.The Full DataPath:



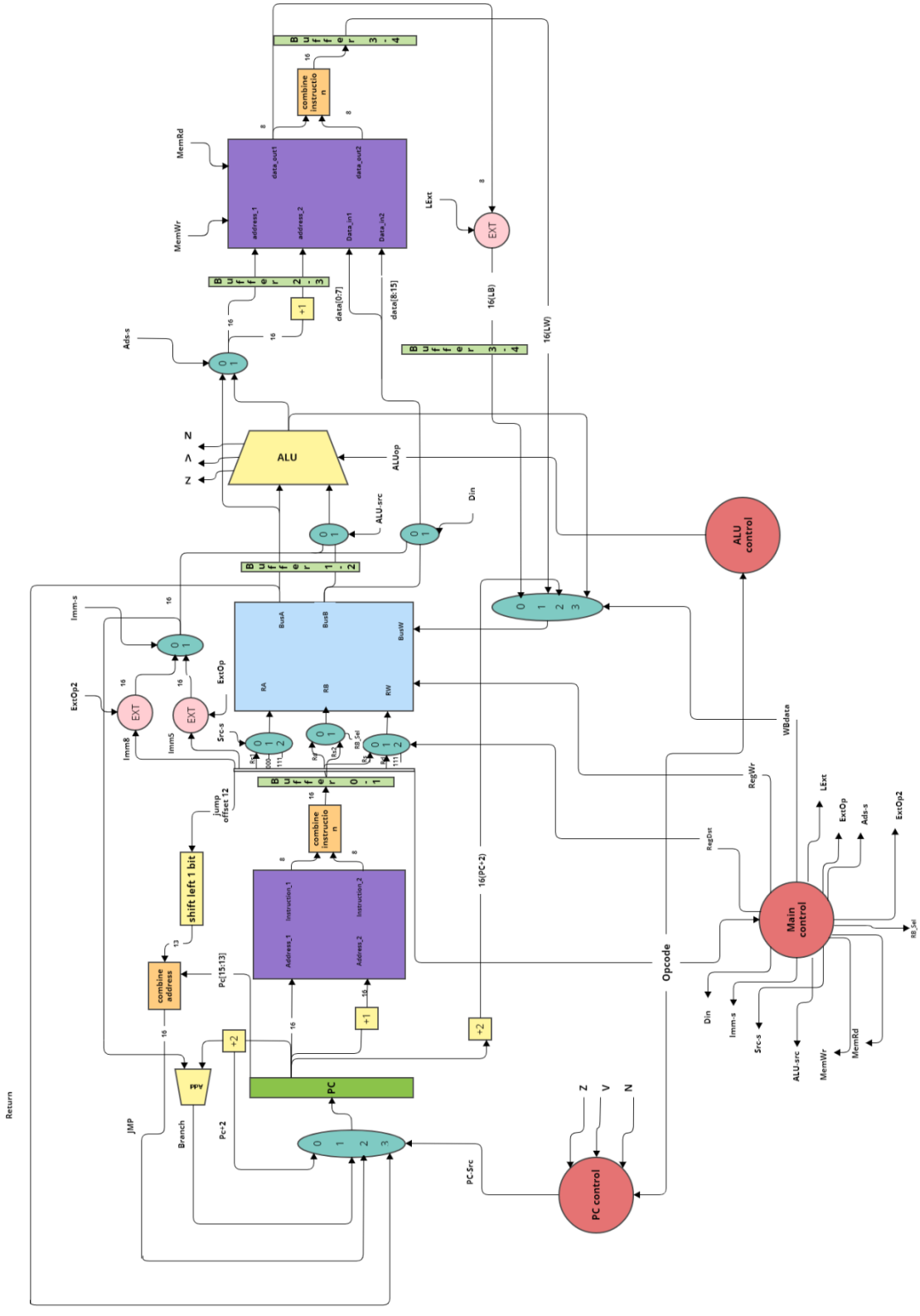



Figure27 :The full datapath

## 7.Testing and simulation:

```
51      reg [15:0] regData [0:7]; // for data stored in reg_file
52          initial begin
53            regData[0] = 16'h0000;
54            regData[1] = 16'h0001;  // 0001
55            regData[2] = 16'h000F;  // 1111
56            regData[3] = 16'h0004;
57            regData[4] = 16'h0000;
58            regData[5] = 16'h0110;
59            regData[6] = 16'h0000;
60            regData[7] = 16'h0007;
61            regData[8] = 16'h0110;
62
63          end
```

Figure 28:Register File Register Value

```
74        reg [7:0] mem [0: 32];// for data stored in data memory
75
76          initial begin
77             mem[2] = 11;
78             mem[3] = 42;
79             mem[4] = 176;
80             mem[5] = -72;
81             mem[6] = -60;
82             mem[7] = 35;
83             mem[8] = 05;
84
85
86          end
```

Figure 29:Data Memory Value

➔**ADD(R-Type):**

The instruction memory for add:

➔mem[16'h0000] = 8'b01010000;

➔mem[16'h0001] = 8'b00011010;

From the two 8 bit instructions in instruction memory that we read them combine them: Opcode :4’b0001 , Rd = 3’b101 ,Rs1=3’b001 ,Rs2=3’b010 , RegData[1] must apper at BusA RegData[2] must apper at BusB

Then BusA and BusB be the operands in AlU with operation Add So we add the values, The AlU result must be 10 and this result must written in regData[5] as clear in the simulation below.

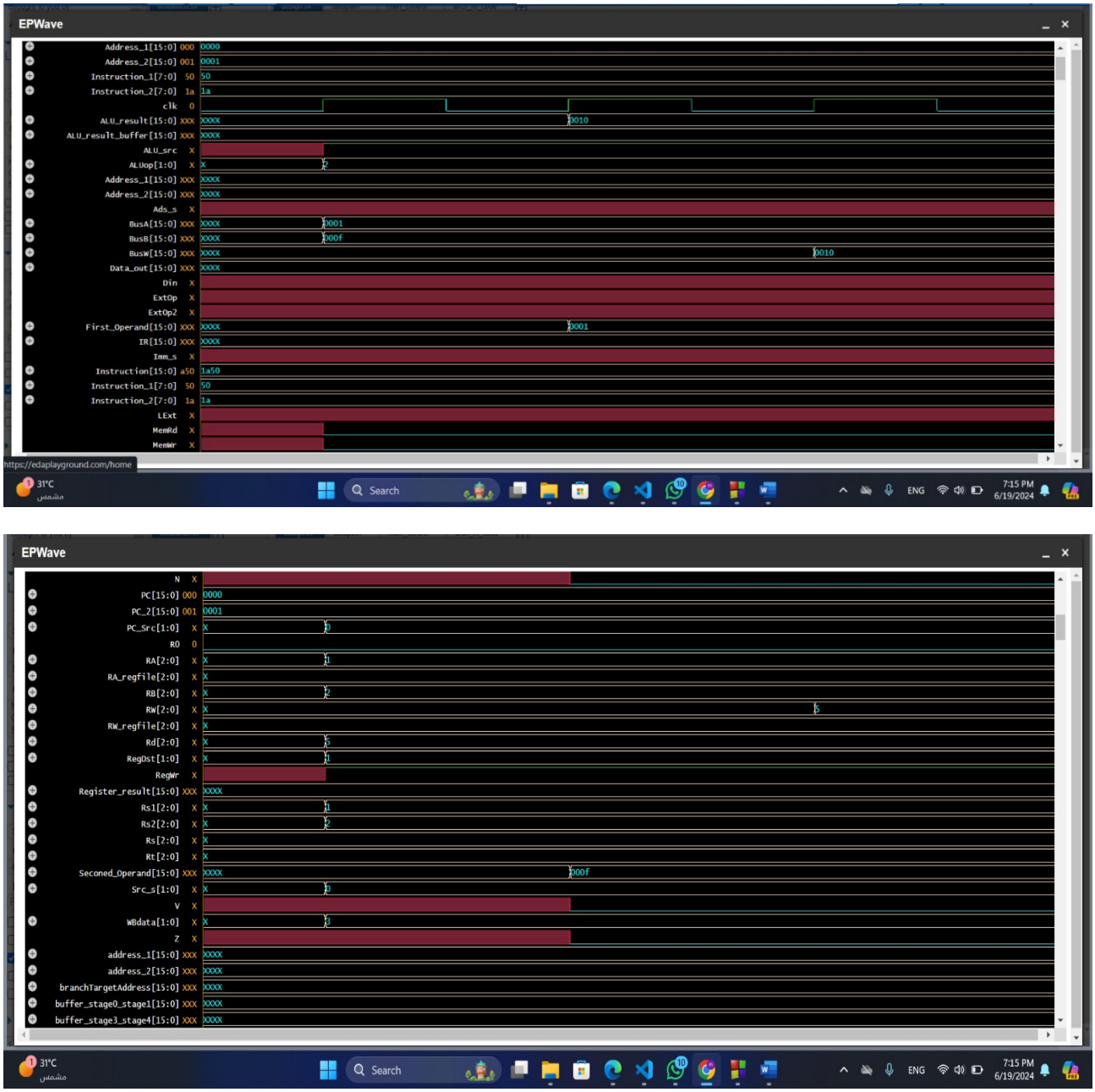



Figure 30:waveform for add instruction

➔**AND(R-Type):**

The instruction memory for and:
➔mem[16'h0000] = 8'b01010000;
➔mem[16'h0001] = 8'b00001010;
From the two 8 bit instructions in instruction memory that we read then combine them:
Opcode :4’b0000 , Rd = 3’b101 ,Rs1=3’b001 ,Rs2=3’b010 , RegData[1] must apper at BusA
RegData[2] must apper at BusB
Then BusA and BusB be the operands in AlU with operation AND So we and the values, The AlU result must be 1 and this result must written in regData[5] as clear in the simulation below.

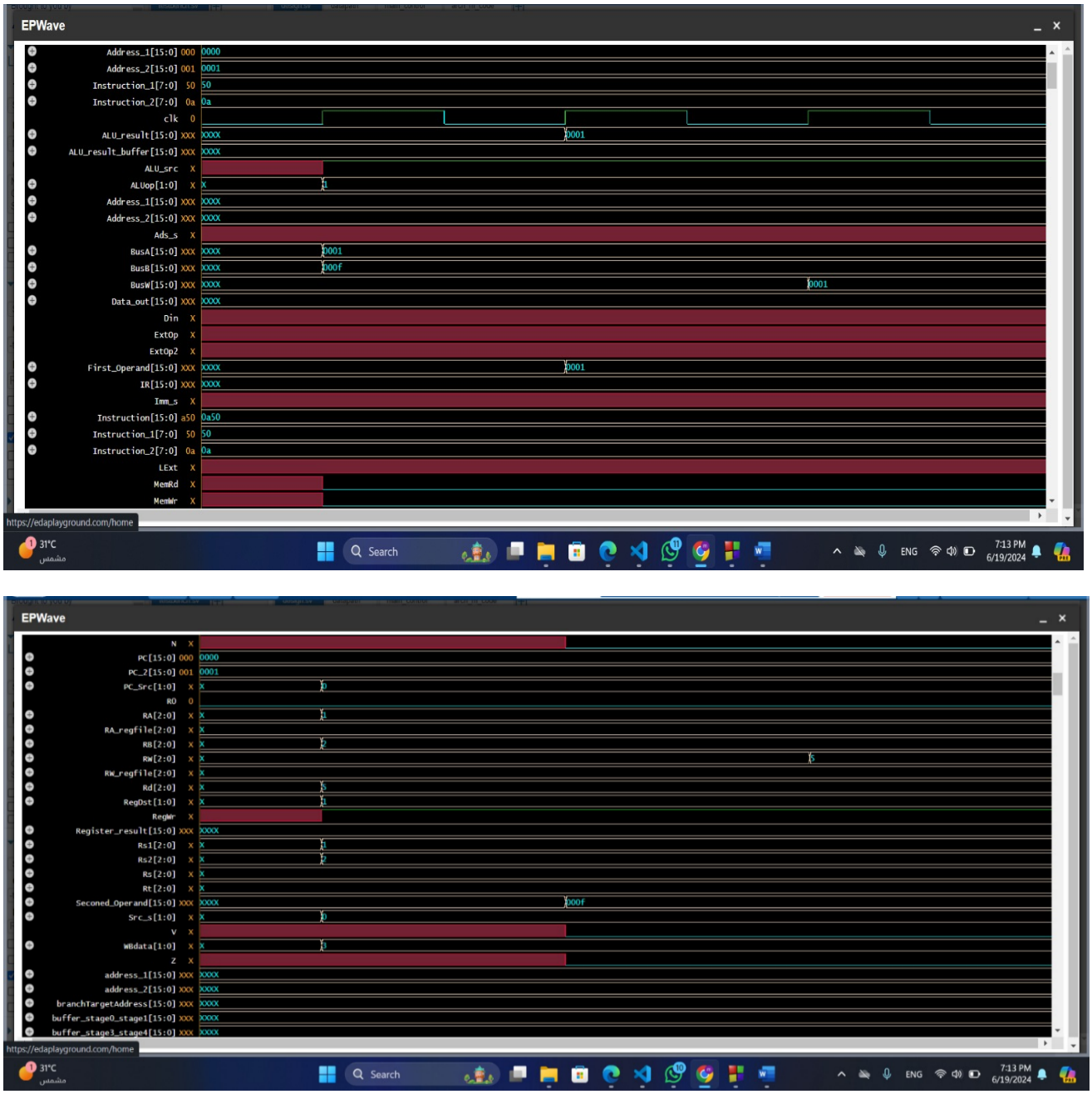


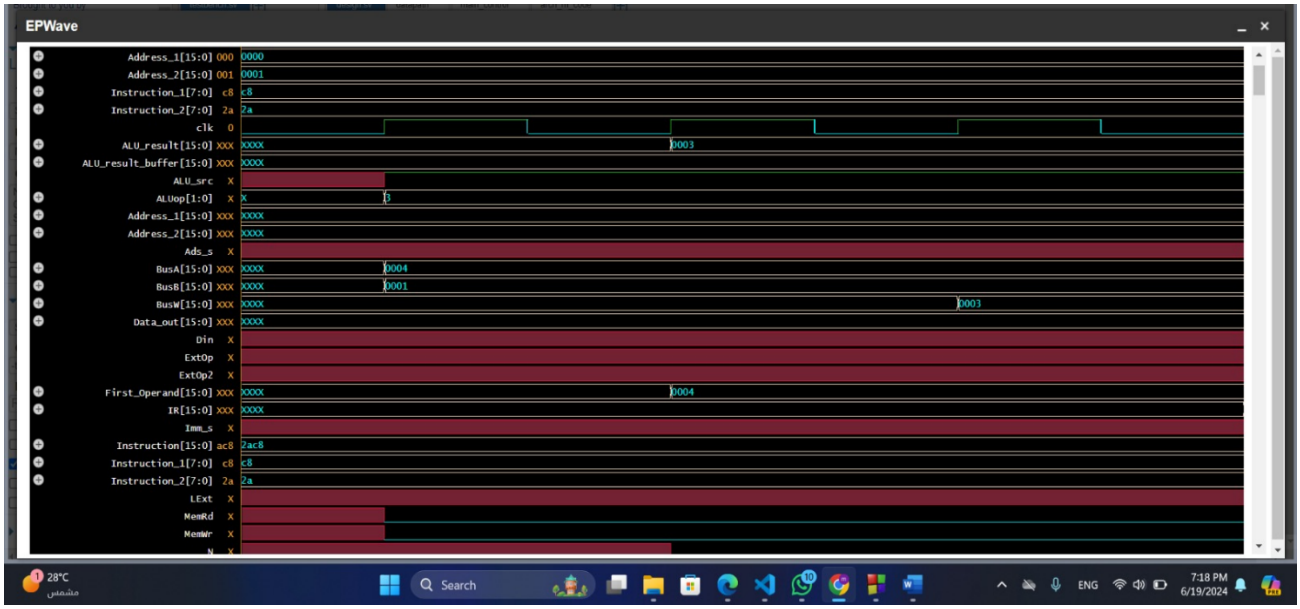




**Figure 31:waveform for and instruction**

➔**SUB(R-Type):**

The instruction memory for sub:
➔mem[16'h0000] = 8'b11001000;
➔mem[16'h0001] = 8'b00101010;
From the two 8 bit instructions in instruction memory that we read then combine them:
Opcode :4’b0010, Rd = 3’b101 ,Rs1=3’b011 ,Rs2=3’b001 , RegData[3] must apper at BusA
RegData[1] must apper at BusB
Then BusA and BusB be the operands in AlU with operation Sub So we sub the values, The AlU result must be 3 and this result must written in regData[5] as clear in the simulation below.



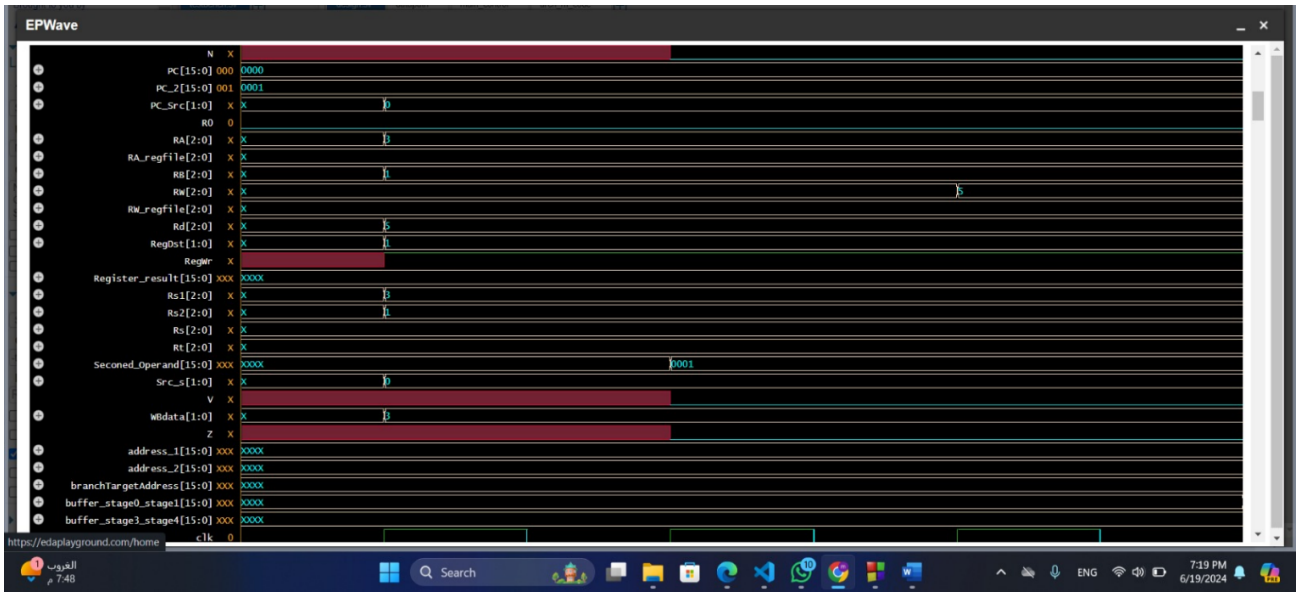




Figure 32:waveform for SUB instruction

**➔ADDI(I-Type):**

The instruction memory for addi:
➔mem[16'h0000] = 8'b01100011;
➔mem[16'h0001] = 8'b00110101;
From the two 8 bit instructions in instruction memory that we read then combine them: Opcode :4’b0011, Rd = 3’b101 ,Rs1=3’b011 ,imm=5’b00011, RegData[3] must apper at BusA. And imm have to signed extened then added to the value in BUSA inside ALU with ALU operation ADD ,So The AlU result must be 7 and this result must written in regData[5] as clear in the simulation below.

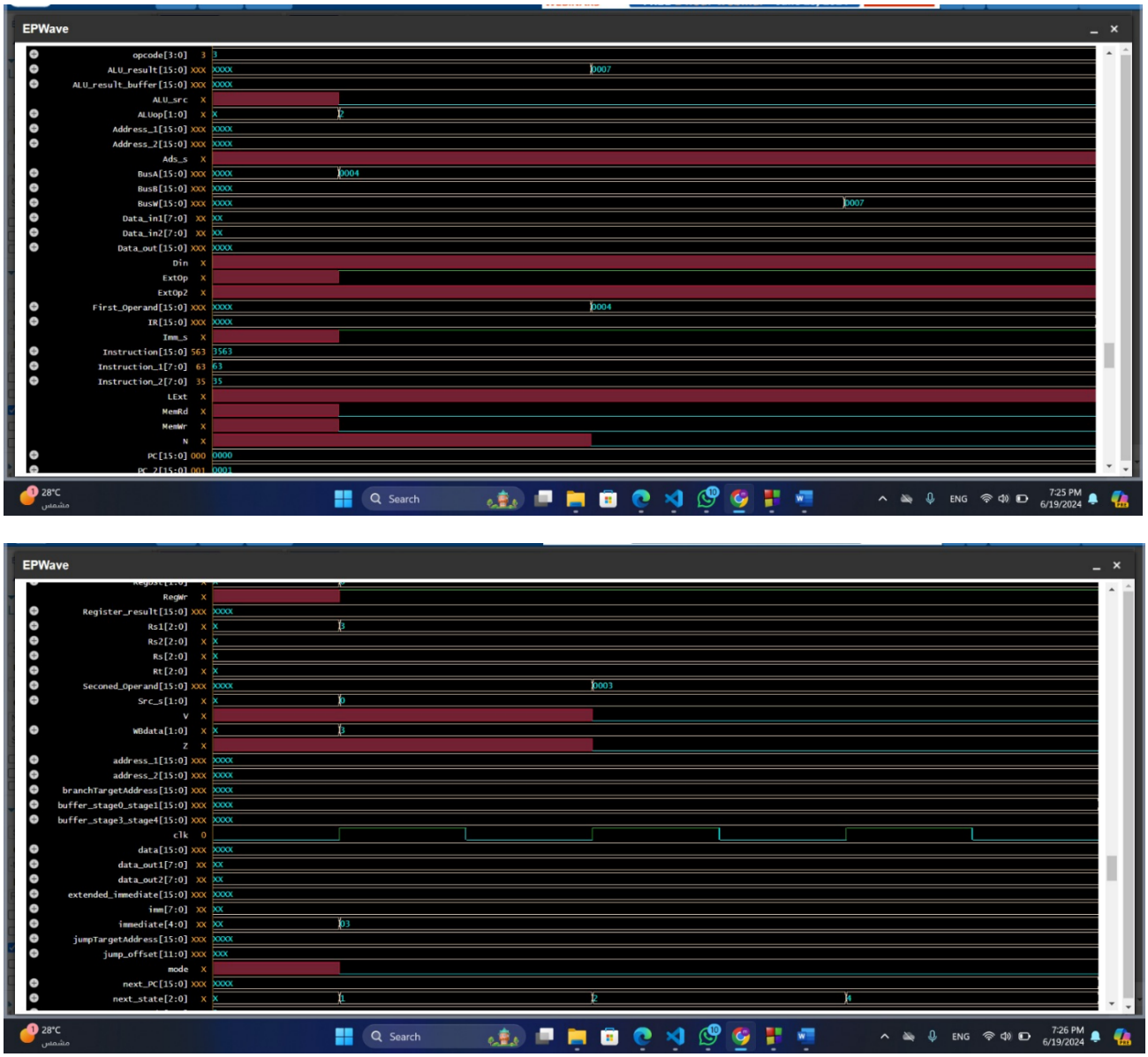



Figure 33:waveform for ADDI instruction

**➔LW:**

The instruction memory foe LW:

➔mem[16'h0000] = 8'b00100001;

➔mem[16'h0001] = 8'b01011101;

From the two 8 bit instructions in instruction memory that we read then combine them:

Opcode :4’b0101, Rd = 3’b10 1,Rs1=3’b001 ,imm=5’b00001, RegData[1] must apper at BusA

And imm have to signed extened then added to the value in BUSA inside the alu with alu operation add. So The AlU result must be 2 and this result is the address_1 and address_2 =address_1 +1=3 of data from data memory that we read then write it in regData[5] as clear in the simulation below.

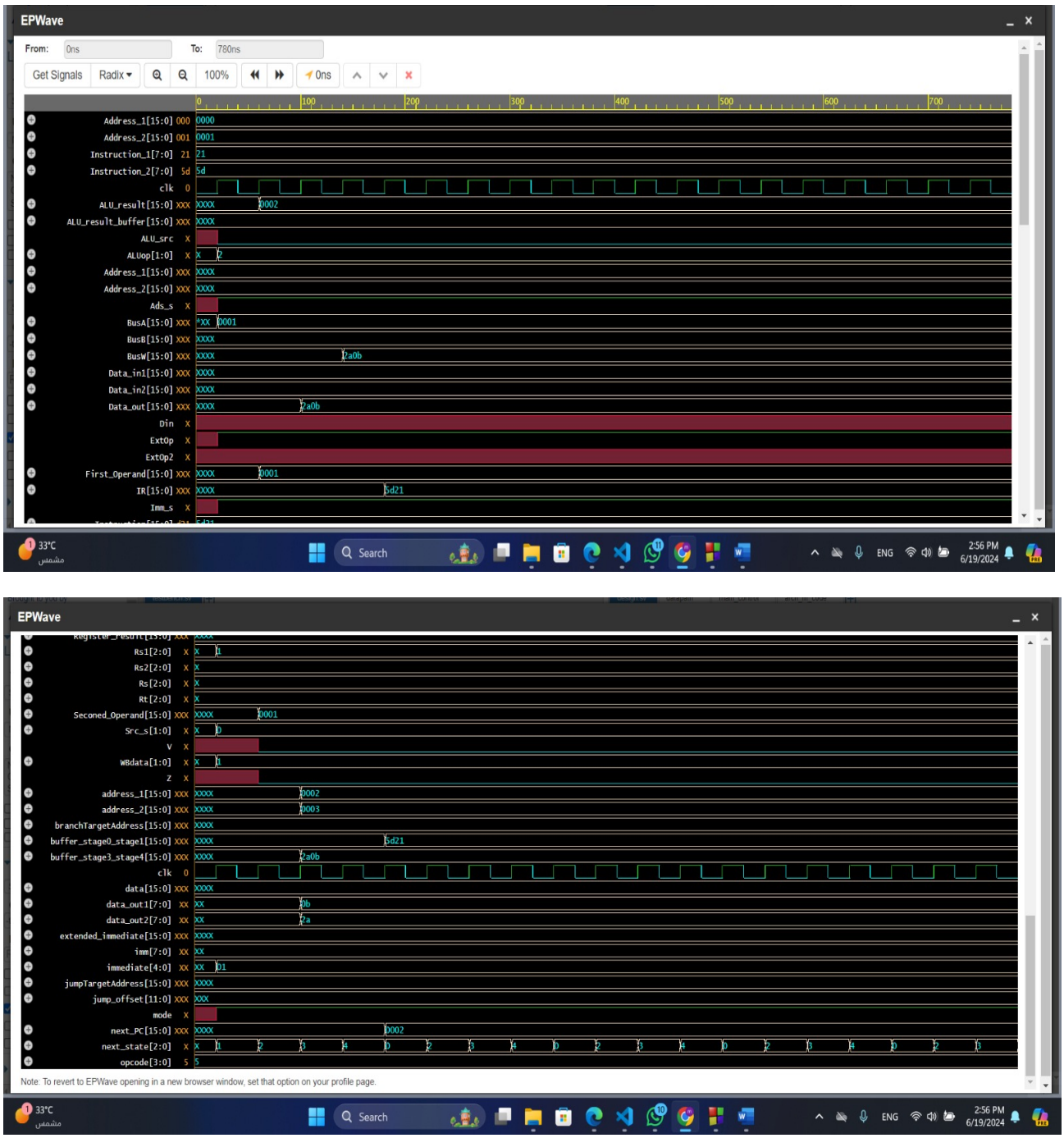



Figure 34:waveform for LW instruction

➔LBu:

The instruction memory foe LBu:

➔mem[16'h0000] = 8'b00100001;

➔mem[16'h0001] = 8'b01100101;

From the two 8 bit instructions in instruction memory that we read then combine them:

Opcode :4’b0110,m=0, Rd = 3’b10 1,Rs1=3’b001 ,imm=5’b00001, RegData[1] must apper at BusA

And imm have to signed extened then added to the value in BUSA inside the alu with alu operation add. So The AlU result must be  2 and this result is the address_1 of data from data memory that we read then write it in regData[5] as clear in the simulation below.

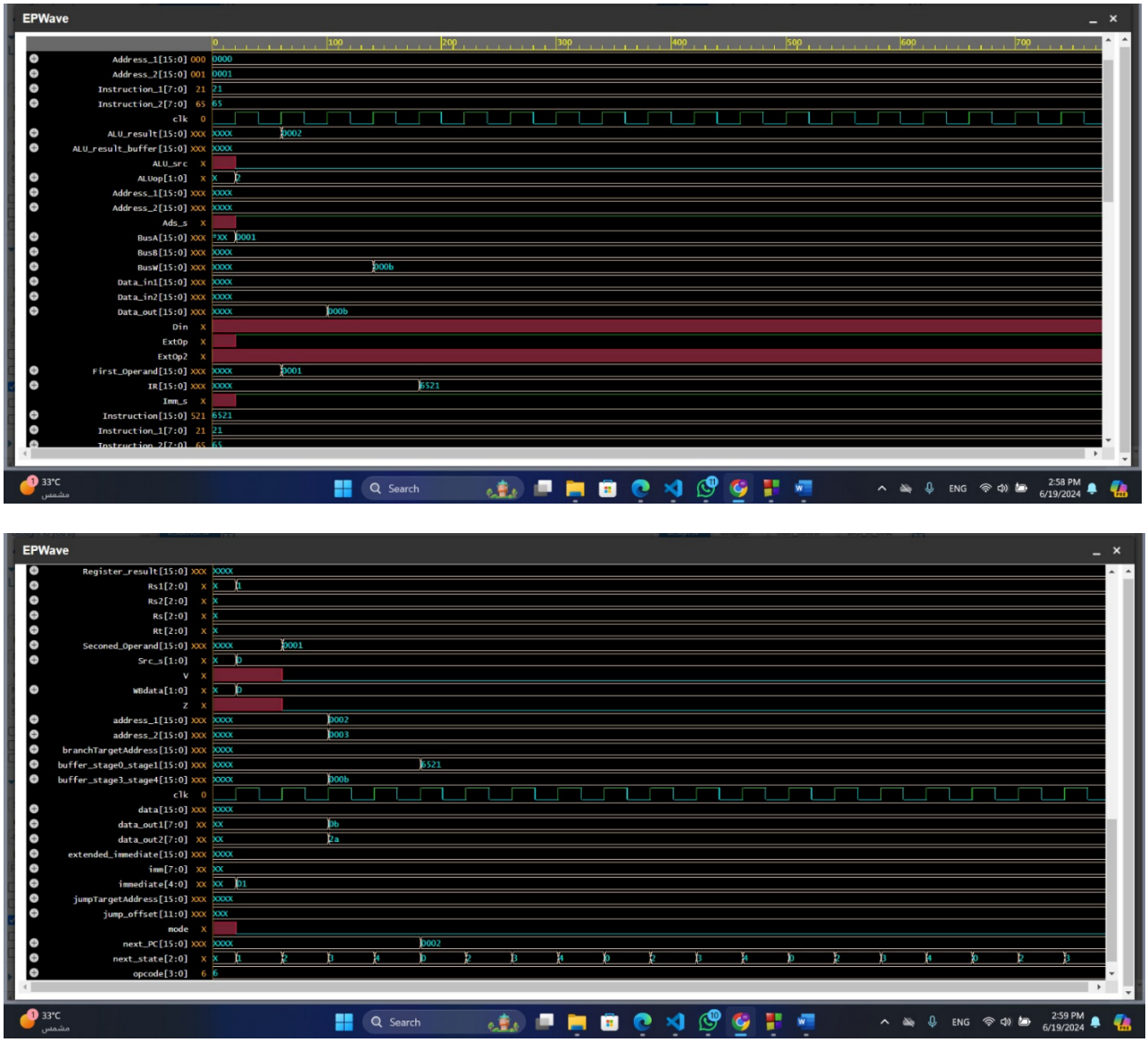






Figure 35:waveform for LBu instruction

➔LBs:

The instruction memory foe LBs:

➔ mem[16'h0000] = 8'b00100011;

➔ mem[16'h0001] = 8'b01101101;

From the two 8 bit instructions in instruction memory that we read then combine them:
Opcode :4’b0110,m=1, Rd = 3’b10 1,Rs1=3’b001 ,imm=5’b00011, RegData[1] must apper at BusA
And imm have to signed extened then added to the value in BUSA inside the alu with alu operation add. So The AlU result must be 4 and this result is the address_1 of data from data memory that we read then write it in regData[5] as clear in the simulation below.


Figure 36:waveform for LBs instruction

➔SW:

The instruction memory foe SW:

➔mem[16'h0000] = 8'b00100011;

➔mem[16'h0001] = 8'b01111101;

From the two 8 bit instructions in instruction memory that we read then combine them:

Opcode :4’b0111,Rd = 3’b101,Rs1=3’b001 ,imm=5’b00011, RegData[1] must apper at BusA

And imm have to signed extened then added to the value in BUSA inside the alu with alu operation add. So The AlU result must be  4 and this result is the address_1 and address_2=address_1 +1 =5 from data memory  that we want to write at it as clear in the simulation below.


Figure 37:waveform for SW instruction

➔BGT:

The instruction memory foe BGT:
➔ mem[16'h0000] = 8'b00100111;
➔mem[16'h0001] = 8'b10000101;
From the two 8 bit instructions in instruction memory that we read then combine them: Opcode :4’b1000,m=0,Rd = 3’b101,Rs1=3’b001 ,imm=5’b00111, then the value at Rd must compare with the value in Rs1 using sub operation in alu and here the value in Rd is larger so the branch is taken and we go to next address which is pc+sign-ext(imm)

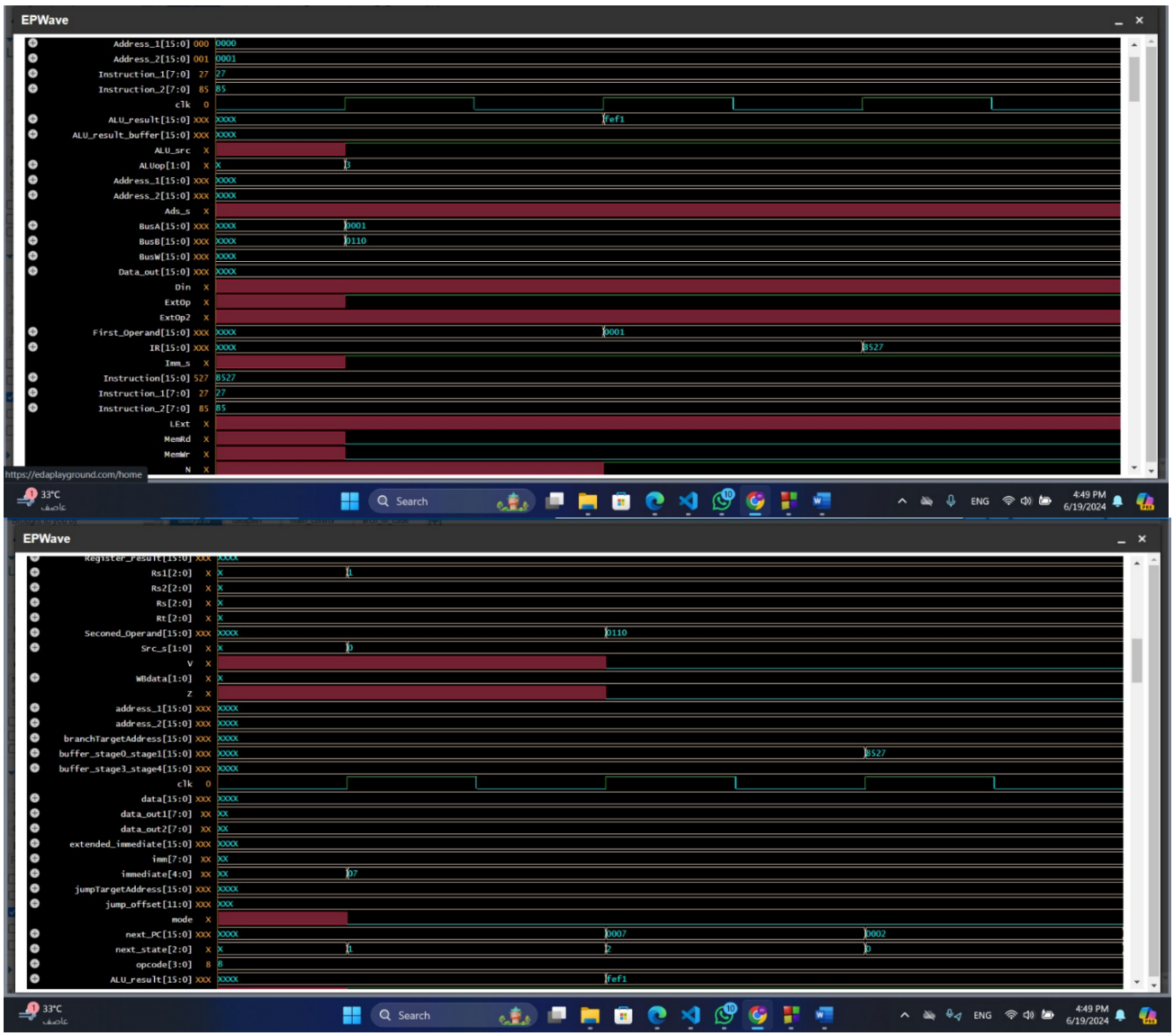



Figure 38:waveform for BGT instruction

➔BGTZ:

The instruction memory foe BGTZ:

➔ mem[16'h0000] = 8'b00100111;

➔ mem[16'h0001] = 8'b10001101;

From the two 8 bit instructions in instruction memory that we read then combine them: Opcode :4’b1000,m=1 ,Rd = 3’b101,Rs1=3’b001 ,imm=5’b00111, then the value at Rd must compare  with the value in R0 which is always 0 using sub operation in alu and here the value in Rd is larger so the branch is taken and we go to next address which is pc+sign-ext(imm)

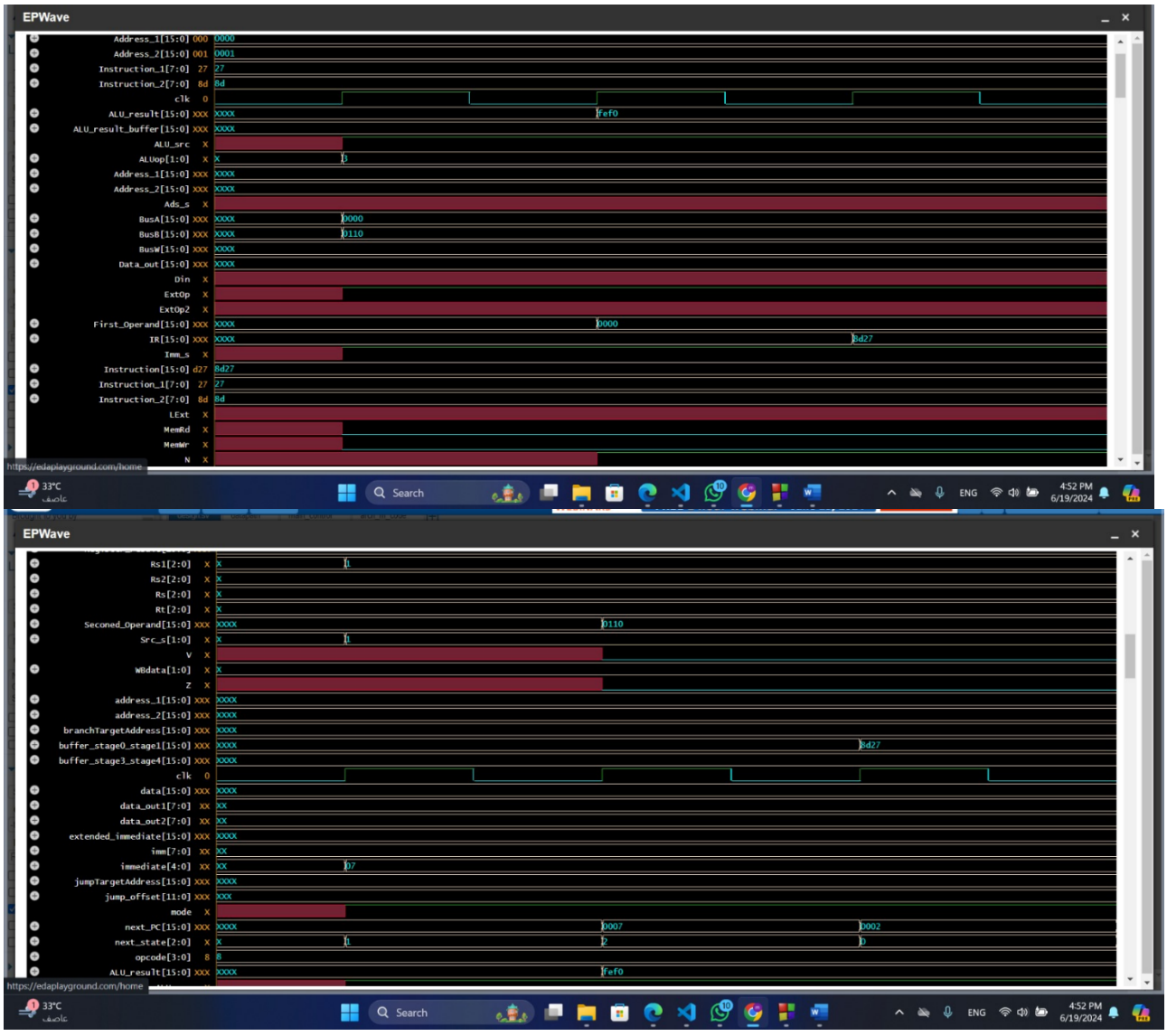



Figure 39:waveform for BGTZ instruction

➔BLT:

The instruction memory foe BLT:

➔ mem[16'h0000] = 8'b00100111;

➔ mem[16'h0001] = 8'b10010101;

From the two 8 bit instructions in instruction memory that we read then combine them:

Opcode :4’b1001,m=0,Rd = 3’b101,Rs1=3’b001 ,imm=5’b00111, then the value at Rd must compare with the value in Rs1 using sub operation in alu and here the value in Rd is larger so the branch is not taken and we go to next address which is pc+sign-ext(imm).

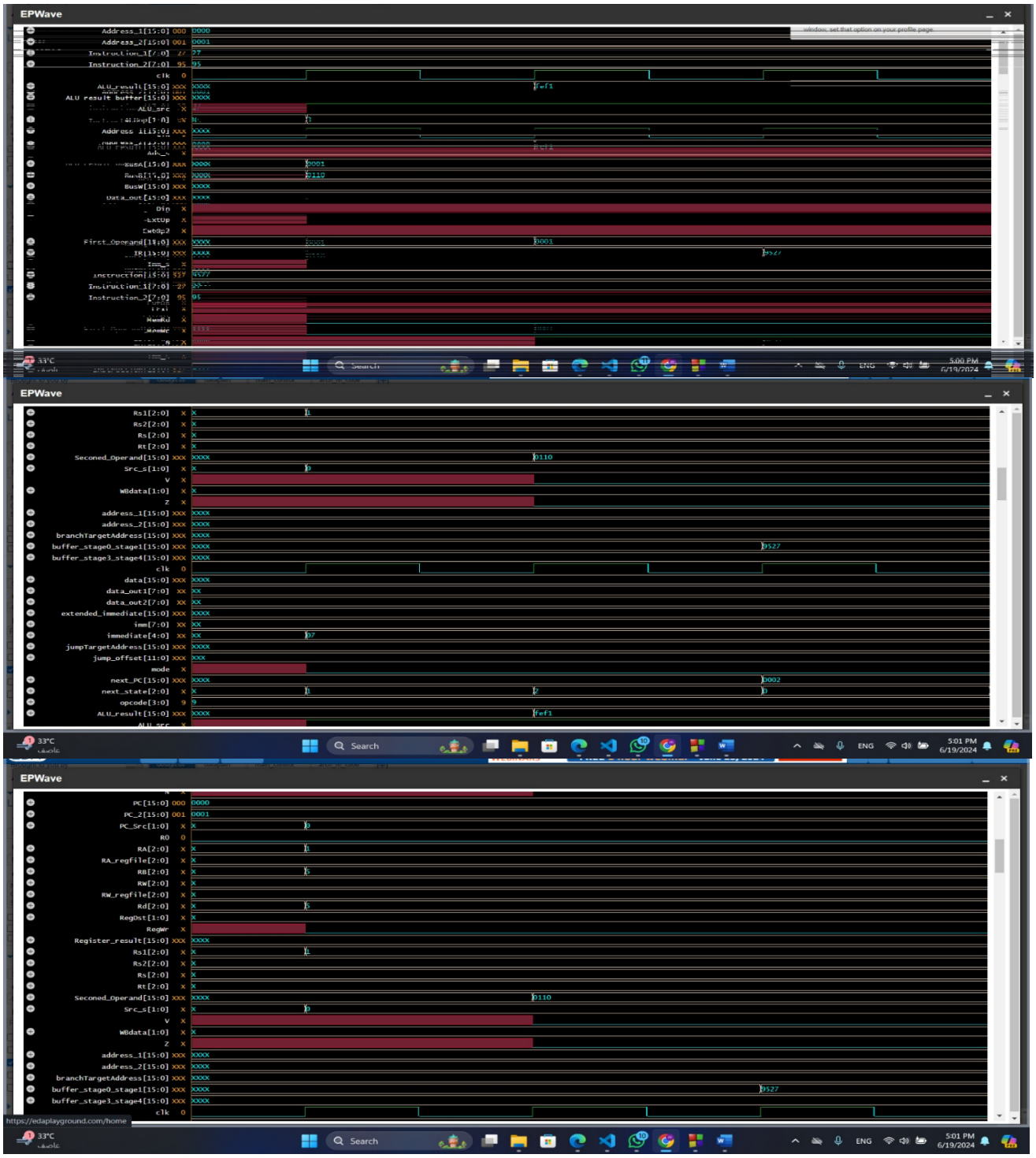



Figure 40:waveform for BLT instruction

➔BEQ:

The instruction memory foe BEQ:

➔ mem[16'h0000] = 8'b00100111;

➔ mem[16'h0001] = 8'b10100101;

From the two 8 bit instructions in instruction memory that we read then combine them:

Opcode :4’b1010,m=0,Rd = 3’b101,Rs1=3’b001 ,imm=5’b00111, then the value at Rd must compare with the value in Rs1 using sub operation in alu and here the value in Rd is not equal to value in Rs1 so the branch is not taken and we go to next address which is pc+2

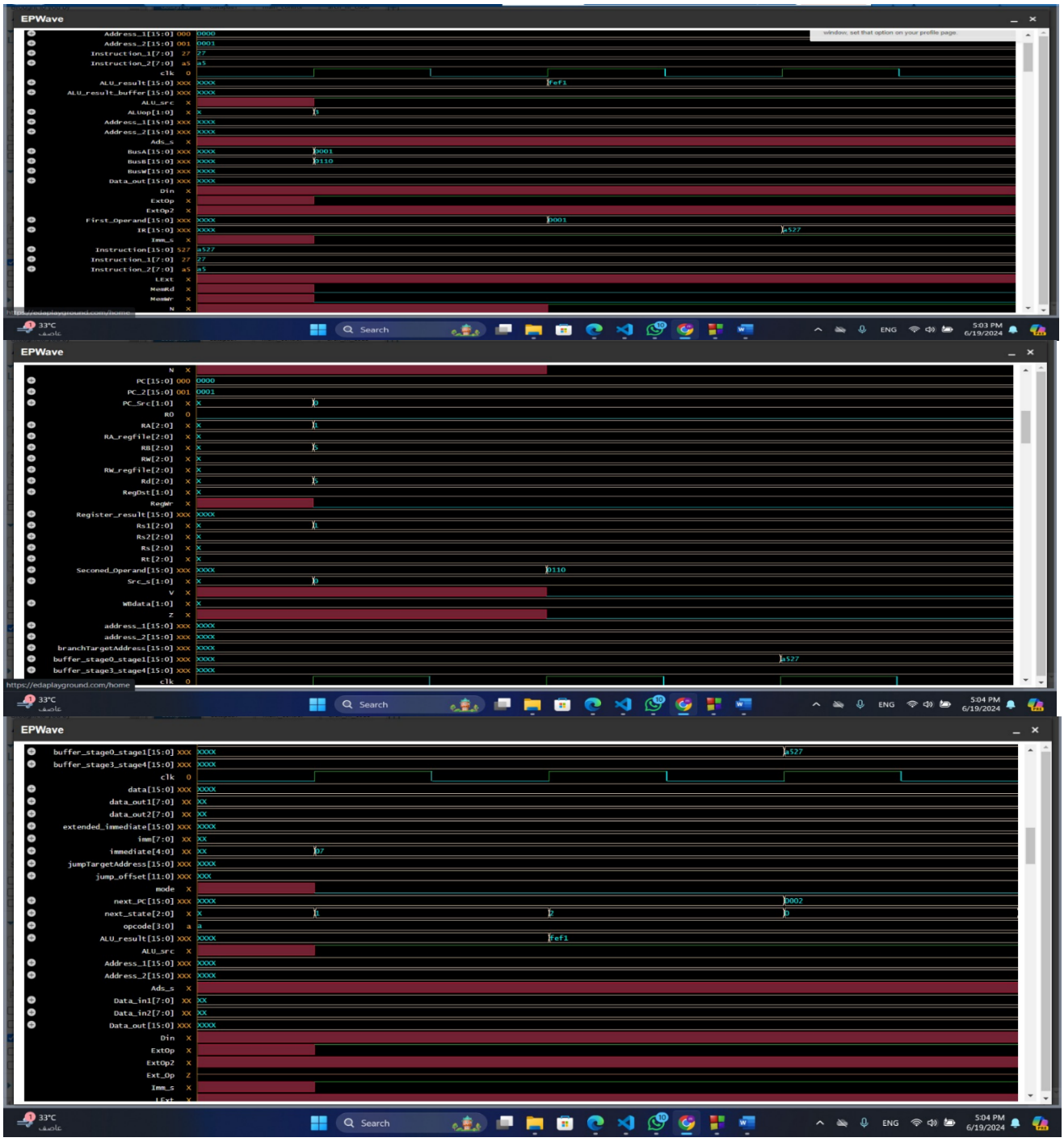



Figure 41:waveform for BEQ instruction

➔BNE:

The instruction memory foe BNE:

➔ mem[16'h0000] = 8'b00100111;

➔ mem[16'h0001] = 8'b10110101;

From the two 8 bit instructions in instruction memory that we read then combine them:

Opcode :4’b1011,m=0,Rd = 3’b101,Rs1=3’b001 ,imm=5’b00111, then the value at Rd must compare with the value in Rs1 using sub operation in alu and here the value in Rd is not equal to value in Rs1 so the branch is taken and we go to next address which is pc+sing-ext(imm)

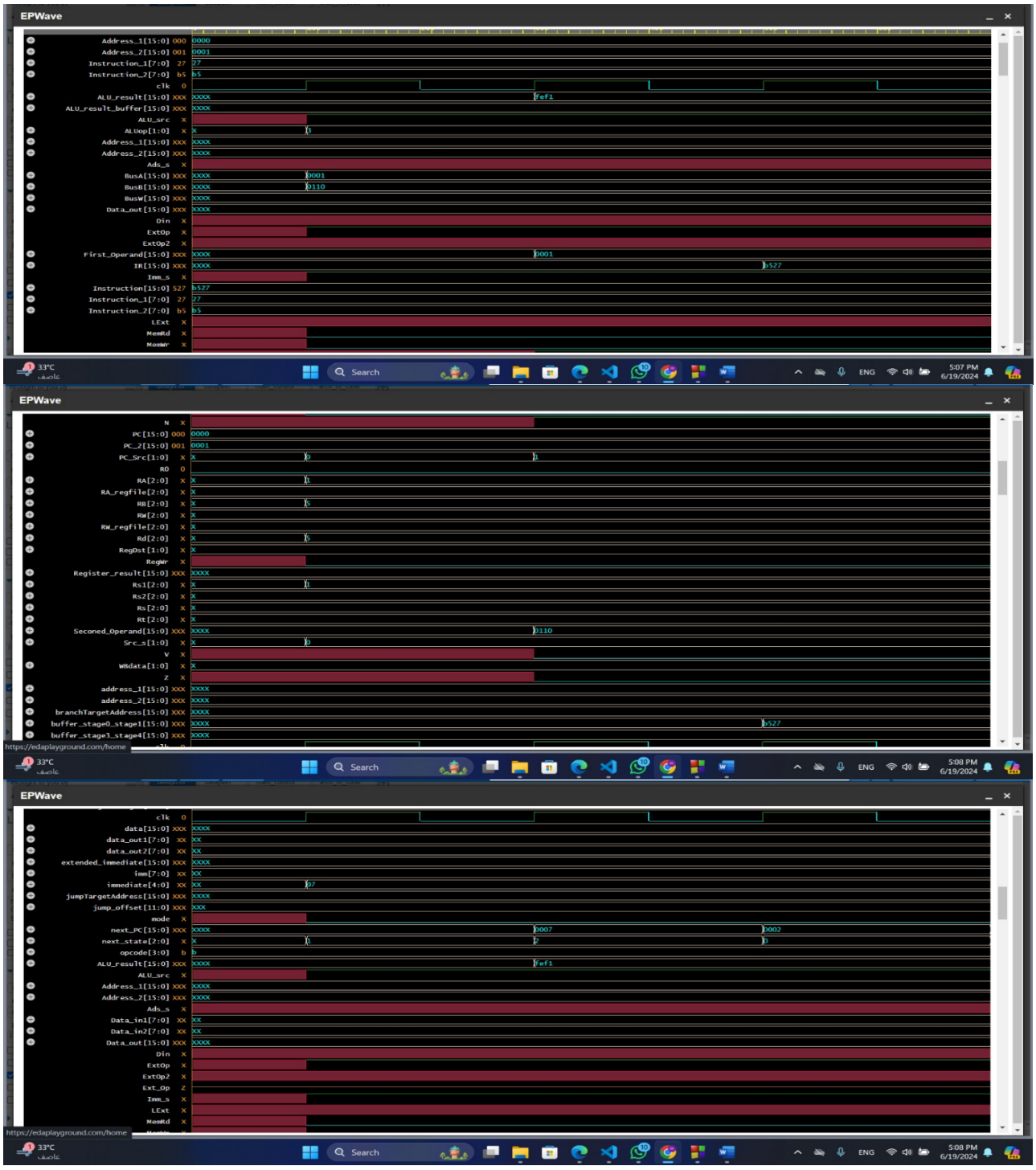



Figure 42:waveform for BNE instruction

➔JMP:

The instruction memory foe JMP:

➔mem[16'h0000] = 8'b00000100;

➔mem[16'h0001] = 8'b11000000;

From the two 8 bit instructions in instruction memory that we read then combine them:

Opcode :4’b1100, jump offset =12’b000000000100

In this case we make shift left 1 bit for the jump offset so it be 8 which is the jump target address and then combine it with 3 most significant bit from pc and it still 8 which is the next pc . And this is clear in figure below

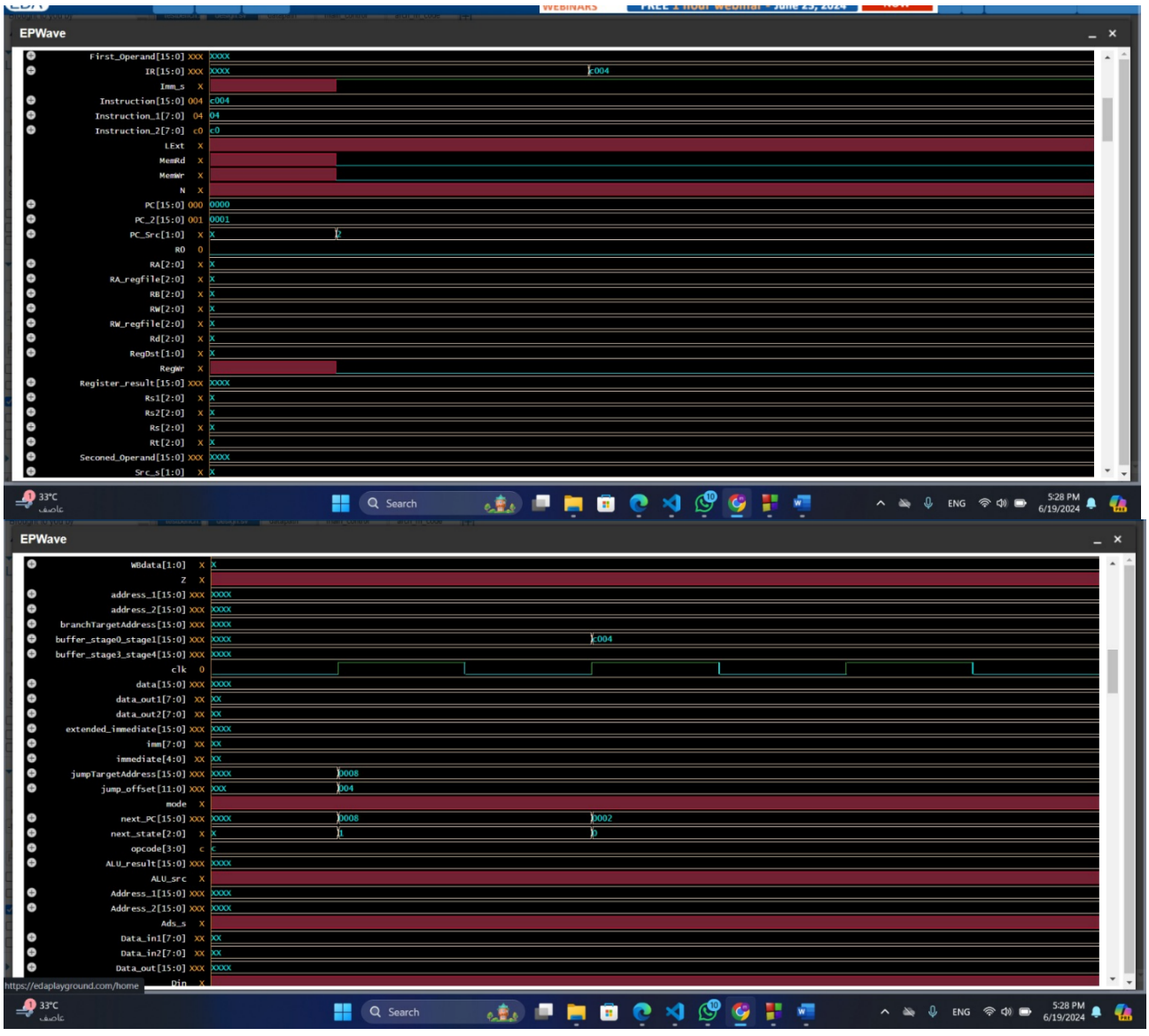



Figure 43:waveform for JMP instruction

➔CALL:

The instruction memory foe CALL:

➔mem[16'h0000] = 8'b00000100;

➔mem[16'h0001] = 8'b11010000;

From the two 8 bit instructions in instruction memory that we read then combine them:

Opcode :4’b1101, jump offset =12’b000000000100

In this case we make shift left 1 bit for the jump offset so it be 8 which is the jump target address and then combine it with 3 most significant bit from pc and it still 8 which is the next pc and the pc+2 which is 2 must store in regData[7] (which is apper in Busw in figure below). And this is clear in figure below

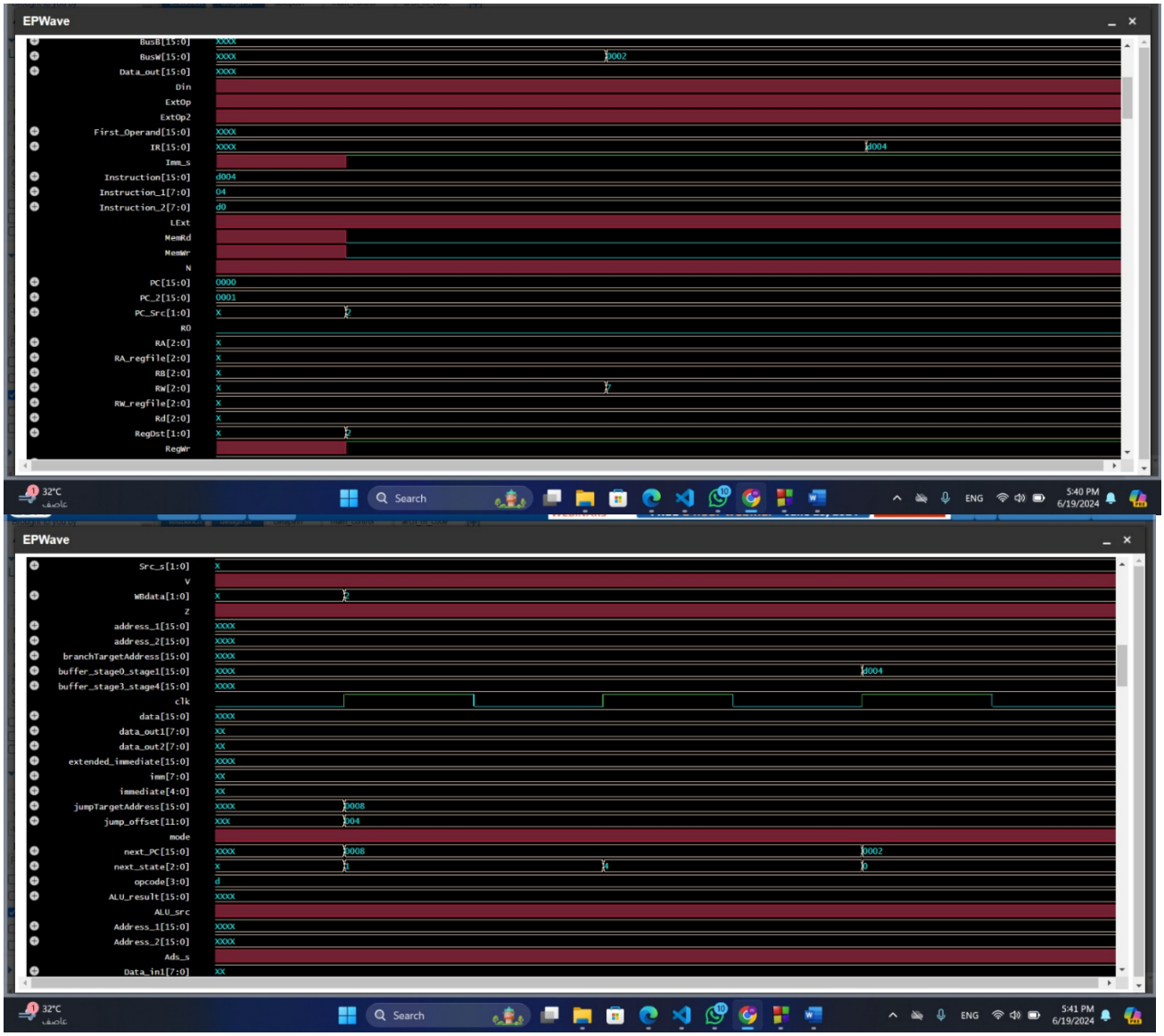



Figure 44:waveform for CALL instruction

➔RET:

The instruction memory foe RET:

➔ mem[16'h0000] = 8'b00000100;

➔ mem[16'h0001] = 8'b11100000;

From the two 8 bit instructions in instruction memory that we read then combine them:

Opcode :4’b1110

In this case we read the value of regData[7]  which is 1 and it be the next pc . And this is clear in figure below

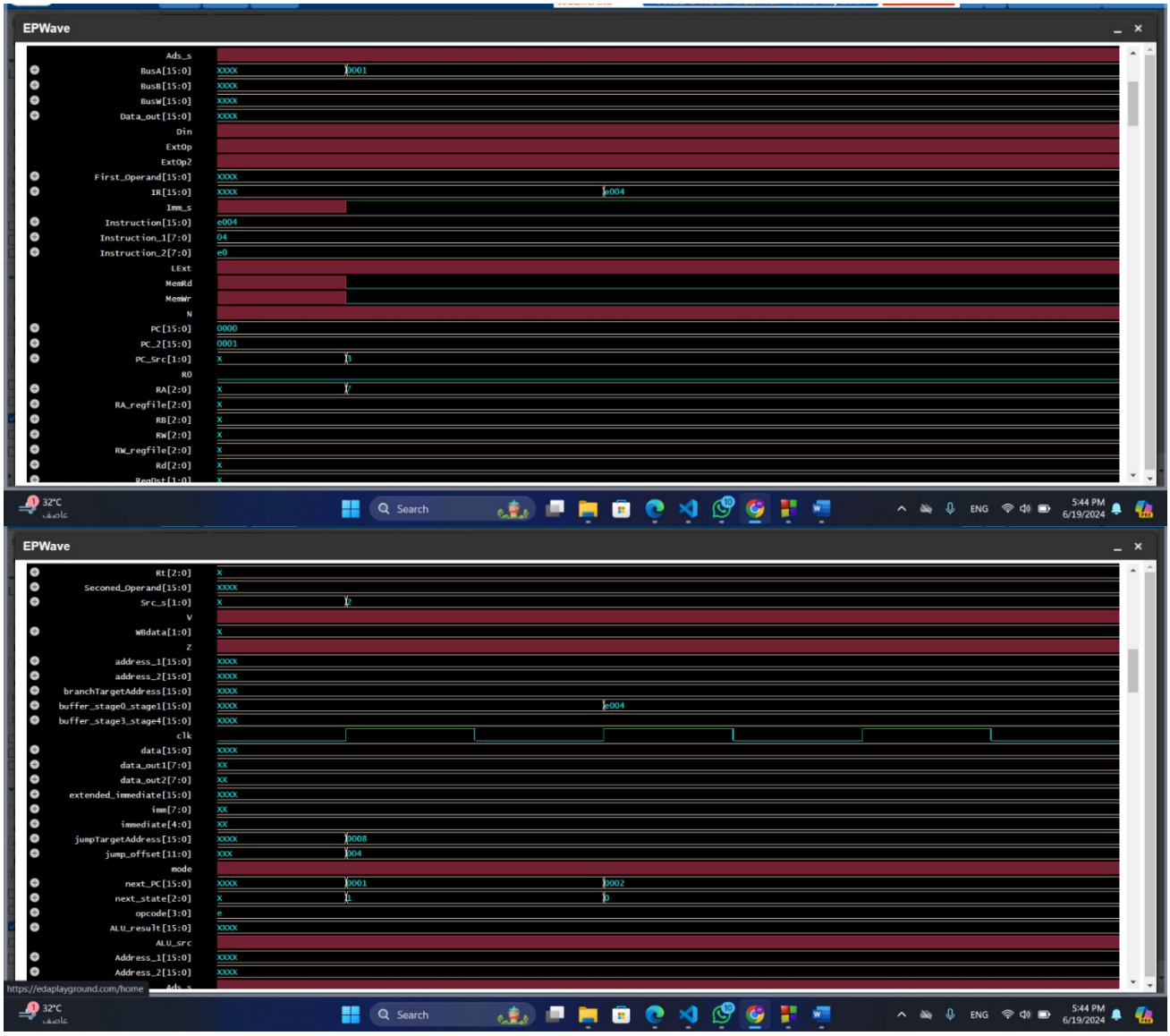



Figure 45:waveform for RET  instruction

➔SV:

The instruction memory foe SV:

➔ mem[16'h0000] = 8'b00100010;

➔mem[16'h0001] = 8'b11111110;

From the two 8 bit instructions in instruction memory that we read then combine them:

Opcode :4’b1111,Rs= 3’b111,imm=8’b00010001, RegData[7] must apper at BusA then it be the address1 which is 7that we want to write first part of the extend imm  and address2 is address1 +1 which is 8bit for the second part of extened imm inside the data memory

as clear in the simulation below.


Figure 46:waveform for SV  instruction

## 8.Teamwork:

All group members participated in all parts of the project, so we made zoom meetings, shared ideas, and did everything together in code, report,digrams,datapath, and test bench.