Faculty of Engineering & Technology

Electrical and Computer Engineering Department

COMPUTER ARCHITECTURE

ENCS4370

Project 2

Prepared by:

Mariam Hamad 1200837 section **2**

Leena Affouri 1200335 section **2**

Amany Hmidan 1200255 section **1**

Instructor: Dr. Aziz Qaroush

Date: 29/1/2024

Table of Contents

List of Figures

Table of tables:

1 – Design and Implementation

A multi-cycle CPU is a type of processor that give each instruction multi cycles as it needed to execute not more not less for example the load instruction need 5 stages to execute it will give it 5 clock cycle to finish its work but the Jump instruction need 2 stages it will give it 2 clock cycle and when the instruction finish its work the system will let the next instruction to enter and that will increased complexity of the design in cooperation with the single cycle which give each instruction just one clock cycle to executed the time for each instruction is equal so instruction like Jump will take more clock cycle than it need and this will make a lot of stall cycle and make the processer slower. So that makes the multicycle better than the single cycle but worse than the pipelining, which is letting the next instruction enter when the previous instruction passes to the next stage. In a clock cycle, the whole path will have five instructions, each one in a stage, but it also makes the design more complex and has a hazard problem. In summary, each processer has benefits and weaknesses.

Our data bath is multi-cycle. To build it, we have to use the single cycle data path, then add register buffers between two stages to store the output from the previous stage, so it will write on the buffer to make the next stage read this output and use it, then store its data on the next buffer, and so on to save the data from being lost when a new clock cycle comes along. To be more clear, reading and writing on the buffer will cost a time, so the multi-cycle will not be exactly 1/5 cycle time of the single cycle, but it is still better than the single cycle. Although execution of the load on the single cycle will be better, we can't judge just from one case that the single cycle is butter.

Data Path:

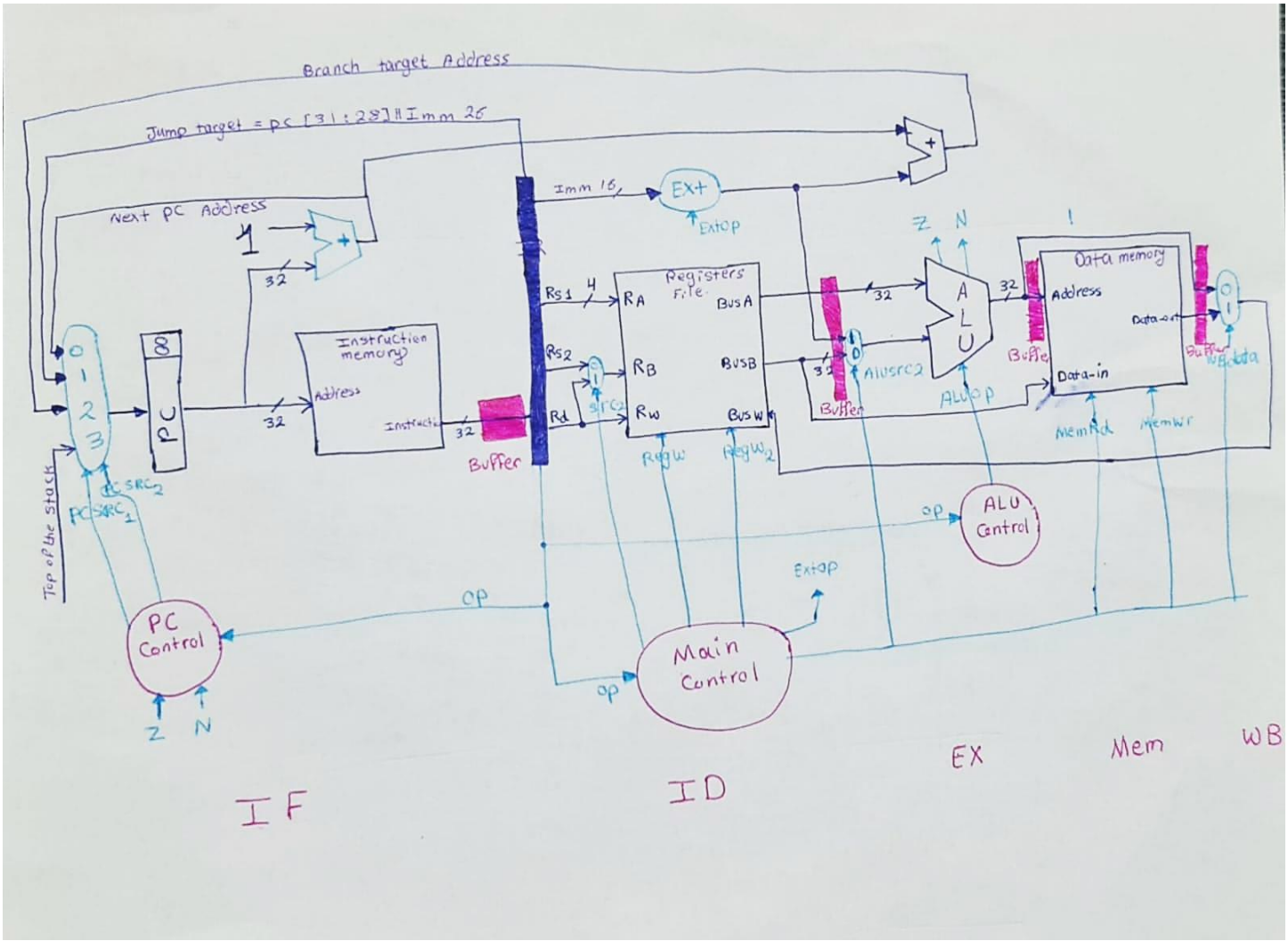


Figure 1: Data Path.

The data path has five stages, each stage takes one clock cycle:

Fetch stage: the instruction stored in the instruction cache memory. In this stage, the instruction is fetched into the control unit of the CPU by supplying the memory with the address of the respective instruction.

Decode stage: in this stage, the instruction type is known from the instruction opcode; also, take from the IR the instruction needed register. Each instruction type uses a different register, but the values in these registers are then taken by the buses as an input to the ALU.

Execution stage: here its input could be two registers or register with immediate value it depends on the instruction type. In this stage, it will execute the instruction operation and change the flags values if needed, then its output will be the input to the memory access.

Memory Access: cache memory is separated into two memory instruction memory in the fetch stage, and the other is data memory, which is here in this stage. This memory is used to store the data. The only instructions that deal with this stage are store and load, store to write on the memory and load to read from the memory.

Write back stage: not all instructions reach this stage; just the instructions that need to write back the data on the register reach here. It even takes the output value from the ALU and writes the data in the second stage on the register, or takes the memory access output if the instruction is loaded and writes the data in stage two on the register.

PC mux:

In our data path we use four muxes first one 4x1 mux it has two selection lines PCsrc0 and PCsrc1 if the value of the two selections is 00 then the output of the mux is the Pc +1 which is next Pc address, else if the two selections value is 01 then the output will be Jump target address which is Pc concatenated with the immediate 26, else if the two selections value is 10 then the output will be the Branch target address which is the Pc address added with the immediate 16 after it get extended to 32 bit, else if the two selections value is 11 then the output will be the top of the stack the address that saved on the SP register .

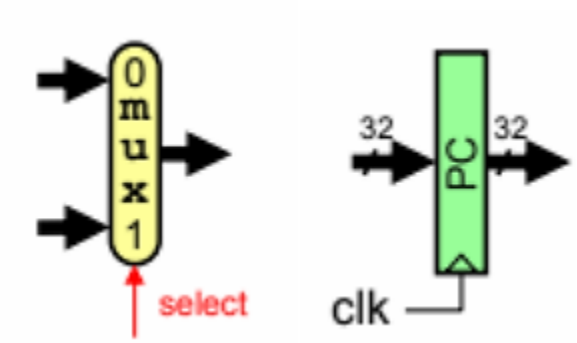


Figure 2: PC Mux.

Instruction memory:

The output of the mux is the input of the PC, which will make the value of the register Pc equal the output of the mux, then enter the memory in the fetch stage to take the instruction to the address that is the PC register. The instruction memory is a cache memory that saves the instruction on it. It takes the address that was saved on the PC as an input and then gets the instruction out to the IR, which stores the instruction.

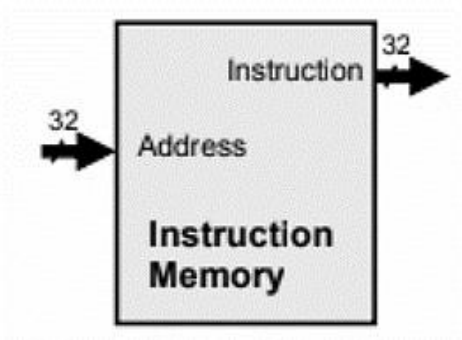


Figure 3: Instruction Memory.

The register file:

The register file component has 16 general peruse registers; each 32-bit register also has a PC register, which is a register for the next address, and a SP register, which is a stack pointer register. It points to the top of the stack. We took three registers from the general purpose register, Rs1, Rs2, and Rd: 4-bit Rs1, which is the first source register, put the value on BusA, 4-bit Rs2, which is the second source register, and 4-bit Rd, which is the destination register. In our path, there is a 2x1 matrix to choose between Rs2 and Rd depending on the value of the selection line. If the value is 0, it takes Rs 2; if it is 1, it takes Rs 3 and puts the value on Bus B. BusA and Bus B go to be input to the ALU.

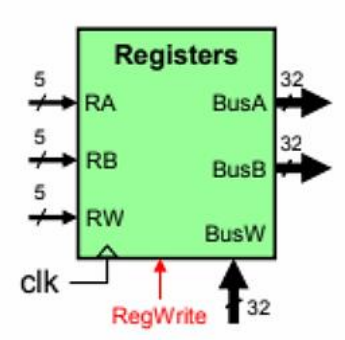


Figure 4: Register File.

The ALU:

The ALU component Its input is the value from BusA, and the output is a 2x1 matrix, which takes the values from BusB, and the immediate 16 after it is extended and becomes 32 depend on the ALUsrc selection line. If its value is 0, it takes BusB. If it's 1, it takes the immediate 32. In this component, it executes the operation for each instruction that is needed based on the ALUOp. Its output has two choices: one to enter into the data memory and the other to do to the mux in the write-back stage.

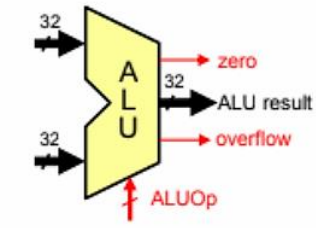


Figure 5: ALU.

Data memory:

Data memory is another type of cache memory that in our project has two types of memory: data memory and stack memory. It took the ALU output as an address input and the BUSB as a data input and got the data output as an output to enter the write-back mux. The only instructions that reach it are the load and the store, the load to read from the memory and the value on Rd to write back, and the store to write to the memory.

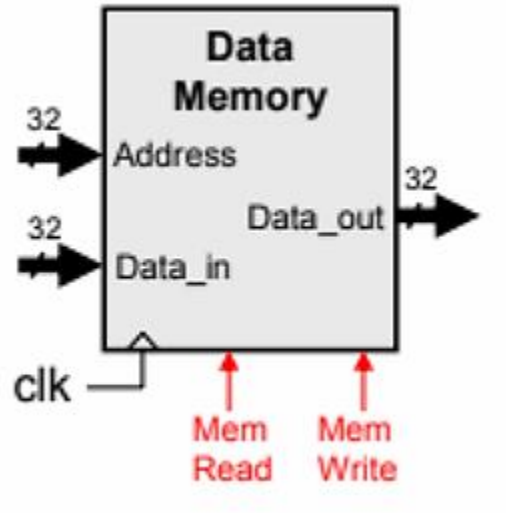


Figure 6: Data Memory.

Write back:

Write back 2x1 mux. If the WB selection line is 0, it will take the output of the ALU and write it back on Rd. If the selection is 1, it will just become 1 for the load instruction.

- **Register (Buffer between Stages)**

In multi cycle data path, the register, acting as a buffer between stages.

Operational synchronization: It makes it easier to synchronize operations across different stages. Because each stage in the multi-cycle data path runs on different clock cycles, the register guarantees that operations remain in sync by serving as an intermediate, ensuring that data is properly transferred to the next stage at the right time. This coordination is essential for data processing to go efficiently across the system.

Preserving intermediate outcomes: At several points in multi-cycle data paths, temporary outcomes are produced. These interim results are stored in the Register until other steps require them. The Register acts as a storage element. This makes it possible to manage data dependencies correctly and permits data to flow via the data stream correctly.

Control timing: In order to provide consistent and readily available data for later stages, the Register regulates the time of data transmission across stages. By allowing a controlled and coordinated data flow and guaranteeing an effective order of processes, it eliminates data hazards such as race conditions and corruption.

Control signal:

PC control:

Its control the address that loading into the Pc register, its generated during the fetch stage, it has three input the 6-bit opcode, ALU zero and negative flags, and two output Pcsrc0 and Pcsrc1 which is the two selection lines for the mux that decide the address that will stores in the Pc register.

Main control:

Its responsible for generation signal to manage the overall operations, it has 6-bit opcode as input and many output signal the src2 which is select the register destination either Rd or Rs2, Regw which is enable or disable writing on the register file for the load, Regw2 which is enable or disable writing on the register file for the Load.POI, Extop it used to ensure that the sign of the immediate number is preserved while extending it to from 16-bit to 32 bit, ALusrc2 it select the second ALU source as BusB or extended immediate, MemRd this signal for load instruction to read from the Data Memory, MemWr this signal for Store instruction to write from the Data Memory, WBdata it select data on BusB as ALU result or Memory Data_out.

ALU control:

Its responsible for choosing the operation inside the ALU, it take 6-bit opcode as input and ALUop as output it control the op inside the ALU for example for the load ita add operation and the branch its sub operation, there is additional control signal which is the zero and negative flage which is for branch control whether branch is taken or not.

Control Signal Truth Tables:

1- Main Control Signal:

Instructions Opcode	*Src2*	*RegW*	*Extop*	*Alusrc2*	*MemRd*	*MemWr*	*WB data*
AND	Rs2=0	1	X	0(BUSB)	0	0	0
ADD	Rs2=0	1	X	0(BUSB)	0	0	0
SUB	Rs2=0	1	X	0(BUSB)	0	0	0
ANDI	X	1	0	1	0	0	0
ADDI	X	1	1	1	0	0	0
LW	X	1	1	1	1	0	1
LW.POS	X	1	1	1	1	0	1
SW	X	0	1	1	0	1	X
BGT	1	0	1	0	0	0	0
BLT	1	0	1	0	0	0	0
BEQ	1	0	1	0	0	0	0
BNE	1	0	1	0	0	0	0
JMP	X	0	X	X	0	0	X
CALL	X	0	X	X	0	1	X
RET	X	X	X	X	1	0	X
PUSH	1	0	X	X	0	1	X
POP	1	1	X	X	1	0	1

Table 1: Main Control Signal.

Boolean Equation

Src2 = (AND + ADD + SUB)`

RegW = (SW + BGT+ BLT+ BEQ+ BNE + JMP+ CALL + PUSH)`

Extop = ANDI`

ALUsrc2 = (AND + ADD + SUB+ BGT+ BLT+ BEQ+ BNE)`

MemRd = (AND + ADD + SUB+ ADDI + ANDI + SW+ BGT+ BLT+ BEQ+ BNE+ JMP+CALL + PUSH)`

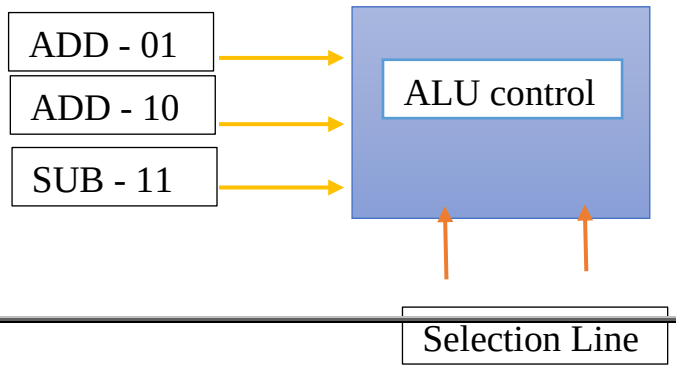
MemWr = SW . CALL. PUSH

WB data = LW . LW POS . POP

2- ALU Control signal:

Instructions Opcode	*Operation*	*Type Of Operation*
AND	AND	01
ADD	ADD	10
SUB	SUB	11
ANDI	AND	01
ADDI	ADD	10
LW	ADD	10
LW.POS	ADD	10
SW	ADD	10
BGT	SUB	11
BLT	SUB	11
BEQ	SUB	11
BNE	SUB	11
JMP	X	X
CALL	X	X
RET	X	X
PUSH	X	X
POP	X	X

Table 2: ALU Control signal.



3- PC Controller:

Instructions Opcode	*Zero Flag*	*Negative Flag*	*PC s/s*
AND	X	X	00
ADD	X	X	00
SUB	X	X	00
ANDI	X	X	00
ADDI	X	X	00
LW	X	X	00
LW.POS	X	X	00
SW	X	X	00
BGT	X	0	10
		1	00
BLT	X	1	10
		0	00
BEQ	1	X	10
	0		00
BNE	0	X	10
	1		00
JMP	X	X	01
CALL	X	X	11
RET	X	X	11
PUSH	X	X	11
POP	X	X	11

Table 3: PC Controller.

If (opcode = jump) pcsrc 01;

Else if Branch = (BEQ . ZeroFlag) + (BNE . ZeroFlag`) + ( BGT . NegFlag`) + (BLT . NegFlag) PCsrc 10;

Else if (opcode = JMP || CALL || RET || POP || PUSH) PCsrc 11;

Else pcsrc = 00;

The Finite State Machine

Here are the stages and states that, would summarize it briefly:

Instruction Fetch: To load an instruction into the instruction register, the CPU must retrieve it from memory.

Instruction Decode: After decoding an instruction, the processor recognizes its type, ascertains the operands required, and saves the pertinent data in internal registers.

Execution: The processor carries out the required actions, which might take several clock cycles, to finish the instruction.

Memory Access: When an instruction calls for reading or writing data to or from memory, the processor makes a memory access to carry out the task.

Write Back: After an operation, the processor writes the outcome back to the relevant memory address or internal register.

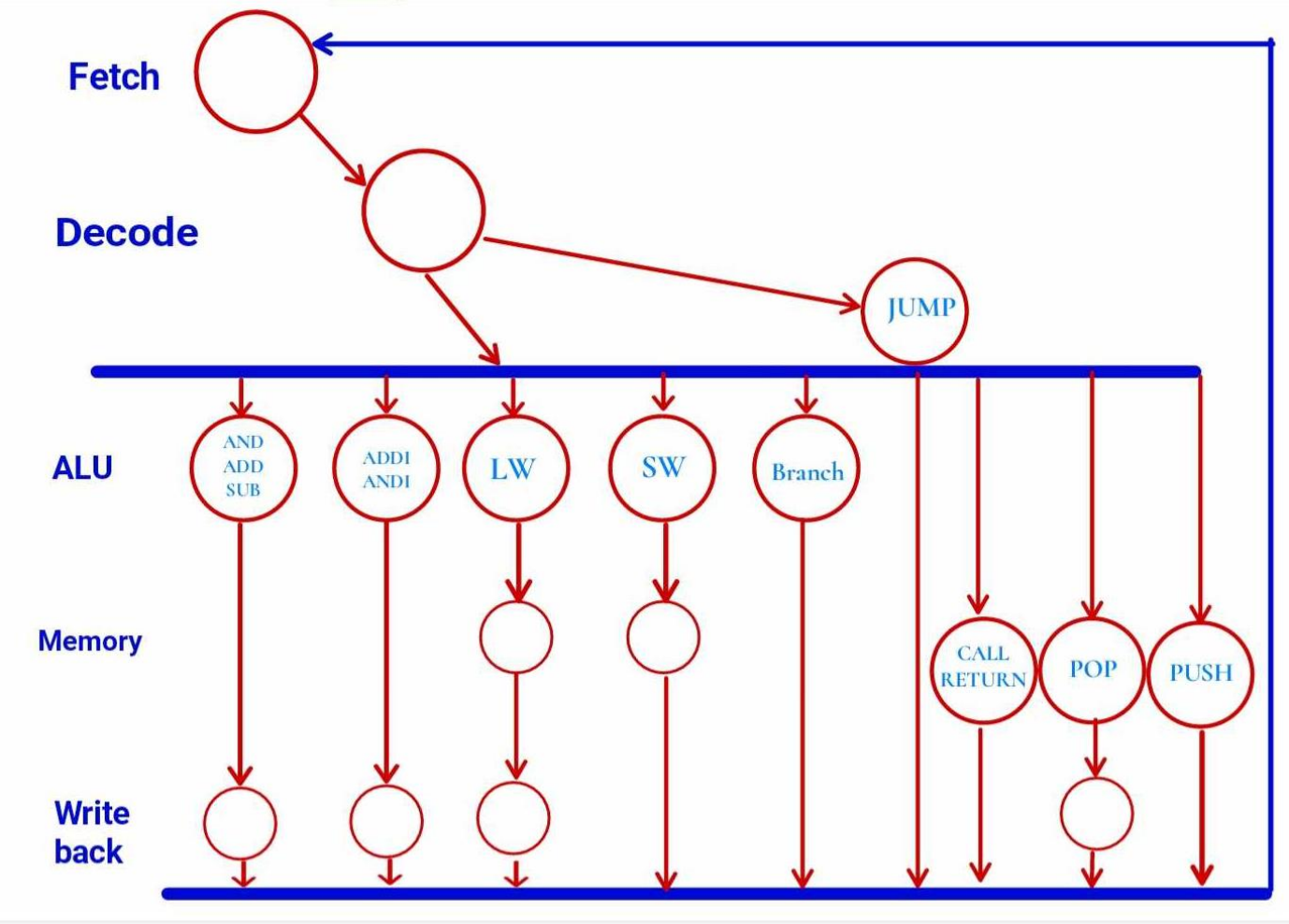


Figure 7: Finite State Machine.

Stages For Each Instruction:

- The R-type instructions (ADD, AND, SUB) need 4 stages

  Stages: Fetch, Decode, Execution (ALU), Write Back.

- The ADDI, ANDI instruction from I-type need 4 stages

  Stages: Fetch, Decode, Execution (ALU), Write Back.

- The LW (Load instruction) from I-type need 5 stages.

  Stages: Fetch, Decode, Execution (ALU), Memory, Write Back.

- The SW (Store instruction) from I-type need 4 stages.

  Stages: Fetch, Decode, Execution (ALU), Memory.

- The all Branch instructions from I-type need 3 stages.

  Stages: Fetch, Decode, Execution (ALU).

- The Jump instruction from J-type need 2 stages.

  Stages: Fetch, Decode.

- The Call and Return instructions from J-type need 3 stages.

  Stages: Fetch, Decode, Memory.

- The Pop instructions from S-type need 4 stages.

  Stages: Fetch, Decode, Memory, Write Back.

- The Push instructions from S-type need 3 stages.

  Stages: Fetch, Decode, Memorys.

RTL Design:

R-type (AND, ADD, SUB)

Fetch instruction: Instruction ← MEM[PC]

Fetch operands: BusA ← Reg(RS1), BusB ← Reg(RS2)

Execute operation: ALU_result ← func(BusA, BusB)

Write ALU result: Reg(RD) ← ALU_result

Next PC address: PC ← PC + 1

I-TYPE (ADDI & ANDI)

Fetch instruction: Instruction ← MEM[PC]

Fetch operands: BusA ← Reg(RS1), BusB ← Extend(imm16)

Execute operation: ALU_result ← op(data1, data2)

Write ALU result: Reg(RD) ← ALU_result

Next PC address: PC ← PC + 1

LW (Load From I-type)

Fetch instruction: Instruction ← MEM[PC]

Fetch base register: base ← Reg(RS1)

Calculate address: address ← base + sign_extend(imm16)

Read memory: data ← MEM[address]

Write register : Reg(RD) ← data

Next PC address: PC ← PC + 1

LW POS (Load From I-type)

Fetch instruction: Instruction ← MEM[PC]

Fetch base register: base ← Reg(RS1)

Calculate address: address ← base + sign_extend(imm16)

Result_sum = base + 1

Read memory: data ← MEM[address]

Write register: Reg(RD) ← data

Reg(RS1) ← Result_sum

Next PC address: PC ← PC + 1

SW (Store From I-type)

Fetch instruction: Instruction ← MEM[PC]

Fetch registers: BusA ← Reg(RS1) BusW← Reg(RD)

Calculate address: address ← BusA + sign_extend(imm16)

Write memory: MEM[address] ← BusW

Next PC address: PC ← PC + 1

BEQ (From I-type)

Fetch instruction: Instruction ← MEM[PC]

Fetch operands: BusA ← Reg(RS1), BusW ← Reg(RD)

Equality: zero ← subtract(BusA, BusW)

Branch: if (zero) PC ← PC + 1 + sign_ext(offset16)

else PC ← PC + 1

BNE (From I-type)

Fetch instruction: Instruction ← MEM[PC]

Fetch operands: BusA ← Reg(RS1), BusW ← Reg(RD)

NotEquality: zero` ← subtract(BusA, BusW)

Branch: if (zero`) PC ← PC + 1 + sign_ext(offset16)

else PC ← PC + 1

BGT (From I-type)

Fetch instruction: Instruction ← MEM[PC]

Fetch operands: BusA ← Reg(RS1), BusW ← Reg(RD)

Greater: NEG` ← subtract(BusA, BusW)

Branch: if (NEG`) PC ← PC + 1 + sign_ext(offset16)

else PC ← PC + 1

BLT (From I-type)

Fetch instruction: Instruction ← MEM[PC]

Fetch operands: BusA ← Reg(RS1), BusW ← Reg(RD)

LESS: NEG ← subtract(BusA, BusW)

Branch: if (NEG) PC ← PC + 1 + sign_ext(offset16)

else PC ← PC + 1

JMP (From J-type)

Fetch instruction: Instruction ← MEM[PC]

Target PC address: target ← PC[31:26] ||Immediate26

Jump: PC ← target

CALL (From J-type)

Fetch instruction: Instruction ← MEM[PC]

Target PC address: target ← PC[31:26] ||Immediate26

INC ← PC + 1

CALL: PC ← target

Write memory: MEM[SP] ← INC

Return (From J-type)

Fetch instruction: Instruction ← MEM[PC]

Read memory: data ← MEM[SP]

return: PC ← data

PUSH (From S-type)

Fetch instruction: Instruction ← MEM[PC]

Fetch operands: BusW ← Reg(RD)

Write memory: MEM[SP] ← BusW

POP (From S-type)

Fetch instruction: Instruction ← MEM[PC]

Fetch operands: BusW ← Reg(RD)

Read memory: data ← MEM[SP]

Write register : Reg(RD) ← data

2 – Simulation and Testing

Here is a part of our instruction memory, and the simulation results for these instructions will be explained.

```
initial begin

        //ADD inst: R[3] = R[1] + R[2]   aluOut = 9
        Instruction_Mem[0] = 32'b 000001_0011_0001_0010_00000000000000;

        //ANDI inst:  R[4] = R[1] & 15
        Instruction_Mem[1] = 32'b 000011_0100_0001_0000000000001111_00;

        //Load inst:  R[7] = Mem(R[1] + 2)   Rd = 7
        Instruction_Mem[2] = 32'b 000101_0111_0001_0000000000000010_00;

        //BEQ inst: Beq R8,R9 if equal go to inst 7
        Instruction_Mem[3] = 32'b 001010_1000_1001_0000000000000011_00;

        //Sub inst: R[2] = R[5] - R[6]
        Instruction_Mem[4] = 32'b 000010_0010_0101_0110_00000000000000;

        //AND inst: R[3] = R[1] & R[2]
        Instruction_Mem[5] = 32'b 000000_0011_0001_0010_00000000000000;

        //Store inst:   Mem(R[8] + 2) = R[5] , data_memory[10] = 7
        Instruction_Mem[6] = 32'b 000111_0101_1000_0000000000000010_00;

        //ADDI inst:  R[4] = R[1] + 14 , alu_out = 12 hexa
        Instruction_Mem[7] = 32'b 000100_0100_0001_0000000000001110_00;
```

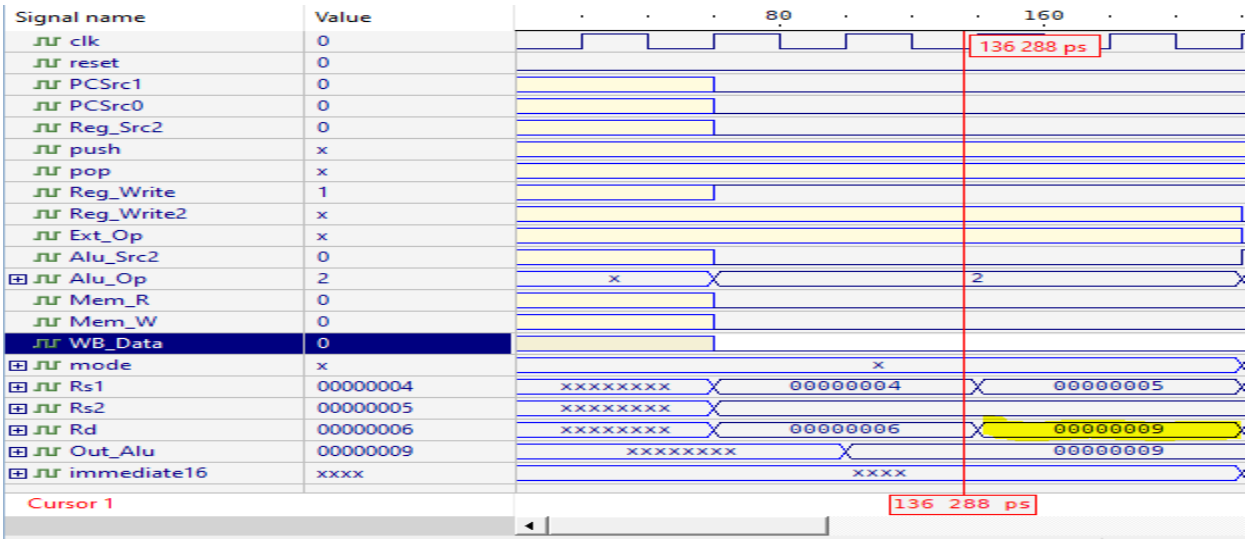
General purpose registers values:

```
initial begin
        GPR[0] = 32'h00000003;
        GPR[1] = 32'h00000004;
        GPR[2] = 32'h00000005;
        GPR[3] = 32'h00000006;
        GPR[4] = 32'h00000007;
        GPR[5] = 32'h00000007;
        GPR[6] = 32'h00000002;
        GPR[7] = 32'h00000009;
        GPR[8] = 32'h00000008;
        GPR[9] = 32'h00000008;
        GPR[10] = 32'h0000000C;
        GPR[11] = 32'h0000000A;
        GPR[12] = 32'h0000000A;
        GPR[13] = 32'h0000000A;
        GPR[14] = 32'h0000000B;
        GPR[15] = 32'h0000000A;
    end
```

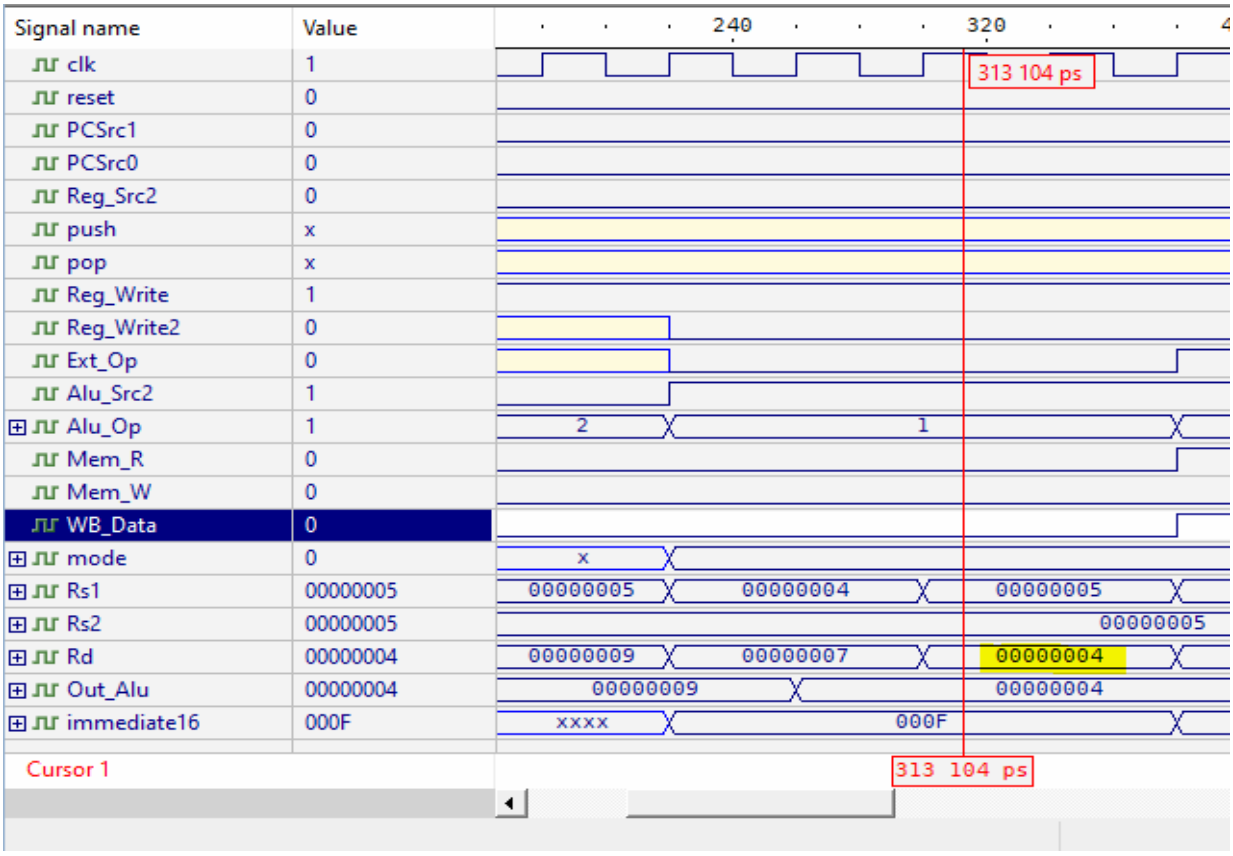
Data memory values:

```
initial begin
        data_memory[0] = 32'h00000001;
        data_memory[1] = 32'h00000002;
        data_memory[2] = 32'h00000003;
        data_memory[3] = 32'h00000004;
        data_memory[4] = 32'h00000005;
        data_memory[5] = 32'h00000006;
        data_memory[6] = 32'h00000007;
        data_memory[7] = 32'h00000008;
        data_memory[8] = 32'h00000009;
        data_memory[9] = 32'h0000000A;
        data_memory[10] = 32'h0000000B;
        data_memory[11] = 32'h0000000C;
        data_memory[12] = 32'h0000000D;
        data_memory[13] = 32'h0000000E;
        data_memory[14] = 32'h0000000F;
        data_memory[15] = 32'h00000001;

    end
```

First instruction is ADD: the waveform shows that it only took 4 clock cycles to finish, and at the last stage which is write back, the result was written on Rd as shown in yellow. The result is ADD between value in Rs1 which is 4 and value in Rs2 which is 5, so result is 9.



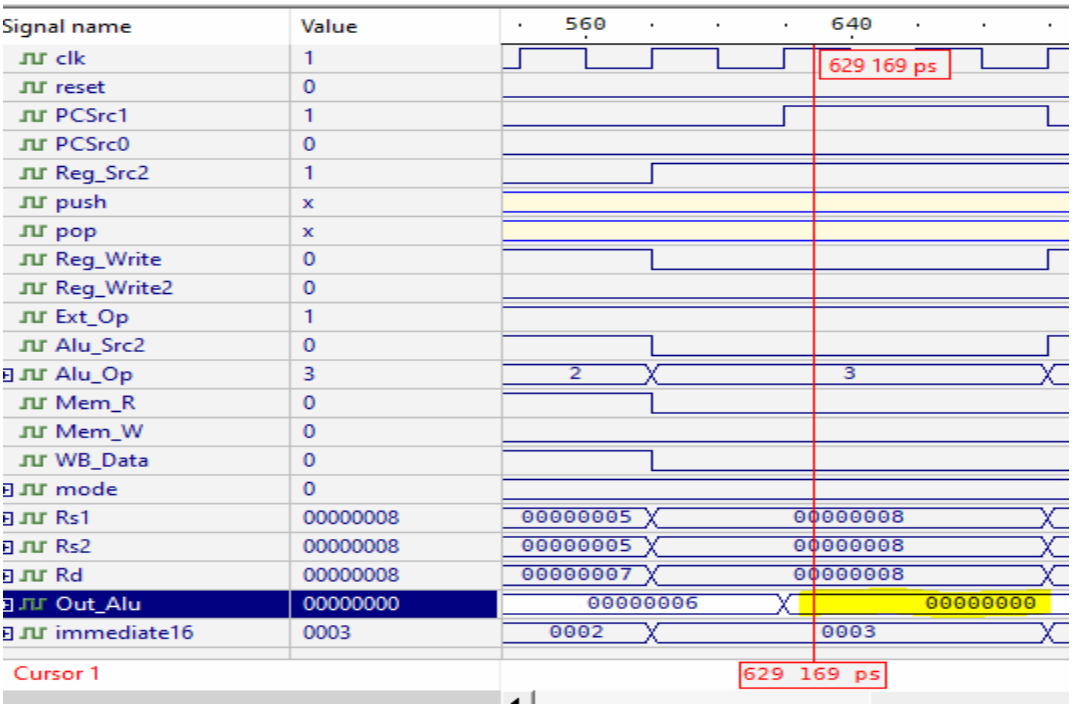
The next instruction is ANDI: between the value in Reg[1] which is 4, and the immediate which is 15, the result is 4 in hexadecimal which is written on Rd on cycle 4.



The next instruction is load: which is R[7] = Mem(R[1] + 2), the memory address is calculated by adding the value in R[1] which is 4 as shown in the figure below, and the value of immediate which is 2, so the memory address is 4 + 2 =6, and the value stored data_memory[6] is 7, then 7 is written back on Rd.

Note that Load took 5 cycles.

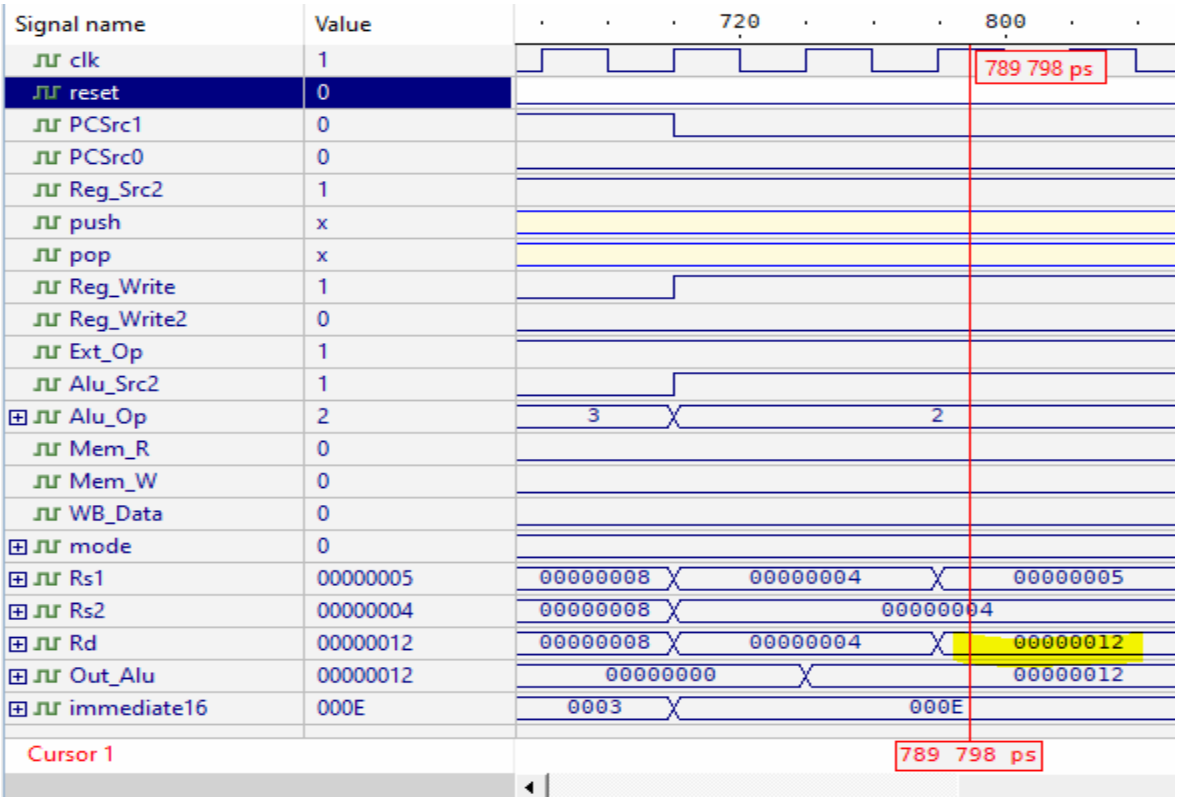
Next instruction is branch: it compares the value in R8 with the value in R9, and they are equal as shown in the registers above. So ALU result will be zero as shown in the next figure and zero flag is 1 so the branch will be taken.



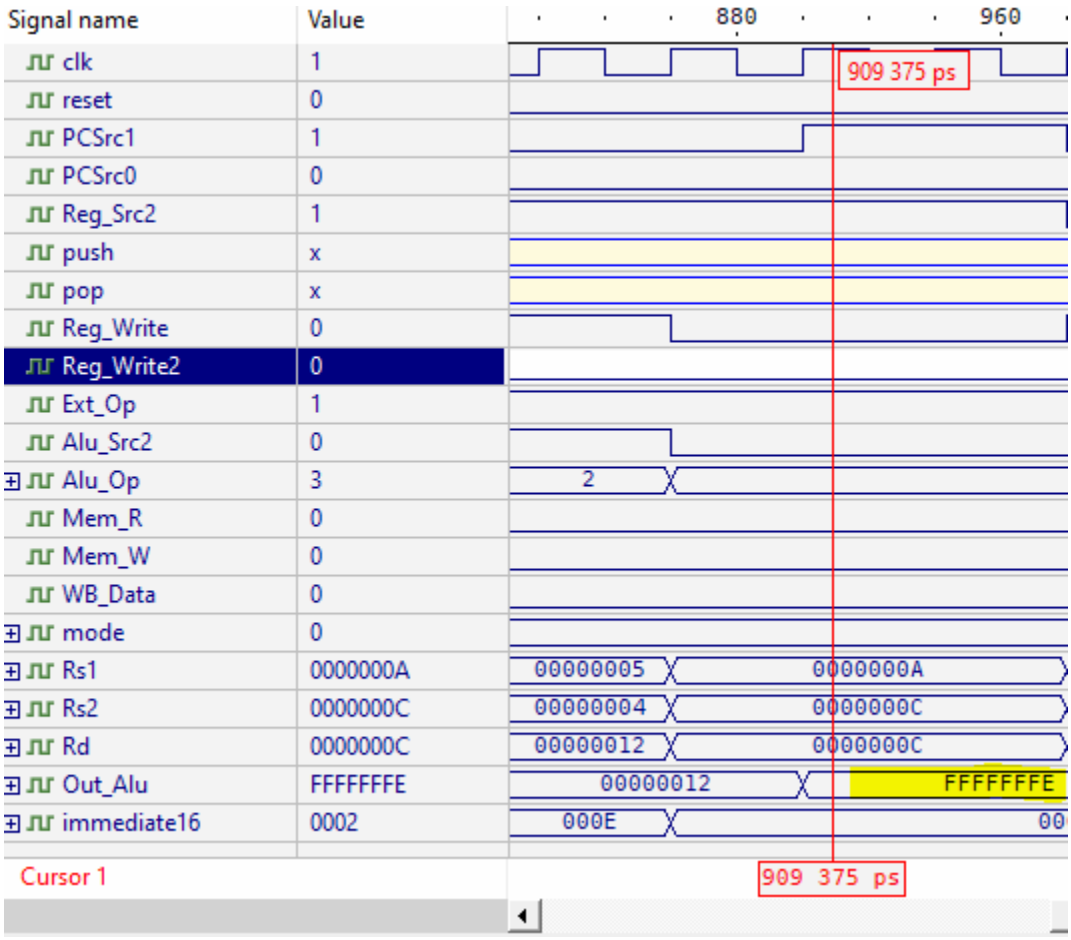
Note that branch only took 3 cycles.

Now if you look back at the instruction memory, the next instruction should be SUB, but because the branch was taken, it jumps to the branch target address which is instruction 7 in this case.

Instruction 7 is ADDI: which is addition between value in R1 which is 4, and immediate 14, the result is 12 in hexadecimal as shown below. And note that it will take 4 clock cycles.



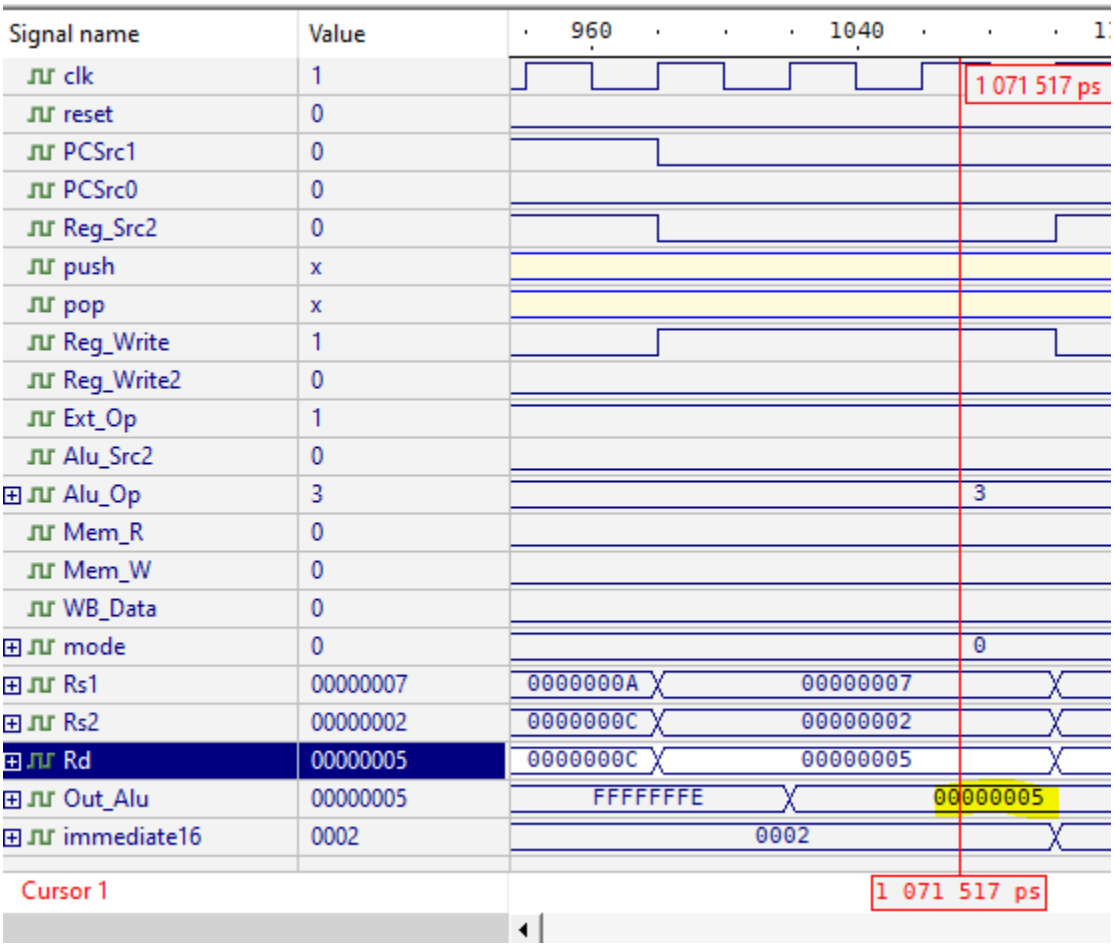
The next instruction is BGT: it compares the values in R10 and R11, value in R10 is bigger so branch will be taken and it will jump to instruction 11 according to the branch target address.



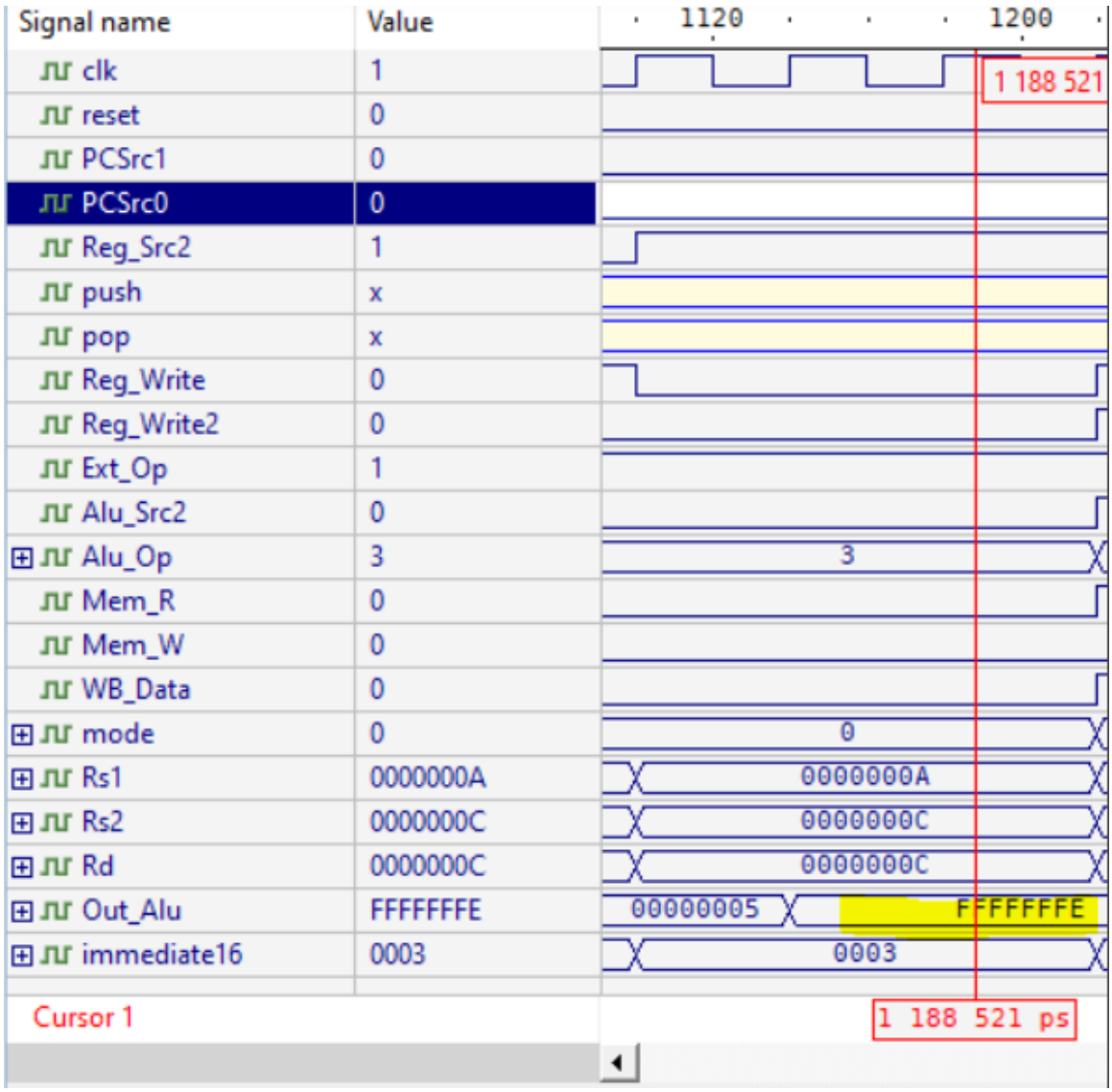
The ALU result is negative so the branch was taken according to the negative flag.

After branch it will reach instruction 11 which is Sub.

The next instruction is SUB: it take the value from Rs1 = 7 and from Rs2 = 2 the result from ALUout =5 then it Write back on Rd.



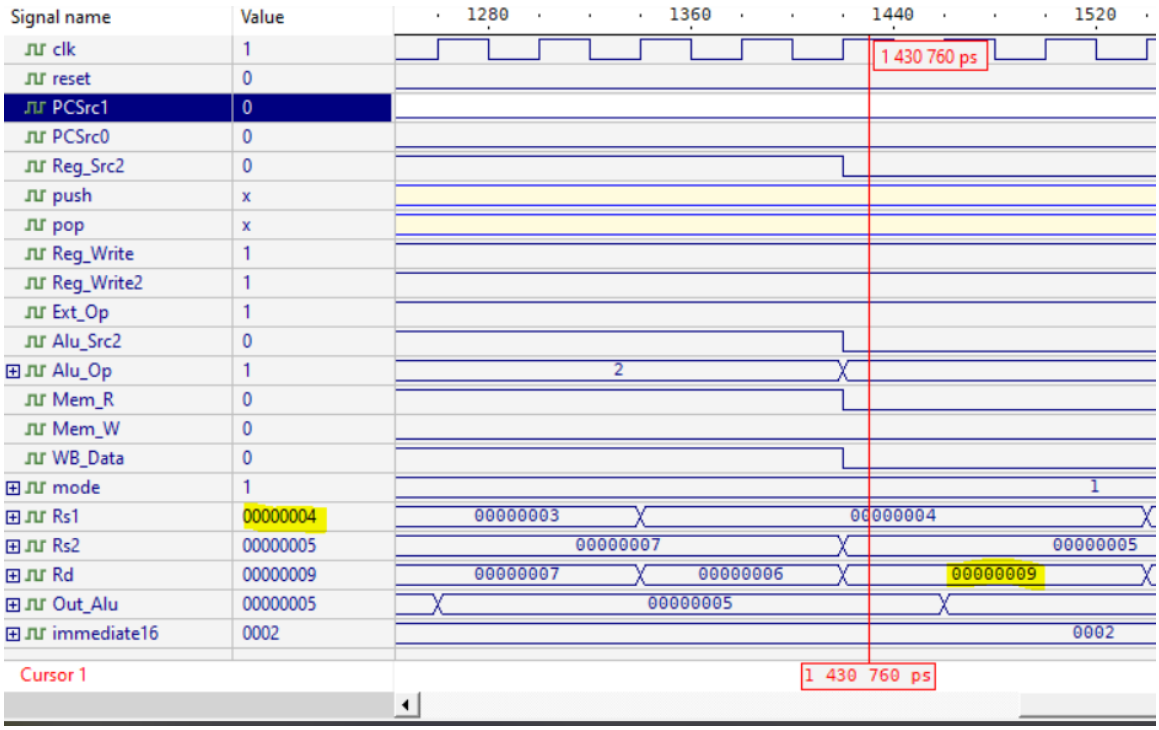
The next branch is BLT between R10 and R11, but this time the branch will not be taken because R10 is bigger than R11 as stated in the previous instruction of BGT. When ALU result is negative, BLT will not be taken according to negative flag.



The next instruction is Load.POS:

Load.PIO inst: R[7] = Mem(5) = 6, R0 =3+1 =4

we can see that the value of Rs1 is increment by 1 from 3 to 4.



Now, we will try puch and pop instructions:

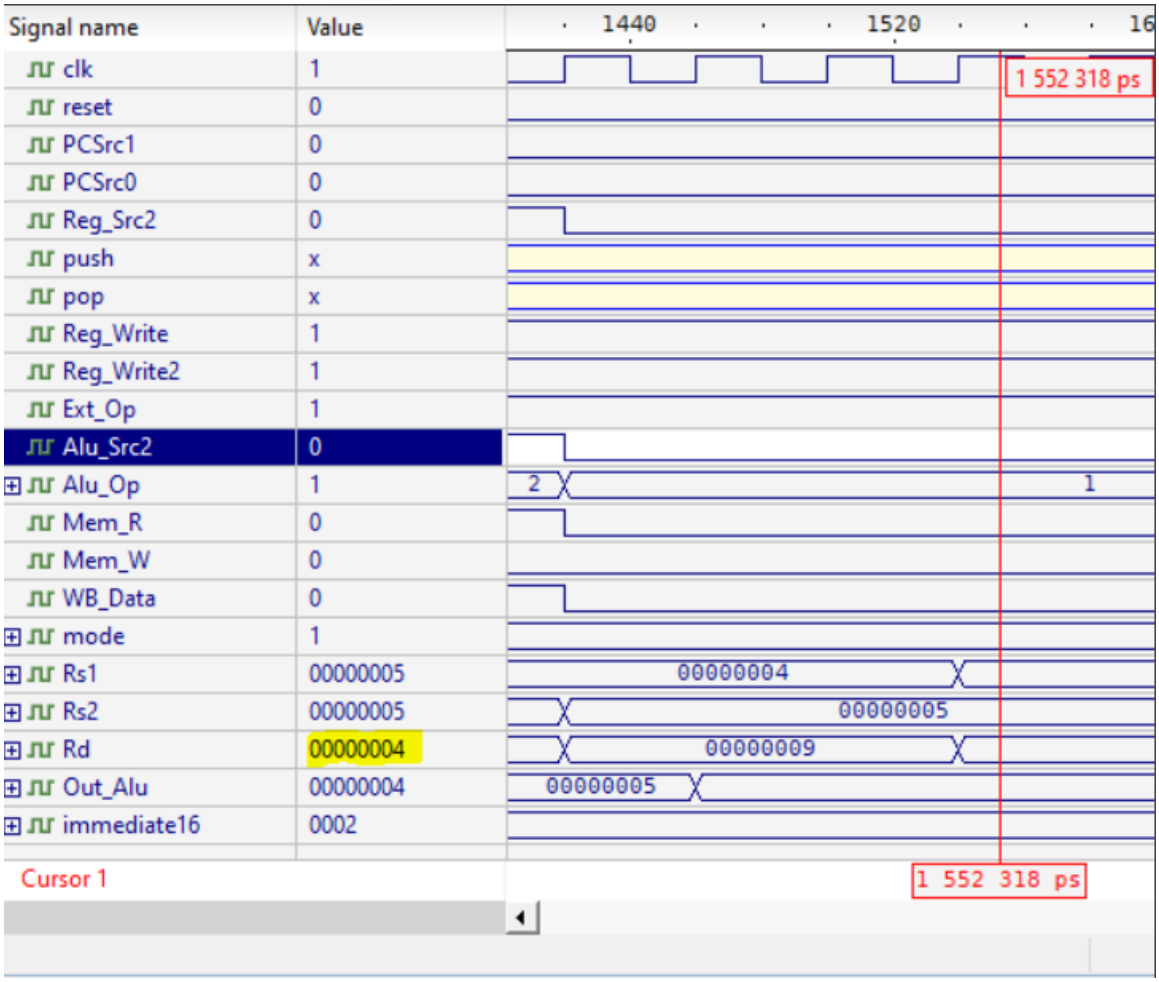
```
//AND inst: R[3] = R[1] & R[2]   aluOut = 4  R3=4
Instruction_Mem[14] = 32'b 000000_0011_0001_0010_00000000000000;

//PUSH inst, push value of R3 on stack
Instruction_Mem[15] = 32'b 001111_0011_00000000000000000000;

// next inst will change value of R3

//ADD inst: R[3] = R[1] + R[2]   aluOut = 9
Instruction_Mem[16] = 32'b 000001_0011_0001_0010_00000000000000;

// pop the value of R3 to see how the previos value was preserved
//  PoP inst, pop value of R3 from stack
Instruction_Mem[17] = 32'b 010000_0011_00000000000000000000;
```

AND make R3 =4, then we make Puch instruction and it put the value of R3 on the top of the Stack (SP =30). We make ADD operation to make the value of Rd = 9.

Then to ensure that Puch instruction work well we pop the value Rd = 4 from the top of the stack as shown in the figure below.

```
# KERNEL: Time= 1540000,in stage: 4, opcode = 0
# KERNEL: Time= 1580000,in stage: 0, PC = 16
# KERNEL: Time= 1620000,in stage: 1, opcode = f
# KERNEL: Time= 1660000,in stage: 3, opcode = f
# KERNEL: PUSH:OPcode= 1111, SP= 30,data_memory[30] = 4
# KERNEL: Time= 1700000,in stage: 0, PC = 17
# KERNEL: Time= 1740000,in stage: 1, opcode = 1
# KERNEL: Time= 1780000,in stage: 2, opcode = 1
# KERNEL: first input = 5, second input = 5
# KERNEL: opcode = 1, aluOut = 10
# KERNEL: Time= 1820000,in stage: 3, opcode = 1
# KERNEL: Time= 1860000,in stage: 4, opcode = 1
# KERNEL: Time= 1900000,in stage: 0, PC = 18
# KERNEL: Time= 1940000,in stage: 1, opcode = 10
# KERNEL: Time= 1980000,in stage: 3, opcode = 10
# KERNEL: POP:OPcode= 10000, SP= 29, Rd = 4,
# KERNEL: Time= 2020000,in stage: 4, opcode = 10
# KERNEL: Time= 2060000,in stage: 0, PC = 19
# KERNEL: Time= 2100000,in stage: 1, opcode = 2
# KERNEL: Time= 2140000,in stage: 2, opcode = 2
# KERNEL: first input = 7, second input = 2
```

Rd = 4


Signal name
Value
1440
1520
16
clk
1
1 552 318 ps
reset
0
PCSrc1
0
PCSrc0
0
Reg_Src2
0
push
x
pop
x
Reg_Write
1
Reg_Write2
1
Ext_Op
1
Alu_Src2
0
Alu_Op
1
2
1
Mem_R
0
Mem_W
0
WB_Data
0
mode
1
Rs1
00000005
00000004
Rs2
00000005
00000005
Rd
00000004
00000009
Out_Alu
00000004
00000005
immediate16
0002
Cursor 1
1 552 318 ps

3- Teamwork

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

The report: Mariam: Control siginal tables with Boolean equations with a finite state diagram

Leena: draw a datapath with a description for it and a component description

Amany - stage and control signal and buffer description

And other things were shared while working.