Faculty of Engineering & Technology Electrical & Computer Engineering Department

Computer Architecture

Project II

Simple Multicycle RISC Processor

Prepared by: Shahd Arda 1211034 Dr. Ayman 2

Mohammad Salameh 1212109 Dr. Ayman 2

Mai Beitnoba 1210260 Dr. Aziz 1

Date: 6/20/2024.

Abstract

This report covers the design and verification of a simple pipelined RISC processor implemented in Verilog. The processor, which uses a 16-bit instruction and word size, includes eight 16-bit general-purpose registers and a 16-bit program counter (PC). It supports four instruction types: R-type, I-type, J-type, and S-type, and employs separate data and instruction memories with byte-addressable, little-endian format.

The design process involved writing RTL code for each instruction, constructing the data path, and determining the necessary control signals. Boolean equations for control signals were developed, followed by writing and testing the Verilog code. Verification was done using a detailed testbench and simulation tools, ensuring the processor's functional correctness and confirming the successful execution of all instruction types within the pipelined architecture.

Contents

Table Of Figures:

Tables Content:

Theory

RISC and CISC in Computer Organization

CISC (Complex Instruction Set Computer) processors are designed to execute a wide range of instructions, enabling them to perform functions in the most efficient way possible. CISC architecture allows each instruction to perform multiple low-level operations, such as memory storage, memory loading, and arithmetic calculations, all within a single instruction. This design minimizes the number of instructions per program by incorporating complex instructions that can execute several steps or address various modes in one command.[1]

RISC (Reduced Instruction Set Computer) is a type of microprocessor architecture that employs a small, highly-optimized set of instructions. This is in contrast to the highly-specialized set of instructions typically found in other architectures, such as Complex Instruction Set Computing (CISC). RISC is widely regarded as one of the most efficient CPU architecture technologies available today.[2]

The RISC architecture is based on the design principle of simplified instructions that perform fewer operations but can be executed more rapidly. This approach leads to improved performance by allowing for faster execution of instructions. A key feature of RISC is its ability to increase the register set and enhance internal parallelism. This is achieved by increasing the number of parallel threads executed by the CPU and boosting the speed of instruction execution.[2]

An example of RISC architecture is the ARM (Advanced RISC Machine) family, developed by Arm Ltd. ARM processors are prevalent in smartphones, tablets, laptops, gaming consoles, desktops, and an expanding array of other intelligent devices.[2]

Advantages of RISC

RISC processors offer several key advantages due to their design. First, they use a smaller set of simple instructions, which simplifies decoding and allows for quick execution, resulting in faster processing times. This simpler instruction set enables RISC processors to execute instructions more rapidly than their CISC counterparts. Additionally, RISC processors are known for their lower power consumption, making them an ideal choice for portable devices where energy efficiency is crucial.[3]

Multicycle processor

A multi-cycle data path breaks down instructions into separate steps, each taking a single clock cycle. This approach reduces the average instruction time and allows each functional unit to be reused within an instruction as long as it occurs in different clock cycles. Consequently, it minimizes the amount of hardware needed, enhancing efficiency.

The multi-cycle data path operates through the following stages:

- **Fetch Instruction**:
  The instruction stored in memory is fetched into the control unit of the CPU. This is done by

supplying the memory with the address of the instruction, enabling the CPU to retrieve the required data.

- **Decode (Interpret Instruction)**:
  The control unit decodes the fetched instruction to determine the sequence of operations required for execution. This step involves interpreting the instruction's opcode and operands to prepare for the subsequent actions.
- **Execute**:
  The processor performs the operations specified by the instruction. This may involve arithmetic or logical operations, control flow changes, or other computations.
- **Memory Access**:
  If the instruction requires reading from or writing to memory, this step accesses the memory. For example, a load instruction would read data from memory, while a store instruction would write data to memory.
- **Register Write**:
  The results of the executed instruction are written back to the appropriate register. This step ensures that the outcomes of computations or data retrievals are stored for future use.

By breaking instructions into these discrete steps, the multi-cycle data path enhances the efficiency and flexibility of the processor, leading to improved overall performance.[4]

Pre-implementation

The pre-implementation stage was crucial for correctly understanding the problem and laying a solid foundation for the processor design. Initially, the RTL implementation for each instruction was written to determine the required operands for each stage. This step was instrumental in formulating the state diagram and designing the data path.

Following the RTL implementation, the data path was drawn. This visual representation facilitated the writing of the Verilog code, showing how to connect components and manage each instruction using multiplexers to select the correct inputs. This process also helped identify the necessary control signals.

Subsequently, control signals for each instruction were determined, and Boolean expressions for each signal were formulated. This ensured accurate control logic to guide the data path components during execution.

Finally, the state diagram was developed to decide the progression through stages for each instruction. This comprehensive pre-implementation process ensured a clear understanding of the design requirements and streamlined the implementation phase.

Register-Transfer Level (RTL)

The following is how each instruction's RTL implementation was written:

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

    - IF: Instruction ← MEM[PC], pc = pc+2
    - ID: data1 ← Reg(rs1), data2 ← Reg(rs2)
    - EX: ALU_result ← OPCODE (data1, data2)
    - WB: Reg(rd) ← ALU_result

4. ADDI I-Type Reg (Rd) = Reg (Rs1) + Imm 0011 (**signed)**
5. ANDI I-Type Reg (Rd) = Reg (Rs1) + Imm 0100 (**unsigned)**
    - IF: Instruction ← MEM [PC], pc = pc+2
    - ID: data1 ← Reg(rs1), data2 ← Extend(imm11)
    - EX: ALU_result ← OPCODE (data1, data2)
    - WB: Reg(rd) ← ALU_result

6. LW I-Type Reg (Rd) = Mem (Reg (Rs1) + Imm) 0101
7. LBu I-Type Reg (Rd) = Mem (Reg (Rs1) + Imm) 0110 0 (**unsigned**)
8. LBs I-Type Reg (Rd) = Mem (Reg (Rs1) + Imm) 0110 1 (**signed**)
    - IF: Instruction ← MEM [PC], pc = pc+2
    - ID: data1 ← Reg(rs1), data2 ← Extend(imm11)
    - EX: ALU_result ← ADD (data1, data2)
    - M: memo_out←MEM[ALU_result]
    - Wb: Reg(rd)←memo_out

9. SW I-Type Mem (Reg (Rs1) + Imm) = Reg (Rd) 0111

- IF: Instruction ← MEM[PC], pc = pc+2
- ID: data1 ← Reg(rs1), data2 ← Extend(imm11)
- EX: ALU_result ← ADD (data1, data2)
- M: data← rd, address ← ALU_result

10. BGT I-Type if (Reg(Rd) > Reg(Rs1)) 1000 0

Next PC = PC + sign_extended (Imm11)

else PC = PC + 2

11. BGTZ I-Type if (Reg (Rd) > Reg (0)) 1000 1

Next PC = PC + sign_extended (Imm)

else PC = PC + 2

- IF: Instruction ← MEM[PC]
- ID: data1 ← Reg(rd), data2 ← Reg(rs2)
- Negative, zero← subtract (data1, data2)

if (N == 0 && V == 0 && Z=0)

Next_PC = PC + sign_extend (Imm11)

else PC = PC + 2

BGTZ ➔ RS=R0

12. BLT I-Type if (Reg (Rd) < Reg (Rs1)) 1001 0

Next PC = PC + sign_extended (Imm)

else PC = PC + 2

13. BLTZ I-Type if (Reg (Rd) < Reg(R0)) 1001 1

Next PC = PC + sign_extended (Imm)

else PC = PC + 2

- IF: Instruction ← MEM[PC]
- ID: data1 ← Reg(rd), data2 ← Reg(rs2)
- Negative, zero ← subtract (data1, data2)
- if (N == 1 && V == 0 && Z==0)

Next_PC = PC + sign_extend (Imm)

else PC = PC + 2

BLTZ ➔ RS=R0

14. BEQ I-Type if (Reg (Rd) == Reg (Rs1)) 1010 0

Next PC = PC + sign_extended (Imm)

else PC = PC + 2

15. BEQZ I-Type if (Reg (Rd) == Reg (R0)) 1010 1

Next PC = PC + sign_extended (Imm)

else PC = PC + 2

- IF: Instruction ← MEM[PC]
- ID: data1 ← Reg(rd), data2 ← Reg(rs2)
- Negative, Zero← subtract (data1, data2)
- if (Z == 1)

PC = PC + sign_extend (Imm)

else PC = PC + 2

BEQZ ➔ RS=R0

16. BNE I-Type if (Re g(Rd)! =Reg (Rs1)) 1011 0

Next PC = PC + sign_extended (Imm)

else PC = PC + 2

17. BNEZ I-Type if (Reg (Rd)! = Reg (Rs1)) 1011 1

Next PC = PC + sign_extended (Imm)

else PC = PC + 2

- IF: Instruction ← MEM[PC]
- ID: data1 ← Reg(rd), data2 ← Reg(rs2)
- Negative, zero ← subtract (data1, data2)
- if (Z == 0)

PC = PC + sign_extend (Imm)

else PC = PC + 2

BNEZ ➔ RS=R0

18. JMP J-Type Next PC = {PC [15:13], Immediate,0} 1100

- IF: Instruction ← MEM[PC]

- PC ← PC [15:13] || address12|| ‘0’

19. CALL J-Type Next PC = {PC [15:12], Immediate} PC + 2 is saved on r7 1101
    - IF: Instruction ← MEM[PC]
    - PC ← PC [15:13] || address12||0
    - r7 ← PC+2

20. RET J-Type Next PC = R7 1110
    - IF: Instruction ← MEM[PC]
    - ID: Rs =R7
    - Pc=R7

21. Sv S-Type M[rs1] = imm 1111
    - IF: Instruction ← MEM[PC]
    - ID: Rs
    - M: address← Rs, data_in←ext (imm)

Datapath Implementation

Prior to implementing the data path, it was essential to comprehend the format of each instruction. The table below shows the structure for each type of instruction:

Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
R-type	op	op	op	op	rd	rd	rd	Rs1	Rs1	Rs1	Rs2	Rs2	Rs2	U	U	U
I-type	op	op	op	op	m	rd	rd	rd	rs	rs	rs	im m	im m	im m	im m	im m
J-type	op	op	op	op	im m	im m	im m	im m	im m	im m	im m	im m	im m	im m	im m	im m
S-type	op	op	op	op	rs	rs	rs	im m	im m	im m	im m	im m	im m	im m	im m	im m

Table 1, Structure of Types

The table indicates that the opcode occupies 4 bits [15:12] for all instruction types. In R-type instructions, the destination register (rd) is specified by bits [11:9], and the source registers are given by rs1[8:6] and rs2[5:3], with the remaining bits being unused. For I-type instructions, there is a mode bit [11], rd is indicated by bits [10:8], rs by bits [7:5], and the immediate value by bits [4:0]. In J-type instructions, the immediate value spans bits [11:0]. In S-type instructions, the source register is indicated by bits [11:9], and the immediate value is in bits [8:0].

Additionally, it is known that certain instructions require direct access to R7 (e.g., for call and return operations) or R0 (e.g., for instructions like BGTZ and BLTZ). Based on this information, the data path shown below was constructed.

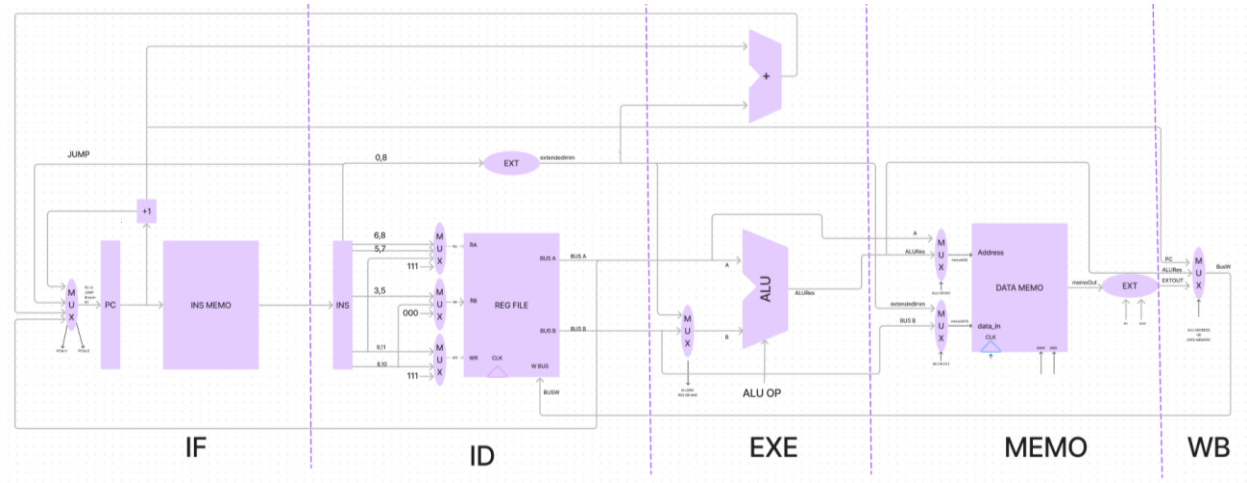


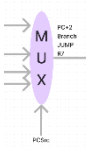
Figure 1, DataPath Implementation

Detailed description of the data path

This datapath is divided into several stages, allowing it to function in a multi-cycle manner: Instruction Fetch (IF), Instruction Decode (ID), Execute (EXE), Memory (MEMO), and Write Back (WB). Each stage is controlled by a specific enable signal (control signals will be discussed later).

The datapath comprises several components, including the PC register, instruction register, instruction memory, data memory, extender, multiplexers (MUXes), and ALU. Each of these components will be discussed in detail below.

PC MUX



This multiplexer (MUX) is designed to select the appropriate PC address based on the opcode. For instance, the default selection is PC+2, but if the opcode indicates a RET instruction, it will select the value from R7. Therefore, a control signal (PCSrc) is necessary to manage this selection.

Figure 2, PC MUX

Program Counter (PC)

The PC register holds the address of the next instruction to be fetched from instruction memory. It is updated at the beginning of each IF cycle to point to the next instruction address.

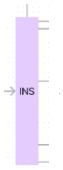
Figure 3, PC

Instruction memory

The Instruction Memory stores the program instructions to be executed by the processor. During the Instruction Fetch (IF) stage, the address held in the PC (Program Counter) register is used to fetch the corresponding instruction from the Instruction Memory. This instruction is then stored in the instruction register and passed to the Instruction Decode (ID) stage. The Instruction Memory ensures that the correct instructions are retrieved efficiently, enabling the processor to execute the program.

Figure 4, INSTRUCTION MEMORY

The Instruction Register

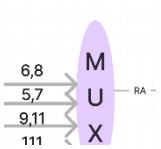


The Instruction Register captures and holds the fetched instruction from the Instruction Memory during the Instruction Fetch (IF) stage. This register ensures that the instruction is stable and available for the next stage, Instruction Decode (ID). It acts as a buffer to prevent any changes in the fetched instruction before it is fully decoded and executed.

Figure 5, INST REG

After fetching the instruction, it will be divided based on the information required and connected via wires to serve as inputs for other components.

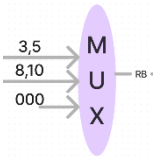
RA MUX



This multiplexer (MUX) determines the address of the first output of the register file (RA). It selects between part [8:6] of the instruction (RS1) for R-type instructions, part [7:5] (RS) for I-type instructions, part [11:9] (RS) for S-type instructions, or the special case value 3'b111 for the RET instruction. Consequently, a 2-bit selection line is required, which will be generated by the control unit based on the opcode.

Figure 6, RA SOURCE MUX

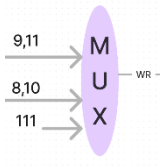
RB MUX



Similarly, the RB MUX determines the address of the second output of the register file. It selects between part [5:3] (RS2) for R-type instructions, part [10:8] (RD) for I-type instructions, and the special case value 3'b000 for the BGTZ, BLTZ, BEQZ, and BNEZ instructions. This MUX also requires a 2-bit selection line, which will be generated by the control unit.

Figure 7, RB Source MUX

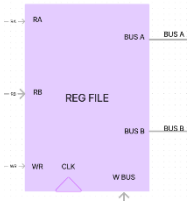
WR MUX



The WR MUX selects the address of the register that will be written to. It chooses between part [11:9] (RD) for R-type instructions, part [10:8] (RD) for I-type instructions, or the special case value 3'b111 for CALL instructions. This MUX also requires a 2-bit selection line, which also will be generated by the control unit.

Figure 8, Reg Destination MUX

Register File



The Register File consists of eight 16-bit general-purpose registers. It has two read buses (A and B) and one write bus, allowing simultaneous reading from two registers and writing to one register. A write enable signal, generated by the control unit, controls whether data is written to the register file during the Write Back (WB) stage.

Figure 9, REG FILE

Immediate Extender

Figure 10, EXTENDER

The Immediate Extender takes part [8:0] of the instruction and processes it based on the opcode. For I-type instructions, it operates on the least significant 5 bits. If the instruction is a logical operation (such as ANDI), it extends the immediate value with zeros from 5 bits to 16 bits. For arithmetic operations, it performs sign extension. For S-type instructions, the extender works with the entire 9 bits, performing sign extension.

This component can be viewed as three separate extenders combined into one: one for I-type unsigned, one for I-type signed, and one for S-type signed extensions.

The extender requires two control signals: one to determine the instruction type and the other to decide whether the extension should be signed or unsigned.

ALU MUX

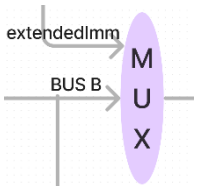


Figure 11, ALU Operands MUX

This multiplexer selects the appropriate ALU operand based on the opcode. It either chooses the extended immediate value (the extender output) or takes the value from bus B (the register file output). It requires a one-bit selection line, which will be generated by the control unit.

ALU

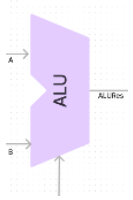


Figure 12, ALU

This ALU takes two inputs and has a 2-bit selection line that determines the operation to perform based on the opcode. It can execute four operations: AND, ADD, SUB, and SUB inverse. Additionally, it generates flags for zero, negative, and overflow conditions.

Memory Address MUX

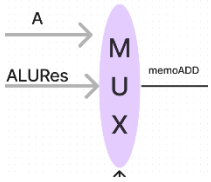


Figure 13, Memory Address MUX

This multiplexer selects the memory address input based on the opcode, choosing either the ALU result or the first output from the register file (A value). It requires a one-bit selection line to perform this operation.

Memory Data MUX

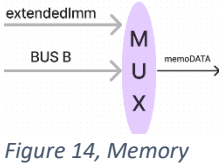


Figure 14, Memory Data MUX

This multiplexer selects the memory data input based on the opcode, choosing either the second output from the register file (Bus B value) or the extended immediate. It requires a one-bit selection line to make this choice.

Data Memory

The data memory module, labeled as "DATA MEMO" in the diagram, is a crucial component in the processor's architecture, responsible for storing and retrieving data required during program execution. This module interacts with several input and output signals to perform its functions efficiently.

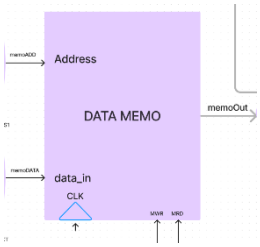


Figure 15, Data Memory

The module receives an address input (memoADD), which specifies the memory location from which data is to be read or to which data is to be written. Alongside the address input, the data input (data_in) line receives data that is to be written into the memory at the specified address.

Control signals play an essential role in determining the memory operations. The Memory Write (MWR) control signal indicates when a write operation should occur. When this signal is active, data from the data_in line is written to the specified address in the memory. Conversely, the Memory Read (MRD) control signal indicates when a read operation should be performed. When active, data from the specified address is read and outputted.

The output from the data memory module is carried by the memory output (memoOut) line. This output provides the data read from the memory, which is then used by other parts of the processor.

Extender 2

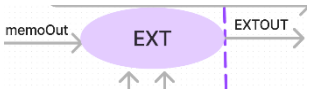


Figure 16, 2nd Extender

This extender module processes the output of the data memory when enabled. It takes the first 8 bits (a byte) and extends them to 16 bits. The extension can be either signed or unsigned, depending on the opcode and a mode bit. The operation of this extender requires two control signals, which are generated by the control unit (CU).

Write Back MUX

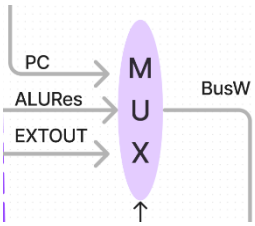


Figure 17, WRITE BACK SRC MUX

This multiplexer selects the appropriate data to be written to the register file. It chooses between the ALU result, the data read from the data memory, or the PC value. The output wire from this multiplexer serves as the input wire to the write bus in the register file, ensuring that the correct data is written based on the current operation. It also needs 2-bit selection line.

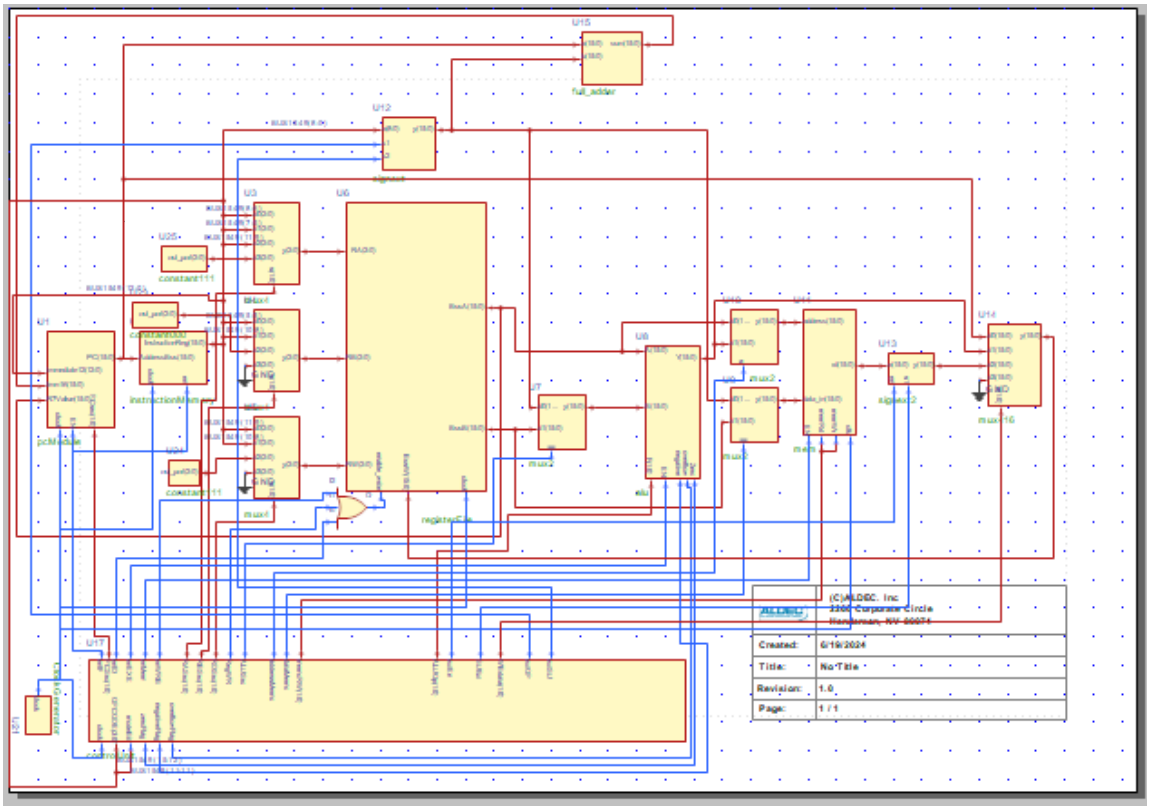


Figure 18:detailed Block Diagram

Control signals design

After designing the data path and understanding the requirements for the control unit, we carefully determined the value of each control signal for every instruction. This detailed process was crucial for developing the Boolean expressions that define the control logic. By analyzing the requirements of each instruction, we specified the exact values needed for all control signals as shown in the tables below.

	Op	RegWr	ExtOp	Ext Signed/Unsigned	ALUSrc	MemRD	MemWR	ALUOp1	ALUOp2	RASrc1	RASrc2
AND	**0**	1	x	x	Rs2	0	0	0	0	0	0
ADD	**1**	1	x	X	Rs2	0	0	0	1	0	0
SUB	**10**	1	x	X	Rs2	0	0	1	0	0	0
ADDI	**11**	1	I-type	Signed	IMM	0	0	0	1	0	1
ANDI	**100**	1	I-type	unsigned	IMM	0	0	0	0	0	1
LW	**101**	1	I-type	Signed	IMM	1	0	0	1	0	1
LBu	**110**	1	I-type	Signed	IMM	1	0	0	1	0	1
LBs	**110**	1	I-type	Signed	IMM	1	0	0	1	0	1
SW	**111**	0	I-type	Signed	IMM	0	1	0	1	0	1

BGT	**1000**	0	I-type	Signed	RS2	0	0	1	1	0	1
BGTZ	**1000**	0	I-type	Signed	RS2	0	0	1	1	0	1
BLT	**1001**	0	I-type	Signed	RS2	0	0	1	1	0	1
BLTZ	**1001**	0	I-type	Signed	RS2	0	0	1	1	0	1
BEQ	**1010**	0	I-type	Signed	RS2	0	0	1	1	0	1
BEQZ	**1010**	0	I-type	Signed	RS2	0	0	1	1	0	1
BNE	**1011**	0	I-type	Signed	RS2	0	0	1	1	0	1
BNEZ	**1011**	0	I-type	Signed	RS2	0	0	1	1	0	1
JMP	**1100**	0	x	X	X	0	0	X	x	X	x
CALL	**1101**	1	x	X	X	0	0	X	x	X	x
RET	**1110**	0	x	X	X	0	0	X	x	1	1
Sv	**1111**	0	S-type	Signed	x	0	1	x	x	1	0

Table 2, Control Signals Design Table, Part(1)

	Op	**RBSrc1**	**RBSrc2**	**RDSrc1**	**RDSrc2**	**Address InMemo**	**Data_in**	**WBMUX1**	**WBMUX2**	**EnExt**	**s/u**
AND	**0**	0	0	0	0	X	X	0	0	X	x
ADD	**1**	0	0	0	0	X	X	0	0	X	x
SUB	**10**	0	0	0	0	X	X	0	0	X	x
ADDI	**11**	X	x	0	1	X	X	0	0	X	x
ANDI	**100**	X	x	0	1	X	X	0	0	X	x
LW	**101**	X	x	0	1	ALU	X	0	1	0	x
LBu	**110**	x	x	0	1	ALU	X	0	1	1	U
LBs	**110**	x	x	0	1	ALU	x	0	1	1	S
SW	**111**	0	1	X	x	ALU	RB	X	x	X	X
BGT	**1000**	0	1	x	x	x	x	X	x	X	X
BGTZ	**1000**	1	0	x	x	x	x	X	x	X	X
BLT	**1001**	0	1	x	x	x	x	X	x	X	X
BLTZ	**1001**	1	0	x	x	x	x	X	x	X	X
BEQ	**1010**	0	1	x	x	x	x	X	x	X	X
BEQZ	**1010**	1	0	x	x	x	x	X	x	X	X
BNE	**1011**	0	1	x	x	x	x	X	x	X	X
BNEZ	**1011**	1	0	x	x	x	x	X	x	X	X
JMP	**1100**	X	x	X	x	X	X	X	x	X	X

CALL	**1101**	X	x	1	0	X	X	1	0	X	X
RET	**1110**	X	x	X	x	X	X	X	x	X	X
Sv	**1111**	x	x	x	x	RA	IMM	x	x	x	x

Table 3, Control Signals Design Table, Part(2)

	op	**z-flag**	**v-flag**	**n-flag**	**PCSrc1**	**PCSrc2**
	0-7	x	x	x	0	0
BGT	**1000**	0	0	0	0	1
	1000	else above line			0	0
BLT	**1001**	0	0	1	0	1
	1001	else above line			0	0
BEQ	**1010**	1	0	0	0	1
	1010	else above line			0	0
BNE	1011	0	0	x	0	1
	1011	else above line			0	0
JUMP	1100	x	x	x	1	0
CALL	1101	x	x	x	1	0
RET	1110	x	x	x	1	1
SV	1111	x	x	x	0	0

Table 4, Control Signals Design Table, Part(3)

Now, we're ready to find the Boolean equations for each control signal.

Register Write Signal

The RegWr signal determines whether data should be written to the register file. It is set to 1 for instructions that require writing to the register file and 0 for those that do not.

RegWr = AND + ADD + SUB + ADDI + ANDI + LW + LB + CALL

0 → don't write on the register file.

1 → write on the register file.

Extension I signals

The EXT OP signal determines whether the opcode is I-type or S-type. If the opcode is S-type, the least significant 9 bits are sign-extended. If it is I-type, the extension depends on whether the opcode represents a logical operation or not.

EXT OP = ~Sv

0 → S-type (sign-extend the least 9 bits).

1 → I-type (check EXT s/u for further extension rules).

EXT s/u Signal: The EXT s/u signal checks if the opcode is logical. If it is ADDI, the least significant 5 bits are unsigned-extended. Otherwise, the least significant 5 bits are sign-extended.

EXT s/u = ~ADDI

0 → Unsigned extension.

1 → Signed extension.

ALU source signal

The ALUSrc signal determines which input is provided to the ALU based on the opcode. It selects between the extended immediate value or the output from the register file.

ALUSrc = ADDI + ANDI + LW + LB +SW

0 → REG FILE OUT

1 → Immediate

Read Write Memory Signals

These signals control whether data memory is read from, written to, or disabled:

memRD = LW + LB

0 → don't read from memory

1→ read from memory

memWR = SW + Sv

0 → don't write to memory

1→ write to memory

ALU operation

This signal dictates the operation that the ALU will execute, which includes AND, ADD, SUB, and SUBINV. It determines whether the ALU performs logical AND, addition, subtraction, or inverted subtraction operations.

ALUOP [0] = ~ (AND + SUB + ANDI)

ALUOP [1] = ~ (AND + ADD + ADDI + ANDI + LW + LB + SW)

00 → And

01 → Add

10 → Subtract

11 → inverted subtract

RA source

This signal determines the address of the first output of the register file.

RASrc [0] = ~ (AND + ADD + SUB + Sv)

RASrc [1] = RET + Sv

00 → ins [8:6]

01 → ins [7:5]

10 → ins [11:9]

11 → 3'b111

RB source

This signal determines the address of the second output of the register file.

RBSrc [0] = SW + BGT + BLT +BEQ +BNE

RBSrc [1] = BGTZ + BLTZ + BEQZ + BNEZ

00 → ins [5:3]

01 → ins [10:8]

10 → 3'b000

RD source

This signal selects the address of the register that will be written to.

RDSrc [0] = ~ (AND + ADD + SUB + CALL)

RDSrc [1] = CALL

00 → ins [11:9]

01 → ins [10:8]

10 → 3'b111

Data Memory input signals

The memory address signal specifies the location in memory where data will be written, while the data input signal specifies the actual data that will be written into that memory location.

AddressInMemo = ~ Sv
0 → A
1 → ALU OUTPUT
DataInMemo = SW
0 → Immediate
1 → Bus B

Write back signal

This signal specifies the data that will be written into the register file.

WB [0] = LW + LB

WB [1] = CALL

00 → ALU output

01 → Data out from memory

10 → PC + 2

Extension II signals

The signal EXTen controls whether the extension is enabled or disabled, while the signal EXTSU determines whether the extension operates in signed or unsigned mode.

EXTen = ~LW

EXTSU = mode bit

PC source

This signal selects the correct value for the Program Counter (PC)

PCSrs [0] = (BGT. ~Z. ~V. ~N) + (BLT. ~Z. ~V. N) + (BEQ. Z. ~V. ~N) + (BNE. ~Z. ~V) + RET

PCSrs [1] = JMP + CALL + RET

00 → PC + 2

01 → Branch target address

10 → Concatenated Jump

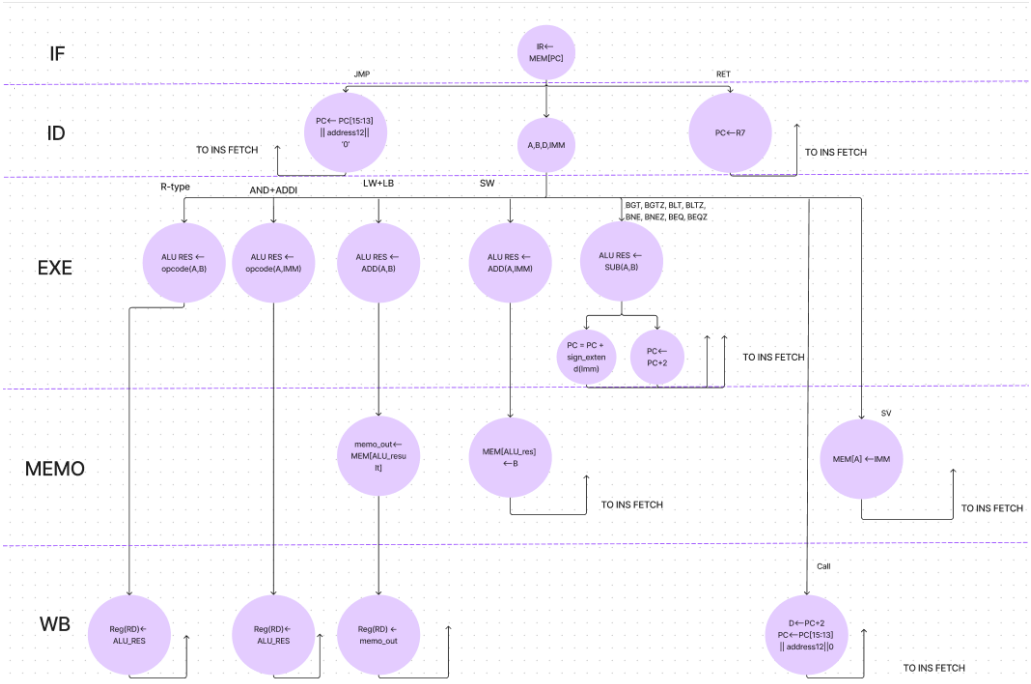
11 → R7 value

State Diagram

Before we start writing the Verilog code for the processor, it's essential to draw the state diagram for each instruction. This diagram provides a clear, visual representation of the processor's control flow, making it easier to understand the sequence of operations and how different instructions traverse through various states. The state diagram serves as a crucial reference throughout the development process, aiding in debugging and verification. By comparing the expected flow of operations as depicted in the diagram with the actual behavior of the Verilog code, we can ensure accuracy and correctness.

Each instruction type, such as R-type, load/store, branch, and jump, follows a specific path through these stages. For instance, R-type instructions perform an ALU operation between two registers in the EXE (execution) stage and write the result back to the register file in the WB (write-back) stage. Load instructions calculate the memory address in the EXE stage, fetch data from memory in the MEMO (memory) stage, and write it to a register in the WB stage. Similarly, store instructions compute the memory address in the EXE stage and write data to memory in the MEMO stage. Branch instructions evaluate conditions in the EXE stage and modify the program counter (PC) based on the result, while jump instructions directly update the PC.

Understanding this helps us determine the next stage for each instruction and identify which components to enable or disable at each step. This structured approach ensures that each instruction is processed correctly and efficiently, maintaining the overall integrity and performance of the processor.



Simulation And Testing:

First of all, we've created a separate code module with its test bench for each block in data path, then we connected all the modules with processor module.

1-Register File module:

```
module registerFile(
    input wire clock,
    input wire enable_write,
    input wire EN,
    input wire [2:0] RA, RB, RW,
    input wire [15:0] BusW,
    output reg [15:0] BusA, BusB,

);

    reg [15:0] registers_array [7:0];


    always @(posedge clock) begin
        if(EN) begin
            BusA <= registers_array[RA];
            BusB <= registers_array[RB];
        end
    end


    always @(posedge clock) begin
        if (enable_write && RW != 3'b000 && EN) begin
            registers_array[RW] <= BusW;
        end
    end

    initial begin
        registers_array[0] <= 16'h0000;
        registers_array[1] <= 16'h0000;
        registers_array[2] <= 16'h0000;
        registers_array[3] <= 16'h0000;
        registers_array[4] <= 16'h0000;
        registers_array[5] <= 16'h0000;
        registers_array[6] <= 16'h0000;
        registers_array[7] <= 16'h0000;
    end

endmodule
```

Figure 19, Reg File Module

Register File module have clock, enable_write, RA,RB,RW, BusW as inputs and BusA, BusB as outputs.

Also there is a registers array that represents the number of registers (8 registers (16 bit) as required).

The EN input controls the ID stage, if the EN is set, then the reg file module will be enabled and we can take values for BusA and BusB through registers array[RA] or [RB].

Also if the enable_write is set which is we have to write at destination register so the registers_array[RW] will take BusW value.

The registers array values initiated with zero.

4-to-1 Multiplexer Module (mux4):

```
module mux4(input [2:0] d0, d1, d2, d3,
    input [1:0] s,
    output reg [2:0] y);

    always @( * )
    case(s)
    2'b00: y <= d0;
    2'b01: y <= d1;
    2'b10: y <= d2;
    2'b11: y <= d3;
    endcase
endmodule
```

Figure 20, 4to1 Mux Module

This code defines a 4-to-1 multiplexer module in Verilog. The module has the following components:

Inputs are d0, d1, d2, d3: 3-bit input data lines,

s: 2-bit select signal.

Output: y: 3-bit output data line.

Functionality:

The output y is determined by the select signal s. Depending on the value of s, one of the input data lines (d0, d1, d2, d3) is assigned to the output y.

The always block is sensitive to changes in s or any of the input data lines, and it uses a case statement to assign the appropriate input to y.

2-to-1 Multiplexer Module (mux2):

```
module mux2(input [15:0] d0, d1,
    input s,
    output [15:0] y);
    assign y = s ? d1 : d0;
endmodule


module tb_mux2to1;


reg [15:0] a;
reg [15:0] b;
reg  sel;
wire [15:0] y;


mux2 mux2to1 (a,b,sel,y);


initial begin


        // Test case 1: sel = 0
    a = 16'h0000; b = 16'hFFFF; sel = 0; #10;  // Expected output: y = 0000
    a = 16'h1234; b = 16'h5678; sel = 0; #10;  // Expected output: y = 1234
    a = 16'hABCD; b = 16'hEF01; sel = 0; #10;  // Expected output: y = ABCD
    a = 16'hFFFF; b = 16'h0000; sel = 0; #10;  // Expected output: y = FFFF

    // Test case 2: sel = 1
    a = 16'h0000; b = 16'hFFFF; sel = 1; #10;  // Expected output: y = FFFF
    a = 16'h1234; b = 16'h5678; sel = 1; #10;  // Expected output: y = 5678
    a = 16'hABCD; b = 16'hEF01; sel = 1; #10;  // Expected output: y = EF01
    a = 16'hFFFF; b = 16'h0000; sel = 1; #10;  // Expected output: y = 0000

    // End simulation
    $finish;
end

endmodule
```

Figure 21, 2to1 Mux Module

This code defines a 2-to-1 multiplexer module in Verilog. The module has the following components:

Inputs:

d0, d1: 16-bit input data lines.
s: 1-bit select signal.
Output:
y: 16-bit output data line.
Functionality:
The output y is determined by the select signal s. If s is 0, y is assigned the value of d0; if s is 1, y is assigned the value of d1.

Memory Module:

```
module mem(
    input clk,
    input memRd, memWr,
    input [15:0] address,
    input [15:0] data_in,
    output reg [15:0] rd,
    input EN
);

    reg [7:0] RAM[255:0];

    initial begin
        RAM[0] <= 8'hFF;
        RAM[1] <= 8'hFF;
        RAM[2] <= 8'h00;
        RAM[3] <= 8'h05;
        RAM[4] <= 8'h20;
        RAM[5] <= 8'h03;
        RAM[6] <= 8'h00;
        RAM[7] <= 8'h0c;
        RAM[8] <= 8'h20;
        RAM[9] <= 8'h67;
        RAM[10] <= 8'hff;
        RAM[11] <= 8'hf7;
        RAM[12] <= 8'h00;
        RAM[13] <= 8'he2;
        RAM[14] <= 8'h20;
        RAM[15] <= 8'h25;
        RAM[16] <= 8'h00;
        RAM[17] <= 8'h64;
        RAM[18] <= 8'h28;
        RAM[19] <= 8'h24;
        RAM[20] <= 8'h00;
        RAM[21] <= 8'ha4;
        RAM[22] <= 8'h28;
        RAM[23] <= 8'h20;
        RAM[24] <= 8'h10;
        RAM[25] <= 8'ha7;
        RAM[26] <= 8'h00;
        RAM[27] <= 8'h0a;
        RAM[28] <= 8'h00;
        RAM[29] <= 8'h64;
        RAM[30] <= 8'h20;

    end

    always @(posedge clk) begin
        if (memWr && EN) begin
            RAM[address] <= data_in[7:0];
            RAM[address + 1] <= data_in[15:8];
        end
    end

```

This code defines a memory module in Verilog with the following components:

Inputs:

clk: Clock signal.
memRd: Control signal to read from memory.
memWr: Control signal to write to memory.
address: 16-bit address for memory access.
data_in: 16-bit data input for writing to memory.
EN: Enable signal to activate the memory module.

Output:

rd: 16-bit data output for reading from memory.
Internal Registers:

RAM: Array of 256 bytes (8 bits each), representing the memory, the Original Size is 2^16, but it's 256 for simplicity.

Initialization:

The memory (RAM) is preloaded with specific values during the initial block.

Functionality:

Write Operation: If memWr and EN are both high on the rising edge of clk, the module writes the lower byte of data_in to RAM[address] and the upper byte to RAM[address + 1].

Figure 22, Memory Module

Read Operation: If memRd and EN are both high on the rising edge of clk, the module reads a 16-bit value from RAM, combining RAM[address + 1] and RAM[address] into rd.

```
        .EN(EN)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // Clock period is 10 time units
    end

    // Test procedure
    initial begin
        // Initial values
        EN = 1;
        memRd = 0;
        memWr = 0;
        address = 0;
        data_in = 0;

        // Display the results
        $monitor("At time %t: EN = %b, memRd = %b, memWr = %b, address = %h, data_in = %h -> rd = %h", $time, EN, memRd, memWr, address, data_in, rd);

        // Wait for a few clock cycles
        #20;

        // Test Case 1: Write data to address 0x0002
        address = 16'h0002;
        data_in = 16'h1234;
        memWr = 1;
        memRd = 0;
        #10; // Wait for one clock cycle
        memWr = 0;

        // Check the memory content at 0x0002 and 0x0003
        #5; // wait for half clock cycle to ensure data is written
        $display("Memory[0x0002] = %h", uut.RAM[16'h0002]);
        $display("Memory[0x0003] = %h", uut.RAM[16'h0003]);

        // Test Case 2: Read data from address 0x0002
        memRd = 1;
        #10; // Wait for one clock cycle
        memRd = 0;

        // Test Case 3: Write data to address 0x0010
        address = 16'h0010;
        data_in = 16'h5678;
        memWr = 1;
        memRd = 0;
        #10; // Wait for one clock cycle
        memWr = 0;

        // Check the memory content at 0x0010 and 0x0011
        #5; // wait for half clock cycle to ensure data is written
        $display("Memory[0x0010] = %h", uut.RAM[16'h0010]);
        $display("Memory[0x0011] = %h", uut.RAM[16'h0011]);

        // Test Case 4: Read data from address 0x0010
        memRd = 1;
        #10; // Wait for one clock cycle
        memRd = 0;

        // End the simulation
        $finish;
    end

endmodule
```

Figure 23, Memory Module TestBench

ALU MODULE:

```
module alu(
    input [15:0] A, B,
    input [1:0] F,
    output reg [15:0] Y,
    output reg Zero, negative, overflow,
     input EN
);


    reg signed [15:0] signed_A, signed_B, signed_Y;

    always @(*) begin

         if (EN)begin

        case (F)
            2'b00: Y = A & B; // AND
            2'b01: Y = A + B; // ADD
            2'b10: Y = A - B; // SUB
            2'b11: Y = B - A; // SUBINV
            default: Y = 16'b0; // default to 0, should not happen
        endcase
          end
    end


    assign Zero = (Y == 16'b0);
    assign negative = Y[15];

     initial begin
         if (EN == 0)begin
             Y = 16'd0;
         end
     end

    // Overflow detection logic
    always @(*) begin
        signed_A = A;
        signed_B = B;
        signed_Y = Y;

        if (F == 2'b01) // ADD
            overflow = (signed_A > 0 && signed_B > 0 && signed_Y < 0) || (signed_A < 0 && signed_B < 0 && signed_Y > 0);
        else if (F == 2'b10) // SUB
            overflow = (signed_A > 0 && signed_B < 0 && signed_Y < 0) || (signed_A < 0 && signed_B > 0 && signed_Y > 0);
        else if (F == 2'b11) // SUBINV
            overflow = (signed_B > 0 && signed_A < 0 && signed_Y < 0) || (signed_B < 0 && signed_A > 0 && signed_Y > 0);
        else
            overflow = 1'b0;
    end

endmodule
```

This code defines an Arithmetic Logic Unit (ALU) module in Verilog with the following components:

Inputs:

A, B: 16-bit input operands.
F: 2-bit function selector.
EN: Enable signal to activate the ALU.

Outputs:

Y: 16-bit result of the ALU operation.
Zero: Flag indicating if the result is zero.
negative: Flag indicating if the result is negative.
overflow: Flag indicating if an overflow occurred.

Internal Registers:

signed_A, signed_B, signed_Y: Signed versions of the input operands and result for overflow detection.

Figure 24, ALU MODULE

Functionality:

Operation Selection: The ALU performs different operations based on the value of F:

2'b00: AND operation (Y = A & B)

2'b01: Addition (Y = A + B)
2'b10: Subtraction (Y = A - B)
2'b11: Subtraction with operands swapped (Y = B - A)

The ALU is enabled only if EN is high.

Zero and Negative Flags:

Zero is set if the result Y is zero.
negative is set if the most significant bit (MSB) of the result Y is 1.

Overflow Detection:

Overflow is detected for addition (F == 2'b01), subtraction (F == 2'b10), and subtraction with operands swapped (F == 2'b11) using signed arithmetic.

Initialization:

If EN is low, Y is initialized to zero.

```
module tb_alu;

    // Testbench signals
    reg [15:0] A, B;
    reg [1:0] F;
    reg EN;
    wire [15:0] Y;
    wire Zero, negative, overflow;

    // Instantiate the ALU module
    alu uut (
        .A(A),
        .B(B),
        .F(F),
        .Y(Y),
        .Zero(Zero),
        .negative(negative),
        .overflow(overflow),
        .EN(EN)
    );

    // Testbench procedure
    initial begin
        // Initialize signals
        A = 0;
        B = 0;
        F = 0;
        EN = 0;

        // Monitor signals
        $monitor("time = %0t, A = %h, B = %h, F = %b, Y = %h, Zero = %b, negative = %b, overflow = %b", $time, A, B, F, Y, Zero, negative, overflow);

        // Apply test vectors
        #10;
        EN = 1;
        A = 16'h0301;
        B = 16'h0101;
        F = 2'b00; // AND
        #10;

        F = 2'b01; // ADD
        #10;

        F = 2'b10; // SUB
        #10;

        F = 2'b11; // SUBINV
        #10;

        A = 16'h0200;
        B = 16'h1000;
        F = 2'b01; // ADD (overflow)
        #10;

        A = 16'h7FFF;
        B = 16'hFFFF;
        F = 2'b10; // SUB (overflow)
        #10;

        A = 16'hFFFF;
        B = 16'h7FFF;
        F = 2'b11; // SUBINV (overflow)
        #10;

        EN = 0;
        #10;

        // Finish simulation
        $finish;
    end

endmodule
```

Figure 25, ALU TestBench

ADDER MODULE:

```
1  module full_adder (
2      input [15:0]a,b,
3      input cin,
4      output [15:0]sum,
5      output carry
6  );
7
8      assign {carry,sum} = a + b + cin;
9
10
11 endmodule
```

The full_adder Verilog module implements a 16-bit full adder circuit. It computes the sum of two 16-bit inputs (a and b) along with a carry-in (cin), producing a 16-bit sum (sum) and a carry-out (carry). The addition operation is straightforward, handling of carry propagation in arithmetic operations.

Figure 26, Adder Module

Extender (11 bit or 8 bit) Module:

```
1  module ext(
2      input [11:0] a,
3      input [1:0] s1,
4      input s2,
5      output reg [15:0] y);
6
7      always @(*) begin
8          if (s1 == 2'b01) begin
9              if (s2)
10                 y = {{11{a[4]}}, a[4:0]};
11             else
12                 y = {{11{1'b0}}, a[4:0]};
13         end else begin
14             y = {{8{a[7]}}, a[7:0]};
15         end
16     end
17 endmodule
18
```

Figure 27, Extender Module

The ext module efficiently extends a 12-bit input vector to a 16-bit output vector based on the control signals s1 and s2. It offers flexibility in extension methods, providing either sign-extension or zero-extension depending on the configuration of s2. S1 controls the width of extension, either 11 bit or 8 bits.

Extender (8 bit) Module:

```
module signext2(
    input [15:0] a,
    input s1,
    input en,
    output reg [15:0] y);

    always @(*) begin
        if (en) begin
            if (s1)
                y = {{8{a[7]}}, a[7:0]};
            else
                y = {{8{1'b0}}, a[7:0]};
        end
            else
                begin
            y = a[15:0];
        end
    end
endmodule
```

The width of memory output is 16 bit with 0 extension, so it's actually 8 bit, when signext2 module called, it gives sign extension for the output based on input 7th bit sign, either 1 or zero.

Figure 28, 2nd Extender Module

Instruction Memory:

```
`include "constants.v"
module instructionMemory(clock, AddressBus, InstructionReg);

    input wire clock;
    input wire [15:0] AddressBus;
    output reg [15:0] InstructionReg;

    // instruction memory
    reg [7:0] instruction_memory [255:0];

    assign InstructionReg = {instruction_memory[AddressBus[15:0]+1],instruction_memory[AddressBus[15:0]]};

    initial begin
      {instruction_memory[1],instruction_memory[0]} <= {16'b0};
     {instruction_memory[3],instruction_memory[2]} <= {ADDI,1'b0,R1,R1,5'd13} ;  // r1=   13
     {instruction_memory[5],instruction_memory[4]} <=  {ANDI,1'b0,R2,R1,5'd4} ; //r2 = 4
     //
     {instruction_memory[7],instruction_memory[6]} <= {ADD,R3,R1,R2,3'b000};   //r3 = 17
     {instruction_memory[9],instruction_memory[8]} <= {SUB,R4,R2,R1,3'b000};     //r4 = -9

    {instruction_memory[11],instruction_memory[10]} <= {AND,R5,R1,R2,3'b000} ;    //R5 = 4
     {instruction_memory[13],instruction_memory[12]} <= {LW,1'b0,R6,R5,5'b0};       //R6 = 5
     {instruction_memory[15],instruction_memory[14]} <= {SW,1'b0, R4, R2, 5'd0} ;   //MEM4 = -9
     {instruction_memory[17],instruction_memory[16]} <= {LB,u, R7 , R2,5'b0}    ;   //R7 = 247
     {instruction_memory[19],instruction_memory[18]} <= {LB,s, R1 , R2,5'b0} ;   //r1 = -9
     {instruction_memory[21],instruction_memory[20]} <= {BGT,1'b0, R3, R2, 5'd4} ;
     {instruction_memory[23],instruction_memory[22]} <= {BEQ,1'b0, R5, R2, 5'd6} ;
     {instruction_memory[25],instruction_memory[24]} <= {BGT,1'b1, R3, R5, 5'd4} ;
     {instruction_memory[27],instruction_memory[26]} <= {BGT,1'b0, R3, R2, 5'd14} ;
     {instruction_memory[29],instruction_memory[28]} <= {BGT,1'b0, R3, R2, 5'b11110} ;
     {instruction_memory[31],instruction_memory[30]} <= {BEQ,1'b0, R1, R2, 5'd6} ;  /
     {instruction_memory[33],instruction_memory[32]} <= {BLT,1'b0, R6, R5, 5'd4} ;
     {instruction_memory[35],instruction_memory[34]} <= {BEQ,Z, R1, R0, 5'd4} ;
     {instruction_memory[37],instruction_memory[36]} <= {BLT,1'b0, R1, R2, 5'b11110} ;
     {instruction_memory[39],instruction_memory[38]} <= {BNE,1'b0, R1, R2, 5'd4} ;
     {instruction_memory[41],instruction_memory[40]} <= {JMP,12'd22} ;
     {instruction_memory[43],instruction_memory[42]} <= {BNE,Z, R1, R2, 5'b11110} ;
     {instruction_memory[45],instruction_memory[44]} <= {CALL, 12'd24 } ;
     {instruction_memory[47],instruction_memory[46]} <= {Sv, R2, 9'd30} ;  //mem[4] = 30
     {instruction_memory[49],instruction_memory[48]} <= {RET, 12'd0} ;



    end
```

Instruction Memory Holds the Instructions inside it, the memory is byte addressable so every memory slot is 8 bit, the actual size is 2^16 but here It is 256 for simplicity.
The module uses AddressBus to index into instruction_memory to fetch instructions.

Figure 29, INSTRUCTION MEMORY

The assignment InstructionReg = {instruction_memory[AddressBus[15:0]+1], instruction_memory[AddressBus[15:0]]};

constructs a 16-bit instruction by concatenating two 8-bit values from instruction_memory.
The concatenation {instruction_memory[AddressBus[15:0]+1], instruction_memory[AddressBus[15:0]]} forms the 16-bit instruction, where instruction_memory[AddressBus[15:0]+1] occupies the higher 8 bits (most significant byte) and instruction_memory[AddressBus[15:0]] occupies the lower 8 bits (least significant byte).

PARAMETERS:

```
parameter
    // opcode
    // R-type
    AND = 4'b0000,
    ADD = 4'b0001,
    SUB = 4'b0010,

    // I-type
    ADDI = 4'b0011,
    ANDI = 4'b0100,

    LW = 4'b0101,
    LB = 4'b0110,

    u = 1'b0,
    s = 1'b1,

    //LBu = 5'b01100,
    //LBs = 5'b01101,

    SW = 4'b0111,

    BGT = 5'b1000,
    //BGTZ = 5'b10001,

    BLT = 5'b1001,
    //BLTZ = 5'b10011,

    BEQ = 5'b1010,
    //BEQZ = 5'b10101,

    BNE = 5'b1011,
    //BNEZ = 5'b10111,

    Z=1'b1,

    // J-type
    JMP = 4'b1100,
    CALL = 4'b1101,
    RET = 4'b1110,

    // S-type
    Sv = 4'b1111,

    // 8 registers
    R0 = 3'd0, // zero register
    R1 = 3'd1,
    R2 = 3'd2,
    R3 = 3'd3,
    R4 = 3'd4,
    R5 = 3'd5,
    R6 = 3'd6,
    R7 = 3'd7 ;
```

Parameters are for simplicity in writing the code.

Figure 30, Parameters

MUX 4-1 (16 BIT):

```
module mux416(
    input [15:0] d0, d1, d2, d3,
    input [1:0] s,
    output reg [15:0] y);

    always @( * )
    case(s)
    2'b00: y <= d0;
    2'b01: y <= d1;
    2'b10: y <= d2;
    2'b11: y <= d3;
    endcase
endmodule
```

Mux for 16 bit inputs.

Figure 31, 16 bit 4-1 MUX

PC MODULE:

```
// 0: Pc = Pc + 2
// 1: PC =  PC + sign_extended (Imm16)
// 2: PC = {PC[15:13], Immediate13 }
// 3: PC =   R7


module PCmodule(clock, PC, PCsrc, immediate12, Imm16, R7Value, EN);

    input wire clock;

    input wire [1:0] PCsrc;
    input wire [11:0] immediate12;
    input wire [15:0] R7Value;
    input wire signed [15:0] Imm16;
    input wire EN;

    // PC Output
    output reg [15:0] PC;


    // To store assignments
    wire [15:0] NextPC;
    wire [15:0] JumpConc;


    // PC + 2
    assign NextPC = PC + 16'd2;
    // Concatinate PC and immediate
    assign JumpConc = {{PC[15:13], immediate12, 1'b0}};

    initial begin
        PC <= 16'd0;
    end


    always @(posedge clock) begin
        #1
        if(EN) begin
            case (PCsrc)
                2'b00: begin
                    // PC = PC + 2
                    PC = NextPC;
                end
                2'b01:  begin

                    PC = Imm16 + PC;
                end
                2'b10: begin
                   PC = JumpConc;
                end
                2'b11: begin

                    PC = R7Value + 2;
                end
            endcase
        end
    end
endmodule
```

Figure 32, PC Module

The PCmodule provides a flexible and controlled way to update the Program Counter (PC) in a digital system.
NextPC: Holds the value of PC + 2.
JumpConc: Holds the concatenated value of the 3 bit from current PC and the 12-bit immediate value.

Initial Setup:

The PC is initialized to 0 at the start of the simulation.

PC Update Logic:

The always block is triggered on the positive edge of the clock.
If EN is set, the PC is updated based on the value of PCsrc:
2'b00: Increment PC by 2 (NextPC).
2'b01: Add Imm16 to the current PC.
2'b10: Concatenate PC and immediate12 to form the new PC value (JumpConc).
2'b11: Set PC to R7Value + 2.

```
module tb_PCmodule;

    // Testbench signals
    reg clock;
    reg [1:0] PCsrc;
    reg [11:0] immediate12;
    reg [15:0] R7Value;
    reg signed [15:0] Imm16;
    reg EN;
    wire [15:0] PC;

    // Instantiate the PCmodule
    PCmodule uut (
        .clock(clock),
        .PC(PC),
        .PCsrc(PCsrc),
        .immediate12(immediate12),
        .Imm16(Imm16),
        .R7Value(R7Value),
        .EN(EN)
    );

    // Clock generation
    always begin
        #5 clock = ~clock;
    end

    initial begin
        // Initialize signals
        clock = 0;
        PCsrc = 0;
        immediate12 = 0;
        R7Value = 0;
        Imm16 = 0;
        EN = 0;

        // Monitor signals
        $monitor("time = %0t, PCsrc = %b, immediate12 = %h, Imm16 = %h, R7Value = %h, PC = %h", $time, PCsrc, immediate12, Imm16, R7Value, PC);

        // Reset PC
        #10;
        EN = 1;

        // Test PC = PC + 2
        PCsrc = 2'b00;
        #10;

        // Test PC = PC + Imm16
        Imm16 = 16'h0010;
        PCsrc = 2'b01;
        #10;

        // Test PC = {PC[15:13], immediate12}
        immediate12 = 12'h7FF;
        PCsrc = 2'b10;
        #10;

        // Test PC = R7 + 2
        R7Value = 16'h1234;
        PCsrc = 2'b11;
        #10;

        // Disable the module
        EN = 0;
        #10;

        // Finish simulation
        $finish;
    end

endmodule
```

Figure 33, PC Test Bench

Clock Generator Module:

Clock every 5ns.

```
module ClockGenerator (clock);

    initial begin
        $display("(%0t) > initializing clock generator ...", $time);
    end

    output reg clock=0;

    always #5 begin
        clock = ~clock;
    end

endmodule
```

Figure 34, Clock Generator Module

Control Signal Unit:

```
`include "constants.v"

module controlUnit (
    input wire clock,
    input wire [3:0] OPCODE,
    input wire modeBit,
    input wire zeroFlag,
    input wire negativeFlag,
    input wire overflowFlag,

    output reg enIF,  // enable instruction fetch
    output reg enID,  // enable instruction decode
    output reg enEXE,  // enable execute
    output reg enMem,  // enable memory
    output reg enWRB,  // enable write back

    output reg [1:0] PCSrc,
    output reg [1:0] RASrc,
    output reg [1:0] RBSrc,
    output reg [1:0] RDSrc,
    output reg RegWR,
    output reg ALUSrc,
    output reg [1:0] ALUOp,
    output reg addressMemo,
    output reg dataMemo,
    output reg [1:0] memoRW,
    output reg extEn,
    output reg SUExt,
    output reg [1:0] WBdata,
    output reg extOP,
    output reg extSU
);


    // Define stages
    `define IF 3'b000
    `define ID 3'b001
    `define EXE 3'b010
    `define MEM 3'b011
    `define WRB 3'b100
    `define INIT 3'b101

    reg [2:0] currentStage = `INIT;
    reg [2:0] nextStage = `IF;


    always @(posedge clock) begin

        case (currentStage)
            `INIT: begin
                enIF = 1'b0;
                //nextStage <= `IF;
                        currentStage <= `IF;
            end
```

Figure 35, Control Unit 1

The controlUnit module in Verilog manages the stages and control signals for a processor. It controls when each stage of instruction processing (fetch, decode, execute, memory access, write back) is enabled based on the current stage, opcode, and condition flags.

Inputs:

clock: Synchronizes the stages.

OPCODE: Determines the operation to be performed.

modeBit, zeroFlag, negativeFlag, overflowFlag: Additional control signals and flags.

Outputs:

Enable signals for each stage: enIF, enID, enEXE, enMem, enWRB.

Control signals for PC source (PCSrc), register sources and destinations (RASrc, RBSrc, RDSrc), ALU operations (ALUSrc, ALUOp), memory operations (addressMemo, dataMemo, memoRW), and extensions (extEn, SUExt, WBdata, extOP, extSU).

Functionality

Initial Stage:

Disables instruction fetch (enIF).

Moves to the instruction fetch (IF) stage.

Instruction Fetch (IF) Stage:

Disables previous stages.

Sets memory read/write and register write signals to default.

Enables instruction fetch (enIF).

Determines the next PC source based on OPCODE and condition flags.

Moves to the instruction decode (ID) stage.

This module ensures the sequential and conditional progression of instructions through the pipeline, enabling efficient processing and control flow management in a processor.

```
55
56          `IF: begin
57                $display("Stage 1");
58              // Disable all previous stages leading up to IF
59              enID = 1'b0;
60              enEXE = 1'b0;
61              enMem = 1'b0;
62              enWRB = 1'b0;
63              memoRW = 2'b00;
64              RegWR = 1'b0;
65
66              // Enable IF
67              enIF = 1'b1;
68
69              // Determine PCSrc
70              PCSrc[1] = (OPCODE == JMP) || (OPCODE == CALL) || (OPCODE == RET);
71              PCSrc[0] = ((OPCODE == BGT) && ~(zeroFlag) && ~(overflowFlag) && ~(negativeFlag)) ||
72                         ((OPCODE == BLT) && ~(zeroFlag) && ~(overflowFlag) && (negativeFlag)) ||
73                         ((OPCODE == BEQ) && (zeroFlag) && ~(overflowFlag) && ~(negativeFlag)) ||
74                         ((OPCODE == BNE) && ~(zeroFlag) && ~(overflowFlag)) ||
75                         (OPCODE == RET);
76
77              // Determine next stage
78             // nextStage <= `ID;
79               currentStage <= `ID;
80          end
81
82          `ID: begin
83            enIF = 1'b0;
84            enID = 1'b1;
85
86            $display("Stage 2");
87
88            // Next stage is determined by opcode
89            if (OPCODE == JMP || OPCODE == RET) begin
90                //   nextStage <= `IF;
91                     currentStage <= `IF;
92            end
93
94            else if (OPCODE == CALL)begin
95                 //nextStage <= `WRB;
96                 currentStage <= `WRB;
97            end
98
99            else if (OPCODE == Sv)begin
100               //nextStage <= `MEM;
101               currentStage <= `MEM;
102           end
103
104           else begin
105               //nextStage <= `EXE;
106               currentStage <= `EXE;
107           end
108
109           // Determine RASrc
```

Figure 36, Control Unit 2

Instruction Decode (ID) Stage:
Enables instruction decode (enID).
Disables instruction fetch (enIF).
Determines the next stage based on OPCODE:
IF for JMP and RET.
WRB for CALL.
MEM for Sv.
EXE for all other instructions.
Sets RASrc, RBSrc, RDSrc based on OPCODE.
Enables register write (RegWR) for arithmetic and load operations.
Configures extension operations (extOP, extSU).

```
109         // Determine RASrc
110         RASrc[1] = (OPCODE == RET) || (OPCODE == Sv);
111         RASrc[0] = ~((OPCODE == AND)|| (OPCODE == ADD)||(OPCODE == SUB)||(OPCODE == Sv));
112
113         // Determine RBSrc
114         RBSrc[1] = ((OPCODE == BGT) && (modeBit == 1)) || ((OPCODE == BLT) && (modeBit == 1)) || ((OPCODE == BEQ) && (modeBit == 1)) || ((OPCODE == BNE) && (modeBit == 1));
115         RBSrc[0] = (OPCODE == SW) ||((OPCODE == BGT) && (modeBit == 0)) || ((OPCODE == BLT) && (modeBit == 0)) || ((OPCODE == BEQ) && (modeBit == 0)) || ((OPCODE == BNE) && (modeBit == 0));
116
117          // Determine RDSrc
118          RDSrc[1] = (OPCODE == CALL);
119          RDSrc[0] = ~ ((OPCODE == ADD)|| (OPCODE == AND) ||(OPCODE == SUB) || (OPCODE == CALL));
120
121          //Determine RegWR
122          RegWR = (OPCODE == AND) ||  (OPCODE == ADD) || (OPCODE == SUB) || (OPCODE == ADDI) ||(OPCODE == ANDI) || (OPCODE == LW) || (OPCODE == LB) || (OPCODE == CALL);
123
124          extOP = ~(OPCODE == Sv);
125          extSU = ~(OPCODE == ANDI);
126
127         end
128
129         `EXE : begin
130          enID = 1'b0;
131          enEXE = 1'b1;
132
133
134          if(OPCODE == LW || OPCODE == LB || OPCODE == SW ) begin
135               currentStage = `MEM   ;
136
137          end
138
139          else if(OPCODE == BGT || OPCODE == BLT || OPCODE == BEQ || OPCODE == BNE ) begin
140                  currentStage <= `IF;
141         end
142
143         else begin
144              currentStage <= `WRB;
145         end
146
147         ALUSrc = (OPCODE == ADDI)||(OPCODE == ANDI)|| (OPCODE == LW) || (OPCODE == LB)||(OPCODE == SW);
148         ALUOp[1] = ~((OPCODE == AND) ||(OPCODE == ADD) ||(OPCODE == ADDI)||(OPCODE == ANDI)|| (OPCODE == LW) || (OPCODE == LB)||(OPCODE == SW));
149         ALUOp[0] = ~((OPCODE == AND) ||(OPCODE == SUB) ||(OPCODE == ANDI));
150
151         end
152
153         3'b011 : begin
154              enEXE = 1'b0;
155              enID = 1'b0;
156              enMem = 1'b1;
157
158              if(OPCODE == SW || OPCODE == Sv )begin
159
160                   currentStage <= `IF;
161              end
162              else begin
163                   currentStage <= `WRB;
```

Figure 37, Control Unit 3

Execute (EXE) Stage:
Enables execute (enEXE).
Determines the next stage based on OPCODE.
Configures ALU control signals (ALUSrc, ALUOp).

```
                    currentStage <= `WRB;
                end

            addressMemo =  ~(OPCODE == Sv);
            dataMemo = (OPCODE == SW);
            memoRW[1] = (OPCODE == LW )||(OPCODE == LB);
            memoRW[0] = (OPCODE == SW )||(OPCODE == Sv);
            extEn = ~(OPCODE == LW);
            SUExt = modeBit;
            end

            `WRB: begin
                enEXE = 1'b0;
                enID = 1'b0;
                enMem = 1'b0;
                enWRB = 1'b1;
                    currentStage <= `IF;

                WBdata[1] = (OPCODE == CALL);
                WBdata[0] = (OPCODE == LW ) || (OPCODE == LB);

                end
                default: $display("Hey default");
                endcase
                end
            endmodule


module tb_controlUnit;

    reg clock;
    reg [3:0] OPCODE;
    reg modeBit;
    reg zeroFlag;
    reg negativeFlag;
    reg overflowFlag;

    wire enIF;
    wire enID;
    wire enEXE;
    wire enMem;
    wire enWRB;
    wire [1:0] PCSrc;
    wire [1:0] RASrc;
    wire [1:0] RBSrc;
    wire [1:0] RDSrc;
    wire RegWR;
    wire ALUSrc;
    wire [1:0] ALUOp;
    wire addressMemo;
    wire dataMemo;
    wire [1:0] memoRW;
    wire extEn;
    wire SUExt;
    wire [1:0] WBdata;
```

Figure 38, Control Unit 4

```
    .ALUSrc(ALUSrc),
    .ALUOp(ALUOp),
    .addressMemo(addressMemo),
    .dataMemo(dataMemo),
    .memoRW(memoRW),
    .extEn(extEn),
    .SUExt(SUExt),
    .WBdata(WBdata),
    .extOP(extOP),
    .extSU(extSU)
);

// Clock generation
initial begin
    clock = 0;
    forever #5 clock = ~clock; // Clock period is 10 time units
end

// Test procedure
initial begin
    // Initial values
    OPCODE = 4'b0000;
    modeBit = 0;
    zeroFlag = 0;
    negativeFlag = 0;
    overflowFlag = 0;

    // Display the results
    //$monitor("At time %t: OPCODE = %0, modeBit = %0, zeroFlag = %0, negativeFlag = %0, overflowFlag = %0", $time, OPCODE, modeBit, zeroFlag, negativeFlag, overflowFlag);
    //$monitor("enIF = %0, enID = %0, enEXE = %0, enMem = %0, enWRB = %0", enIF, enID, enEXE, enMem, enWRB);
    //$monitor("PCSrc = %0, RASrc = %0, RBSrc = %0, RDSrc = %0, RegWR = %0, ALUSrc = %0, ALUOp = %0", PCSrc, RASrc, RBSrc, RDSrc, RegWR, ALUSrc, ALUOp);
    //$monitor("addressMemo = %0, dataMemo = %0, memoRW = %0, extEn = %0, SUExt = %0, WBdata = %0, extOP = %0, extSU = %0", addressMemo, dataMemo, memoRW, extEn, SUExt, WBdata, extOP, extSU);

    // Wait for a few clock cycles
    //#20;

    //Test Case 1: JMP
    //OPCODE = JMP;
    // #40; // Wait for a few clock cycles

    // Test Case 2: CALL
    //OPCODE = CALL;
    //#70; // Wait for a few clock cycles

    // Test Case 3: RET
    //OPCODE = RET;
    //#70; // Wait for a few clock cycles

    // Test Case 4: BGT
    //OPCODE = BGT;
    //zeroFlag = 1;
   // negativeFlag = 0;
    //overflowFlag = 0;
    //modeBit = 0;
    //   #70; // Wait for a few clock cycles

    // Test Case 5: BLT
   // OPCODE = BLT;
   // zeroFlag = 0;
   // negativeFlag = 1;
   // overflowFlag = 0;
   // #20; // Wait for a few clock cycles

    // Test Case 6: BEQ
    //OPCODE = BEQ;
    //zeroFlag = 1;
    //negativeFlag = 0;
    //overflowFlag = 0;
    //#20; // Wait for a few clock cycles

    // Test Case 7: BNE
    //OPCODE = BNE;
    //zeroFlag = 0;
    //negativeFlag = 0;
    //overflowFlag = 0;
    //#20; // Wait for a few clock cycles
```

Figure 39, Control Unit TestBench

Memory Access (MEM) Stage:
Enables memory access (enMem).
Determines the next stage based on OPCODE.
Configures memory control signals (addressMemo, dataMemo, memoRW).

Write Back (WRB) Stage:
Enables write back (enWRB).
Determines the next stage to be Instruction Fetch (IF).
Configures write back data (WBdata).

Processor Module:

```
`include "constants.v"
module processor();
    initial begin
        #0
        $display("(%0t) > initializing processor ...", $time);

        #1000 $finish;
    end

    wire clock;
    wire enIF, enID, enE, enMem, enWRB;

    //control unit outputs
    wire[1:0] PCSrc, RASrc, RBSrc, RDSrc,ALUOP,memoRW,WBdata;
    wire regWR, ALUSrc, addressMemo, dataMemo, extEn, SUExt,extOP,extSU;

    // instruction memory wires
    wire [15:0] PC;
    reg [15:0] instruction; // output IF, input to other modules


    //instruction parts
    wire [3:0] OpCode;
    wire mode;

    wire [2:0] p35,p57,p68,p810,p911;
    wire [8:0] imm9;
    wire [11:0] imm12;


    //reg file wires
    wire [2:0] RA, RB, RW;
    wire [15:0] BusA, BusB, BusW;

    //ALU wires
    wire [15:0] A, B;
    wire [15:0] ALURes;
    wire z,v,n;

    //MEMORY WIRES
    wire [15:0] memoADD, memoDATA, memoOut;

    //WB wires
    wire [15:0] EXTOUT;
    wire writebackData;

    //Assignment
    assign OpCode = instruction[15:12];
    assign mode = instruction[11];

    //MUXes inputs
    assign p35 = instruction[5:3];
    assign p57 = instruction[7:5];
    assign p68 = instruction[8:6];
    assign p810 = instruction[10:8];
```

Figure 40, Processor 1

```
    //register file
    wire EnReg;
    assign EnReg = (enID || enWRB);

     assign RA = (RASrc == 2'b00) ? p68 :
                 (RASrc == 2'b01) ? p57 :
                 (RASrc == 2'b10) ? p911 :
                    3'b111;

    assign RB = (RBSrc == 2'b00) ? p35 :
                (RBSrc == 2'b01) ? p810 :
                (RBSrc == 2'b10) ? 3'b000 :
                    3'b000;

    assign RW = (RDSrc == 2'b00) ? p911 :
                (RDSrc == 2'b01) ? p810 :
                (RDSrc == 2'b10) ? 3'b111 :
                    3'b000;


    //ALU

    wire [15:0] extendedImm;

    assign extendedImm = (extOP == 1'b1) ?
                         (extSU ? {{11{imm9[4]}}, imm9[4:0]} : {{11{1'b0}}, imm9[4:0]}) :
                         {{7{imm9[8]}}, imm9[8:0]};
    assign A = BusA;
    assign B = (ALUSrc == 1'b0) ? BusB : extendedImm;


    //MEMO
    mux2 m(A, ALURes,addressMemo, memoADD );
    //mux2 d(extendedImm,BusB,dataMemo ,memoDATA);
    assign memoDATA = dataMemo ? BusB : extendedImm;


    ////////////////////////////////////////////////////////////////////

    ClockGenerator Generator(clock);


    controlUnit cu( clock, OpCode, mode, z,n,v, enIF, enID, enE, enMem, enWRB, PCSrc,
    RASrc, RBSrc, RDSrc, regWR, ALUSrc, ALUOP, addressMemo, dataMemo, memoRW,
    extEn, SUExt, WBdata, extOP, extSU );
```

Figure 41, Processor 2

The processor is initialized at time 0 and is set to run until 1000 time units using $finish.

Clock Generator:
Module: ClockGenerator
Purpose: Generates the clock signal that synchronizes all operations in the processor.
Control Unit:
Module: controlUnit
Inputs: clock, OPCODE, modeBit, zeroFlag, negativeFlag, overflowFlag
Outputs: enIF, enID, enEXE, enMem, enWRB, PCSrc, RASrc, RBSrc, RDSrc, RegWR, ALUSrc, ALUOp, addressMemo, dataMemo, memoRW, extEn, SUExt, WBdata, extOP, extSU
Function: Determines the control signals for different stages based on the opcode and flags. Manages state transitions between stages (IF, ID, EXE, MEM, WRB).
Program Counter (PC) Module:
Module: PCmodule
Inputs: clock, PCsrc, immediate12, Imm16, R7Value, EN
Output: PC
Function: Manages the program counter value, allowing for normal increments and jumps based on control signals.
Instruction Memory:
Module: instructionMemory
Inputs: clock, AddressBus
Output: InstructionReg
Function: Stores and provides instructions based on the current PC value.
Register File:
Module: registerFile
Inputs: clock, RegWrite, EnReg, RA, RB, RW, BusW
Outputs: BusA, BusB
Function: Contains the general-purpose registers. Supports reading and writing based on control signals.
ALU (Arithmetic Logic Unit):
Module: alu
Inputs: A, B, F, EN
Outputs: Y, Zero, Negative, Overflow
Function: Performs arithmetic and logic operations based on the ALU operation code.
Data Memory:
Module: mem
Inputs: clock, memoRW, memoADD, memoDATA
Output: memoOut
Function: Handles read and write operations to memory.
Sign Extension Unit:
Module: ext
Inputs: a, s1, s2
Output: y
Function: Extends immediate values to the required bit-width for operations.
Multiplexers (MUXes):
Function: Select appropriate inputs for various stages based on control signals.
Example: mux2, mux416
Wire Connections and Assignments
Instruction Parts: OPCODE, mode, immediate values
Control Signal Assignments:

```
74      assign RW = (RDSrc == 2'b00) ? p911 :
75                  (RDSrc == 2'b01) ? p810 :
76                  (RDSrc == 2'b10) ? 3'b111 :
77                     3'b000;
78
79
80      //ALU
81
82      wire [15:0] extendedImm;
83
84      assign extendedImm = (extOP == 1'b1) ?
85                           (extSU ? {{11{imm9[4]}}, imm9[4:0]} : {{11{1'b0}}, imm9[4:0]}) :
86                           {{7{imm9[8]}}, imm9[8:0]};
87      assign A = BusA;
88      assign B = (ALUSrc == 1'b0) ? BusB : extendedImm;
89
90
91      //MEMO
92      mux2 m(A, ALURes,addressMemo, memoADD );
93      //mux2 d(extendedImm,BusB,dataMemo ,memoDATA);
94      assign memoDATA = dataMemo ? BusB : extendedImm;
95
96
97      ////////////////////////////////////////////////////////////////////
98
99      ClockGenerator Generator(clock);
100
101
102     controlUnit cu( clock, OpCode, mode, z,n,v, enIF, enID, enE, enMem, enWRB, PCSrc,
103     RASrc, RBSrc, RDSrc, regWR, ALUSrc, ALUOP, addressMemo, dataMemo, memoRW,
104     extEn, SUExt, WBdata, extOP, extSU );
105
106
107     instructionMemory insMem(enIF, PC, instruction);
108     PCmodule pcMod(clock, PC, PCSrc, imm12, extendedImm, A, enIF);
109
110     registerFile regfile( clock, regWR,EnReg ,RA, RB, RW, BusW, BusA, BusB);
111
112     alu op( A, B, ALUOP, ALURes, z, n, v, enE);
113
114     mem memo(clock, memoRW[1], memoRW[0],memoADD, memoDATA,memoOut,enMem);
115
116
117
118     //signext2 ext(memoOUT, SUExt, extEn,EXTOUT);
119     assign EXTOUT = extEn ? (SUExt ? {{8{memoOut[7]}}, memoOut[7:0]} : {{8{1'b0}}, memoOut[7:0]}) : memoOut;
120
121
122     mux416 dataWB(ALURes,EXTOUT,PC,16'd0,WBdata,BusW);
123
124 endmodule
125
126
```

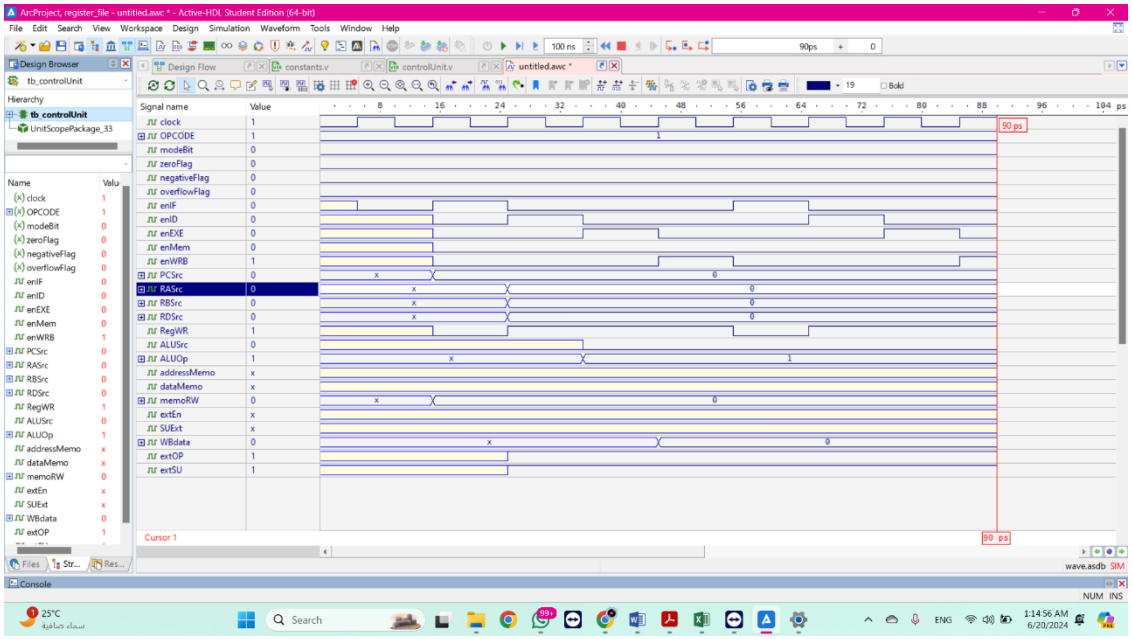
Figure 42, Processor 3,

RA: Selected based on RASrc
RB: Selected based on RBSrc
RW: Selected based on RDSrc
ALU Input: A (BusA), B (BusB or extended immediate value)
Memory Address and Data: memoADD, memoDATA
Write-back Data Selection: WBdata
Summary of Key Stages
Instruction Fetch (IF):
Module: instructionMemory
Action: Fetches the next instruction based on the current PC.
Instruction Decode (ID):
Action: Decodes the fetched instruction to determine the opcode and operand locations. Sets up control signals for subsequent stages.
Execute (EXE):
Module: alu
Action: Performs the specified arithmetic or logical operation on the operands.
Memory Access (MEM):
Module: mem
Action: Accesses memory for load and store instructions. Provides data to be written back to the register file.
Write-Back (WB):
Action: Writes the result of an operation or loaded data back to the appropriate register in the register file.

Control Signal Example
PCSrc: Determines the source for the next PC value (normal increment, jump, etc.).
RASrc, RBSrc, RDSrc: Selects the source registers for ALU operations.
ALUOp: Specifies the operation to be performed by the ALU.
memoRW: Controls read/write operations in data memory.
WBdata: Selects the data to be written back to the register file.
Integration and Interaction
Control Unit: Orchestrates the overall flow by generating appropriate control signals for each stage.
PC Module and Instruction Memory: Work together to fetch the next instruction.
Register File and ALU: Collaborate to perform computations.
Data Memory: Interacts with ALU and register file for load/store operations.
Sign Extension: Ensures immediate values are correctly sized for operations.

WAVE FORMS:

Some of Control Unit Test Benches wave forms that shows that our control unit is working properly.



ADD Instruction, if we compare the results with control unit table it shows that its correct.

Figure 43, Add Instr CU WF

ANDI:

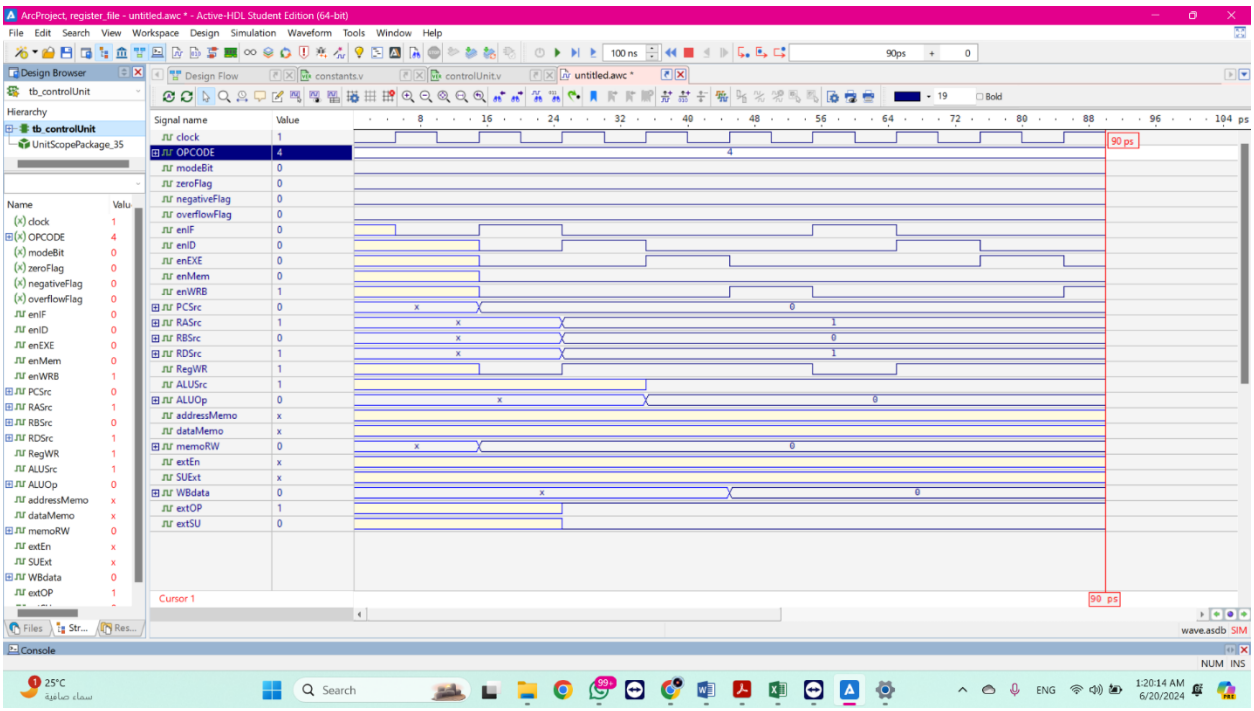


Figure 44, ANDI CU WF

BGT & BGTZ WITH ZERO FLAG:

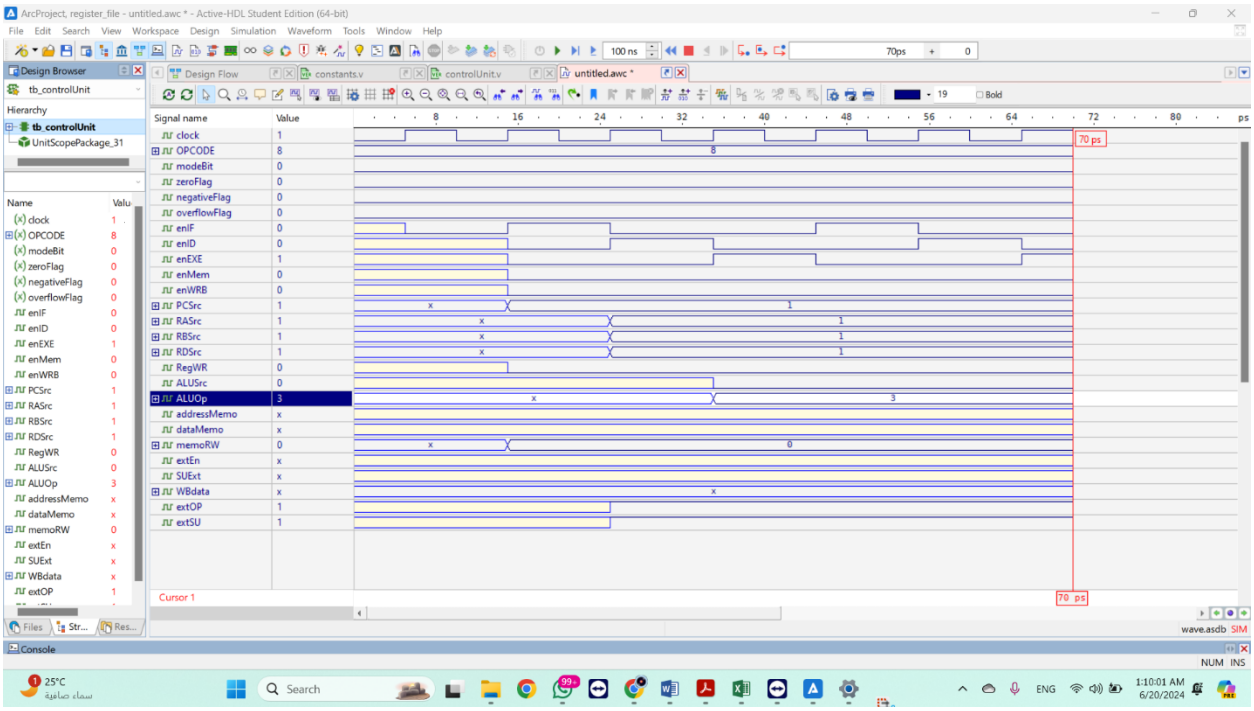


Figure 45, BGT WF CU

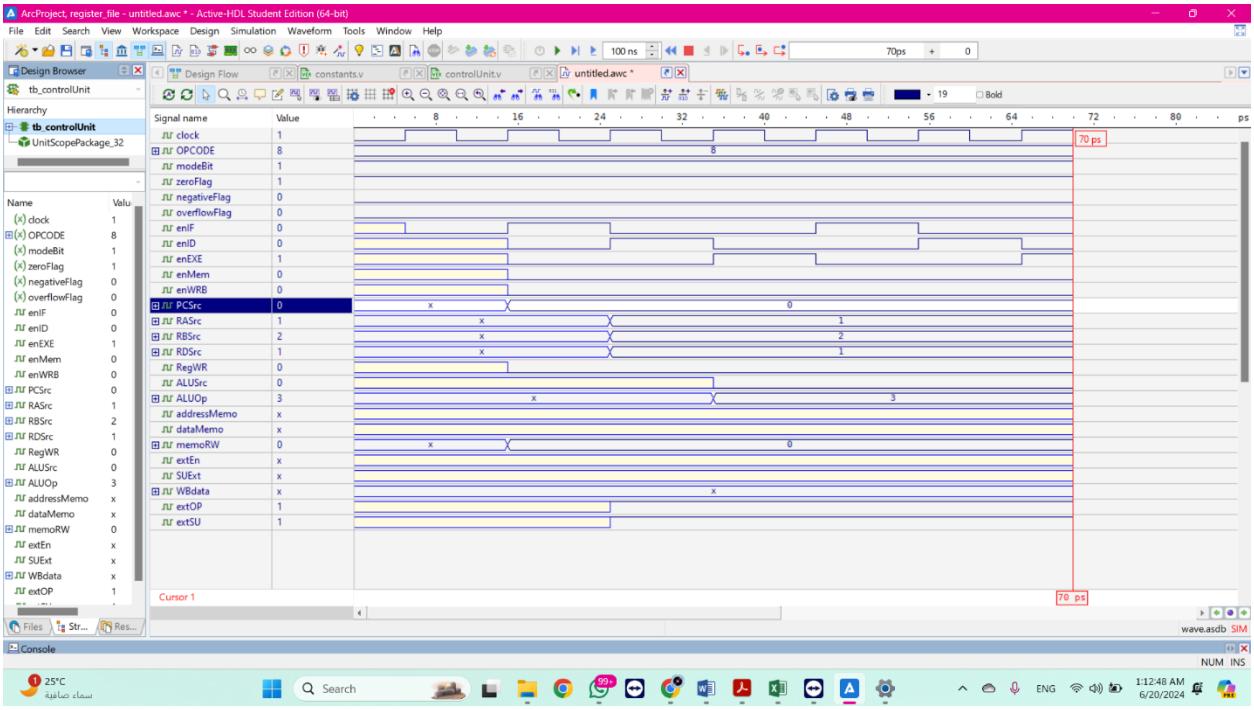


Figure 46, BGTZ CU WF

CALL, JUMP & RETURN:

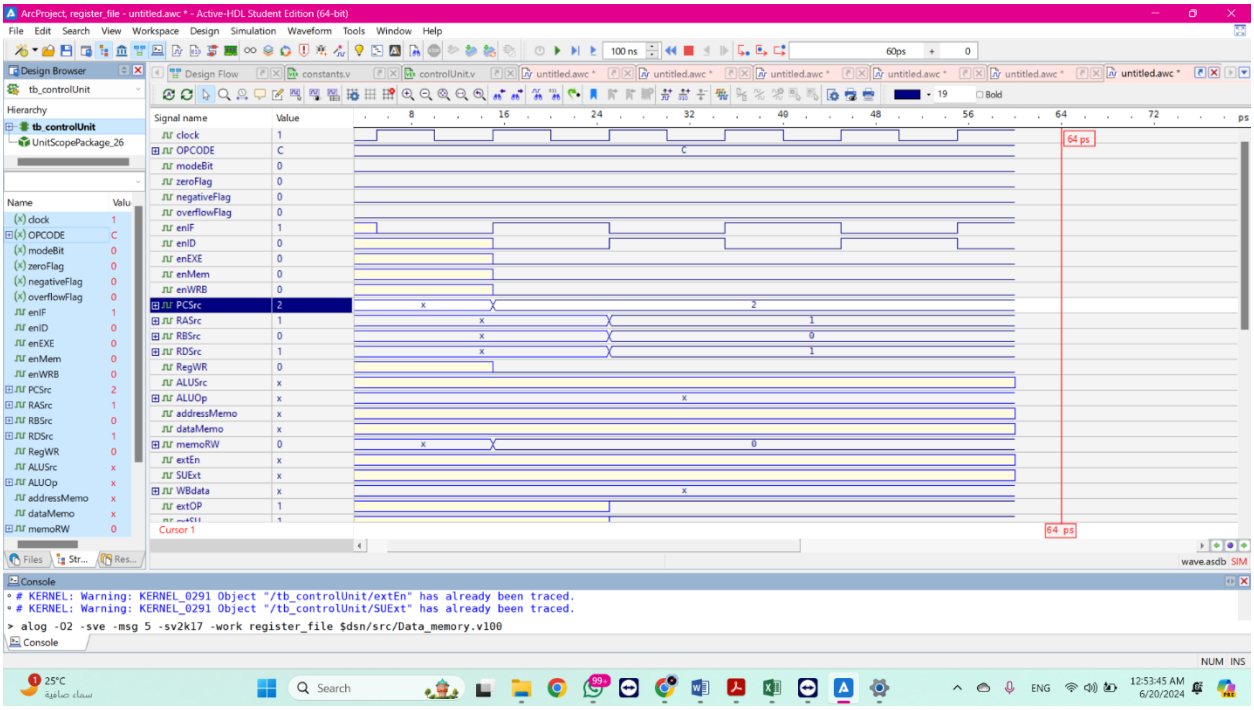


Figure 47, CALL CU WF

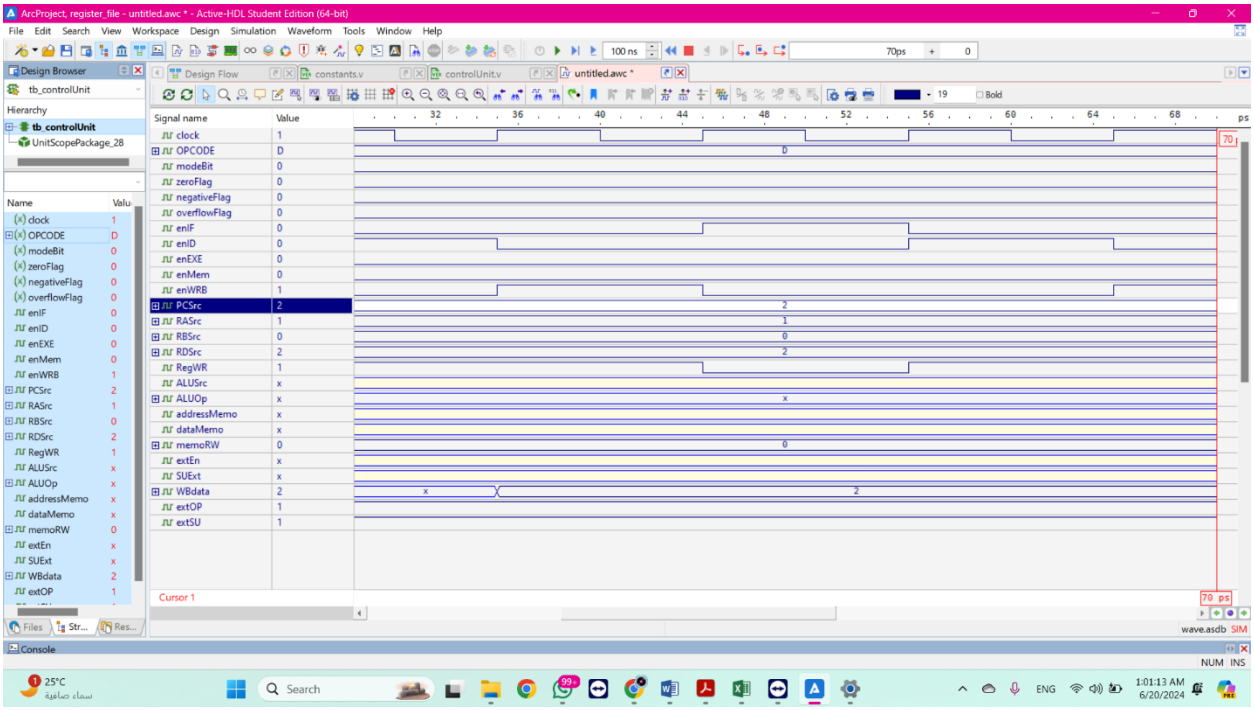


Figure 48, JUMP CU WF

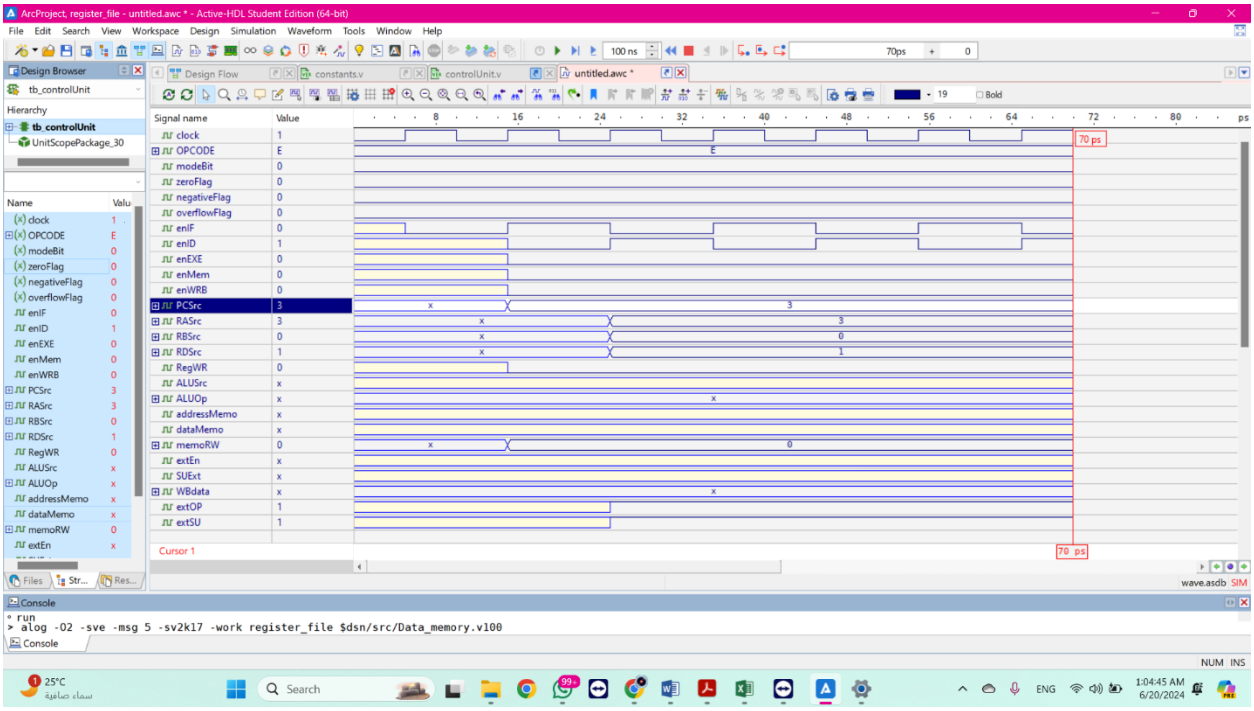


Figure 49, RETURN CU WF

LW & SV:

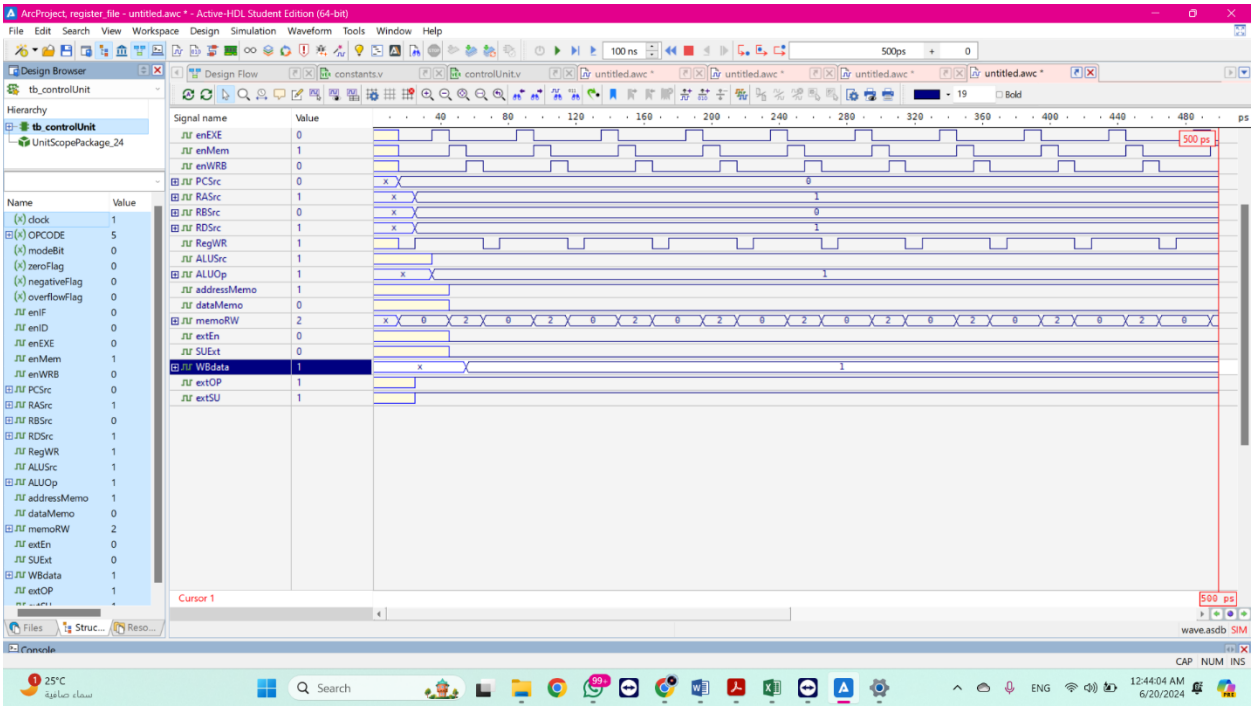


Figure 50, LW WAVEFORM CU

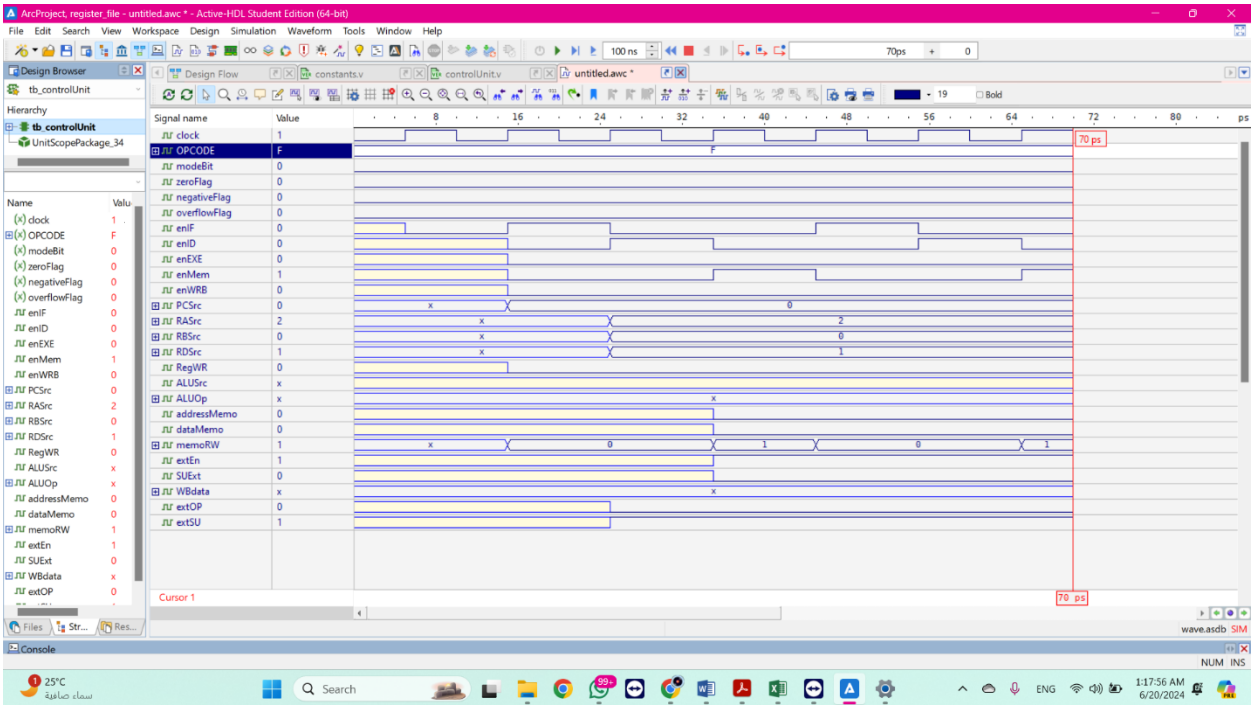


Figure 51, SV WV CU

The results show clearly that the Boolean expressions for control unit were accurate based on many results and based on PCSrc, Flags, Stages Enables, OPCODE, …. ETC

OPCODE WAVEFORMS:

After running the instruction written in the in instruction memory shown above in Figure[29] we obtained the following results:

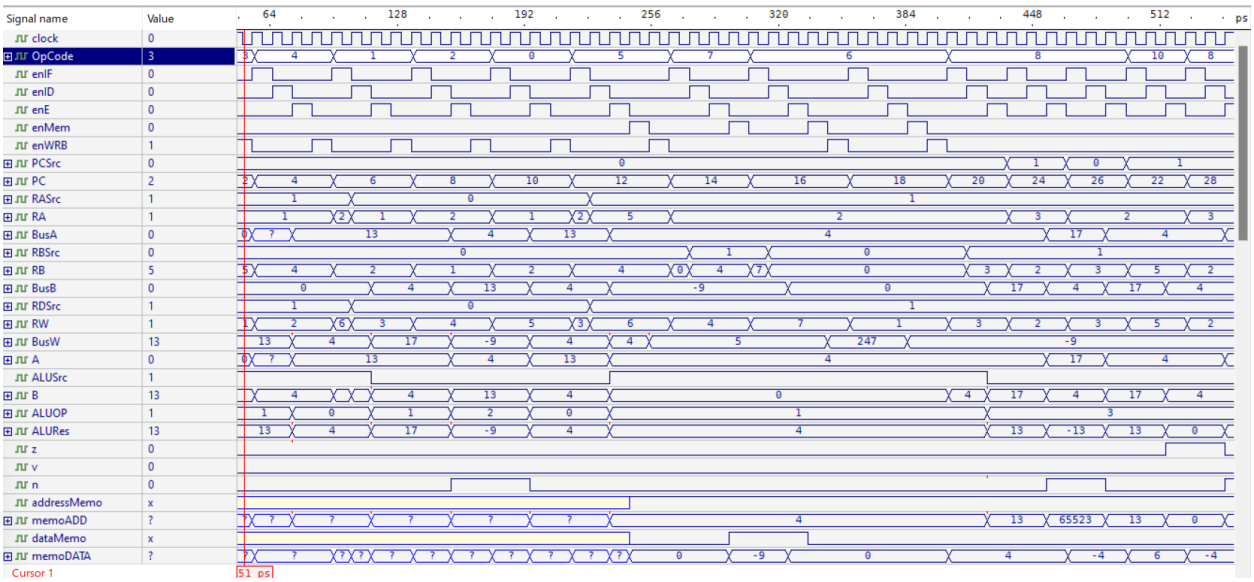


Figure 52:ADDI instruction wave form

The wave form shown above in Figure[52] present the results of execution the instruction:

ADDI R1,R1,13.

As we can see the processor successfully took the register R1 as the first operand to the ALU which we can see in the value of RA on the left and also as a destination register shown in the value of RW and the immediate as the second operand of the ALU as shown in the value of (B), the value in R1 initially is zero and we can see that in the value of BusA and the result of the operation ADDI is presented in the write back Bus (BusW) which is 13 and this value will be written in the register R1 after 4 clock cycles.

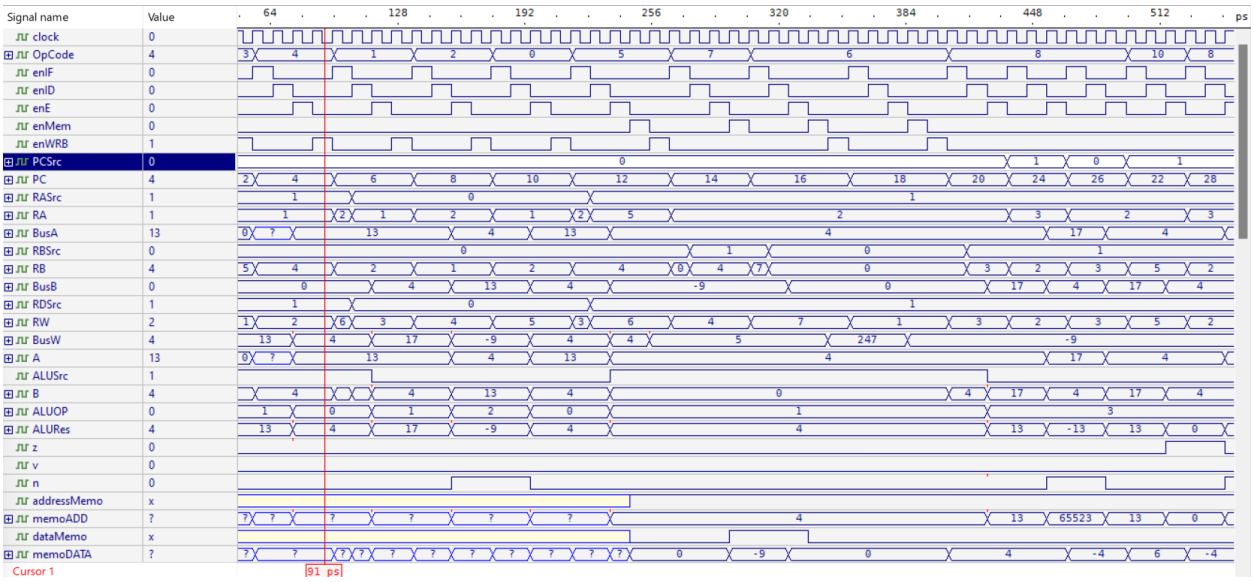


Figure 53:ANDI instruction wave form

The wave form shown above in Figure[53] present the results of execution the instruction:

ANDI R2,R1,4.

As we can see the processor successfully took the register R1 as the first operand to the ALU which we can see in the value of RA on the left and the immediate as the second operand of the ALU as shown in the value of (B) and the register R2 as a destination register shown in the value of RW , the value in R1 is 13 from the previous instruction and we can see that in the value of BusA and the result of the operation ANDI is presented in the write back Bus (BusW) which is 4 and this value will be written in the register R2 after 4 clock cycles.

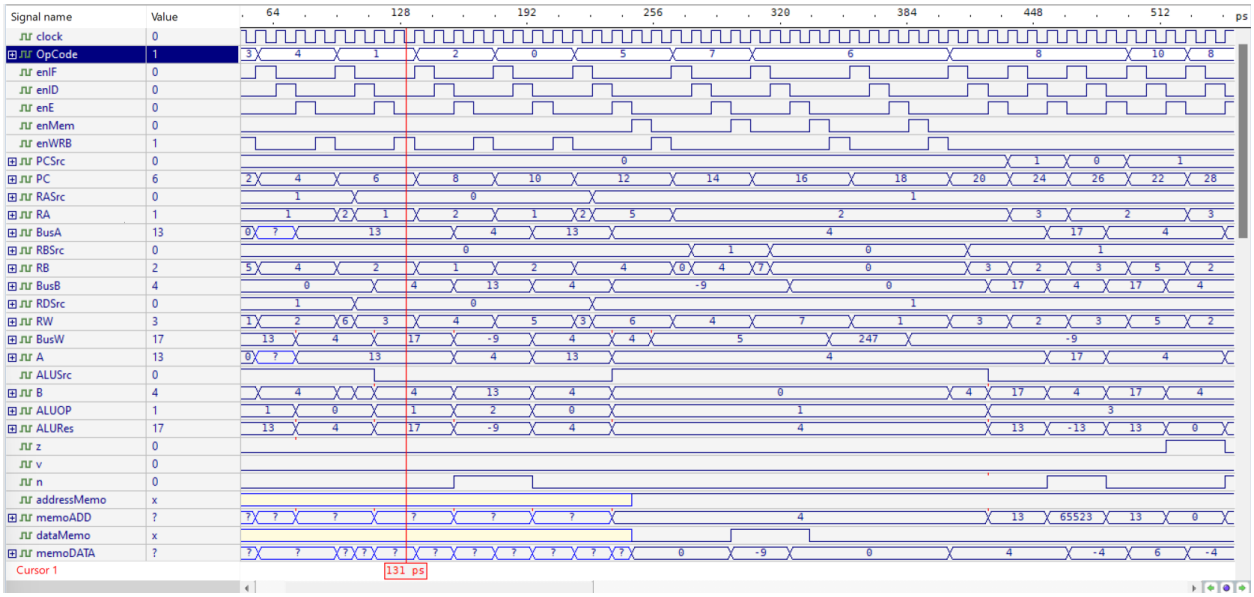


Figure 54:ADD instruction wave form

The wave form shown above in Figure[54] present the results of execution the instruction:

ADD R3, R1, R2.

As we can see the processor successfully took the register R1 as the first operand to the ALU which we can see in the value of RA on the left and the register R2 as the second operand of the ALU as shown in the value of (RB) and the register R3 as a destination register shown in the value of RW , the value in R1 is 13 and we can see that in the value of BusA, the value in R2 is 4 from the previous instruction and we can see that in the value of BusB and the result of the operation ADD is presented in the write back Bus (BusW) which is 17 and this value will be written in the register R3 after 4 clock cycles.

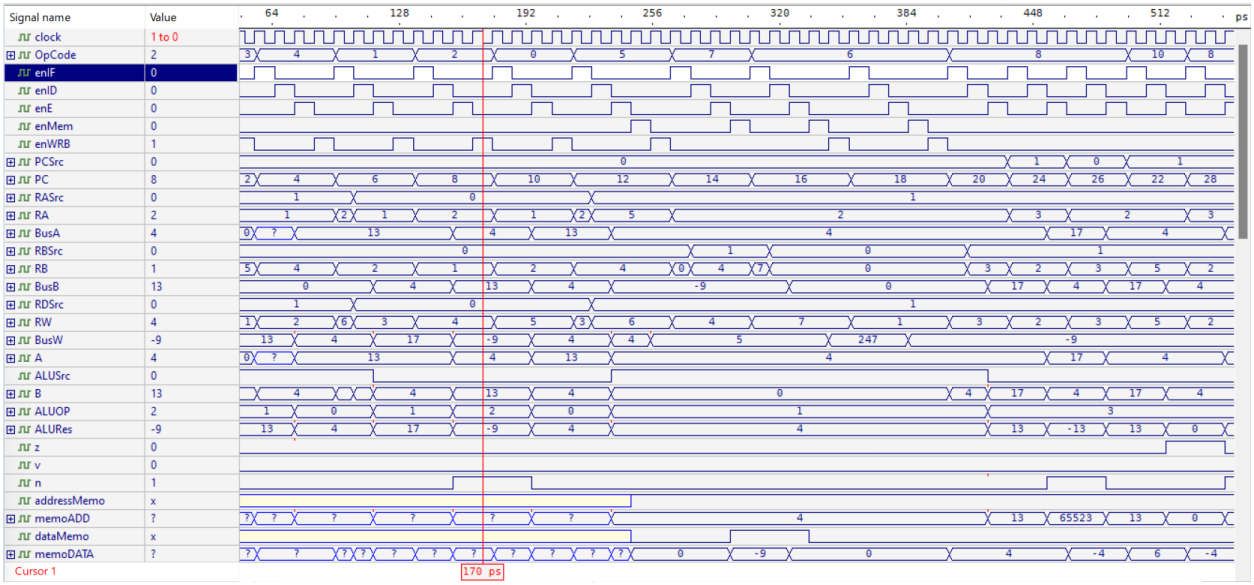


Figure 55:SUB instruction wave form

The wave form shown above in Figure[55] present the results of execution the instruction:

SUB R4, R2, R1.

As we can see the processor successfully took the register R2 as the first operand to the ALU which we can see in the value of RA on the left and the register R1 as the second operand of the ALU as shown in the value of (RB) and the register R4 as a destination register shown in the value of RW , the value in R1 is 13 and we can see that in the value of BusB, the value in R2 is 4 and we can see that in the value of BusA and the result of the operation SUB is presented in the write back Bus (BusW) which is -9 and this value will be written in the register R4 after 4 clock cycles.

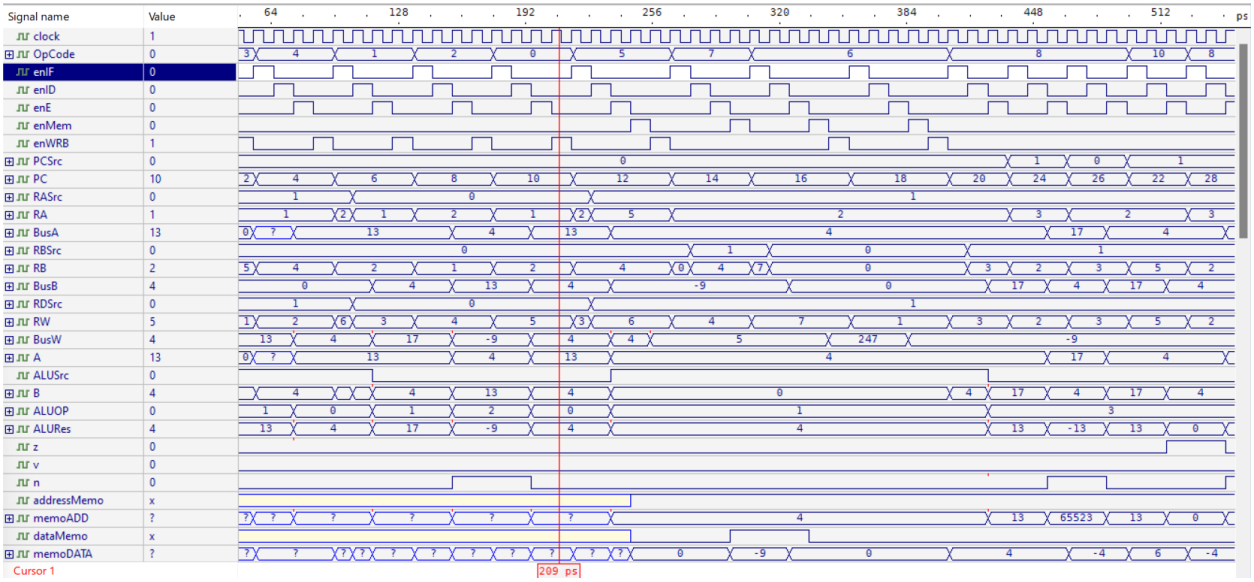


Figure 56:AND instruction wave form

The wave form shown above in Figure[56] present the results of execution the instruction:

AND R5, R1, R2.

As we can see the processor successfully took the register R1 as the first operand to the ALU which we can see in the value of RA on the left and the register R2 as the second operand of the ALU as shown in the value of (RB) and the register R5 as a destination register shown in the value of RW , the value in R1 is 13 and we can see that in the value of BusA, the value in R2 is 4 and we can see that in the value of BusB and the result of the operation AND is presented in the write back Bus (BusW) which is 4 and this value will be written in the register R5 after 4 clock cycles.

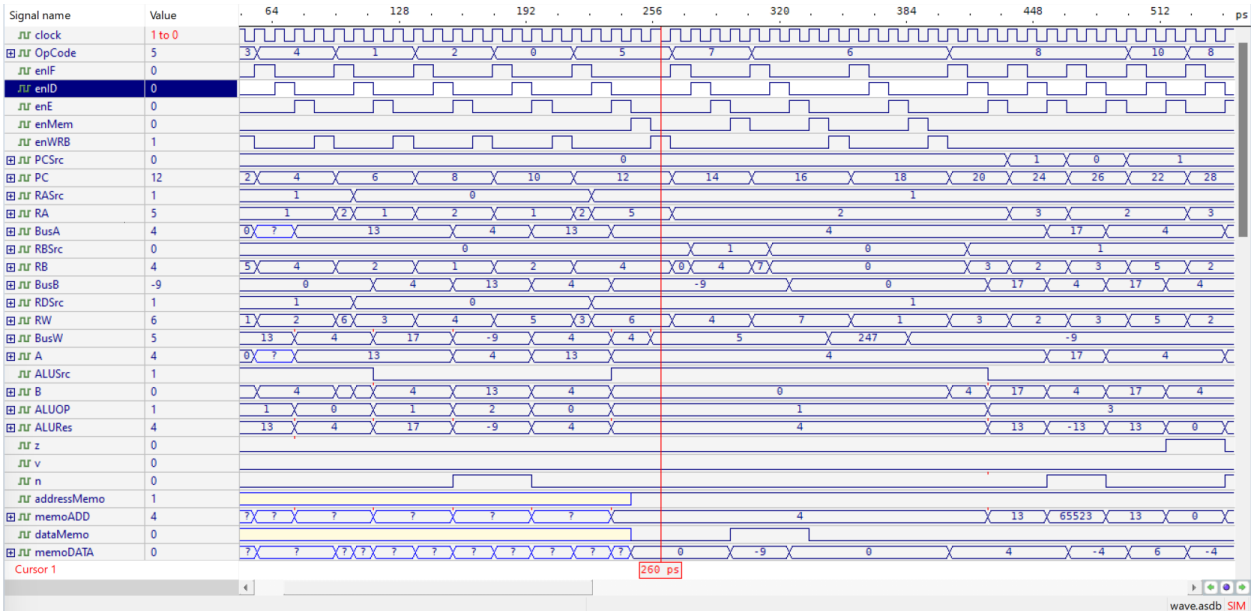


Figure 57:LW instruction wave form

The wave form shown above in Figure[57] present the results of execution the instruction:

LW R6, R5,0.

As we can see the processor successfully took the register R5 as the first operand to the ALU which we can see in the value of RA on the left and the immediate as the second operand of the ALU as shown in the value of (B) and the register R6 as a destination register shown in the value of RW , the value in R5 is 4 from the previous instruction and we can see that in the value of BusA, and the result of the ALU will be the address input to the data memory as shown in the value of (memoADD) which will results to load the value of this address in the destination register R6, the value is shown in the write back Bus (BusW) which is 5 and we can confirm this value by looking at the content of the memory shown in Figure[22] at the address 4, this value will be written in the register R6 after 5 clock cycles.

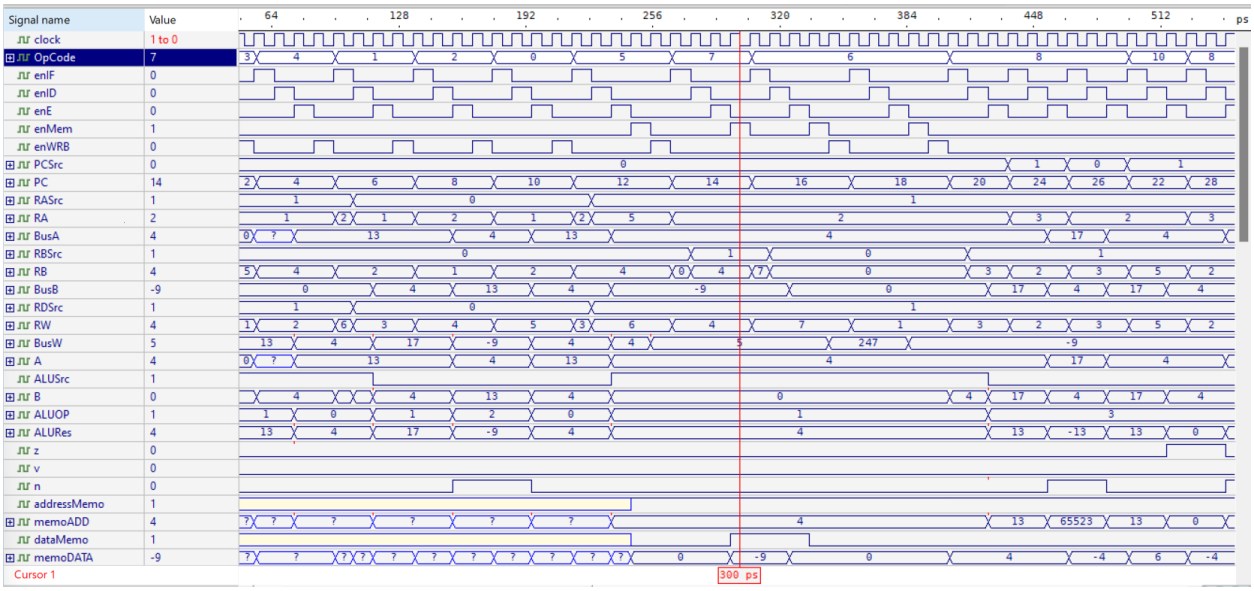


Figure 58:SW instruction wave form

The wave form shown above in Figure[58] present the results of execution the instruction:

SW R4, R2,0.

As we can see the processor successfully took the register R2 as the first operand to the ALU which we can see in the value of RA on the left and the immediate as the second operand of the ALU as shown in the value of (B) and the register R4 as the register that holds the data to be stored in the memory shown in the value of RB , the value in R2 is 4 and we can see that in the value of BusA, and the result of the ALU will be the address input to the data memory as shown in the value of (memoADD) which will results to store the value of the register R4 in this address, the value is shown in the (memoDATA) which is -9 , this value will be stored in the memory after 4 clock cycles.

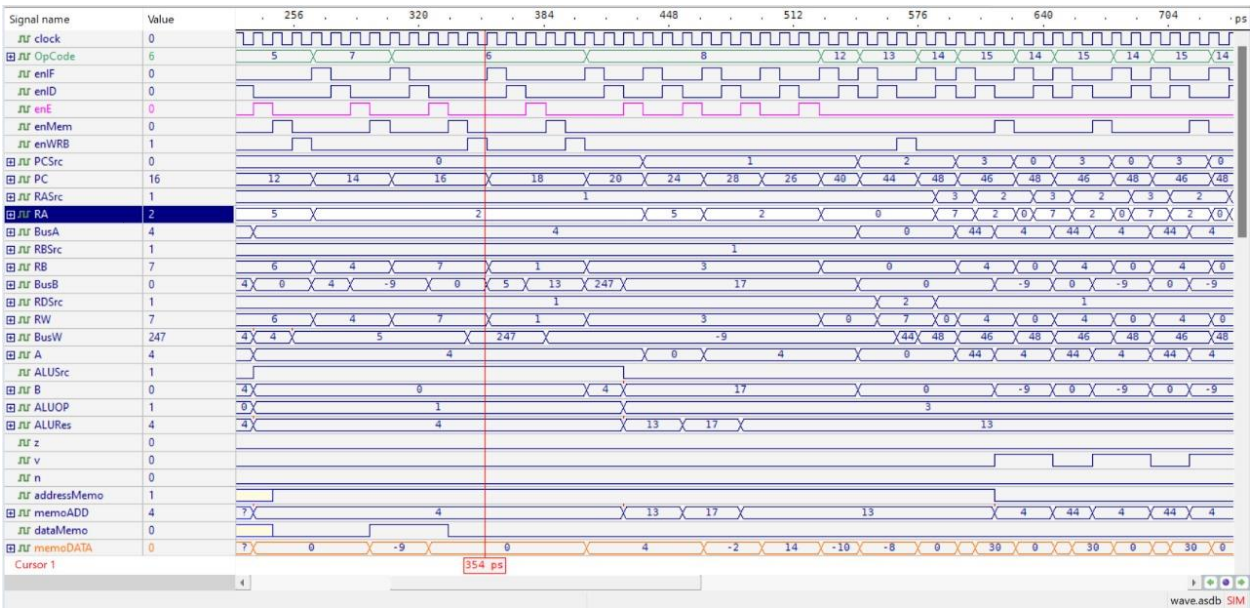


Figure 59:LBu instruction wave form

The wave form shown above in Figure[59] present the results of execution the instruction:

LBu R7, R2,0.

As we can see the processor successfully took the register R2 as the first operand to the ALU which we can see in the value of RA on the left and the immediate as the second operand of the ALU as shown in the value of (B) and the register R7 as a destination register shown in the value of RW , the value in R2 is 4 and we can see that in the value of BusA, and the result of the ALU will be the address input to the data memory as shown in the value of (memoADD) which will results to load the value of this address in the destination register R7, the value is shown in the write back Bus (BusW) which is 247 from a previous instruction after the unsigned extension for the loaded byte from the memory , this value will be written in the register R1 after 5 clock cycles.

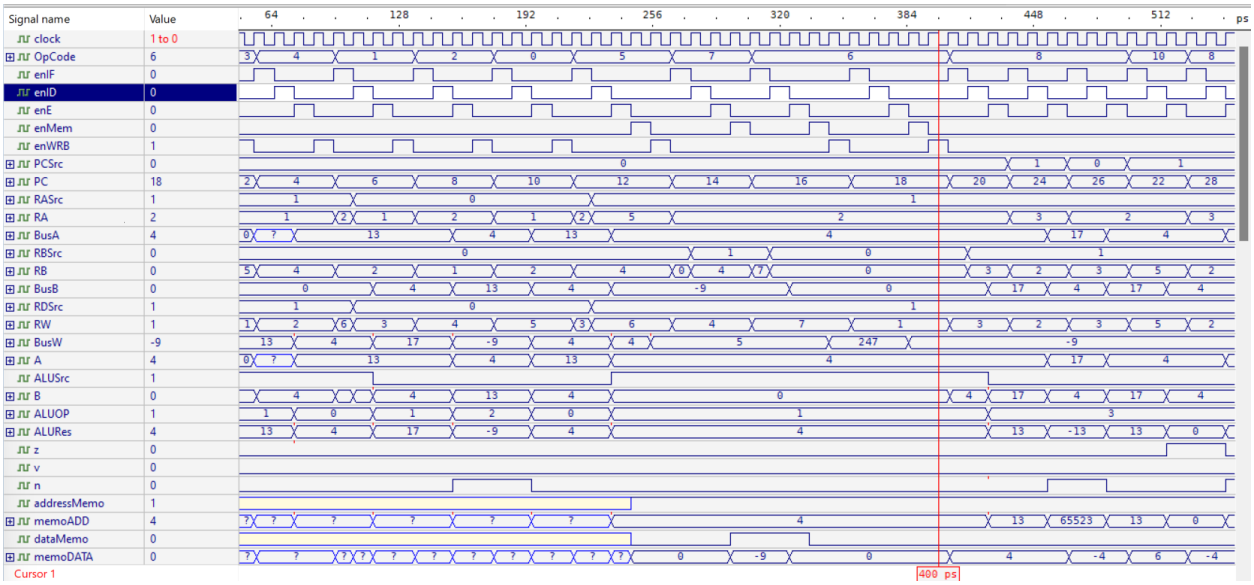


Figure 60:LBs instruction wave form

The wave form shown above in Figure[60] present the results of execution the instruction:

LBs R1, R2,0.

As we can see the processor successfully took the register R2 as the first operand to the ALU which we can see in the value of RA on the left and the immediate as the second operand of the ALU as shown in the value of (B) and the register R1 as a destination register shown in the value of RW , the value in R2 is 4 and we can see that in the value of BusA, and the result of the ALU will be the address input to the data memory as shown in the value of (memoADD) which will results to load the value of this address in the destination register R1, the value is shown in the write back Bus (BusW) which is -9 from a previous instruction after the signed extension for the loaded byte from the memory , this value will be written in the register R1 after 5 clock cycles.

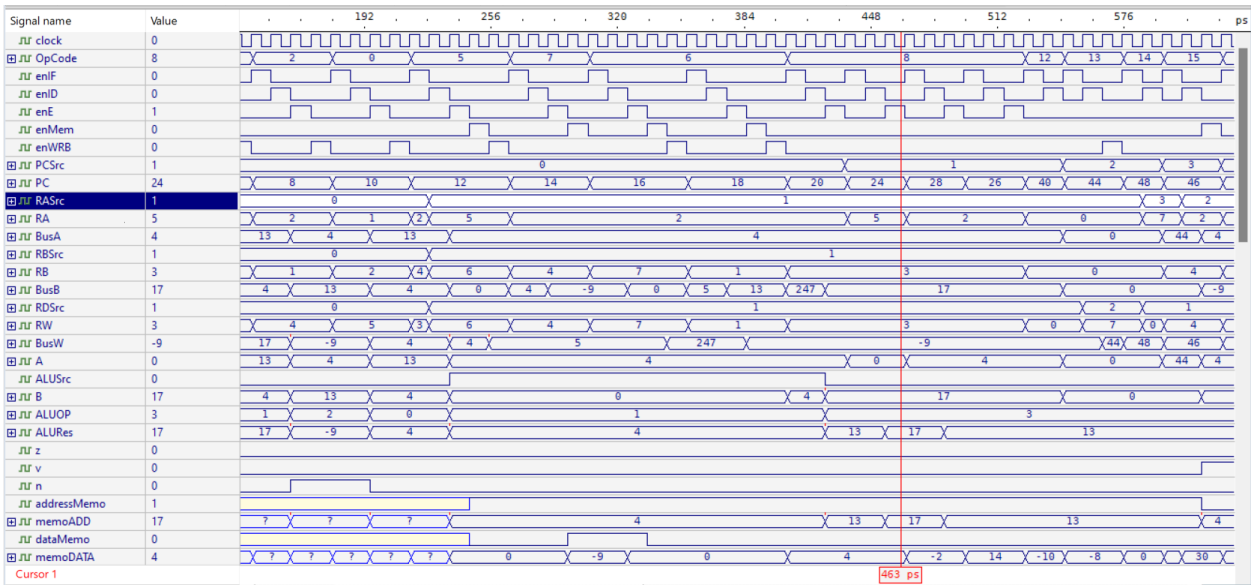


Figure 61:PC changes due to Branches instruction wave form

We have a series of deferent types of Branch instructions that we will analyze their effect by looking and the changes of the PC values from Figure [61]:

BGT R3, R2, 4: PC [20] the condition is true

BEQ R5, R2, 6: PC [22] the condition is true

BGT R3, R5, 4: PC [24] the condition is true

BGT R3, R2, 14: PC [26] the condition is true

BGT R3, R2, -2: PC [28] the condition is true

BEQ R1, R2, 6: PC [30] not reached

BLT R6, R5, 4: PC [32] not reached

BEQZ R1, R2, 4: PC [34] not reached

BLT R1, R2, -2: PC [36] not reached

BNE R1, R2, 4: PC [38] not reached

.

BNEZ R1, R2, -2: PC[42] not reached

After executing BGT R3, R2, 4 at PC [20] we can see that the next PC is 24 not 22 which indicate that the branch condition is true and the next instruction to be executed is BGT R3, R5, 4 at PC [24] and the next PC was 28 not 26, the next instruction to be executed is BGT R3, R2, -2 at PC [28] and the next PC is 26 not 30, back to the instruction that was skipped before at PC [26] BGT R3, R2, 14 the next PC is 40 not 28 which means the condition of the branch was true, the branches works by subtracting the operand

using the ALU which (the ALU) affect the Flags(zeroFlag, negativeFlag, overflowFlag) then they are used to determine what should the next PC be, all of the branch instruction takes 3 clock cycles to exute.

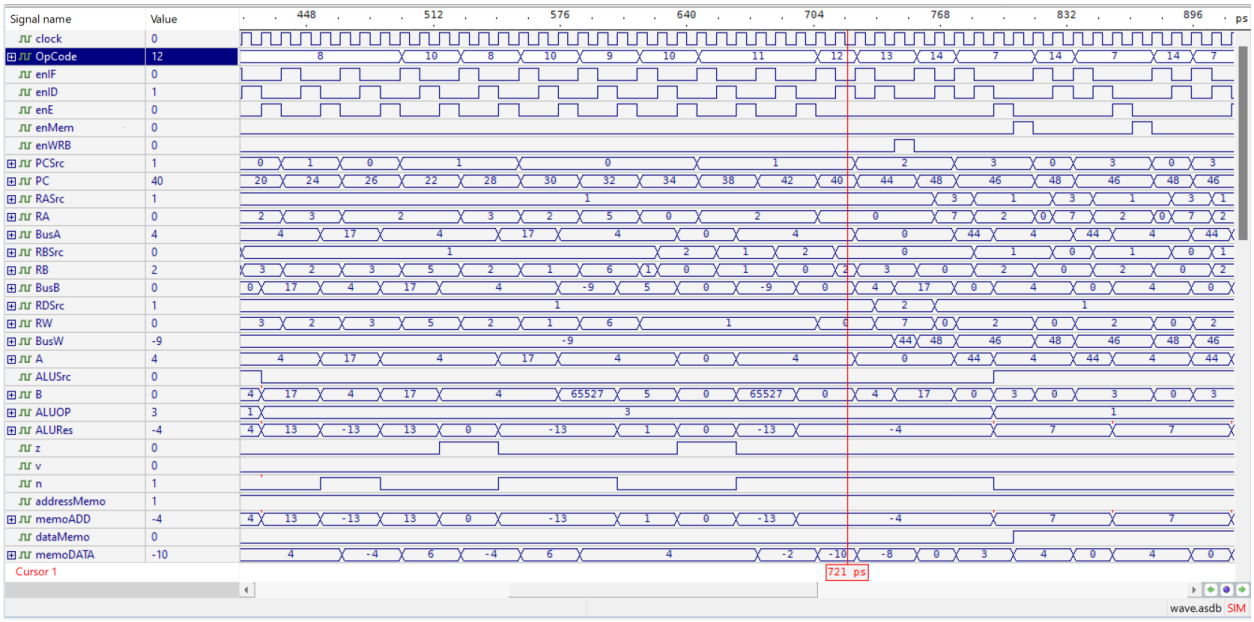


Figure 62:JMP instruction wave form

The wave form shown above in Figure [62] present the results of execution the instruction:

JMP 12.

The Jump instruction instantly jump to the desired PC values that is specified using the immediate part of the instruction unconditionally which we can see the effect of the jump instruction on the next PC that is 44 not 42 and it takes just 2 clock cycles to operate.

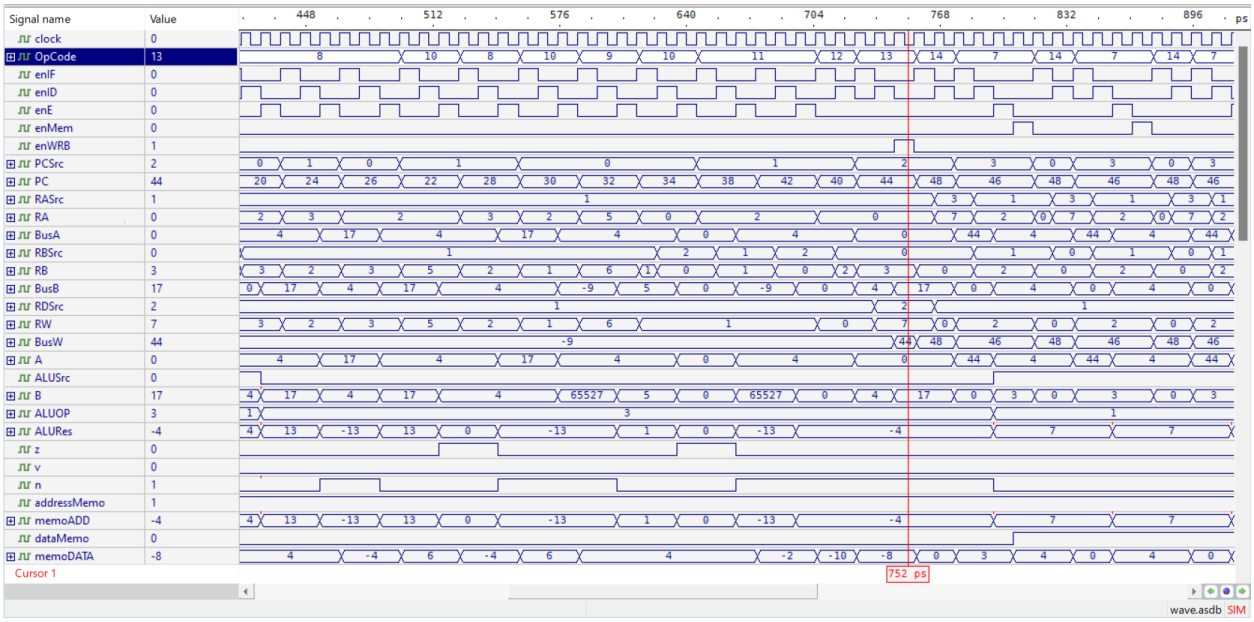


Figure 63:Call instruction wave form

The wave form shown above in Figure [63] present the results of execution the instruction:

CALL ,24.

Similarly to the Jump instruction, the CALL instruction instantly jump to the desired PC values that is specified using the immediate part of the instruction unconditionally which we can see the effect of the CALL instruction on the next PC that is 48 not 46, but the difference is the CALL instruction changes the value of the register R7 to the current PC +2 which we can see that the register R7 to place as a destination register in RW and the value of the write back bus is 44 which is PC+2, the CALL instruction takes 3 clock cycles to operate.

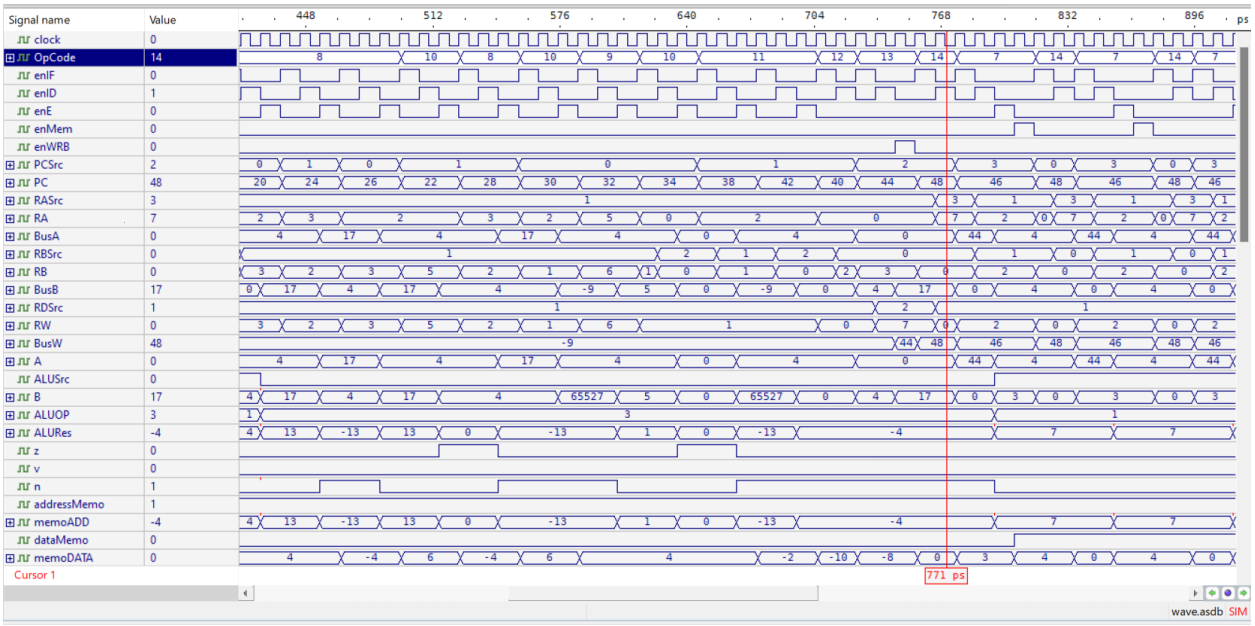


Figure 64:RET instruction wave form

The wave form shown above in Figure [64] present the results of execution the instruction:

RET.

The RET is very simple, it's basically put the value stored in the register R7 into the PC as we can see the value of the operand RA is R7 which contains 46 from the previous instruction, the next PC value is 46 and the instruction takes 2 clock cycles to operate.

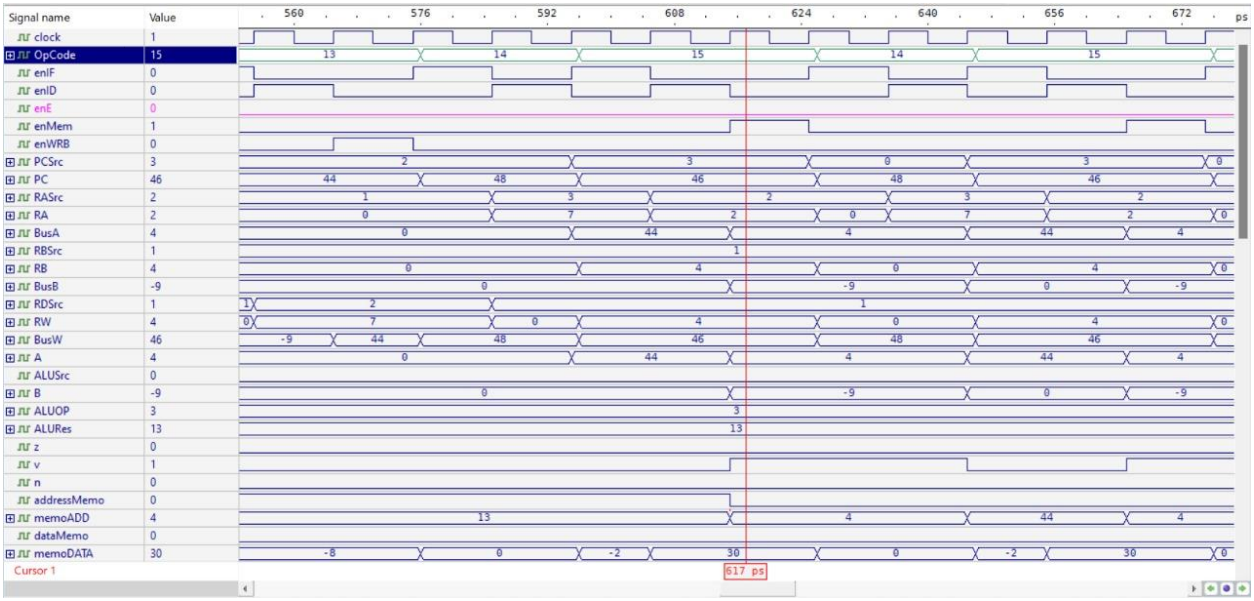


Figure 65:SV instruction wave form

The wave form shown above in Figure [65] present the results of execution the instruction:

Sv R2,30.

As we can see the processor successfully took the register R2 as the first operand to the ALU which we can see in the value of RA on the left and the immediate to be stored in the memory shown in the value of B , the value in R2 is 4 and we can see that in the value of BusA which is going to be the address input to the data memory as shown in the value of (memoADD) which will results to store the value of the immediate in this address , the value is shown in the (memoDATA) which is 30 , this value will be stored in the memory after 3 clock cycles.

Conclusion:

This project required the design, implementation, and verification of a Verilog multicycle RISC processor. We succeeded in achieving the following goals with this project:

1. Design and Implementation: We created a five-stage multicycle processor that included stages for write-back, memory access, fetch, decode, and ALU. Carefully designed datapath and control path to support the provided ISA instructions. Important parts including memory, the ALU, and registers were combined to create a coherent CPU design. We carefully considered our design options to guarantee the accuracy and performance of the processor.

2. Control Signals: We created the control signals required to run the CPU. In order to guarantee correct instruction execution, truth tables and Boolean equations were developed.

3.Verification: A thorough verification environment was created, complete with a working testbench for the CPU. The ability of the processor to correctly handle a variety of instructions was demonstrated by the writing and execution of multiple test programs. Waveform diagrams were employed to confirm the accuracy of the processor and validate the results.

4.Teamwork: The project was a cooperative involving all team members who contributed to the design, implementation, testing, simulation, and report writing. Every phase of the project was fully covered by all the group team.

5. Documentation and Reporting: A thorough report that included test cases, simulation results, control signals, microarchitecture schematics, and design process documentation was put together. This report offers a thorough description of the project's progress and results.

All things considered, the project proved successful in fulfilling the objectives and proving that a multicycle RISC processor could be designed and verified. Our comprehension of computer architecture, Verilog design, and processor verification was improved by the practical experience this project provided.

References

[1]: CISC - Complex Instruction Set Computer

https://www.andersdx.com/cisc-complex-instruction-set-computer/#:~:text=Processors%20designed%20with%20full%20set,works%20several%20low%2Dlevel%20acts.

[2]: ARM

https://www.arm.com/glossary/risc#:~:text=A%20Reduced%20Instruction%20Set%20Computer,typically%20found%20in%20other%20architectures.

[3]: geeksforgeeks

https://www.geeksforgeeks.org/computer-organization-risc-and-cisc/

[4]: geeksforgeeks

https://www.geeksforgeeks.org/multi-cycle-data-path-and-control/