
**Faculty of Engineering and Technology**

**Electrical and Computer Engineering Department**

# Computer Architecture – ENCS4370

## Project#2 – Report

**Prepared By:-**

**Alaa Saleem-1200001** **Section#: 1**

**Afaf Amwas-1203359** **Section#: 2**

**Jenin Karaja-1200540** **Section#: 2**

**Instructors: Dr. Ayman Hroub & Dr. Aziz Qaroush**

**Date of Submission: 29.Jan.2024**

**BIRZEIT**

**Jan – 2024**

## Design and Implementation

We selected a multi-cycle datapath for the processor design, even though it has its limitations such as it doesn't achieve parallelism, it's simple and has a good performance compared to single-cycle. While the pipeline approach has better performance than multi-cycle, its implementation may require more time and testing, and due to our current academic workload the multi-cycle approach was the best choice for us because it's simpler.

### Datapath Components

#### 1. Instruction Memory

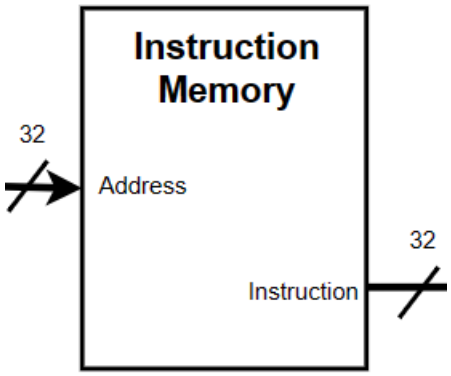


*Figure 1: Instruction Memory Block Diagram*

The instruction memory is used in the fetch stage. It is a word addressable memory, meaning it stores 4 bytes or 32 bits in one memory cell. These are the stored instructions the program should run. The PC (program counter) points at the address of the instruction, thus the 32 bit address of the PC is taken as input to the instruction memory block and outputs a 32 bit instruction. Since the instruction and IR size are both 32 bits, the PC register is incremented by one to move to the next instruction.

## 2. Register File

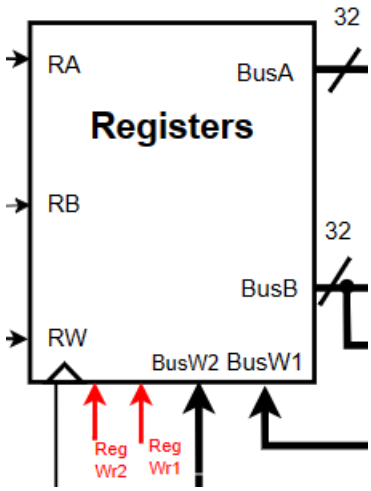


*Figure 2: Register File Block Diagram*

The register file consists of 16 general purpose registers that range from R0-R15, each with a size of 32 bit. In order to access a specific register, a 4 bit address is needed to distinguish one register from another; therefore, in the instruction register, there are 4 bits assigned as addresses for one register. The input of the register file comes from the instruction register. RA is the 4 bits of the first operand, while RB is the 4 bits of the second operand, and RW is the 4 bits of the destination register. To read two operands from the register file, two output buses (BusA and BusB) of the same length of a register (32 bit) is required. There are also two bus inputs to write back onto the register file. The first bus is used to write back ALU results and the output of a load (without increment) instruction. Moreover, the second bus is used in one case only, when the instruction is LW.POI (load post increment), because in this case the base register Rs1 is incremented by one and stored back in the register file in the same time the load result is stored in the destination register. Thus, there are two signals to activate the write back operation on the register file that correspond to the two buses respectively.

## 3. Date Memory

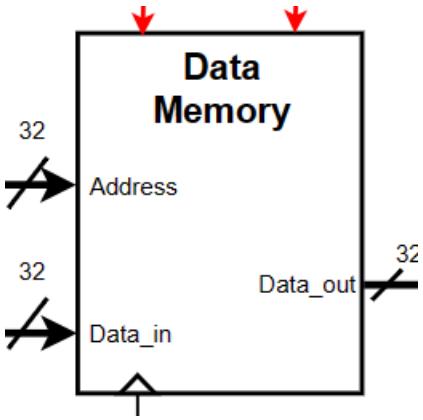


*Figure 3: Data Memory Block Diagram*

The data memory is a word addressable memory that comes with a stack memory. The 32 bit address input of this functional unit can come from either the ALU result or the stack pointer, which is a register that stores the address of the top of the stack element. This memory can be used for both reading and writing. These two signals are needed for this. If the signal "MemRead" is activated, then only the address input is needed to read from the memory and output the data using the "Data_out" bus. If the signal is "MemWrite", then the data memory needs the address and the data that should be stored in the memory. The instructions "push" and "store" are both used to write on the memory, while the instructions "pop", "LW", and "LW.POI" are used to read from the memory.

## 4. Sign Extender

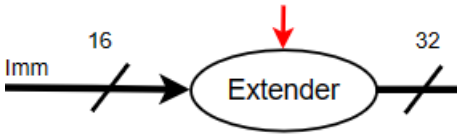


*Figure 4: Extender Block Diagram*

The extender is used in many instructions such as "ANDI", "LW", "BGT", and many other I-type instructions. The input of the extender is a 16 bit immediate value that comes from the instruction register. The signal "ExtOp" is used to either extend that immediate using signed extension ( "ExtOp" = 1 ) or unsigned extension (When "ExtOp" = 0 ). If the "ExtOp" is set, then the most significant bit of the immediate input will determine if the extension is by zeros or

by ones. To do this, the 16th is will go into “AND” operation with bit 0, and if the result is 1, the extender will extend by ones, otherwise it will extend by zeros.

## 5. ALU

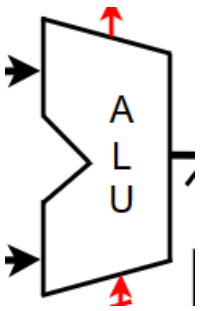


*Figure 5: ALU Block Diagram*

The ALU functional unit supports three distinct operations, which are “AND”, “ADD”, and “SUB”. Therefore there are two bits used for the “ALUOp” signal that chooses between these operations. Two inputs go into this functional unit. The first input comes from “BusA” and the second input comes from either “BusB” or the Extender. For R-type, ALU I-type and branch instructions, the second input of the ALU functional unit is always “BusB”, while in load and store instructions, the second input is the signed extended immediate. After an ALU operation is done, the “zero” and “negative” flag signals are generated and used to check branch conditions.

## 6. Program Counter (PC)

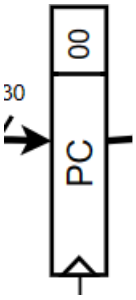


*Figure 6: PC Block Diagram*

The Program counter is a 32 bit register that points at the next instruction. Since the memory and instructions are word addressable, then the pc is incremented by one in order to move from one instruction to the next instruction sequentially.

### 7. Stack Pointer (SP)

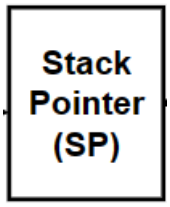


*Figure 7: SP Block Diagram*

The stack pointer is another 32 bit register that holds the address of the top of the stack. If the instruction is “pop”, then the stack pointer is decremented by one. Similarly, if the instruction is “push”, then an element is pushed onto the stack memory thus incrementing the stack pointer by one so that it remains pointing at the top element of the stack.

### 8. 4x1 Mux

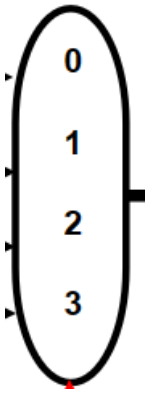


*Figure 8: 4x1 Mux Block Diagram*

This 4x1 mux is used in the project to choose between four different options for the value of the PC register that points at the next instruction. These options include: “PC+1”, “Jump target”, “Branch target address”, and “Return”. The mux chooses one output based on the values of the selection signals called”PCSrc1” and “PCSrc2”.

### 9. 2x1 Mux

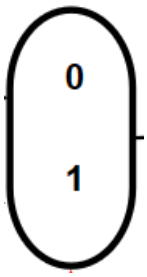


*Figure 9: 2x1 Mux Block Diagram*

This 2x1 mux is used in different places of the datapath. For Example, a 2x1 mux is used to choose between “extender output” or “BusB” as the second operand of the ALU functional unit.

## Datapath Assembly

### 1. Instruction Fetch (IF)

The PC provides the address as input to the Instruction Memory, and the Instruction Memory retrieves the corresponding instruction from the memory as output. As seen in the figure below, a mux is used to choose between the values of the PC. In sequential code where there are no jump or branch instructions, the PC is incremented by one. Otherwise, the PC is changed differently depending on the selection lines.

### 2. Instruction Decode (ID)

The Content of the IR (which is the current instruction) is decoded and the control unit generates control signals based on that instruction. The instruction register is divided into specific fields depending on the opcode. For Example if the opcode is “000000”, then the instruction is divided based on R-type format in which the next 4 bits are for register destination, then 8 bits for the first operand and second operands (stored in registers) respectively, then 14 bits being unused. To encompass all types of instructions, a mux is added to choose between the second operand, which is either “Rs2” or “Rd”. In branch instructions, “Rd” is used as an

operand compared with another operand “Rs1”, therefore this mux will choose “Rd” as the second operand. An Extender is also implemented in this stage for specific instructions.

### 3. Execution (EX)

The execution stage consists of the ALU, and it performs arithmetic and logic operations on the operands based on the control signals. As mentioned above, it supports three distinct operations, which are “AND”, “ADD”, and “SUB”. This stage is not visited by all instructions. For Example the “Jump ” instruction stops at the decode stage and does not continue to other stages.

### 4. Memory Access (MEM)

The Memory stage is the fourth stage in the multicycle CPU implementation. Although, many instructions skip this stage as there is no need to write or read from the memory. For Example R-type and ALU I-type will need to go through instruction fetch, decode, execution and write back, skipping the memory stage. The memory is used in store and load instructions, as well as, “push”, “pop”, “call” and “return” instructions. To write on the memory, a data_in bus is connected from a mux (that is connected to the register file and PC + 1) to the memory input port. This mux chooses the value of PC + 1 in case of “call” instruction, and chooses “BusB” in case of “push” and “store” instructions. In “return” instruction, the top of the stack is the new value of the PC, thus a bus is connected from data output of the memory to the mux that chooses PC values.

### 5. Write Back (WB)

The write back stage is the fifth and final stage where the results are written back to the register file.  To read from memory, a data_out bus is connected from the memory to the register file in cases such as “LW”, “LW.POI”, “pop”. A mux is used to choose between the data_out from the memory or the ALU result and stores this result back into the register file in the destination register. The instruction “LW.POI” will also write back the new value of “Rs1” back

into the register file at the same time the destination register is being written on, thus two buses are used in this case.

## Control Signals

| Main Control Signals | | |
|---|---|---|
| **Signal** | **Effect when '0'** | **Effect when '1'** |
| Reg Src2 | Second source register is the value of Rs2 register | Second source register is the value of Rd register |
| RegWr2 | No write on Rs1 | Source register Rs1 is written with the data on BusW2 |
| RegWr1 | No write on Rd | Destination register Rd is written with the data on BusW1 |
| EXOP | 16-bit immediate is zero-extended | 16-bit immediate is sign-extended |
| ALUSrc | Second ALU operand is the value of register Rt that appears on BusB | Second ALU operand is the value of the extended 16-bit immediate |
| Data_inSig | Data_in = PC + 1 | Data_in = the value of register Rt that appears on BusB |
| AddressSig | Address = ALU result | Address = Top element of stack |
| MemRd | Data memory is NOT read | Data memory is read<br>Data_out ← Memory[address] |
| MemWr | Data Memory is NOT written | Data memory is written<br>Memory[address] ← Data_in |
| WBdata | BusW1 = ALU result | BusW1 = Data_out from Memory |
| Push_popSig | Decrement the Stack Pointer (pop) | Increment the Stack Pointer (push) |

| ALU Control Signals | | | |
|---|---|---|---|
| **Signal** | **Effect when '00'** | **Effect when '01'** | **Effect when '10'** |
| ALUOP | AND between the 2 ALU operands | ADD between the 2 ALU operands | SUB between the 2 ALU operands |

| PC Control Signals | | | | |
|---|---|---|---|---|
| **Signal** | **Effect when '0'** | **Effect when '1'** | **Effect when '2'** | **Effect when '3'** |
| PCSrc | PC = PC+1 | PC = Jump target address | PC = Branch target address | PC = Return address |

## Main and ALU Control Signals - Truth Table

| Type | Instr | Input | Output | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Op | Reg Src2 | RegWr2 | RegWr1 | EXOP | ALUSrc | Data_inSig | AddressSig | MemRd | MemWr | WBdata | Push_popSig | ALUOP |
| R-Type | AND | 000000 | 0 | 0 | 1 | x | 0 | x | x | 0 | 0 | 0 | x | 00 |
| | ADD | 000001 | 0 | 0 | 1 | x | 0 | x | x | 0 | 0 | 0 | x | 01 |
| | SUB | 000010 | 0 | 0 | 1 | x | 0 | x | x | 0 | 0 | 0 | x | 10 |
| I-Type | ANDI | 000011 | x | 0 | 1 | 0 | 1 | x | x | 0 | 0 | 0 | x | 00 |
| | ADDI | 000100 | x | 0 | 1 | 1 | 1 | x | x | 0 | 0 | 0 | x | 01 |
| | LW | 000101 | x | 0 | 1 | 1 | 1 | x | 0 | 1 | 0 | 1 | x | 01 |
| | LW.POI | 000110 | x | 1 | 1 | 1 | 1 | x | 0 | 1 | 0 | 1 | x | 01 |
| | SW | 000111 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | x | x | 01 |
| | BGT | 001000 | 1 | 0 | 0 | 1 | 0 | x | x | 0 | 0 | x | x | 10 |
| | BLT | 001001 | 1 | 0 | 0 | 1 | 0 | x | x | 0 | 0 | x | x | 10 |

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | BEQ | 001010 | 1 | 0 | 0 | 1 | 0 | x | x | 0 | 0 | x | x | 10 |
| | BNE | 001011 | 1 | 0 | 0 | 1 | 0 | x | x | 0 | 0 | x | x | 10 |
| J-Type | JMP | 001100 | x | 0 | 0 | x | x | x | x | 0 | 0 | x | x | xx |
| | CALL | 001101 | x | 0 | 0 | x | x | 0 | 1 | 0 | 1 | x | 1 = push | xx |
| | RET | 001110 | x | 0 | 0 | x | x | x | 1 | 1 | 0 | x | 0 = pop | xx |
| S-Type | PUSH | 001111 | 1 | 0 | 0 | x | x | 1 | 1 | 0 | 1 | x | 1 = push | xx |
| | POP | 010000 | x | 0 | 1 | x | x | x | 1 | 1 | 0 | 1 | 0 = pop | xx |

## PC Control Signal – Truth Table

| | Instr | | | | Output | |
|---|---|---|---|---|---|---|
| | | Op | Negative | Zero | PCSrc1 | PCSrc0 |
| R-Type | AND | 000000 | x | x | 0 | 0 |
| | ADD | 000001 | x | x | 0 | 0 |
| | SUB | 000010 | x | x | 0 | 0 |
| I-Type | ANDI | 000011 | x | x | 0 | 0 |
| | ADDI | 000100 | x | x | 0 | 0 |
| | LW | 000101 | x | x | 0 | 0 |
| | LW.POI | 000110 | x | x | 0 | 0 |
| | SW | 000111 | x | x | 0 | 0 |
| | BGT | 001000 | 0 | x | 0 | 0 |
| | BGT | 001000 | 1 | x | 1 | 0 |
| | BLT | 001001 | 0 | x | 0 | 0 |
| | BLT | 001001 | 1 | x | 1 | 0 |
| | BEQ | 001010 | x | 0 | 0 | 0 |
| | BEQ | 001010 | x | 1 | 1 | 0 |
| | BNE | 001011 | x | 0 | 1 | 0 |
| | BNE | 001011 | x | 1 | 0 | 0 |
| J-Type | JMP | 001100 | x | x | 0 | 1 |
| | CALL | 001101 | x | x | 0 | 1 |
| | RET | 001110 | x | x | 1 | 1 |
| S-Type | PUSH | 001111 | x | x | 0 | 0 |
| | POP | 010000 | x | x | 0 | 0 |

## Boolean Equations

1) **RegSrc2 =** $\overline{AND} \cdot \overline{ADD} \cdot \overline{SUB} = \overline{(AND + ADD + SUB)}$

2) **RegWR2 = LW.POI**

3) **RegWR1 = R-Type . ANDI . ADDI . LW. (LW .POI)**

4) **EXOP =** $\overline{ANDI}$

5) **ALUSrc =** $\overline{(R - Type)} \cdot \overline{BGT} \cdot \overline{BLT} \cdot \overline{BEQ} \cdot \overline{BNE}$
$= \overline{(R - Type + BGT + BLT + BEQ + BNE)}$

6) **Data_inSig =** $\overline{CALL}$

7) **AddressSig =** $\overline{LW} \cdot \overline{(LW \cdot POI)} \cdot \overline{SW} = \overline{(LW + (LW.POI) + SW)}$

8) **MemRd = LW. (LW .POI) . RET . POP**

9) **MemWr = SW . CALL . PUSH**

10) **WBdata =** $\overline{(R - Type)} \cdot \overline{ANDI} \cdot \overline{ADDI} = \overline{((R - Type) + ANDI + ADDI)}$

11) **Push_popSig =** $\overline{RET} \cdot \overline{POP} = \overline{(RET + POP)}$

12) **ALUOP (AB) = A . (**$\overline{AND} \cdot \overline{ADD} \cdot \overline{ANDI} \cdot \overline{ADDI} \cdot \overline{LW} \cdot \overline{(LW \cdot POI)} \cdot \overline{SW}$ **)**

**+ B. (**$\overline{AND} \cdot \overline{SUB} \cdot \overline{ANDI} \cdot \overline{BGT} \cdot \overline{BLT} \cdot \overline{BEQ} \cdot \overline{BNE}$ **)**

$$= \mathbf{A} \cdot \overline{(AND + ADD + ANDI + ADDI + LW + (LW \cdot POI) + SW)}$$

$$+ \mathbf{B} \cdot \overline{(AND + SUB + ANDI + BGT + BLT + BEQ + BNE)}$$

**13) PCSrc1 = BGT . BLT . BEQ . BNE . RET**

**14) PCSrc0 = J-Type**

## Multi-Cycle DataPath and Control

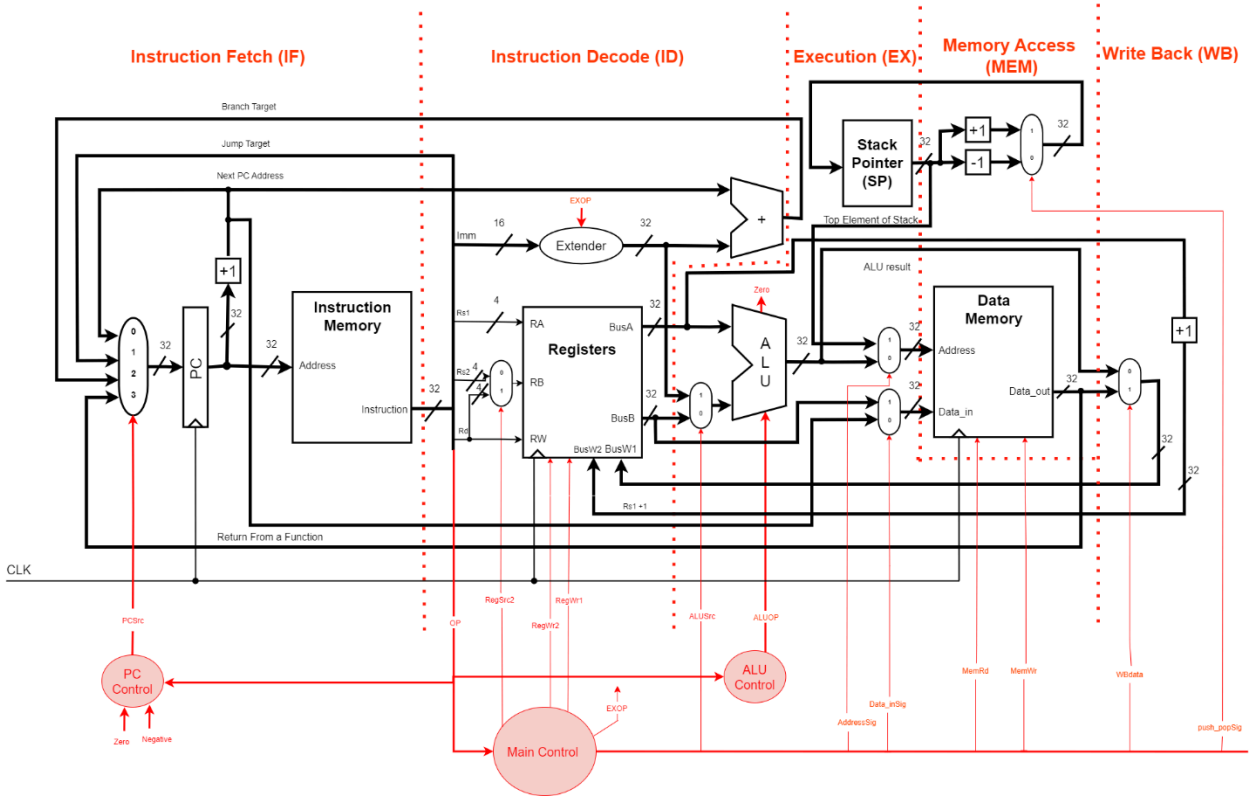


*Figure 12: DataPath and control Block Diagram*

## Instruction Sequence and WaveForms

```
        initial begin
    DataMemory[12] = 32'h00000007;
    DataMemory[14] = 32'h00000006;
    DataMemory[33] = 32'h00000007;
    DataMemory[35] = 32'h00000006;
    DataMemory[36] = 32'h00000006;
            end
initial begin

      Imemory[0] =32'b00000100010010100100000000000000; // ADD   ->  reg(1) =reg(2)+ reg(9) -
      Imemory[1] = 32'b00001100100001000000000000001000; // ANDI   -> reg(2) = reg(1) & 12 =
      Imemory[2] = 32'b00011110111000000000000000011000;   // sw the value 4 to mem[12] (fron
      Imemory[3] = 32'b00011001100101000000000000001101;  //LW.poi  (reg(6)=3) = mem[reg(5)=
      Imemory[4] =32'b00001000100101011000000000000000; // Sub reg(2) = (reg(5) = 10 ) - (reg
      Imemory[5]  =32'b00110100000000000000000000000111; //   call   Imemory[7]
      Imemory[6] = 32'b00100010101011000000000000001000;    //BGT-->taken<---  (reg(10)=12  -
      Imemory[7] =32'b00111000000000000000000000000000; // return   to Imemory[5]+1 = 6
      Imemory[8] =32'b00001100100001000000000001001000;  // just to check  brnash is taken
      Imemory[9]=32'b00111110000000000000000000000000;  // PUSH Reg(8) = 6
      Imemory[10]=32'b00000110001000100100000000000000;    // add then the value in R7 -> reg
      Imemory[11]=32'b01000001110000000000000000000000;  // pop to Reg7 ->  Rd =  6

  end
```

*Figure 13: Instructions*

As seen in the figure above, a sequence of  different instructions are in the memory. These instructions include R-type, I-type, J-type and S-type. First the PC is initialized to point at the first instruction, so in the fetch stage the first instruction in the memory is fetched which is a “ADD” instruction with opcode “000001”.

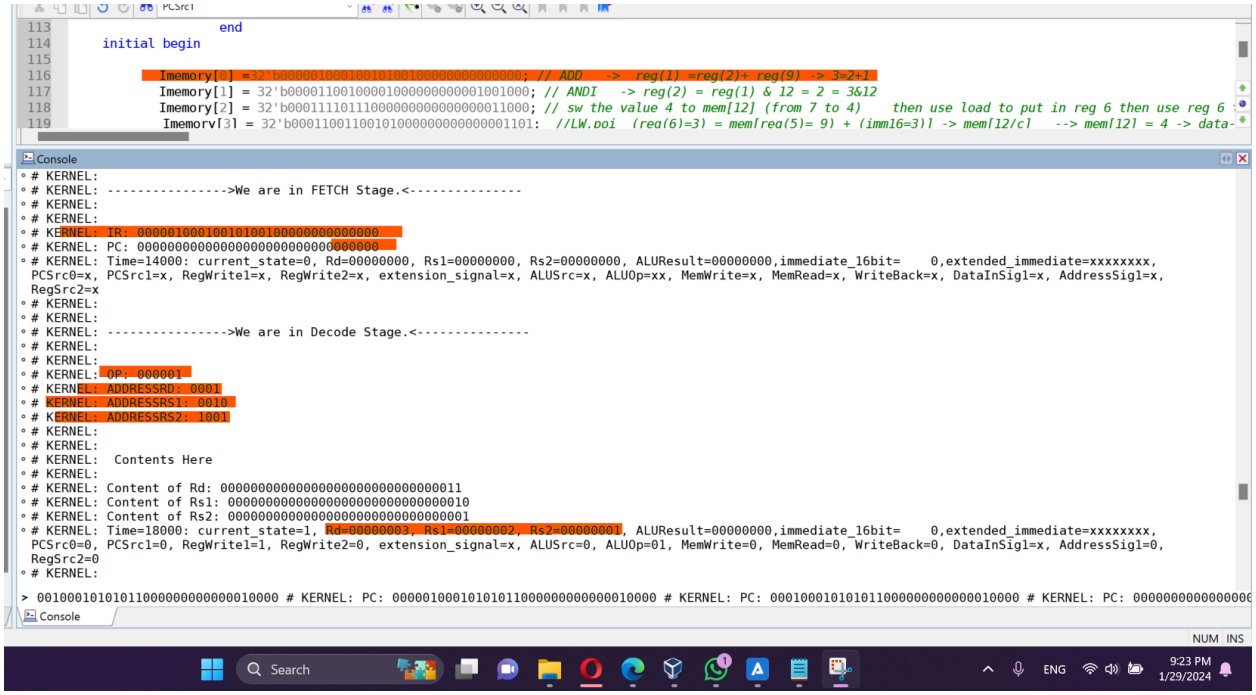


*Figure 14:*

As seen in the figure above, the opcode is “000001” which is the “add” instruction. In the decode stage, the fields are assigned based on the opcode. The destination address is “0001”, thus the result of the add instruction should be stored in R1. The two operands are in the registers R2 “0010” and R9 “1001” respectively, having the following content: 0x2 and 0x1 respectively.

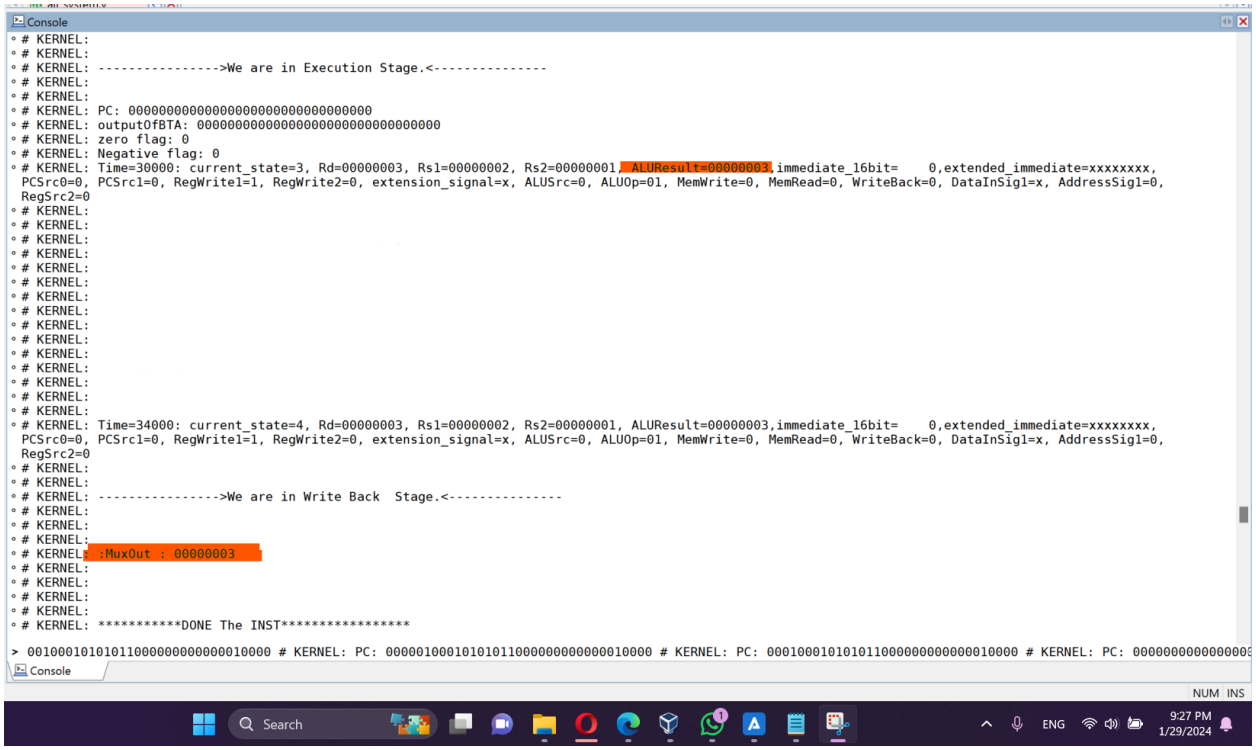


*Figure 15:*

As seen in the figure above, the execution stage is where the addition between the two operands happens. The values 0x2 and 0x1 are added and shown in the “ALUResult” which is 0x3. The “add” instruction does not go to the memory stage, but instead goes directly to the write back stage where the result 0x3 is written back into the register file. The “Muxout” takes the value from the ALU result and is printed on the screen.

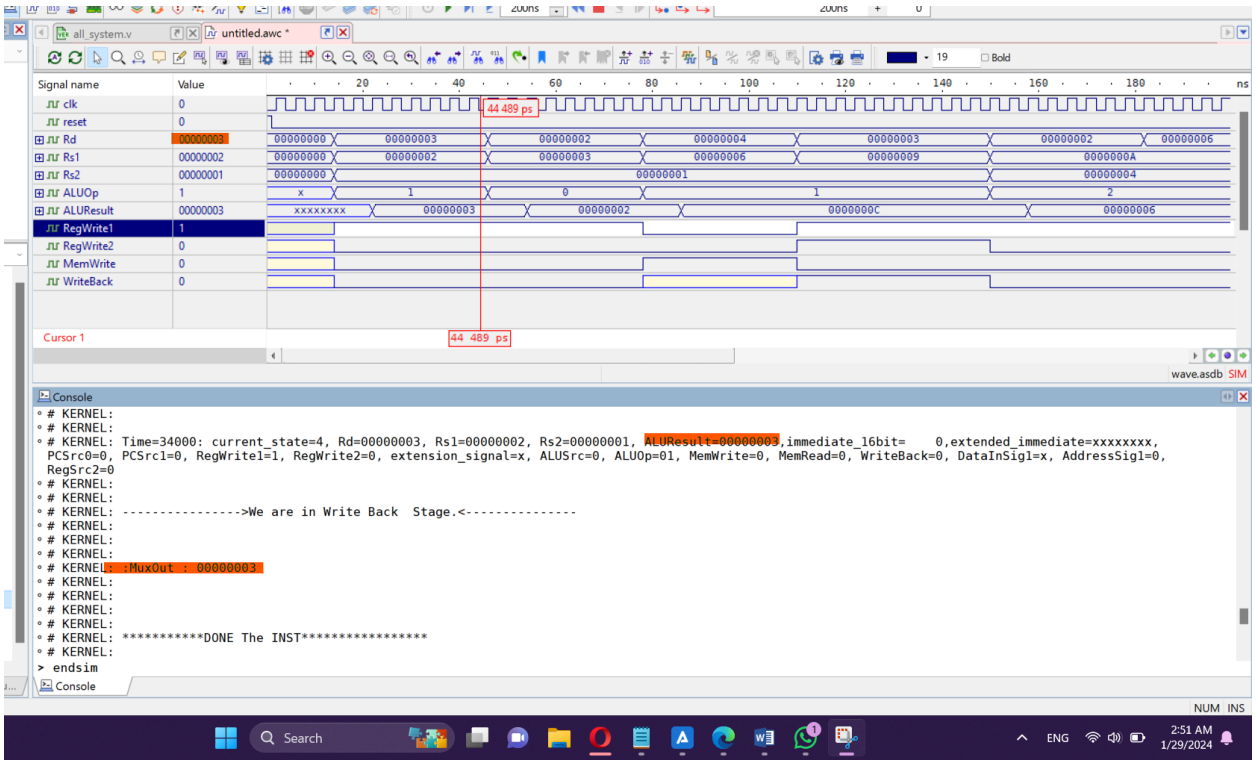


*Figure 16:*

This waveform is a clear display of the instruction “add” with some signals to show how it works. The ALUResult is in fact correct, the signal RegWrite1 is set in order to write back the result in the register file. Since the memory is not being used, both “MemRead” and “MemWrite” are 0’s. Moreover, the signal “WriteBack” is given the value 0 since the output is from ALU not from memory.

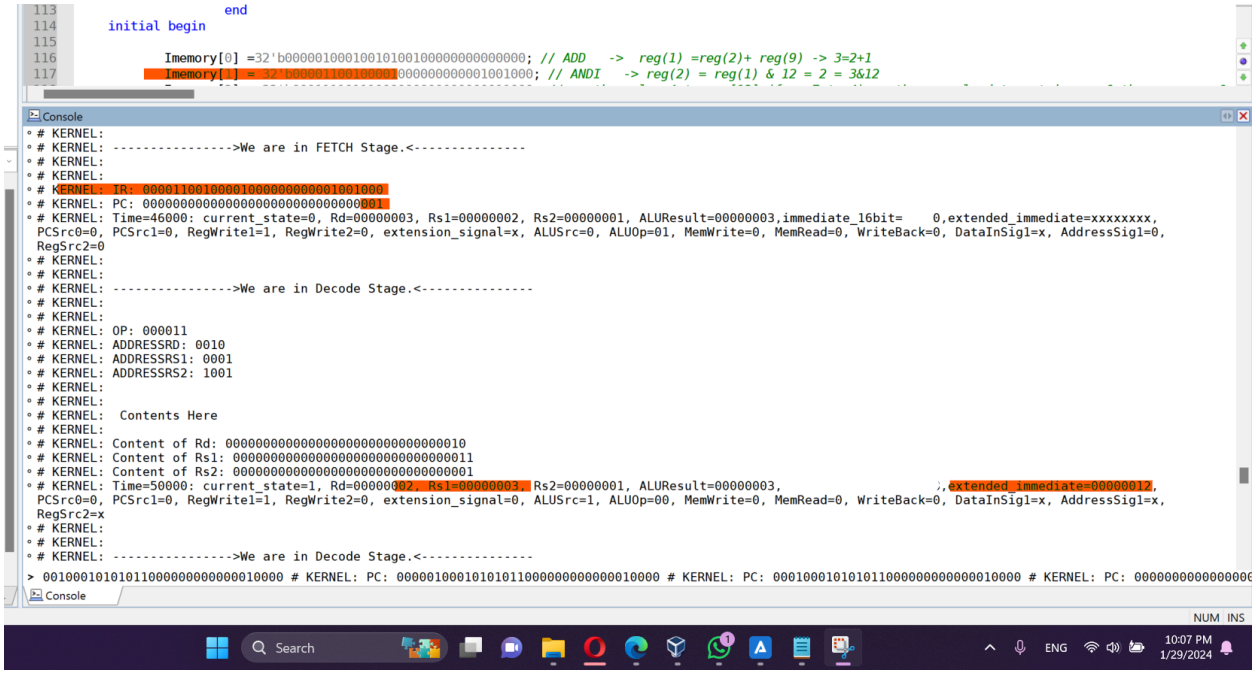


*Figure 17:*

The PC is incremented by one as seen in the figure, and now the next instruction is fetched with opcode “000011” which is the “ANDI” instruction. In the decode stage, the address of Rd is “0010” which is R2, and the address of the first operand is “0001” which is R1. The extended immediate is 0x12 which is the signed extension of the 16 bit immediate in the IR = “0000000000010010” mode =”00”.

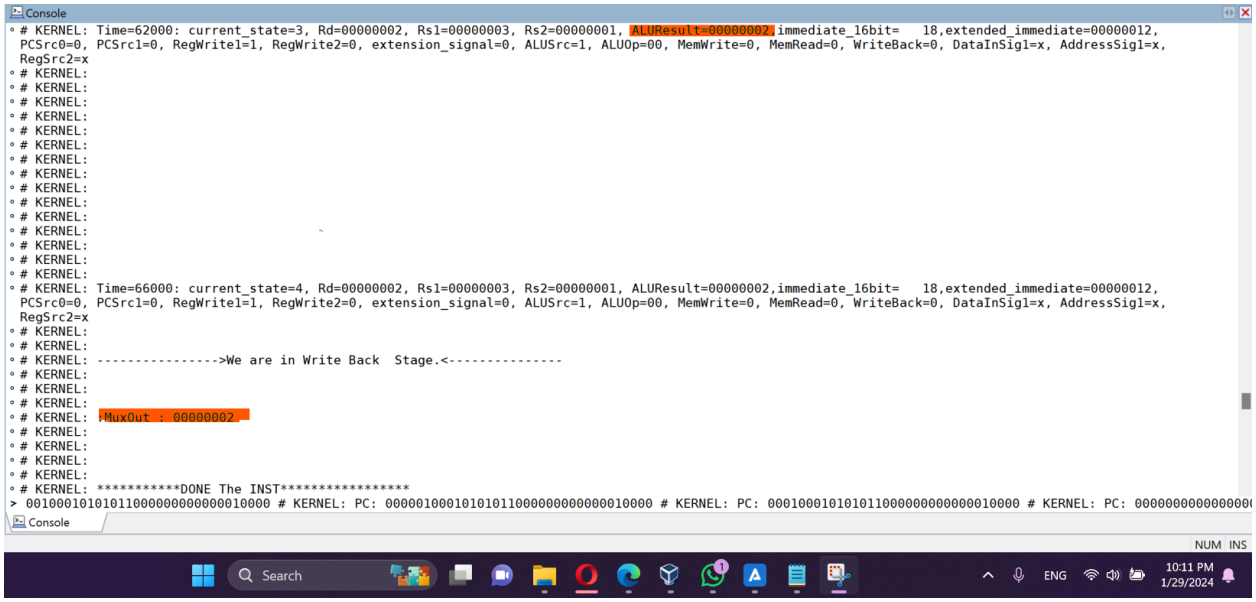


*Figure 18:*

In the Execution stage, the first operand and the signed immediate go through the “AND” operation. As seen in the figure above, the content of Rs1 is 0x3 and the signed extended immediate is 0x12, the “Anding” between these two numbers gives the value 0x2, which is also going to be the output of the mux in the write back stage. This result is stored in the destination register Rd or R2.

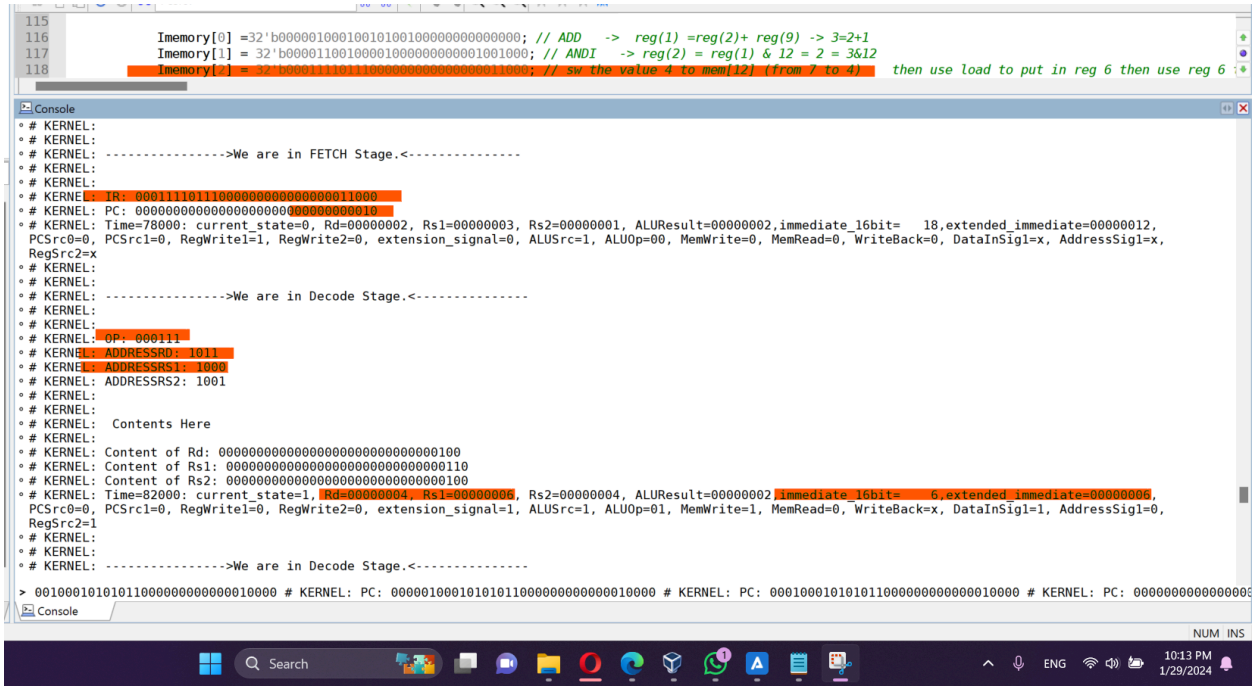


*Figure 19 :*

The next instruction is fetched and the PC is incremented by one. In the decode stage, we can see that the instruction is a “store” instruction. In the store instruction the content of Rd is stored in the data memory with the address calculated in the ALU, taking the content of Rs1 and adding it to the signed immediate value to calculate the address. In this stage, we can see Rs1 = 0x6 and the extended immediate is = 0x6.

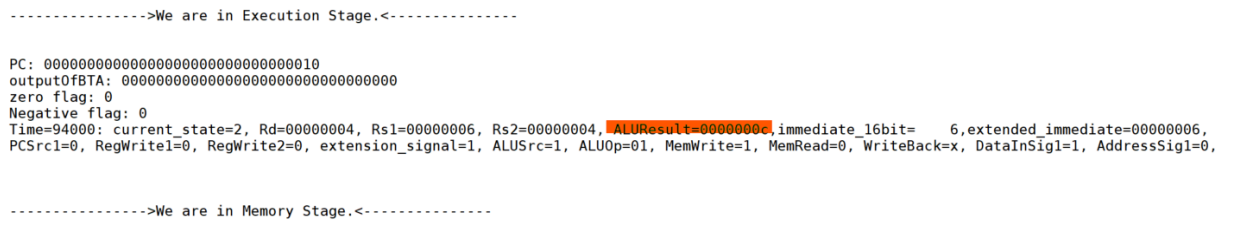


*Figure 20 :*

In the execution stage, the address is calculated taking Rs1 = 0x6 and adding the extended immediate is = 0x6 gives a result of 12 in decimal which is 0xC in hexadecimal. This is shown in the figure above.

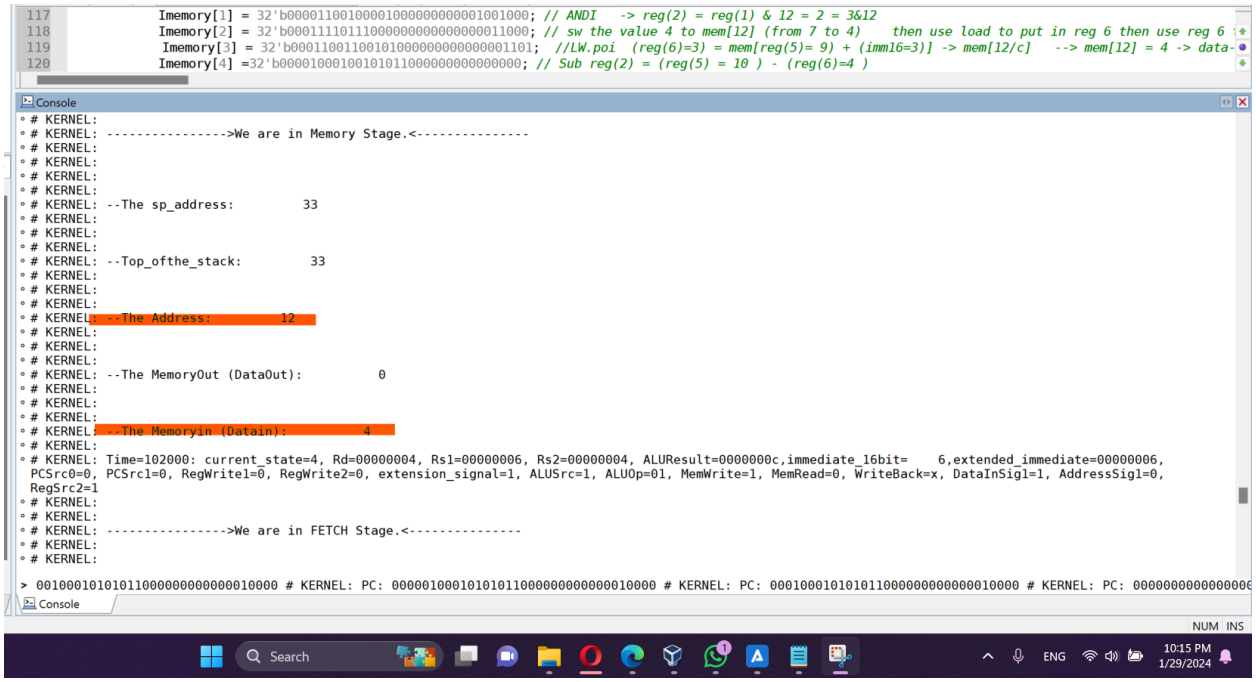


*Figure 20:*

This is the memory stage where the data will be stored in address 12 of the memory. The data, taken from the register file as mentioned in the decode stage, has the value of 0x4.

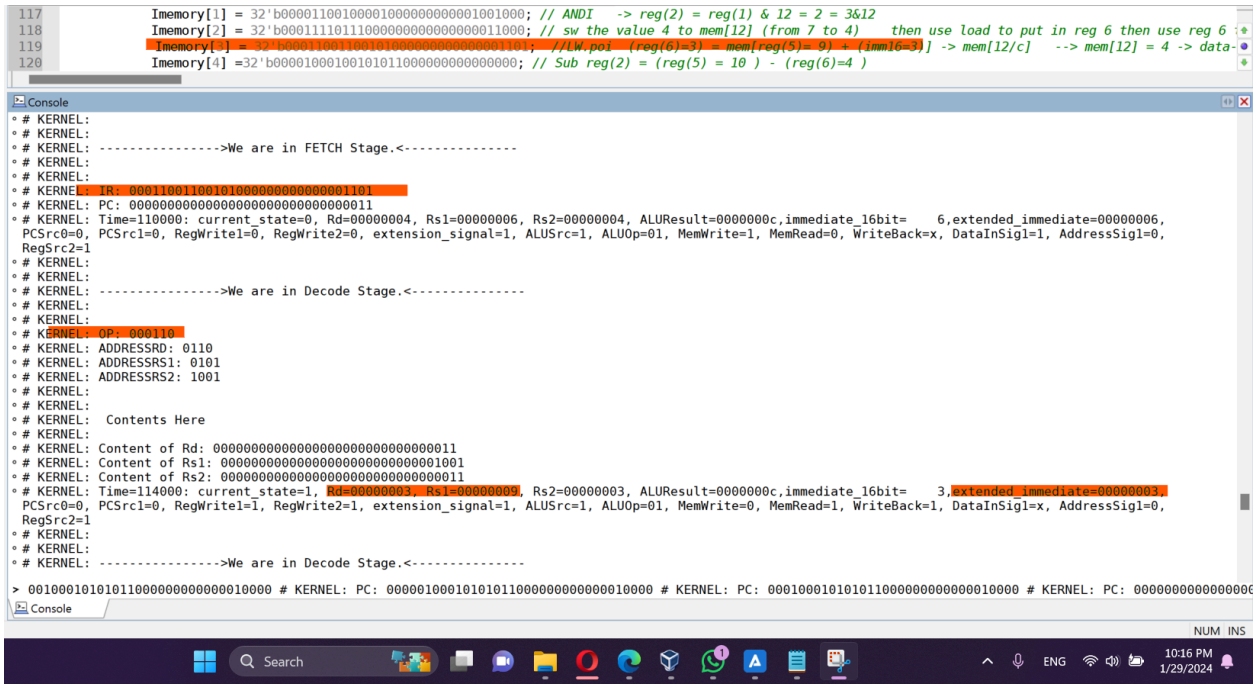


*Figure 21:*

The next Instruction is LW.POI, which has the opcode “000110”. The destination register is “0110” which is R6, the first operand (Rs1) is stored in “0101” which is R5, and the second operand is the signed extended immediate value = “0x3”.

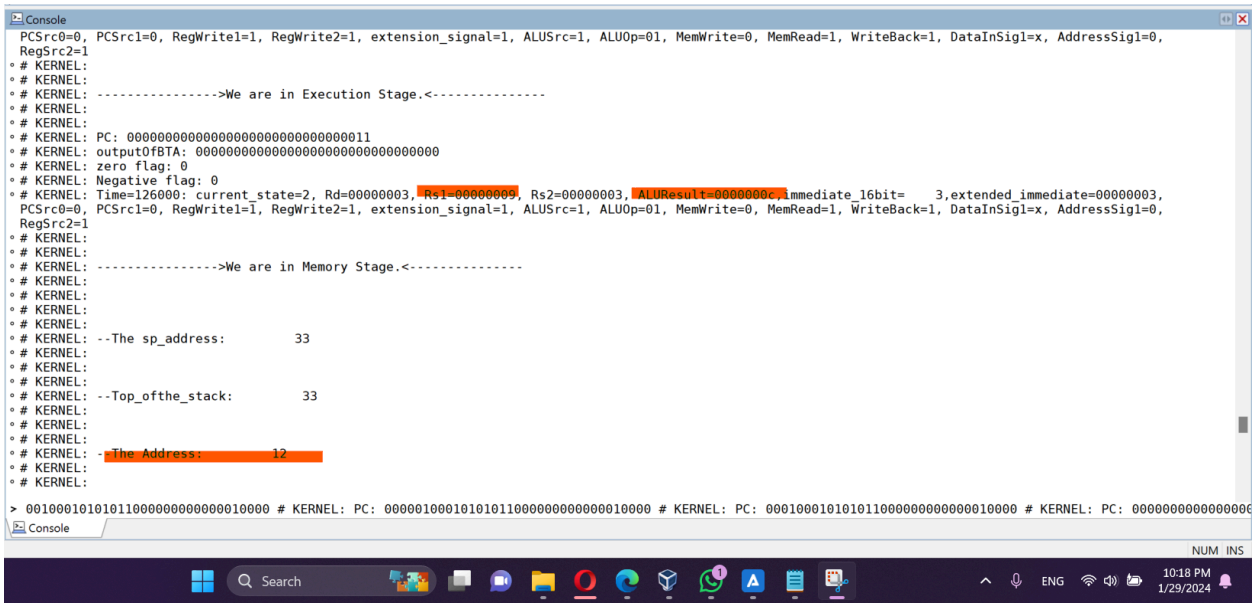


*Figure 22:*

In the execution stage, the address of the memory is calculated by adding the 32 bit register = “0x00000009” with the sign extended immediate = “0x3”. When adding these numbers, the result is 12 in decimal or “0xC” in hexadecimal. Thus, this instruction will load a 32 bit data from memory location with address “0xC”.

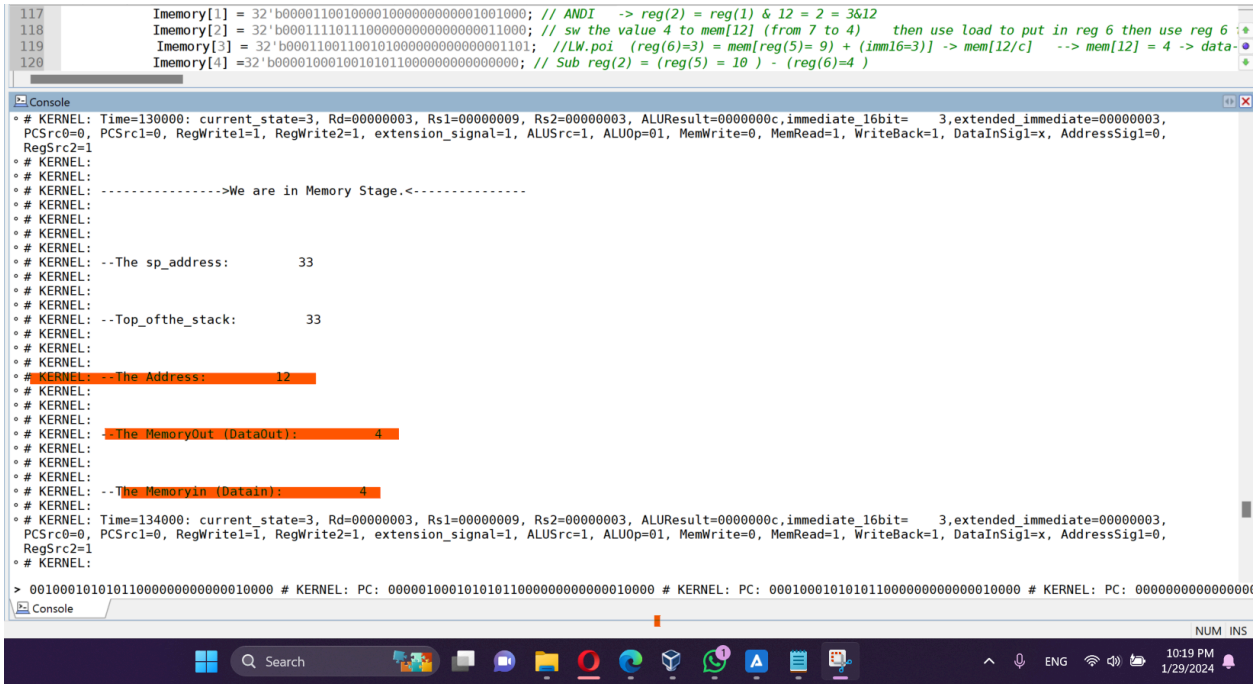


*Figure 23:*

Figure 23 shows the address and the output data that came from the specific memory location “0xC”. As seen the output data has a value of 4.

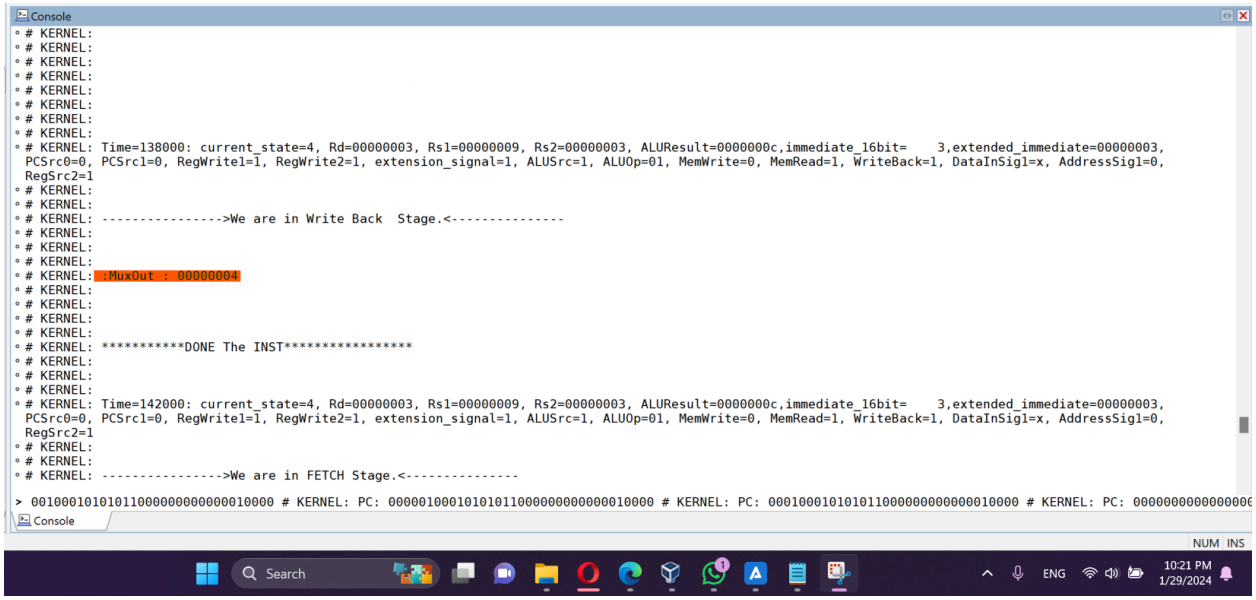


*Figure 24:*

The load instruction will go through all five stages. The write back stage includes a mux that chooses between ALU result or memory output. In this case it will choose the memory data output which is in fact 4.

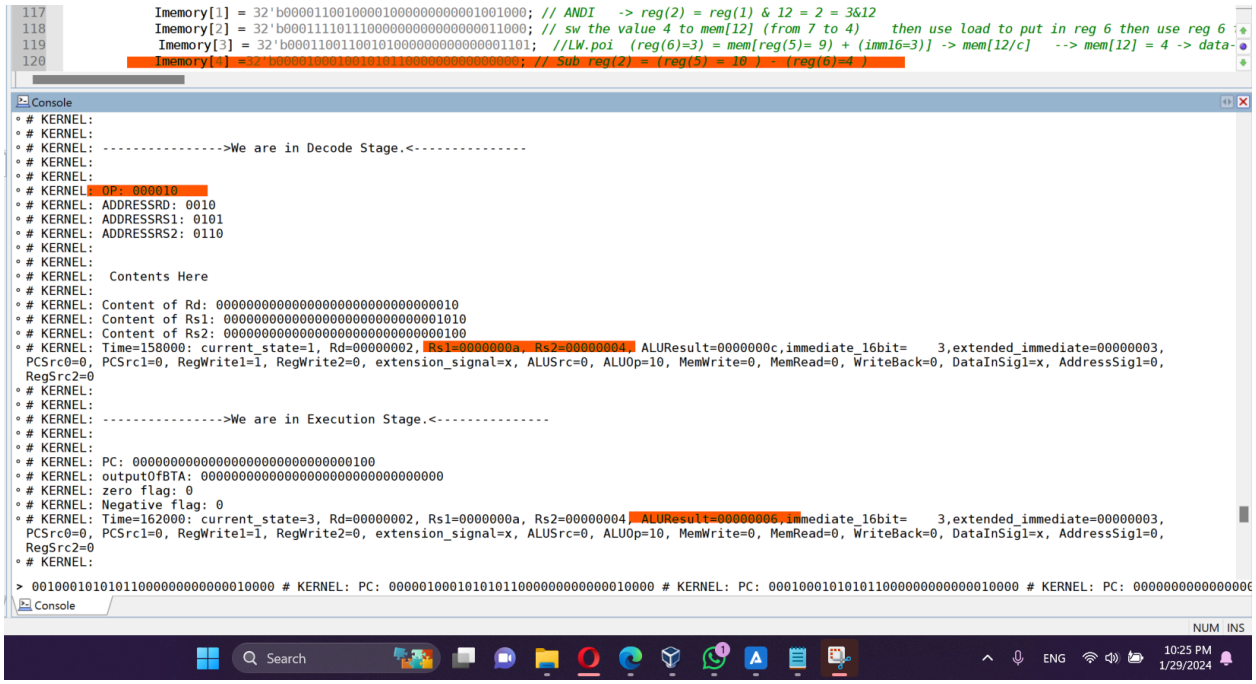


*Figure 25:*

Now a new instruction is being loaded into the register file, but this time Rs1 that was used in LW.POI is being used again to confirm that the value of Rs1 was in fact incremented by 1 in the previous instruction during the write back stage. In the previous instruction Rs1 was R5 which had the content: “0x00000009” and now we can see in the next instruction the value of Rs1 is “0x0000000a”; Therefore, we can guarantee that LW.POI instruction went through all five stages correctly. Figure 25 shows the decode stage for the next instruction that has the opcode = “000010” which is the subtraction instruction. The two operands have the value of = “0xa” and “0x4”. In the Execution stage, they are subtracted given the result = “ 0x6” and is then moved to the write back stage where this value is stored in the destination register.

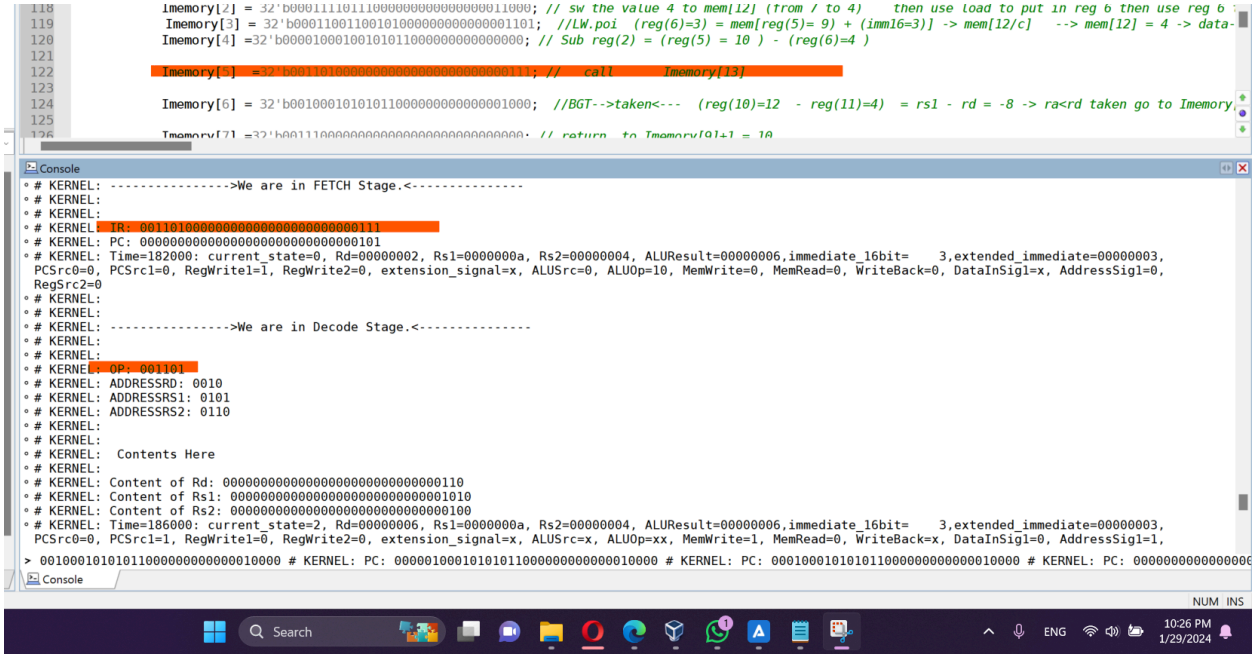


*Figure 26:*

The Next Instruction has the opcode = “001101” which is the call instruction. This instruction takes the 4 most significant bits of the PC and concatenates it with the 26 bit offset. Then, the PC + 1 is pushed onto the stack.

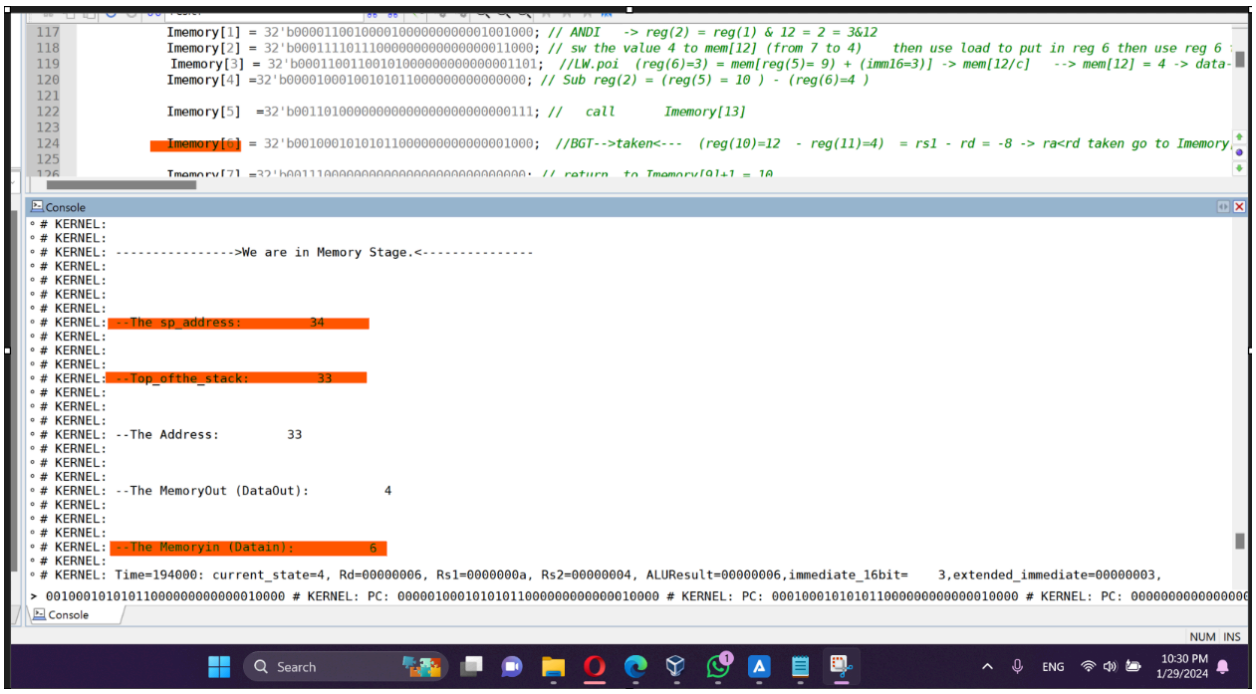

*Figure 27:*

In the memory stage, the address of the top most empty element has the address of = 34 in decimal. The top non-empty element of the stack is 33 in decimal. As seen in the figure, the "DataIn" (data that is pushed in the stack) is the value of PC + 1. 6 is the next instruction which is the BGT instruction, however it will skip that instruction to call and move to a function. Then, the "return" instruction will set back the PC where it was before calling on a function as can be seen in the below figures.

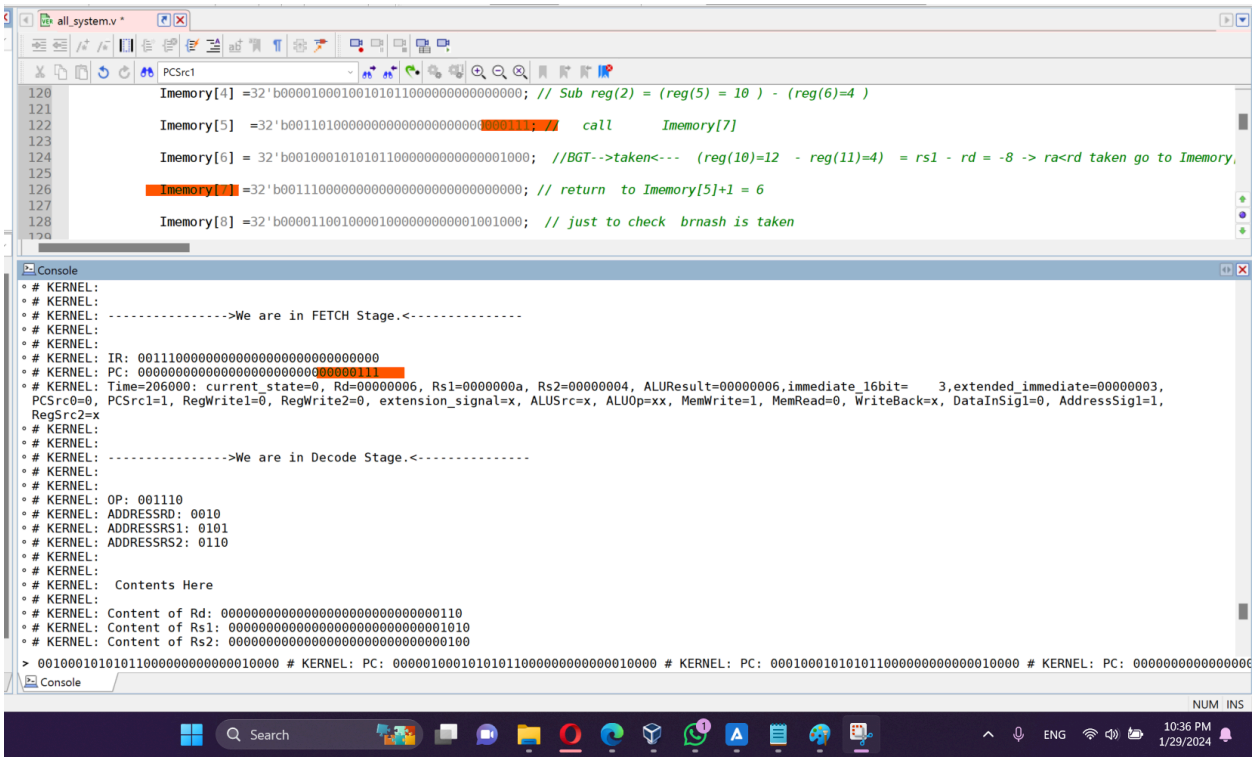


*Figure 28:*

Figure 28 shows the instruction fetch of the "return" instruction with opcode "001110". The return instruction will take the value of the top of the stack and store it in the PC. The top of the stack is "PC + 1 = 6 " which is the branch instruction.

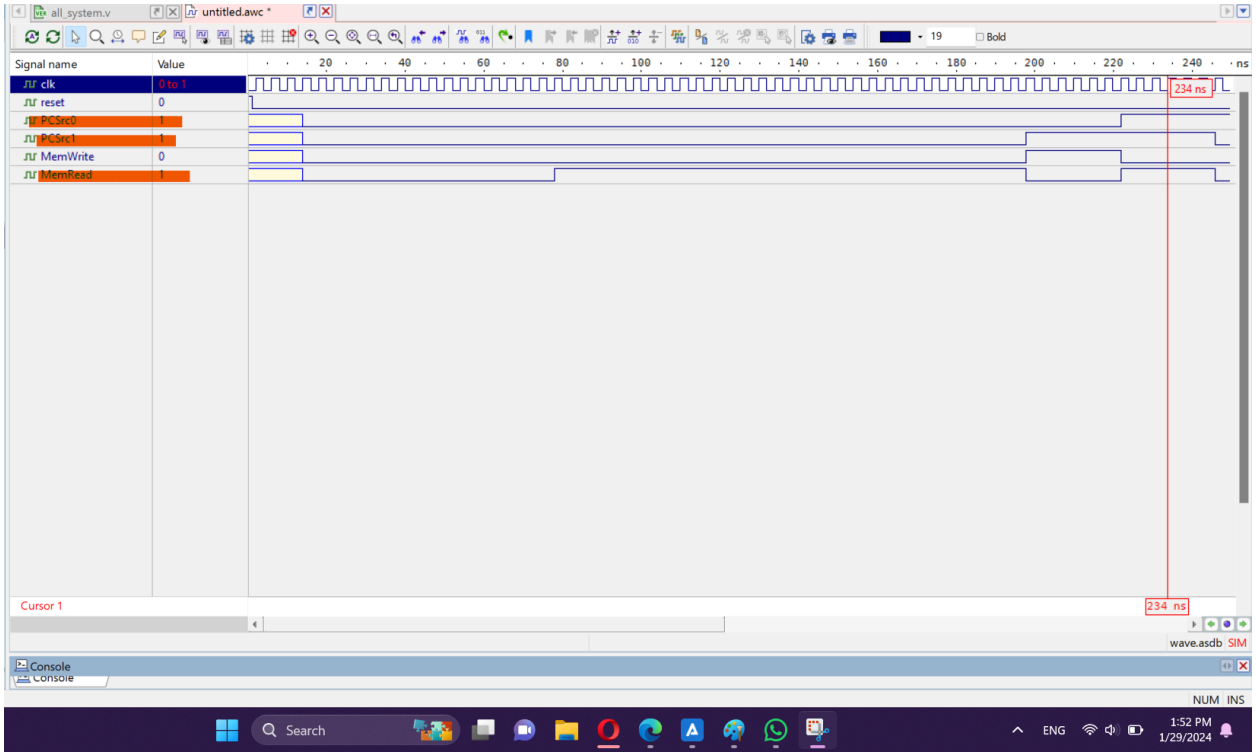


*Figure 29:*

Figure 29 shows the values of the selection line of the PC Mux that will choose the values =”6” which is the “return” PC, meaning the new value of the PC = top of the stack.

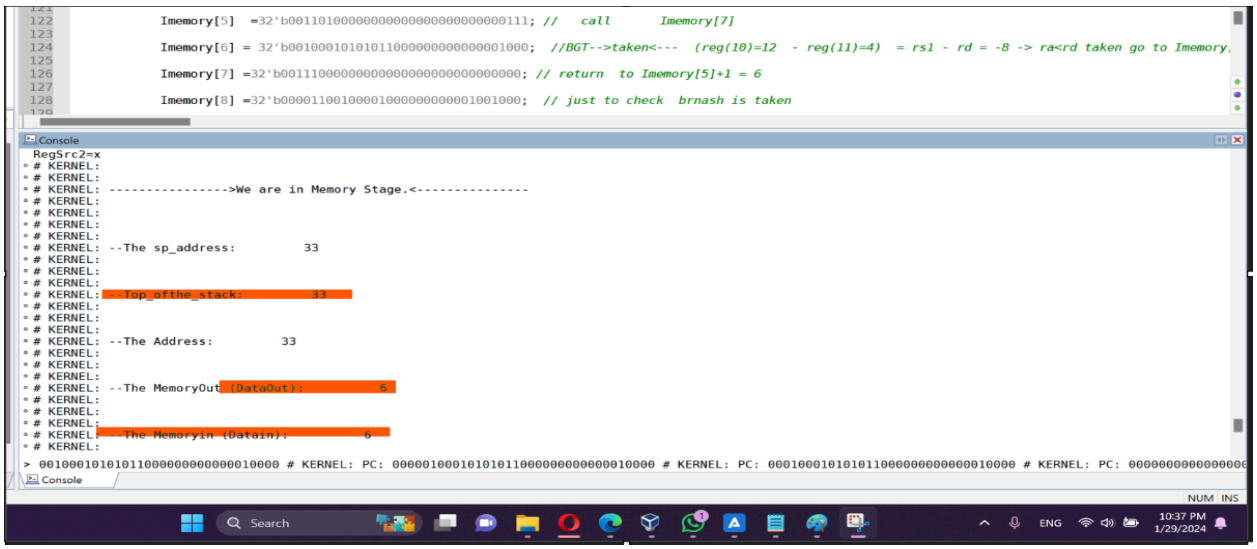


*Figure 30:*

As seen in Figure 30, the top of the stack is popped and the Data out is in fact “PC+1”. Now it will go back to where it left off before calling the function.

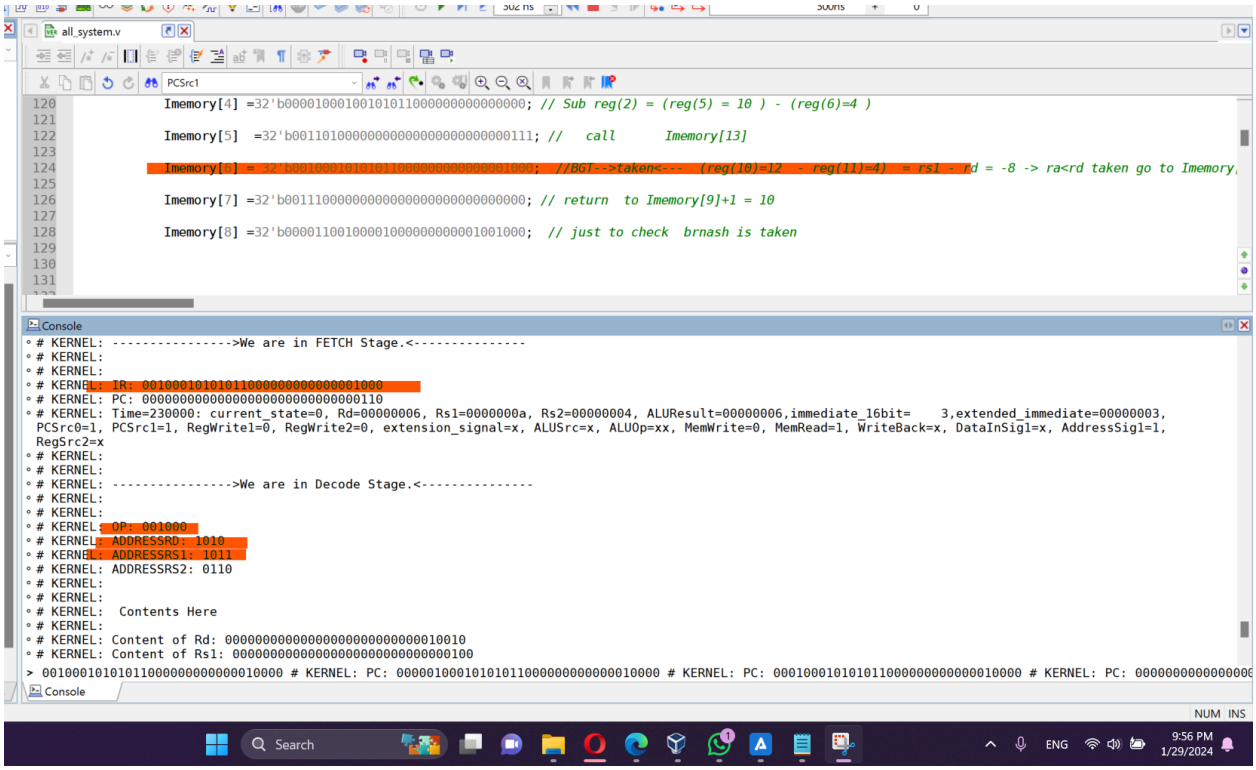


*Figure 31:*

Figure 31 shows the instruction fetch of a branch type instruction with opcode = “001000”. This branch instruction will branch to the BTA (branch target address) if the first operand Rd is greater than the second operand Rs2. The two operands being compared here are in the registers: “1010” and “1011” respectively which have the content “0x12” and “0x4” respectively.

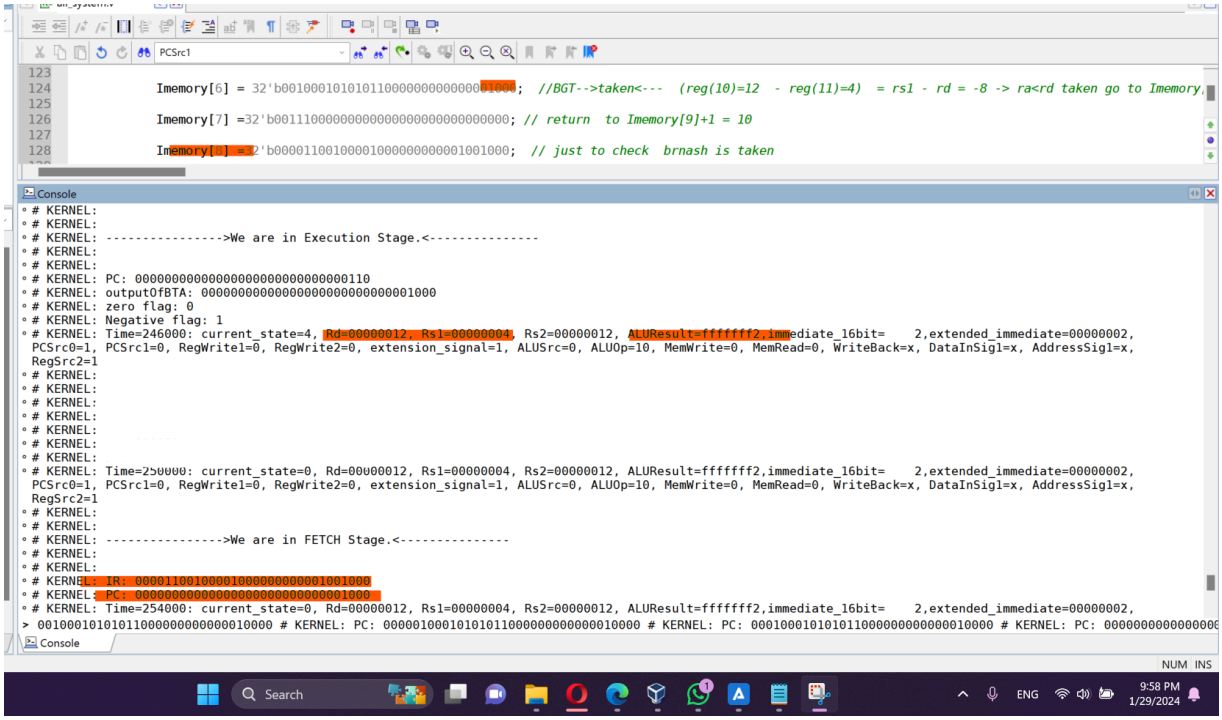


*Figure 32:*

Figure 32 shows the execution stage, these 2 numbers are subtracted in this order = 0x12 - 0x4. Since the result is a positive number then that means Rd is in fact greater than Rs1 and thus the branch is taken. Now it will branch to the address PC+ signed Immediate. Since the PC is 0x6 and the signed immediate is 0x2, then the BTA is 0x8. Therefore the next instruction should be at PC = 0x8 which is an “Add” instruction.

Here are some Define Instructions in the Instruction memory that were used as test cases:

```
initial begin

        Imemory[0] =32'b00000100010010100100000000000000; //
        Imemory[1] = 32'b00001100100001000000000001001000; //
        Imemory[2] = 32'b00011110111000000000000000011000;
        Imemory[3] = 32'b00011001100101000000000000001101;  //
        Imemory[4] =32'b00000100010010101100000000000000; //
        Imemory[5]  =32'b00110100000000000000000000000111; //
        Imemory[6] = 32'b00100010101011000000000000001000;
        Imemory[7] =32'b00111000000000000000000000000000; //
        Imemory[8] =32'b00000110010000100000000001001000;  //
```

Here are some Data in the Data memory:

```
        initial begin

    DataMemory[12] = 32'h00000007;
    DataMemory[14] = 32'h00000006;
    DataMemory[33] = 32'h00000007;
    DataMemory[35] = 32'h00000006;
    DataMemory[36] = 32'h00000006;

            end
```

Here is the register file that consists of 16 registers:

```
//*********************************************************************************
    initial begin
// 16 32-bit general-purpose registers: from R0 to R15.
    Registers[0] = 32'h00000000;
    Registers[1] = 32'h00000003;
    Registers[2] = 32'h00000002;
    Registers[3] = 32'h00000011;
    Registers[4] = 32'h00000006;
    Registers[5] = 32'h00000009;
    Registers[6] = 32'h00000003;
    Registers[7] = 32'h00000005;
    Registers[8] = 32'h00000006;
    Registers[9] = 32'h00000001;
    Registers[10] = 32'h00000012;
    Registers[11] = 32'h00000004;
    Registers[12] = 32'h00000017;
    Registers[13] = 32'h00000020;
    Registers[14] = 32'h00000026;
    Registers[15] = 32'h00000032;

       end

 /////////////////
```

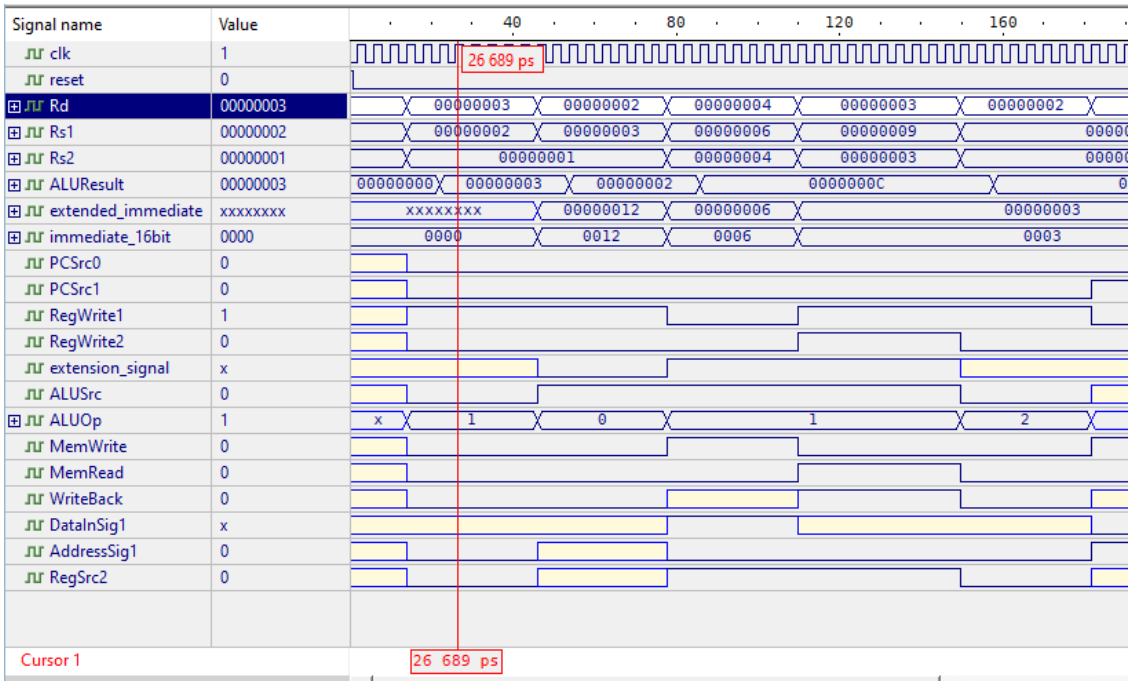

*Figure 33:*

Figure 33 shows the waveform of instructions that were executed in our program. The cursor shows the first instruction which is the “Add” instruction. As seen the two operands 0x2 and 0x1 are added and shown in the ALU_result.

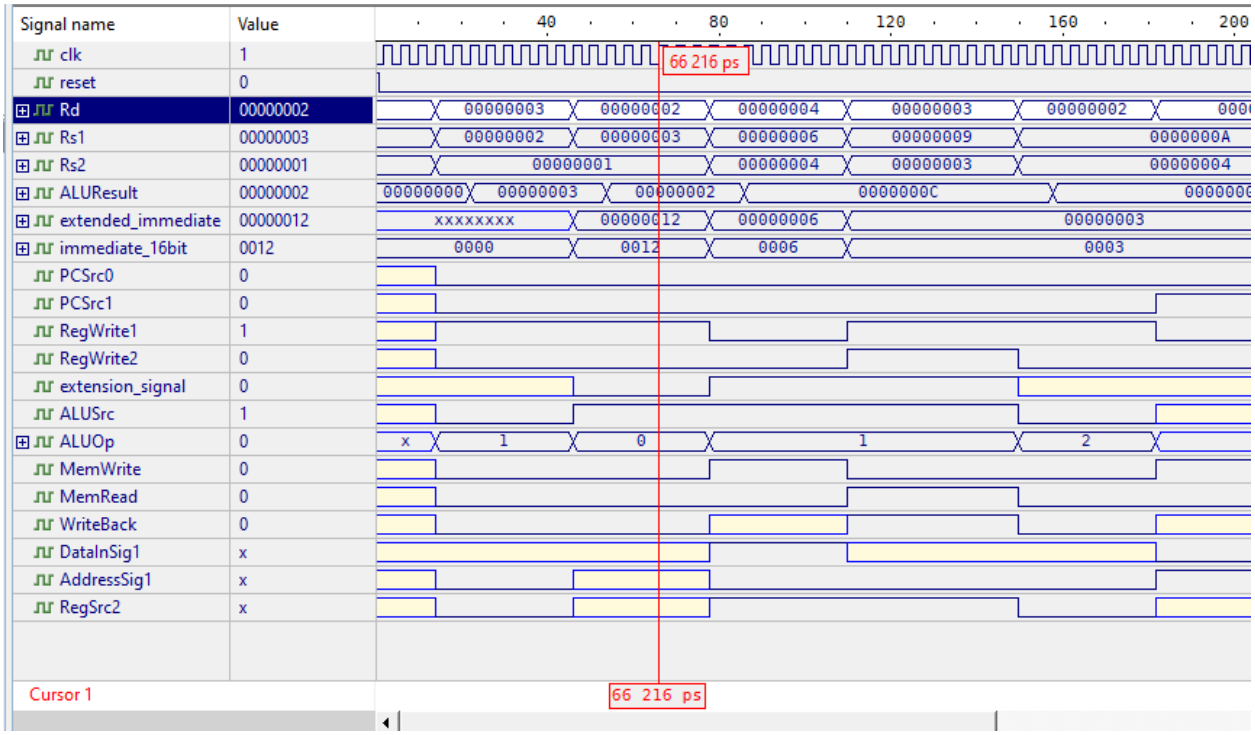


*Figure 34:*

Figure 34 shows the cursor that points at the second instruction which is the “ANDI” instruction. As seen the two operands 0x3 and the immediate (0x12) went through “AND”operation and the result shown in the ALU_result = (0x2). The signals are also shown where PC selection lines are “00”, RegWrite1 is set to 1 to write back in the register file, ALUSrc is also set to choose the extended immediate as the second operand.

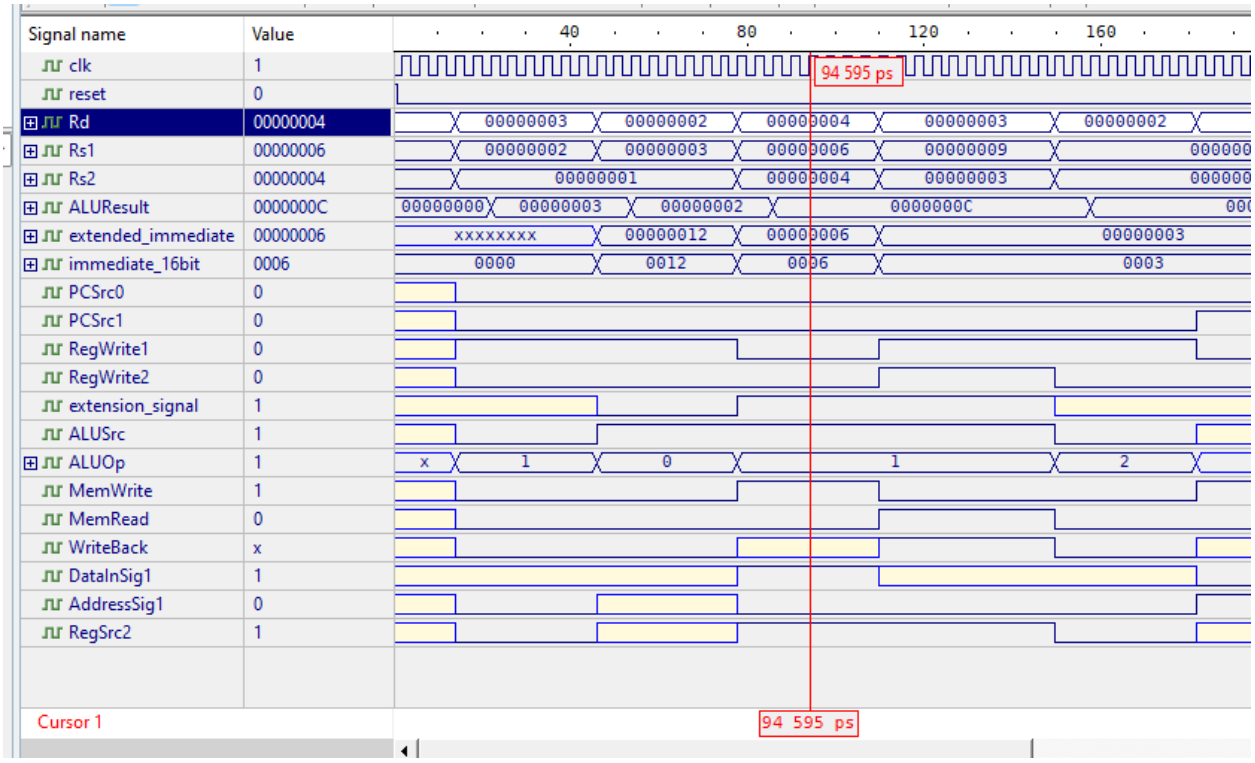


*Figure 35:*

Figure 35 shows the cursor that points at the third instruction which is the “SW” instruction. The signals are shown above as well with MemWrite set to 1 and WriteBack signal is “x” don't care, since the RegWrite1 signal is 0.

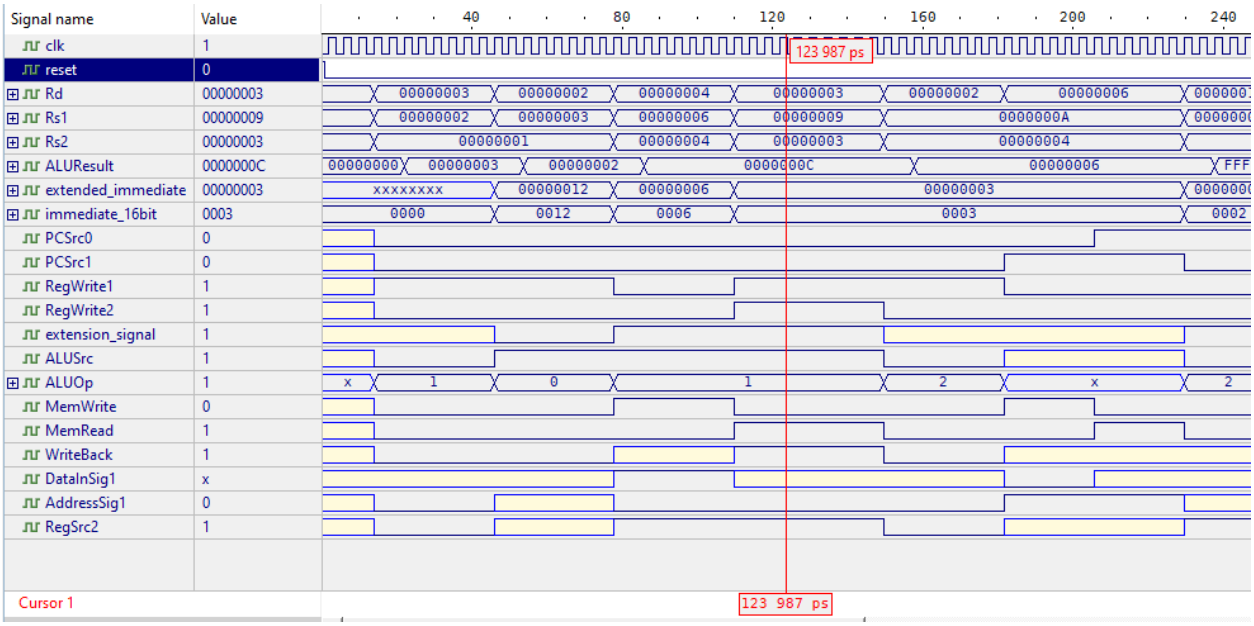


*Figure 36:*

Figure 36 shows the cursor that points at the fourth instruction which is the “LW.POI” instruction. The signals are shown above as well with MemWrite disabled to 0 and WriteBack signal is 1, and both RegWrite1 and RegWrite2 are set to 1 since we will write back at both the destination register and the new value of Rs1.

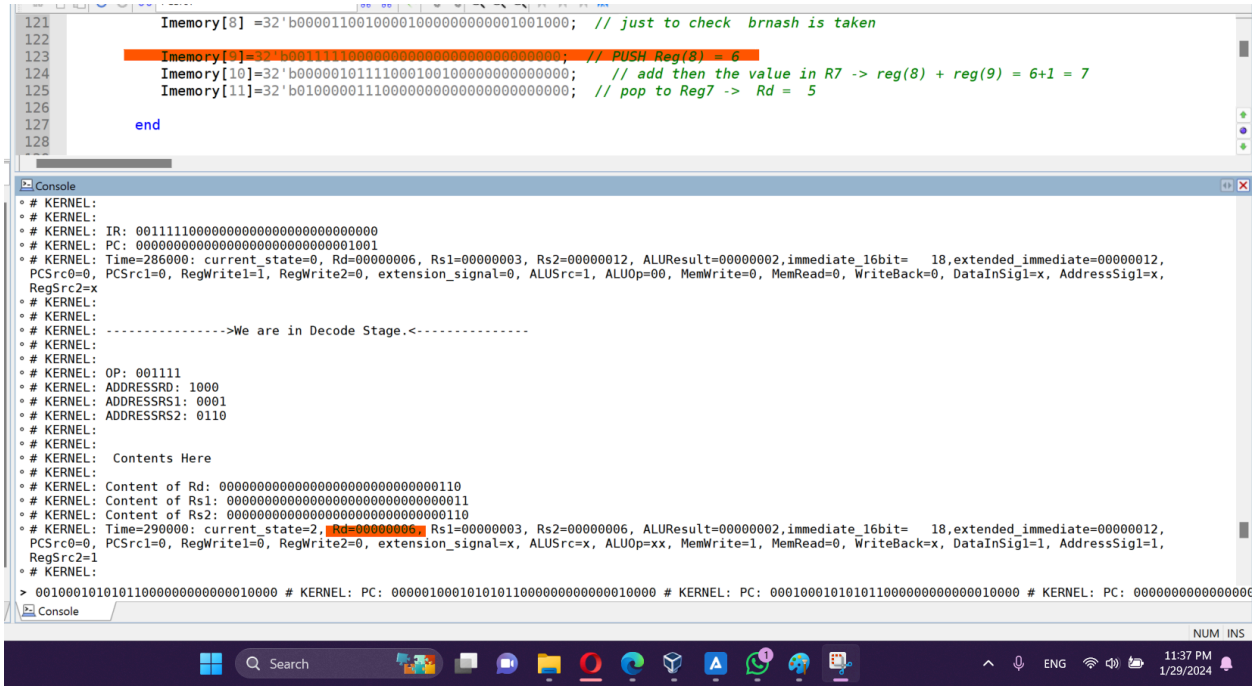


*Figure 37:*

Figure 37 shows the “push” instruction with opcode “001111”. The value of Rd is pushed onto the stack in the position determined by the stack pointer address. As seen in figure 37, the value 0x6 is the data to be pushed on the stack.

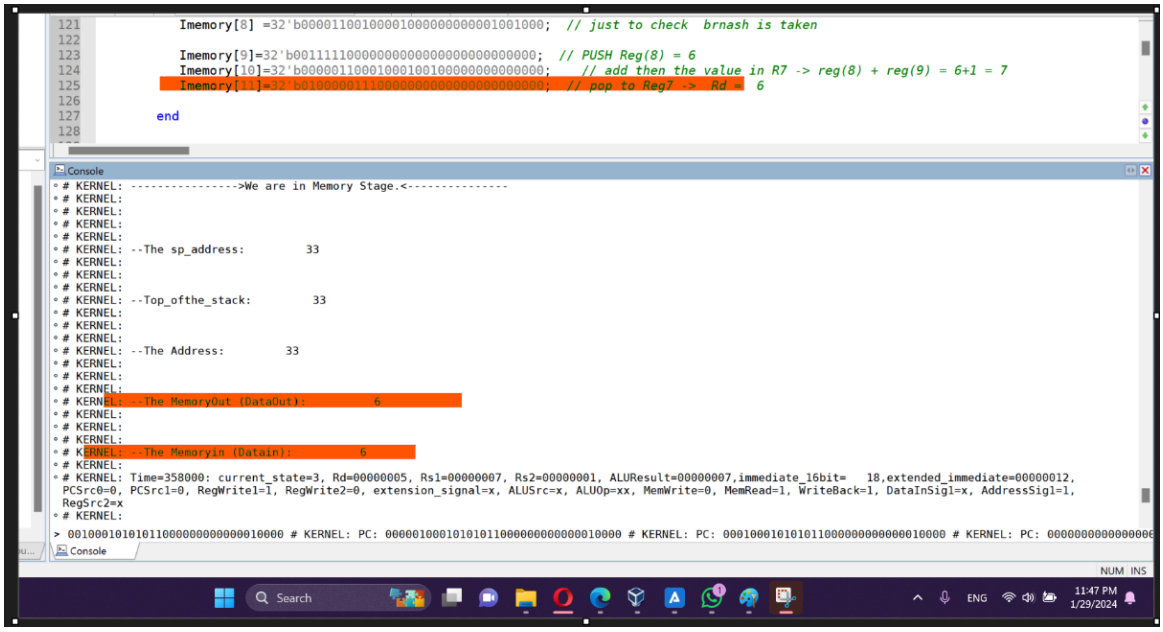


*Figure 37:*

Figure 38, shows the “pop” instruction. To show that both the “push” and “Pop” instructions work, we popped the pushed element from the previous instruction, which is the value 6.