
# EE 224 IITB CPU
# Final Documentation

**Authors:**

| | |
|---|---|
| Amrutanshu Mohanty | 23B1208 |
| Arjoe Basak | 23B1295 |
| Daksh Sawke | 23B1254 |
| Shaileshram Sivakumar | 23B1205 |

December 4, 2024

# Contents

# Overview

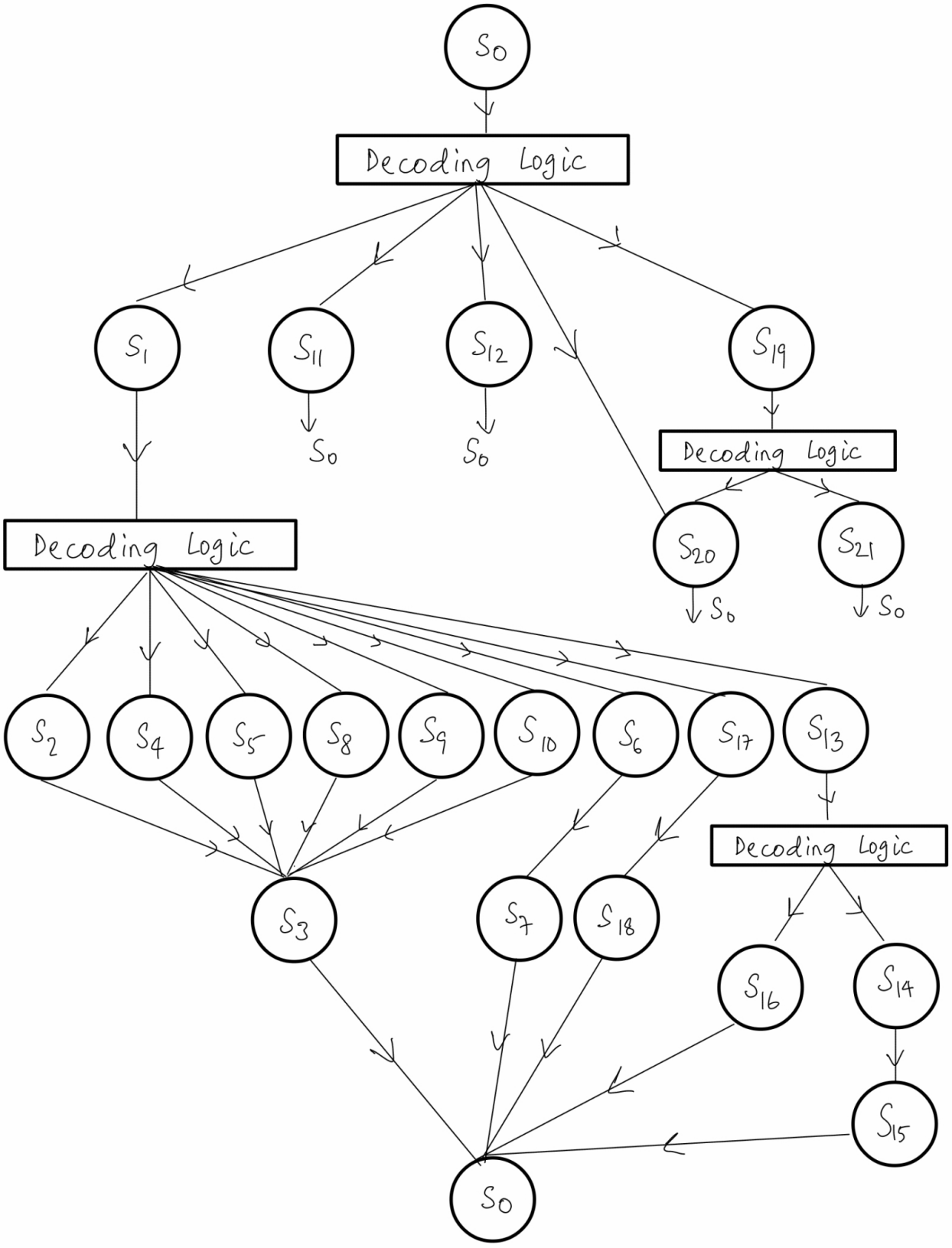
## Problem Statement

We were tasked with building a simple 8-register, 16-bit computer system using point-to-point infrastructure, in a Hardware Descriptive Language. VHDL has been used for the same. The CPU features 8 general-purpose registers **(R0 to R7)**, a Program Counter **(PC)** pointing to the next instruction, and a Condition Code Register with Carry **(C)** and Zero **(Z)** flags. The system supports 15 instructions across three formats (R, I, and J types) and initializes the PC to 0000H on power-up.

## CPU Overview

- Our CPU is a finite state machine with **22** different states. Each instruction has been broken down into 2-3 states, based on the tasks being performed.
- Appropriate state tables are made, which has the tasks to be performed when in a state. A circuit has also been designed for each instruction, based on the components used in the states.
- Finally, a top-level circuit has been designed, which incorporates several multiplexers that are used to control the inputs fed into components. Their select lines controlled by outputs of the FSM.
- **All components** have been designed **structurally** in VHDL, from using logic gates for the 16-bit Kogge-Stone adder to using D-flipflops for every register. Only the final FSM uses behavioural architecture.
- In order to verify our handiwork, the code in VHDL is supported by a comprehensive **Testbench**, to simulate the instructions.
- For the ease of loading various instructions into the CPU, a **python script** has been created which takes in assembly line inputs and generates a text file which is then read by the VHDL code for the memory.
- A **synchronous reset signal** has been implemented, that sets the FSM to a default state while resetting the program counter, tempoarary registers and register file too.

# Finite State Machine



# State Tables

$S_0$ :

| | |
|---|---|
| PC $\longrightarrow$ Mem_A , ALU_A, $T_3$<br>+2 $\longrightarrow$ ALU_B<br>Mem_Dout $\longrightarrow$ IR<br>ALU_C $\longrightarrow$ PC | Mem_read<br>ALU_ADD<br>$T_3$_en<br>IR_en<br>PC_en |

$S_1$ :

| | |
|---|---|
| IR[11:9] $\rightarrow$ RF_$A_1$<br>IR[8:6] $\rightarrow$ RF_$A_2$<br>RF_$D_1$ $\longrightarrow$ $T_1$<br>RF_$D_2$ $\longrightarrow$ $T_2$ | $T_1$_en<br>$T_2$_en |

$S_2$ :

| | |
|---|---|
| $T_1$ $\longrightarrow$ ALU_A<br>$T_2$ $\longrightarrow$ ALU_B<br>ALU_C $\longrightarrow$ $T_3$ | ALU_ADD<br>$T_3$_en |

$S_3$ :

| | |
|---|---|
| $T_3$ $\longrightarrow$ RF_$D_3$<br>IR[5:3] $\rightarrow$ RF_$A_3$ | RF_en |

$S_4$ :

| | |
|---|---|
| $T_1$ $\longrightarrow$ ALU_A<br>$T_2$ $\longrightarrow$ ALU_B<br>ALU_C $\longrightarrow$ $T_3$ | ALU_SUB<br>$T_3$_en |

$S_5$ :

| | |
|---|---|
| $T_1 \longrightarrow$ ALU_A<br>$T_2 \longrightarrow$ ALU_B<br>ALU_C $\longrightarrow T_3$ | ALU_MUL<br>$T_3$ - en |

$S_6$ :

| | |
|---|---|
| $T_1 \longrightarrow$ ALU_A<br>IR [5:0] $\longrightarrow$ SE(16) $\longrightarrow$ ALU_B<br>ALU_C $\longrightarrow T_3$ | ALU_ADD<br>$T_3$ - en |

$S_7$ :

| | |
|---|---|
| IR[8:6] $\longrightarrow$ RF_A3<br>$T_3 \longrightarrow$ RF_D3 | RF_ en |

$S_8$ :

| | |
|---|---|
| $T_1 \longrightarrow$ ALU_A<br>$T_2 \longrightarrow$ ALU_B<br>ALU_C $\longrightarrow T_3$ | ALU_AND<br>$T_3$ - en |

$S_9$ :

| | |
|---|---|
| $T_1 \longrightarrow$ ALU_A<br>$T_2 \longrightarrow$ ALU_B<br>ALU_C $\longrightarrow T_3$ | ALU_OR<br>$T_3$ - en |

$S_{10}$ :

| | |
|---|---|
| $T_1 \longrightarrow$ ALU_A<br>$T_2 \longrightarrow$ ALU_B<br>ALU_C $\longrightarrow T_3$ | ALU_IMP<br>$T_3$ - en |

$S_{11}$ :

| | |
|---|---|
| $IR_{11:9} \rightarrow RF\_A_3$<br>$IR_{7:0} \rightarrow ZL(16) \rightarrow RF\_D_3$ | RF_en |

$S_{12}$ :

| | |
|---|---|
| $IR_{11:9} \rightarrow RF\_A_3$<br>$IR_{7:0} \rightarrow ZH(16) \rightarrow RF\_D_3$ | RF_en |

$S_{13}$ :

| | |
|---|---|
| $T_2 \rightarrow$ ALU_A | |
| $IR_{5:0} \rightarrow$ Signed extension (Pad w/ MSB) $\rightarrow$ ALU_B | ALU_ADD |
| ALU_C $\rightarrow T_3$ | $T_3$ _ en |

$S_{14}$ :

| | |
|---|---|
| $T_3 \rightarrow$ Mem_A | Mem_read |
| Mem_Dout $\rightarrow T_1$ | $T_1$ _ en |

$S_{15}$ :

| | |
|---|---|
| $T_1 \rightarrow RF\_D_3$<br>$IR_{11:9} \rightarrow RF\_A_3$ | RF_en |

$S_{16}$ :

| | |
|---|---|
| $T_3 \rightarrow$ Mem_A<br>$T_1 \rightarrow$ Mem_Din | Mem_write |

$S_{17}$:

| | |
|---|---|
| $T_1 \rightarrow ALU\_A$<br>$T_2 \rightarrow ALU\_B$<br>$ALU\_Z \rightarrow ZR$ | ALU_sub |

$S_{18}$:

| | |
|---|---|
| $T_3 \rightarrow ALU\_A$<br>$IR[5:0] \rightarrow SE(16) \rightarrow$ shift by 1 $\rightarrow ALU\_B$<br>if $(ZR = 1)$<br>$ALU\_C \rightarrow PC$<br>else<br>$PC \rightarrow PC$ } in MUX | ALU_add<br>PC_en |

$S_{19}$:

| | |
|---|---|
| $IR[11:9] \rightarrow RF\_A_3$<br>$T_3 \rightarrow RF\_D_3$ | Rf_en |

$S_{20}$:

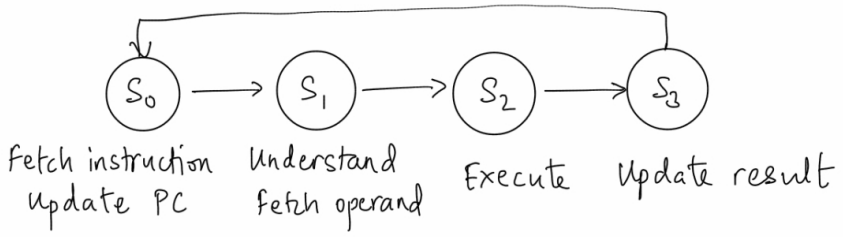
| | |
|---|---|
| $T_3 \rightarrow ALU\_A$<br>$IR[8:0] \rightarrow SE(16) \rightarrow$ shift by 1 $\rightarrow ALU\_B$<br>$ALU\_C \rightarrow PC$ | ALU_add<br>PC_en |

$S_{21}$:

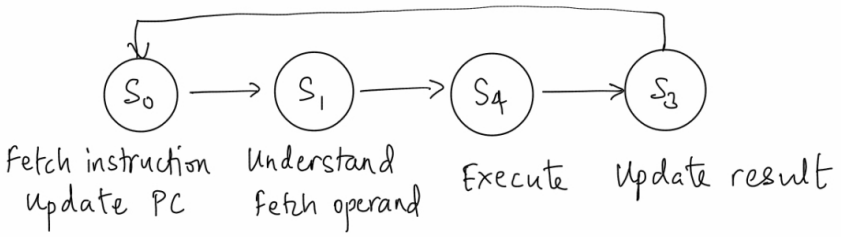
| | |
|---|---|
| $IR[8:6] \rightarrow RF\_A_2$<br>$RF\_D_2 \rightarrow PC$ | PC_en |

# Instruction Implementation

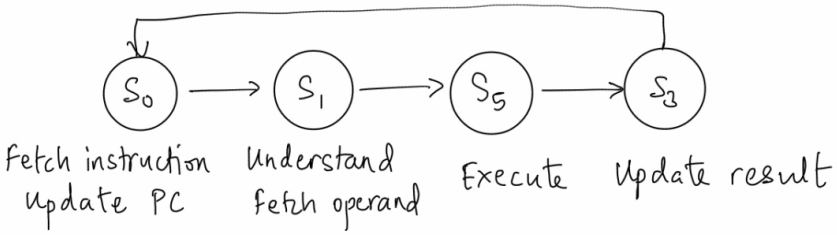
1. ADD (opcode : 0000)



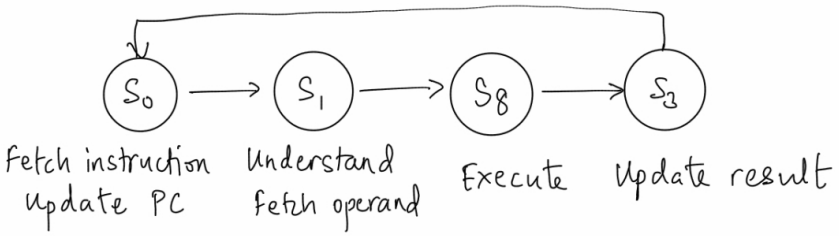

2. SUB (opcode : 0010)



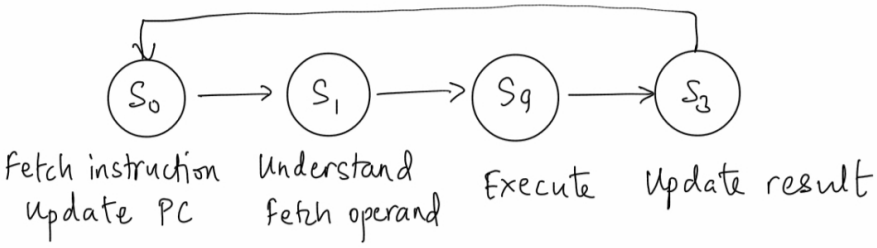

3. MUL (opcode : 0011)



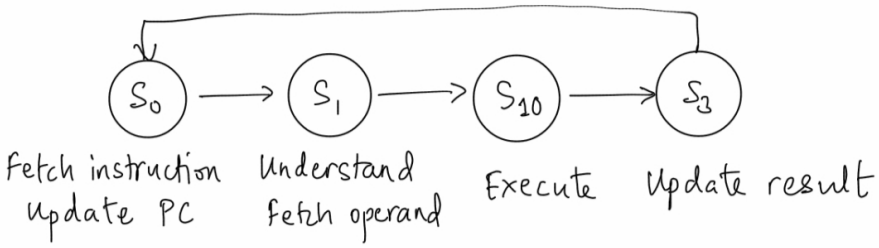

4. AND (opcode : 0100)



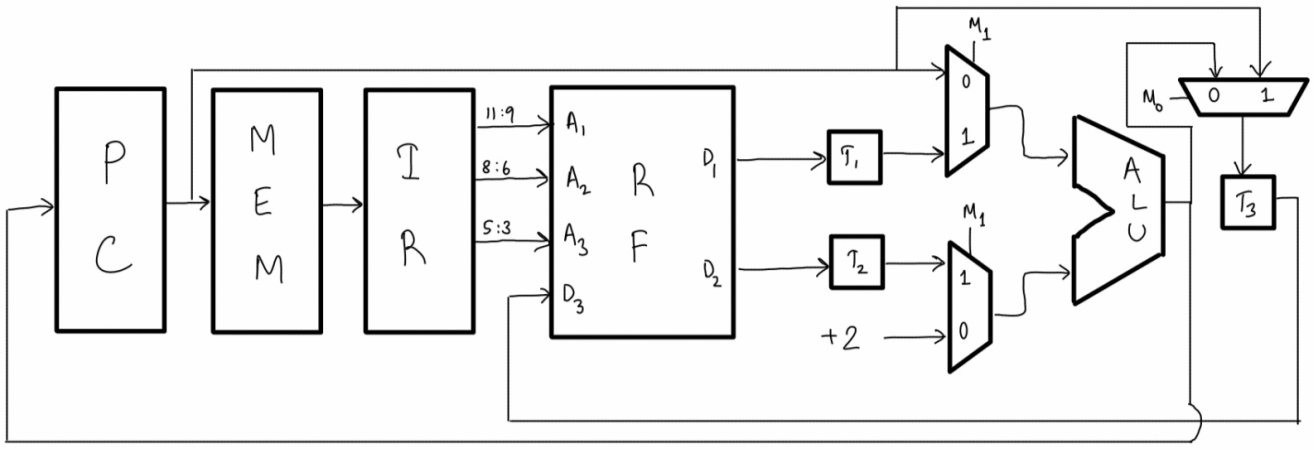
## 5. ORA (opcode: 0101)



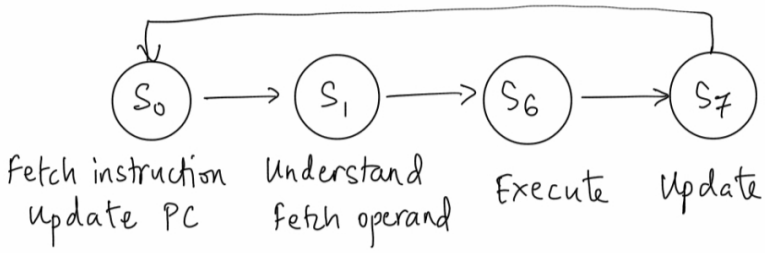

## 6. IMP (opcode: 0110)

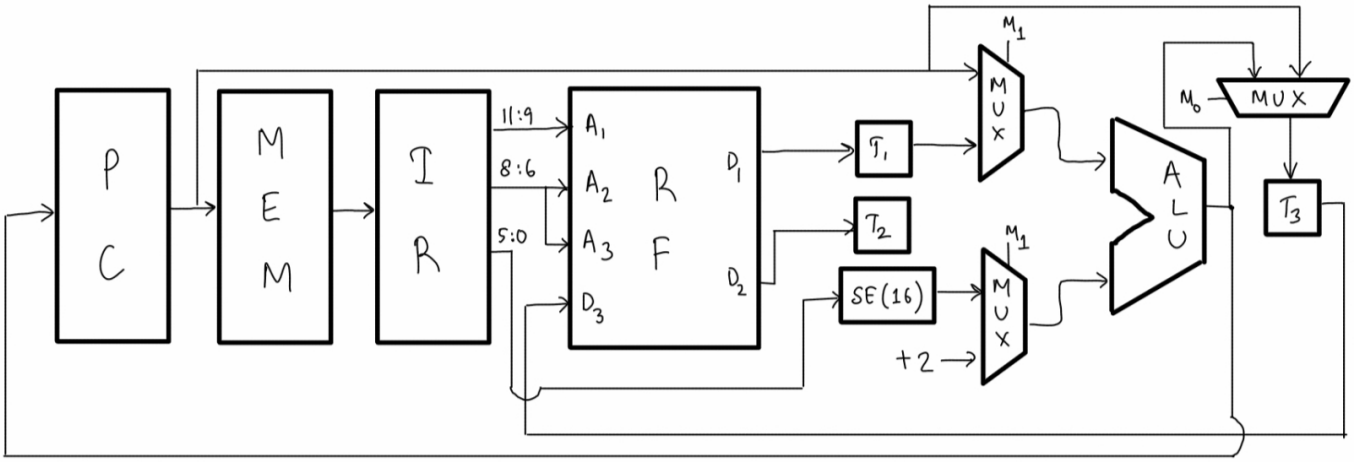



## ADD, SUB, MUL, AND, ORA, IMP

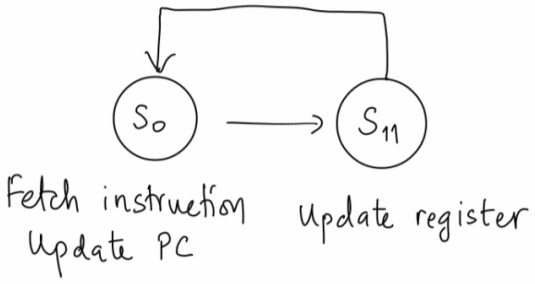


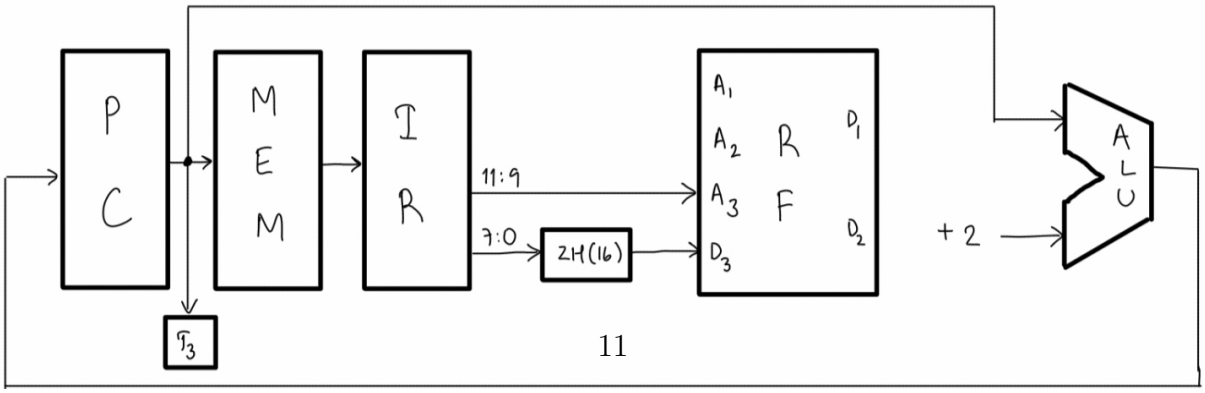
# 7. ADI (opcode : 0001)



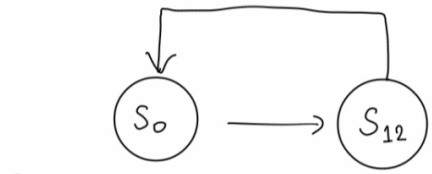



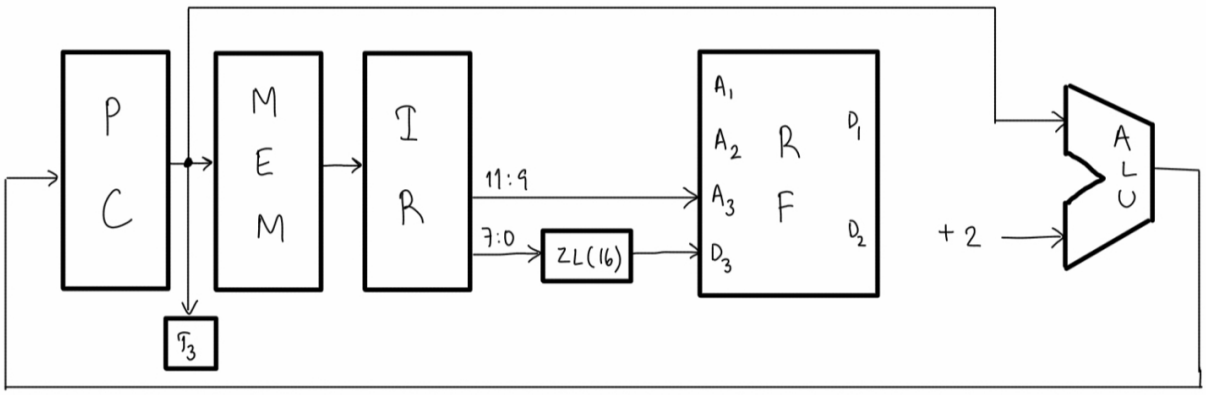

# 8. LHI (opcode: 1000)



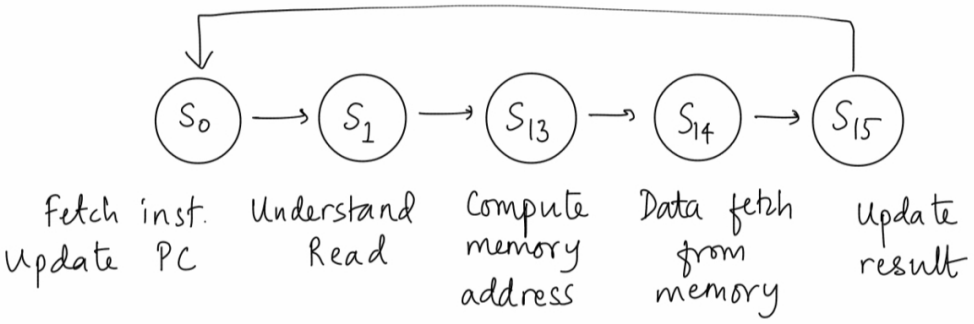



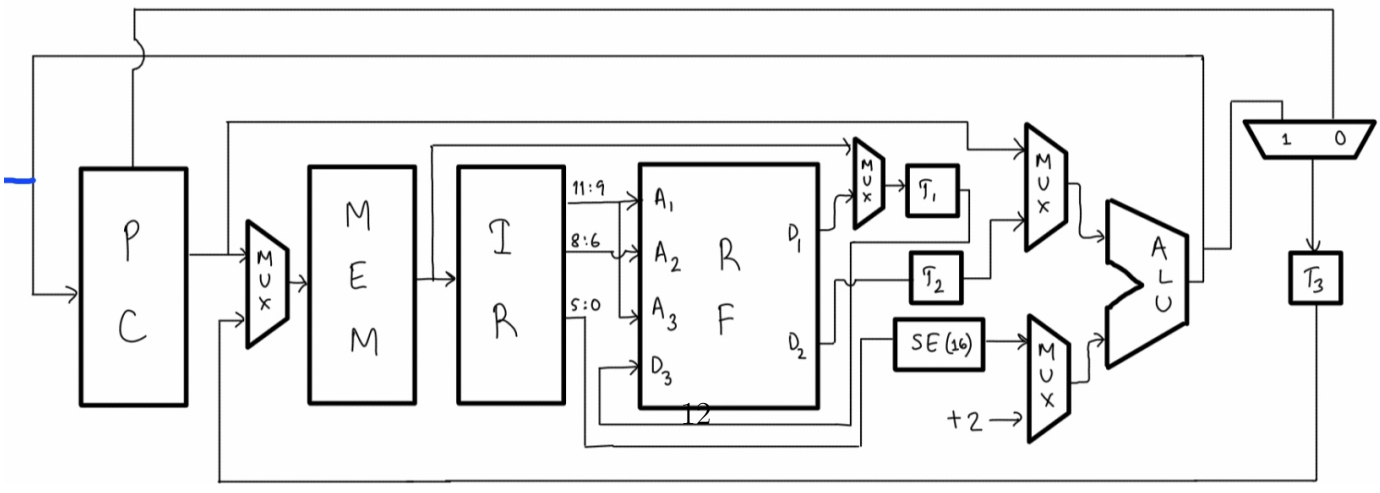
# 9. LLI (opcode: 1001)



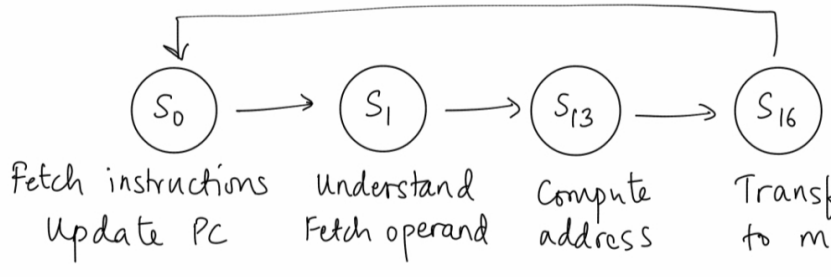



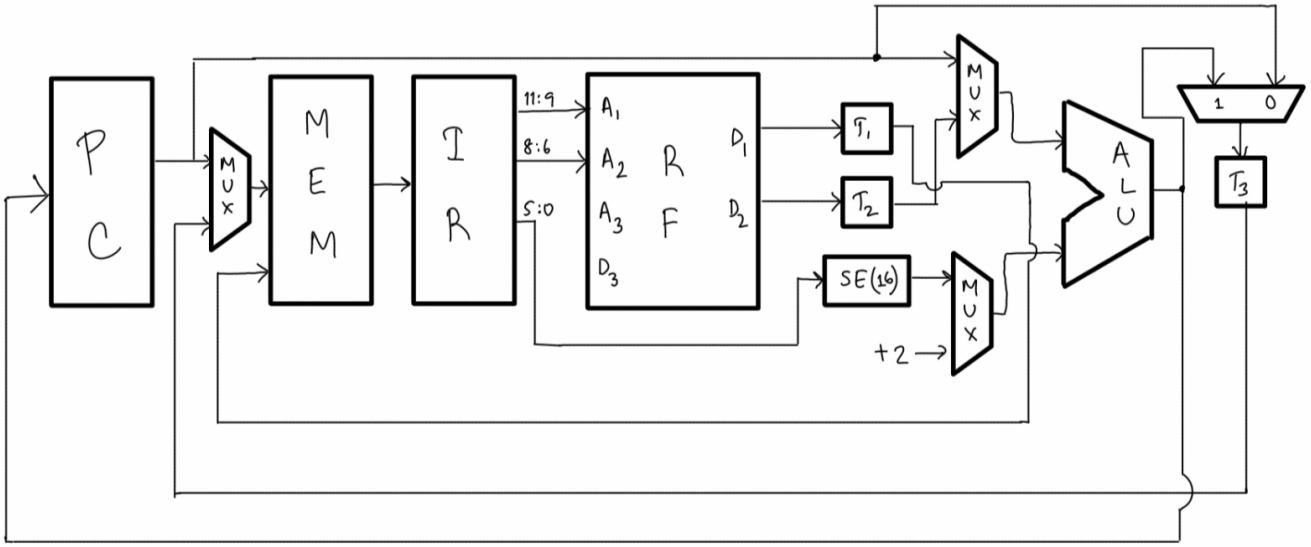

# 10. LW (opcode: 1010)



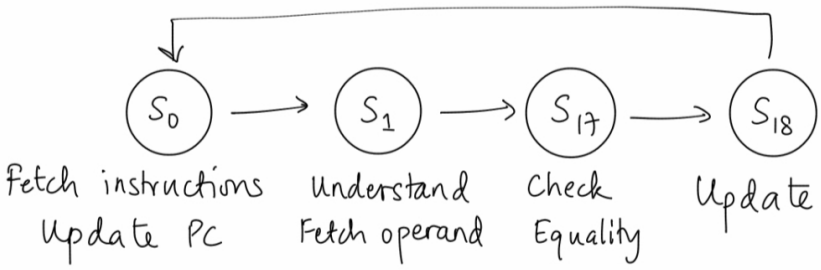



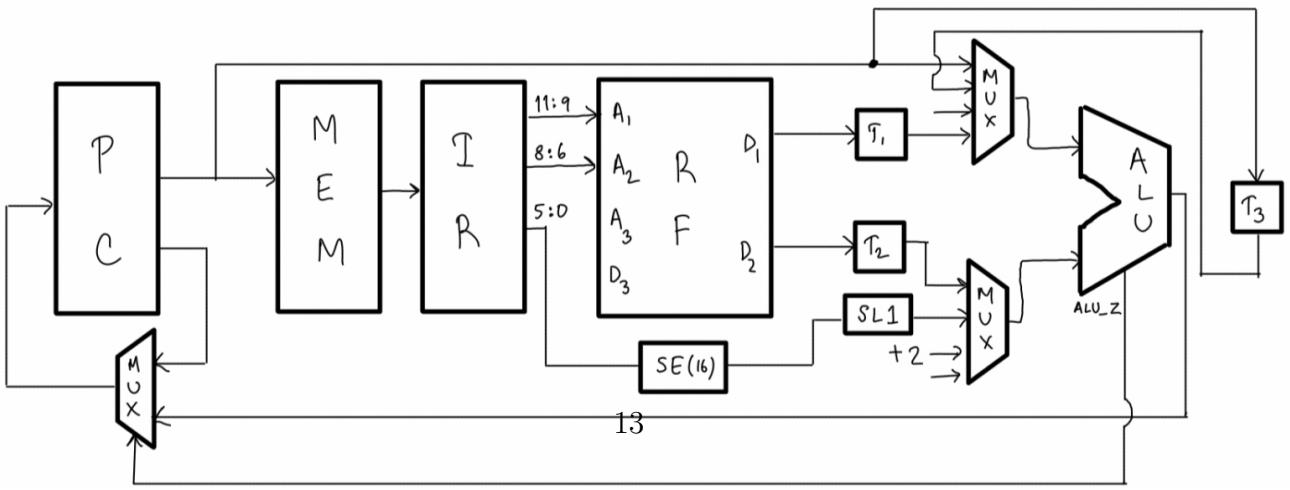
## 11. SW (opcode : 1011)



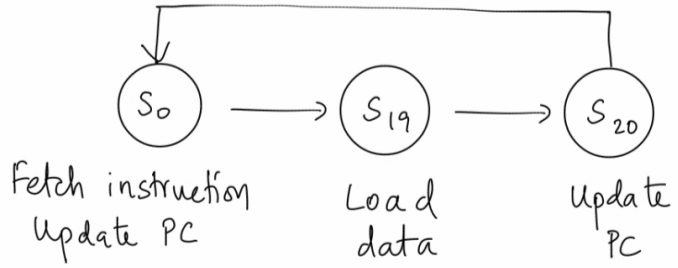



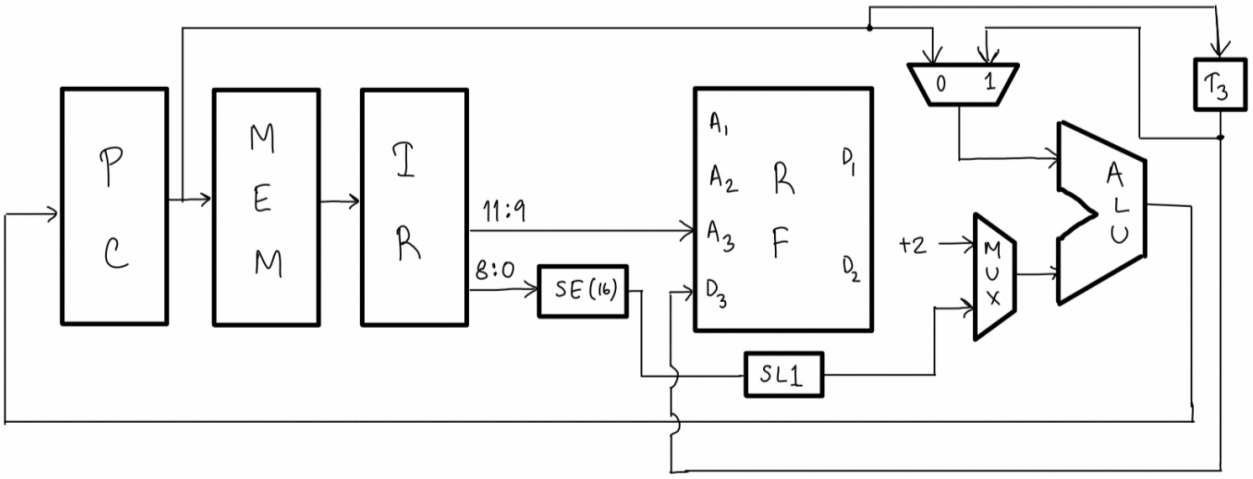

## 12. BEQ (opcode : 1100)



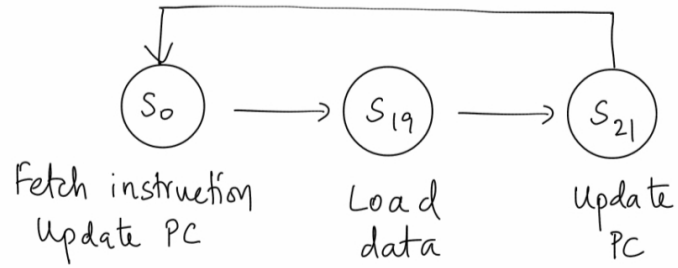



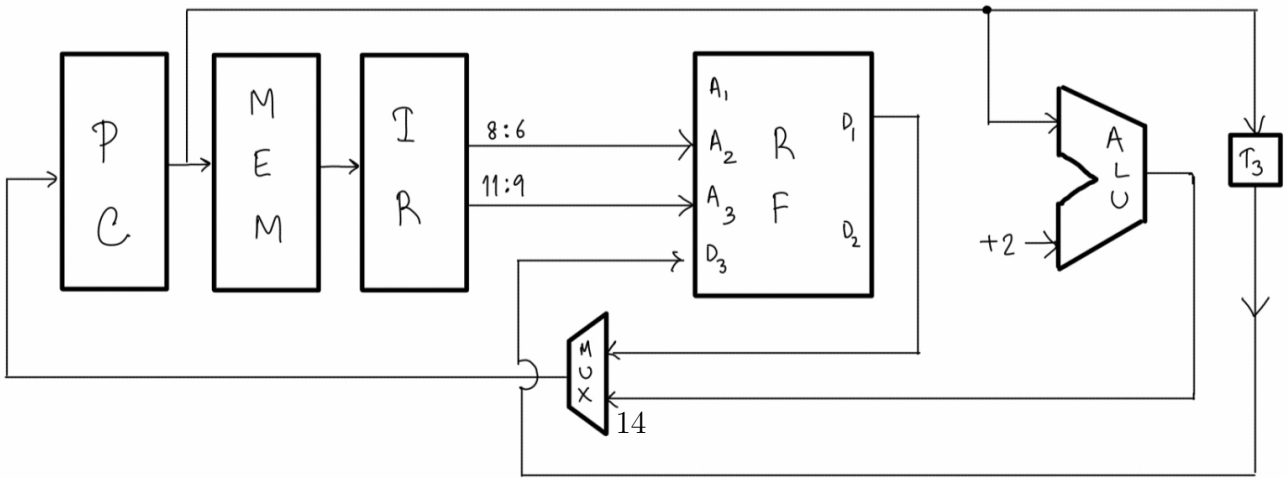
# 13. JAL (opcode : 1101)



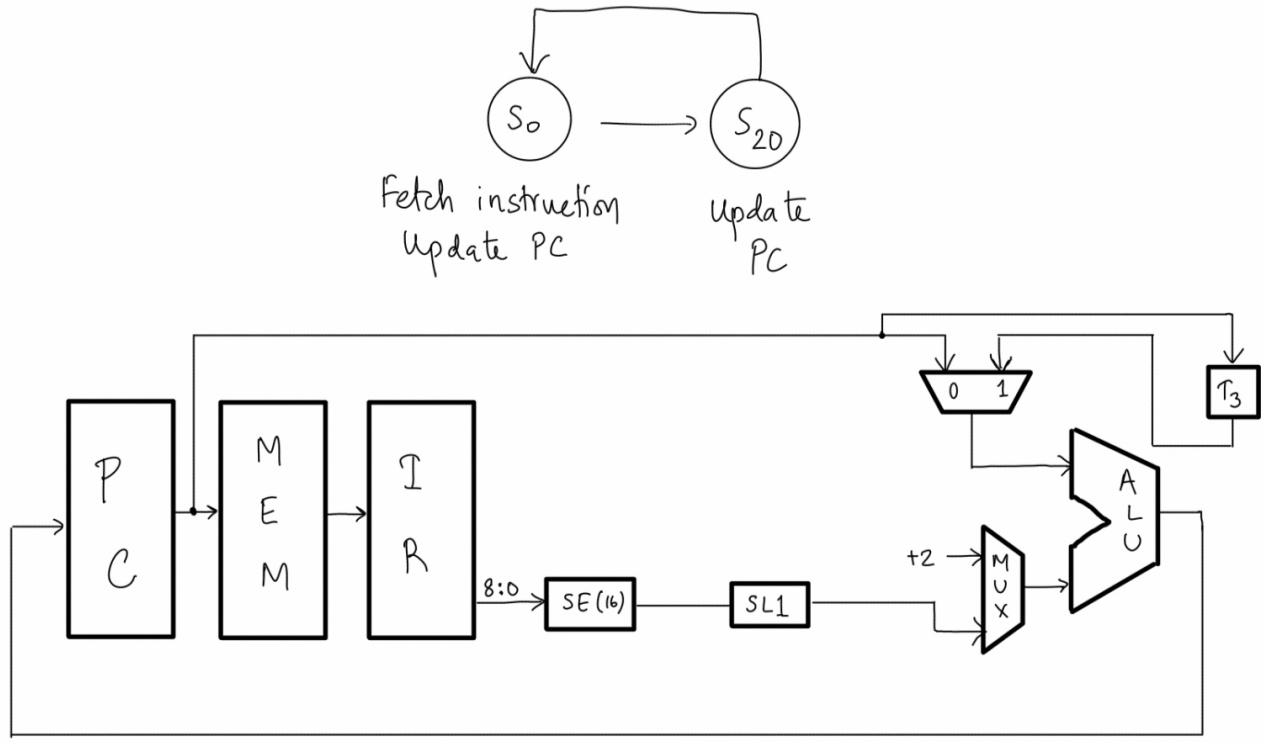




# 14. JLR (opcode : 1111)

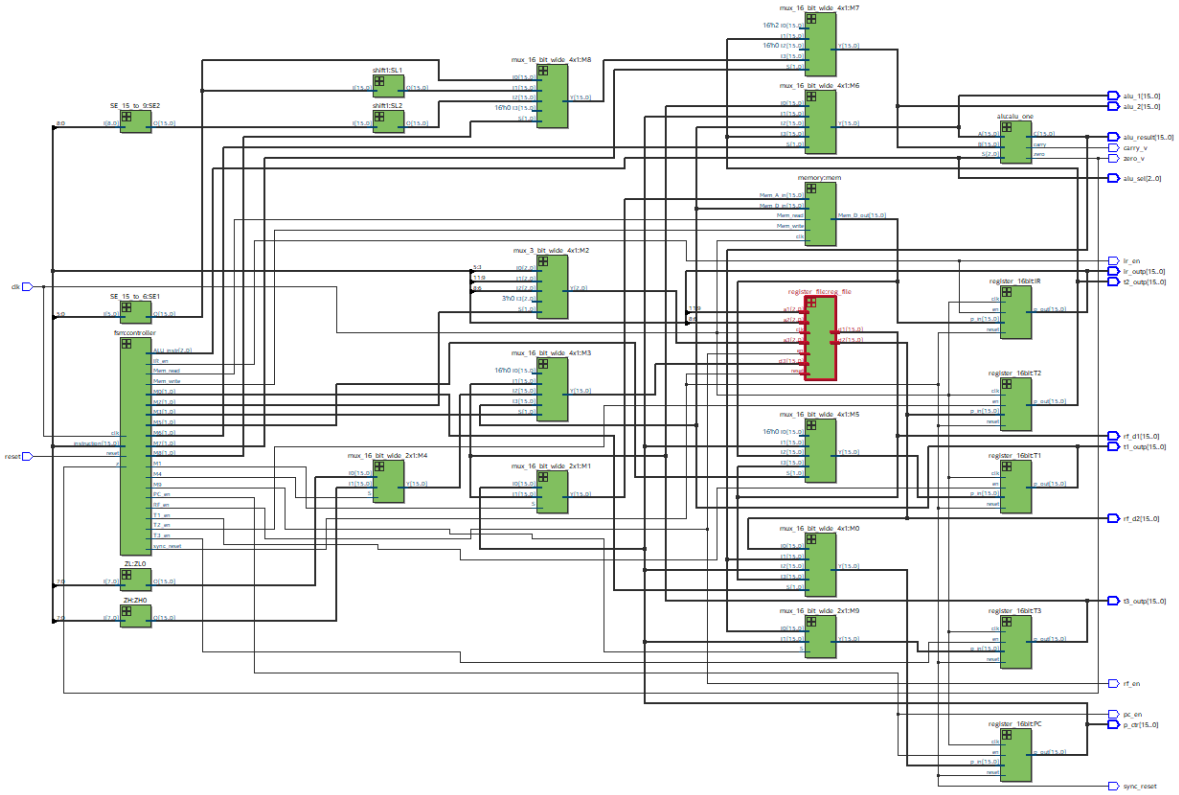


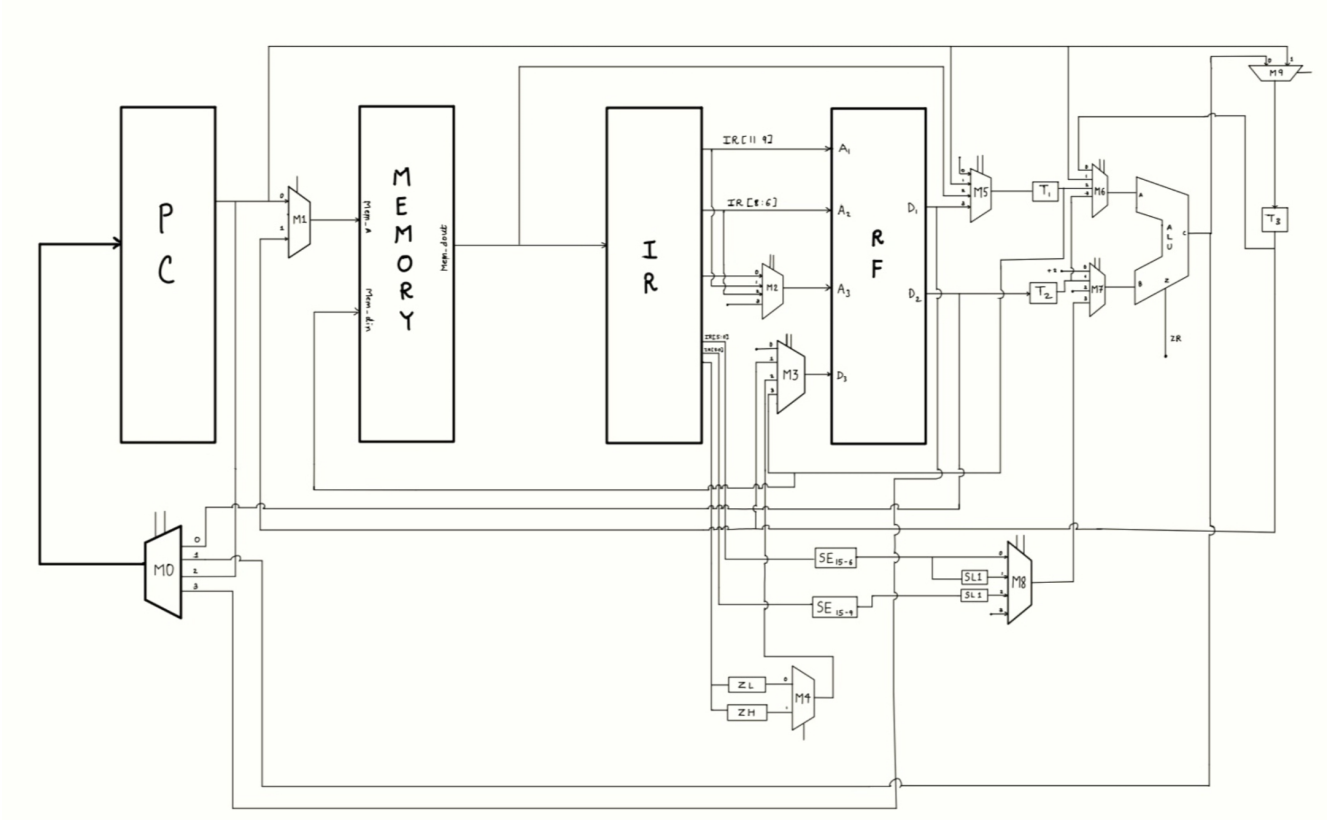



# 15. J (opcode: 1110)



# Top-Level Circuit



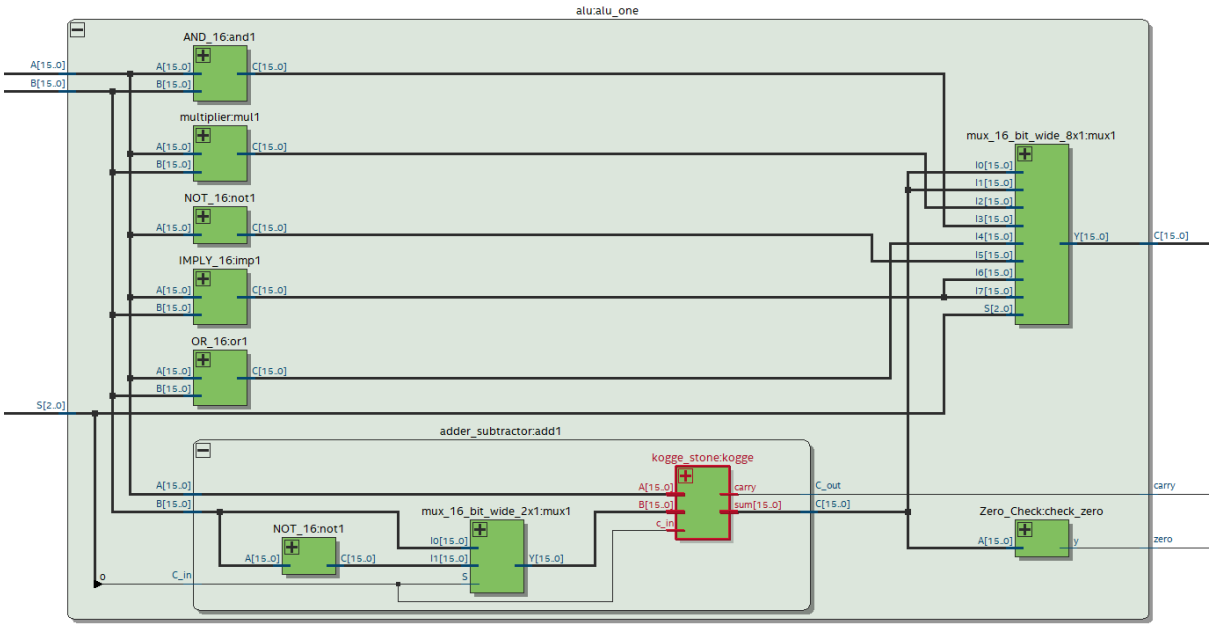


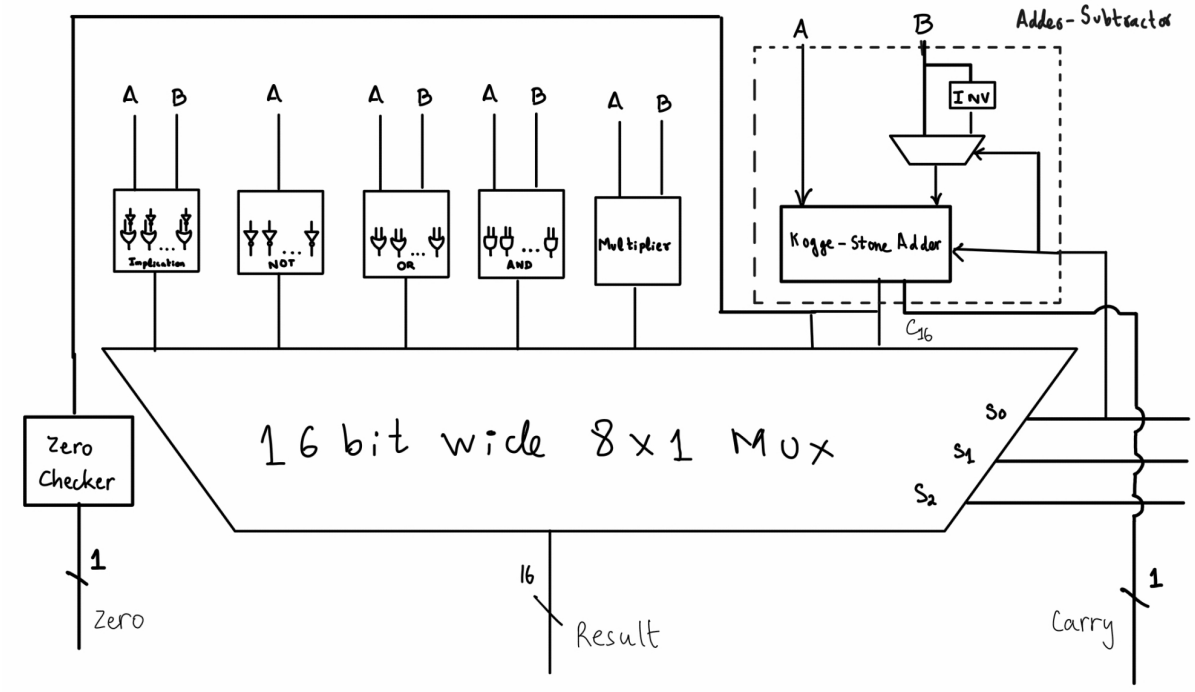
# Control Signals

| States \ Control signal | $M_0$ | $M_1$ | $M_2$ | $M_3$ | $M_4$ | $M_5$ | $M_6$ | $M_7$ | $M_8$ | $M_9$ | PC_en | Mem_Read | Mem_write | IR_en | $T_1$_en | $T_2$_en | $T_3$_en | RF_en | ALU_instr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S_0$ | 01 | 0 | | | | | 01 | 00 | | 1 | 1 | 1 | | 1 | | | 1 | | 000 |
| $S_1$ | | | | | | 11 | | | | | | | | | 1 | 1 | | | |
| $S_2$ | | | | | | | 10 | 01 | | 0 | | | | | | | 1 | | 000 |
| $S_3$ | | | 00 | 01 | | | | | | | | | | | | | | 1 | |
| $S_4$ | | | | | | | 10 | 01 | | 0 | | | | | | | 1 | | 001 |
| $S_5$ | | | | | | | 10 | 01 | | 0 | | | | | | | 1 | | 010 |
| $S_6$ | | | | | | | 10 | 11 | 00 | 0 | | | | | | | 1 | | 000 |
| $S_7$ | | | 10 | 01 | | | | | | | | | | | | | | 1 | |
| $S_8$ | | | | | | | 10 | 01 | | 0 | | | | | | | 1 | | 011 |
| $S_9$ | | | | | | | 10 | 01 | | 0 | | | | | | | 1 | | 100 |
| $S_{10}$ | | | | | | | 10 | 01 | | 0 | | | | | | | 1 | | 110 |
| $S_{11}$ | | | 01 | 10 | 0 | | | | | | | | | | | | | 1 | |
| $S_{12}$ | | | 01 | 10 | 1 | | | | | | | | | | | | | 1 | |
| $S_{13}$ | | | | | | | 11 | 11 | 00 | 0 | | | | | | | 1 | | 000 |
| $S_{14}$ | | 1 | | | | 10 | | | | | | 1 | | | | | | | |
| $S_{15}$ | | | 01 | 11 | | | | | | | | | | | | | | 1 | |
| $S_{16}$ | | 1 | | | | | | | | | | | 1 | | | | | | |
| $S_{17}$ | | | | | | | 10 | 01 | | | | | | | | | | | 001 |
| $S_{18}$ | concat($\overline{2R}$,2R) | | | | | | 00 | 11 | 01 | | 1 | | | | | | | | 000 |
| $S_{19}$ | | | 01 | 01 | | | | | | | | | | | | | | 1 | |
| $S_{20}$ | 01 | | | | | | 00 | 11 | 10 | | 1 | | | | | | | | 000 |
| $S_{21}$ | 00 | | | | | | | | | | 1 | | | | | | | | |

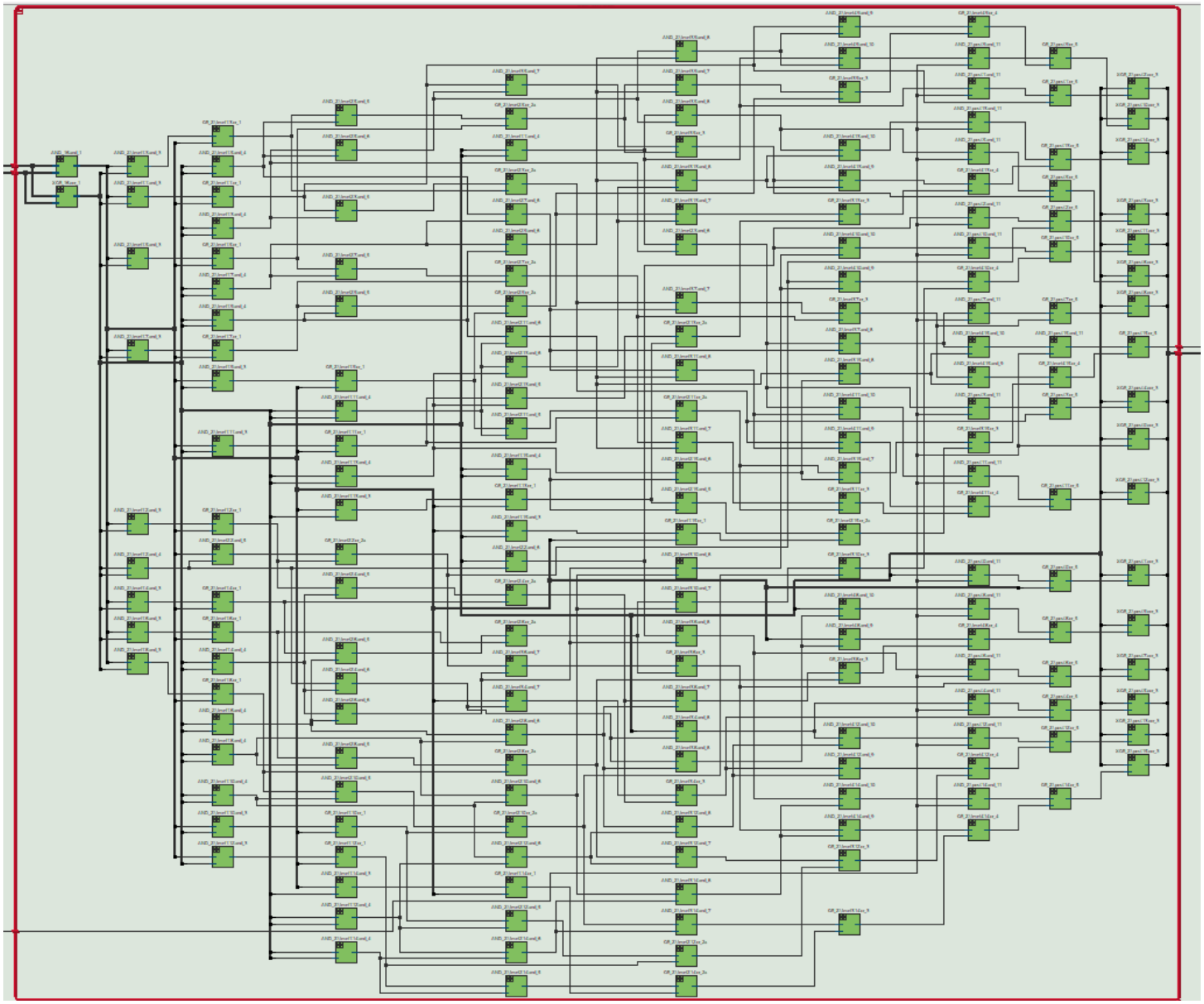
# Circuit Components

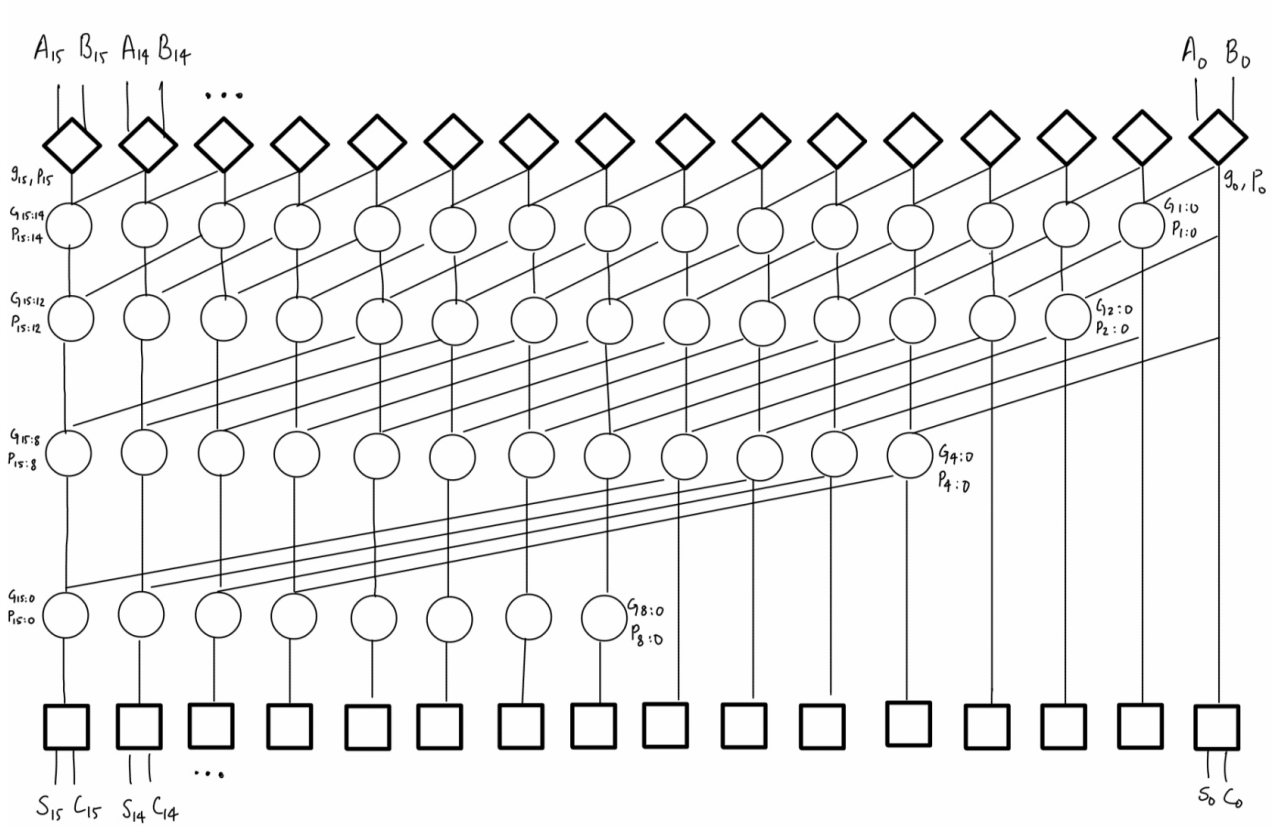
## Arithmetic and Logical Unit

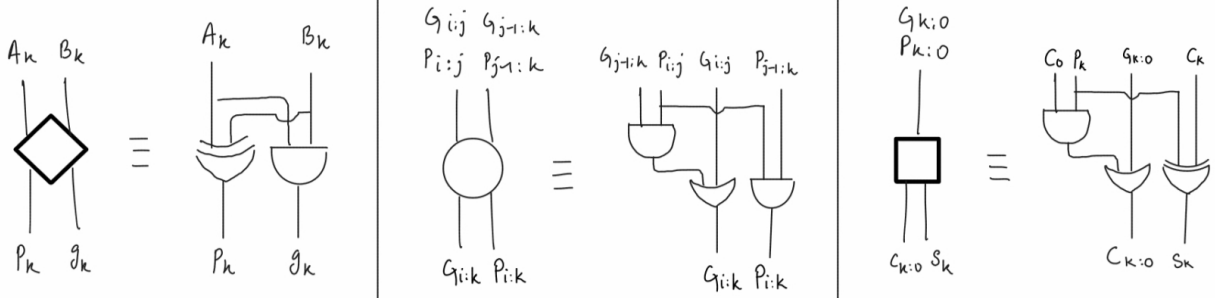


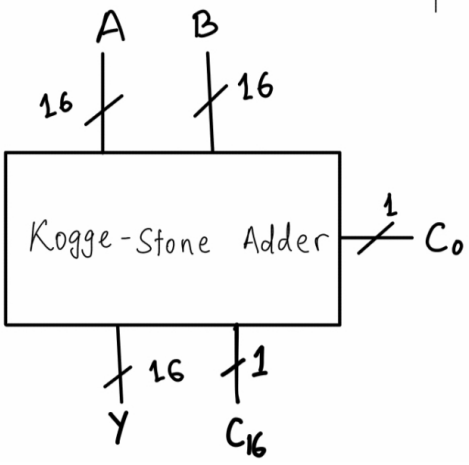



# Kogge Stone Adder

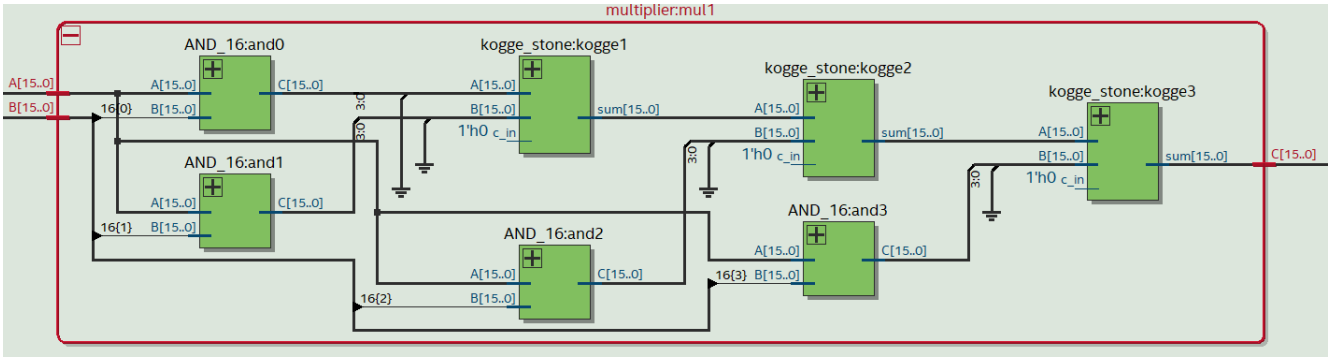


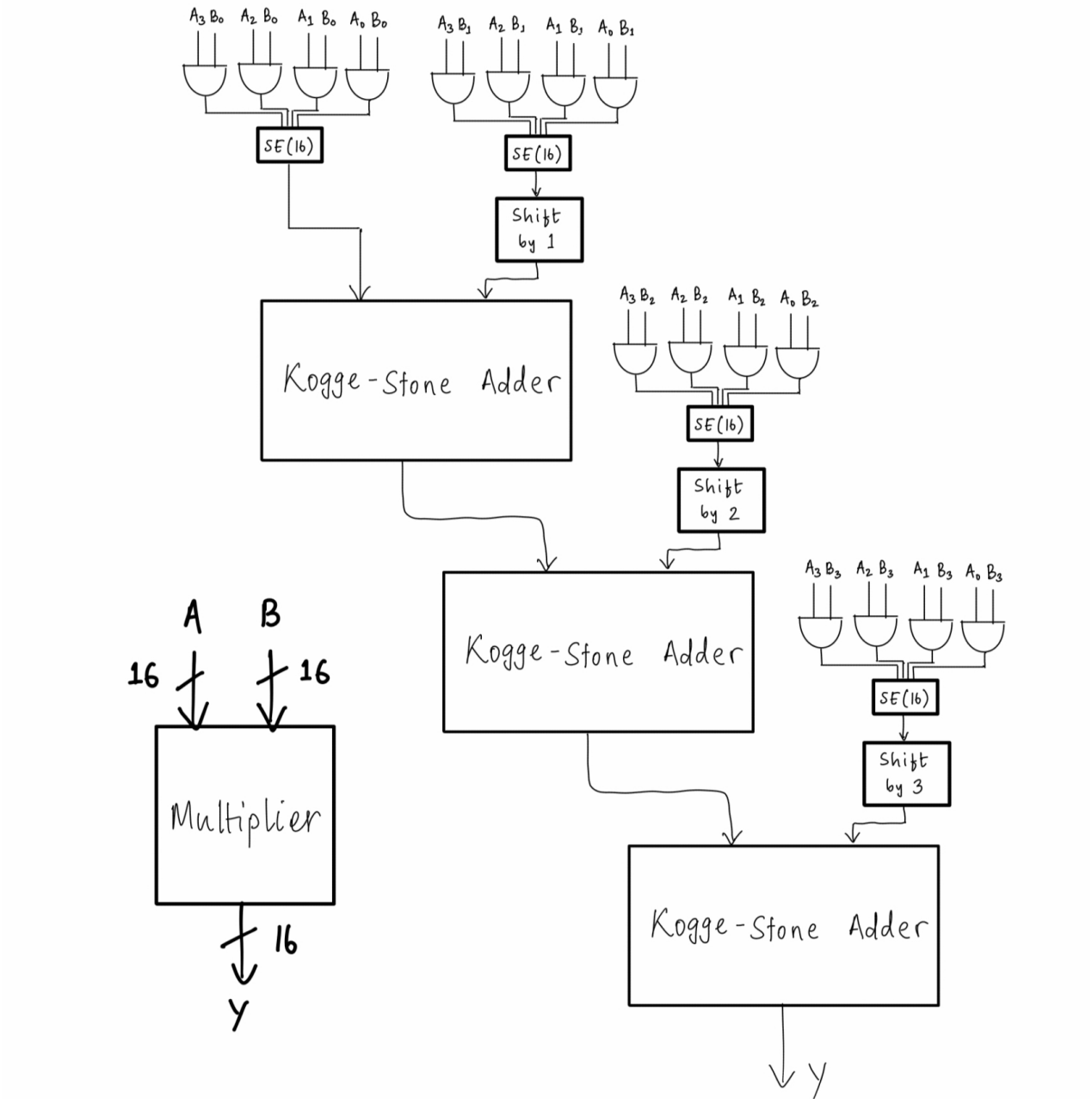

A15 B15 A14 B14
A0 B0
g15, P15
G15:14 P15:14
G15:12 P15:12
G15:8 P15:8
G15:0 P15:0
g0, P0
G1:0 P1:0
G2:0 P2:0
G4:0 P4:0
G8:0 P8:0
S15 C15 S14 C14
S0 C0


Ak Bk
Pk gk
Gi:j Gj-1:k
Pi:j Pj-1:k
Gi:k Pi:k
Gk:0
Pk:0
Ck:0 Sk
Co Pk Gk:0 Ck
Ck:0 Sk

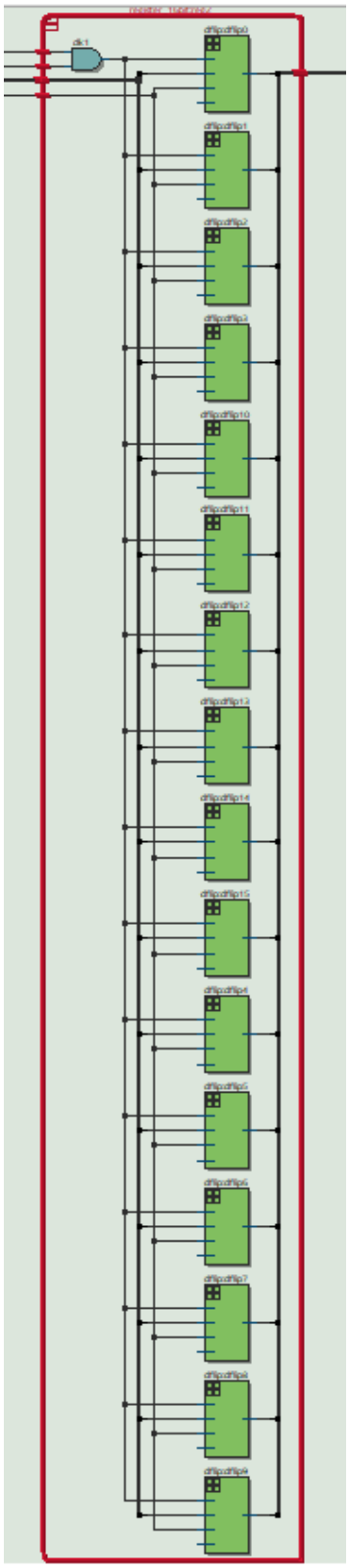

A
B
16
16
Kogge-Stone Adder
1
Co
16
1
Y
C16

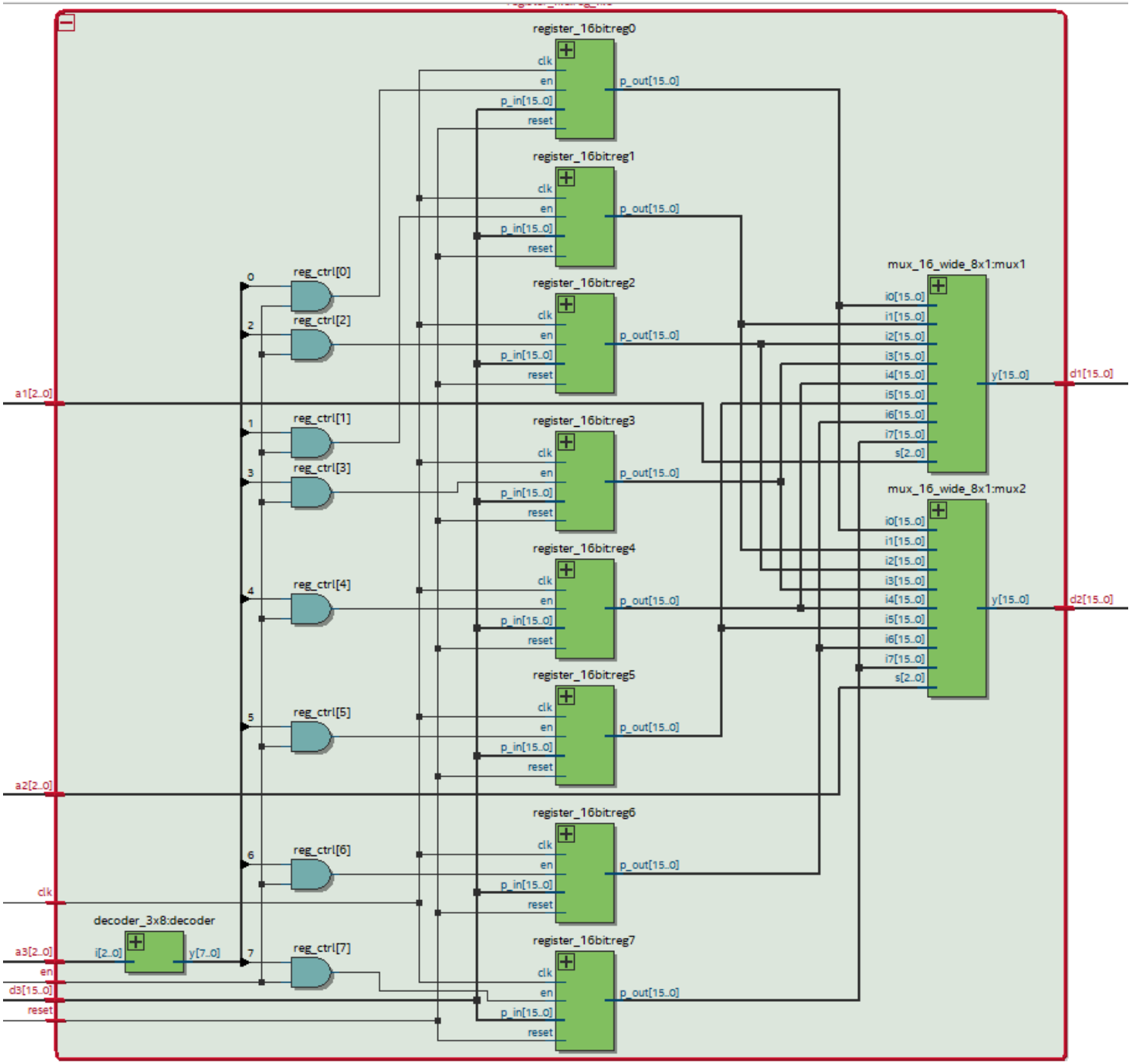
# Multiplier


multiplier:mul1
AND_16:and0
kogge_stone:kogge1
kogge_stone:kogge2
kogge_stone:kogge3
AND_16:and1
AND_16:and2
AND_16:and3
A[15..0]
B[15..0]
C[15..0]
sum[15..0]
1'h0 c_in
16{0}
16{1}
16{2}
16{3}
3:0

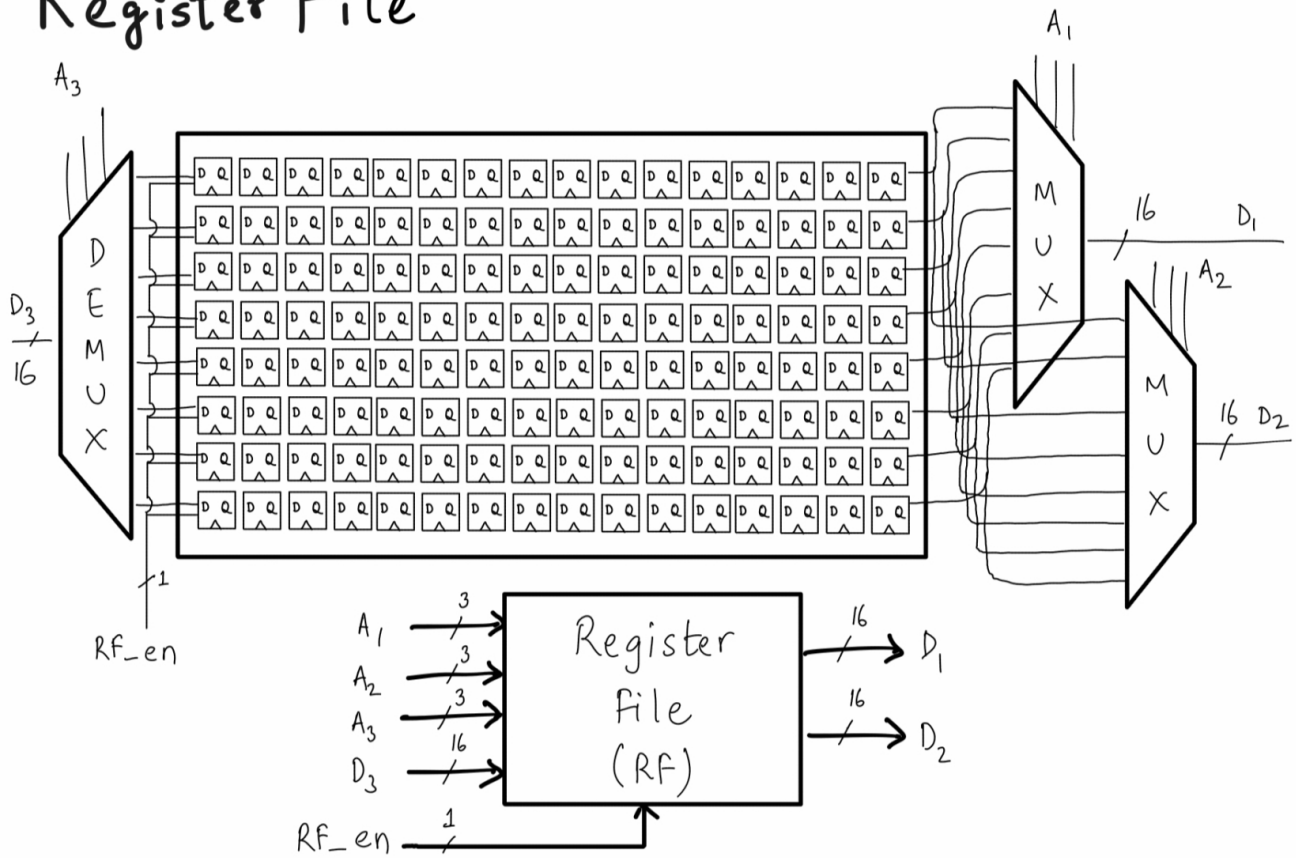

SE(16)
Shift by 1
Shift by 2
Shift by 3
Kogge-Stone Adder
Kogge-Stone Adder
Kogge-Stone Adder
A
B
16
16
Multiplier
16
Y
Y

## Registers and Register File

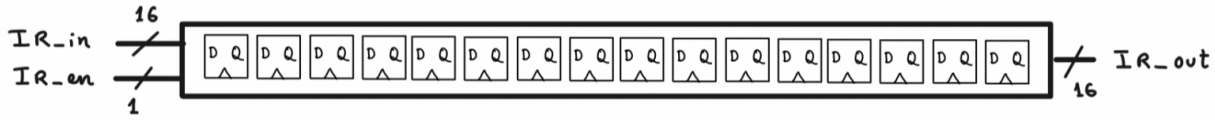



register_16bitreg0
register_16bitreg1
register_16bitreg2
register_16bitreg3
register_16bitreg4
register_16bitreg5
register_16bitreg6
register_16bitreg7
clk
en
p_in[15..0]
reset
p_out[15..0]
reg_ctrl[0]
reg_ctrl[1]
reg_ctrl[2]
reg_ctrl[3]
reg_ctrl[4]
reg_ctrl[5]
reg_ctrl[6]
reg_ctrl[7]
0
1
2
3
4
5
6
7
mux_16_wide_8x1:mux1
mux_16_wide_8x1:mux2
i0[15..0]
i1[15..0]
i2[15..0]
i3[15..0]
i4[15..0]
i5[15..0]
i6[15..0]
i7[15..0]
s[2..0]
y[15..0]
d1[15..0]
d2[15..0]
a1[2..0]
a2[2..0]
a3[2..0]
clk
en
d3[15..0]
reset
decoder_3x8:decoder
i[2..0]
y[7..0]

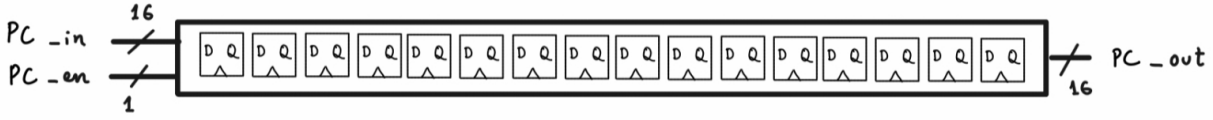
# Register File



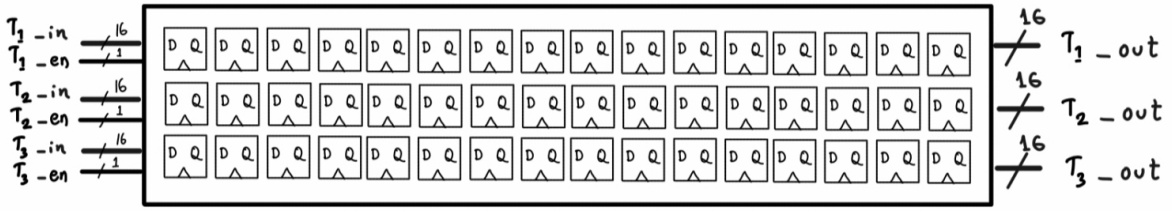

# Instruction Register




# Program Counter




# Temporary Registers



# Supporting Simulations

In this section, we will provide the ModelSim simulation for each instruction and provide a brief explanation for each instruction.

## ADD

### Instruction

```
signal Mem : memory_array := (
    0 => "10010000",
    1 => "11000101",    -- places 197 into reg A (000)
    2 => "10010010",
    3 => "11101011",    -- places 235 into reg B (001)
    4 => "00000000",
    5 => "01010000",    -- adds values in reg A (000) to
                        -- reg B (001) and stores them
                        -- in reg C (010)
    others => (others => '0'));
```

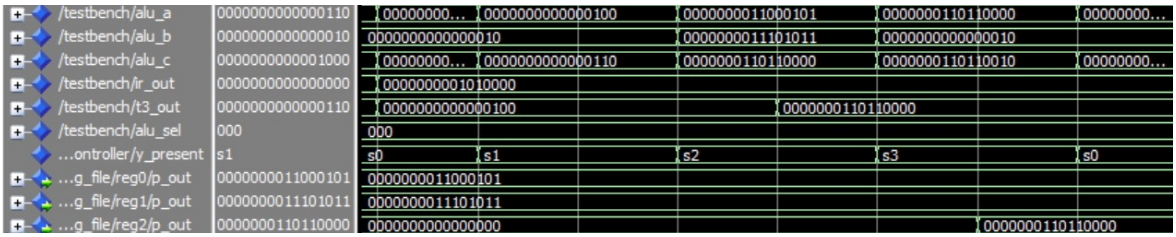



Figure 8.1: Execution of ADD instruction

### Explanation

The state transitions observed in the picture are :
$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3$
We ignore the state transitions involved in loading the data using the LLI command. The state transitions occur according to the FSM we have defined in our instruction table. We can see that the value stored in reg A (000) is 11000101 and that in reg B (001) is 11101011 get added to yield 110110000, which is stored in reg C (010).

# SUB

## Instruction

```
signal Mem : memory_array := (
    0 => "10010000",
    1 => "00001010",    -- places 10 into reg A (000)
    2 => "10010010",
    3 => "00100010",    -- places 34 into reg B (001)
    4 => "00100000",
    5 => "01010000",    -- subtracts values in reg A (000) from
                        -- reg B (001) and stores them
                        -- in reg C (010)
    others => (others => '0'));
```

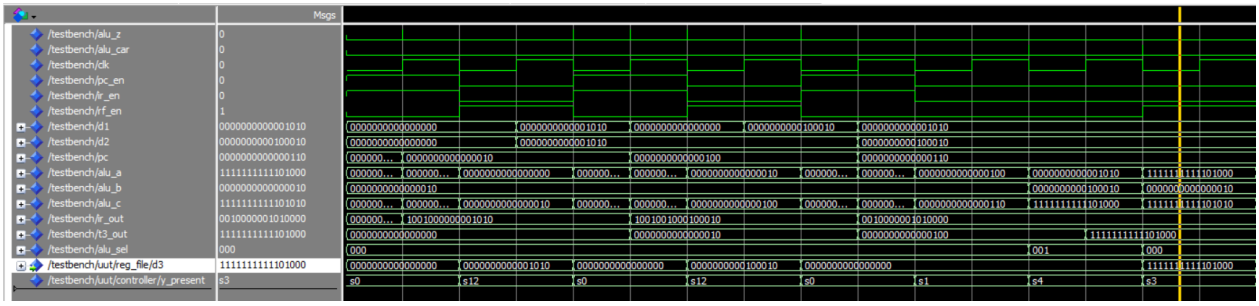



Figure 8.2: Execution of SUB instruction

## Explanation

The state transitions observed in the picture are :

$s_0 \rightarrow s_1 \rightarrow s_4 \rightarrow s_3$

We ignore the state transitions involved in loading the data using the LLI command. The state transitions occur according to the FSM we have defined in our instruction table. We can see that the value stored in reg A (000) is 1010 and that in reg B (001) is 100010. Subtraction occurs in 2's complement to yield 101000, with a sign extension of 1's at the front signifying that the value is negative. This value is stored in reg C (010).

# MUL

### Instruction

```
signal Mem : memory_array := (
    0 => "10010000",
    1 => "00001000",      -- Load binary 8 in Reg A (000)
    2 => "10010010",
    3 => "00000011",      -- Load binary 3 in Reg B (001)
    4 => "00110000",
    5 => "01010000",      -- Multiply 8 and 2, put in Reg C (010)

    others => (others => '0'));
```

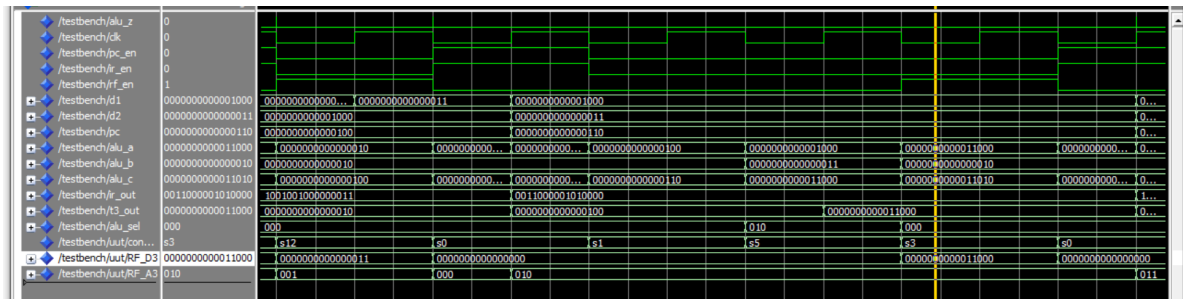



Figure 8.3: Execution of MUL instruction

### Explanation

The state transitions observed in the picture are :
$s_0 \rightarrow s_1 \rightarrow s_5 \rightarrow s_3$
We ignore the state transitions involved in loading the data using the LLI command. The state transitions occur according to the FSM we have defined in our instruction table. We can see that the value stored in reg A (000) is 1000 and that in reg B (001) is 011. Multiplication occurs for the lower 4 bits of each loaded value to yield 11000 which is stored in reg C (010).

## ADI

### Instruction

```
signal Mem : memory_array := (
    0 => "10010000",
    1 => "11010001",     -- places 209 into reg A (000)
```

```
    2 => "00010000",
    3 => "10001011",     -- adds 11 (immediate value to reg A (000)
                         -- and stores the result in reg B (010))

    others => (others => '0'));
```

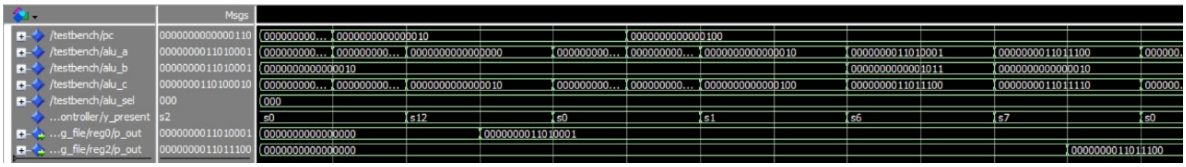



Figure 8.4: Execution of ADI instruction

### Explanation

The state transitions observed in the picture are :
$s_0 \rightarrow s_1 \rightarrow s_6 \rightarrow s_7$
We ignore the state transitions involved in loading the data using the LLI command. The state transitions occur according to the FSM we have defined in our instruction table. We can see that the value stored in reg A (000) is 11010001 we add the immediate value of 1011 to it. This gives the value 011011100, which is stored in reg C (010).

## AND

### Instruction

```
signal Mem : memory_array := (
    0 => "10010000",
    1 => "00001010",     -- places 10 into reg A (000)
    2 => "10010010",
    3 => "00100010",     -- places 34 into reg B (001)
    4 => "01000000"
    5 => "01010000"      -- ANDs values in reg A (000), reg B (001)
                         -- and stores them in reg C (010)
    others => (others => '0'));
```

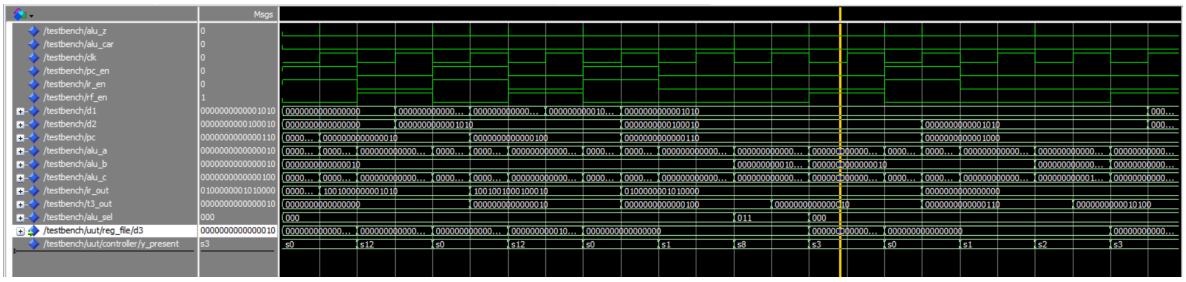


Figure 8.5: Execution of AND instruction

### Explanation

The state transitions observed in the picture are :
$s_0 \rightarrow s_1 \rightarrow s_8 \rightarrow s_3$
We ignore the state transitions involved in loading the data using the LLI command. The state transitions occur according to the FSM we have defined in our instruction table. We can see that the value stored in reg A (000) is 00001010 and that in reg B (001) is 00100010. Performing bitwise AND on these, we get 00000010 which is stored in reg C (010).

## ORA

### Instruction

```
signal Mem : memory_array := (
    .
    .
    .
    6 => "10010110",
    7 => "11011001",  --Load 11011001 to Reg D
    8 => "01010010",
    9 => "11100000",  --Reg B OR Reg D and put in Reg E
    .
    .
    .
    others => (others => '0'));
```

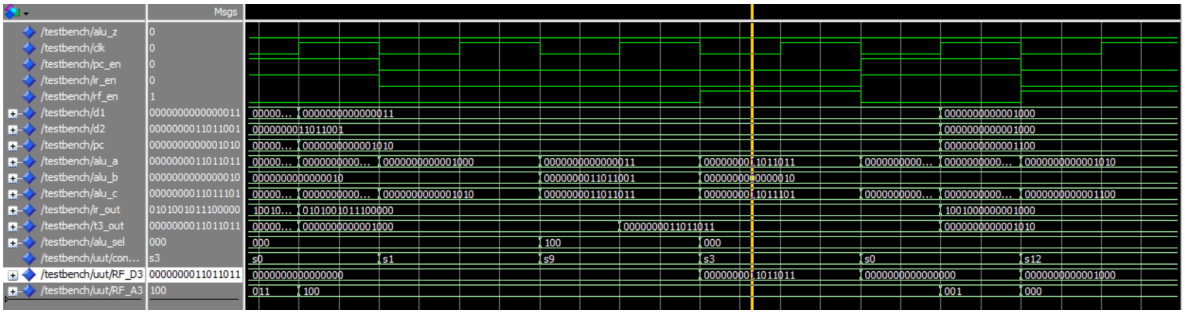



Figure 8.6: Execution of ORA instruction

### Explanation

The state transitions observed in the picture are :
$s_0 \rightarrow s_1 \rightarrow s_9 \rightarrow s_3$
We ignore the state transitions involved in loading the data using the LLI command. 11011001 is loaded to reg D (011) using LLI. Reg B (001) already contains 10 from previous instructions. The state transitions occur according to the FSM we have defined in our instruction table. We can see that the value stored in reg E (100) is 11011011. since we performed bitwise OR for them.

## IMP

### Instruction

```
-- IMP Instruction
     signal Mem : memory_array := (
        0 => "10010000",
        1 => "10111001",    -- places 185 into reg A (000)
        2 => "10010010",
        3 => "01100000",    -- places 96 into reg B (001)
        4 => "01100000",
        5 => "01100000",    -- adds values in reg A (000),
                            -- reg B (001) and stores them
                            -- in reg C (100)
        others => (others => '0'));
```

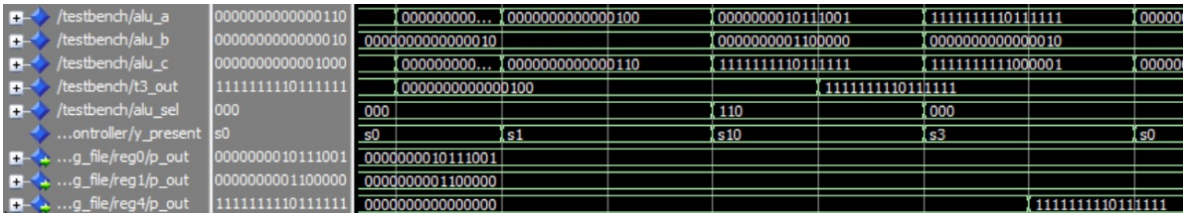



Figure 8.7: Execution of IMP instruction

### Explanation

The state transitions observed in the picture are :
$s_0 \rightarrow s_1 \rightarrow s_{10} \rightarrow s_3$
We ignore the state transitions involved in loading the data using the LLI command. The state transitions occur according to the FSM we have defined in our instruction table. We can see that the value stored in reg A (000) is 0000000010111001 and that in reg B (001) is 0000000001100000. Applying the content of $B \rightarrow A$ bitwise, we we get 1111111110111111, which can be observed to be the value stored in reg C (010).

## LHI

### Instruction

```
signal Mem : memory_array := (
    0 => "10000000",
    1 => "00011010",      --Load binary 26 in Reg A
    others => (others => '0'));
```

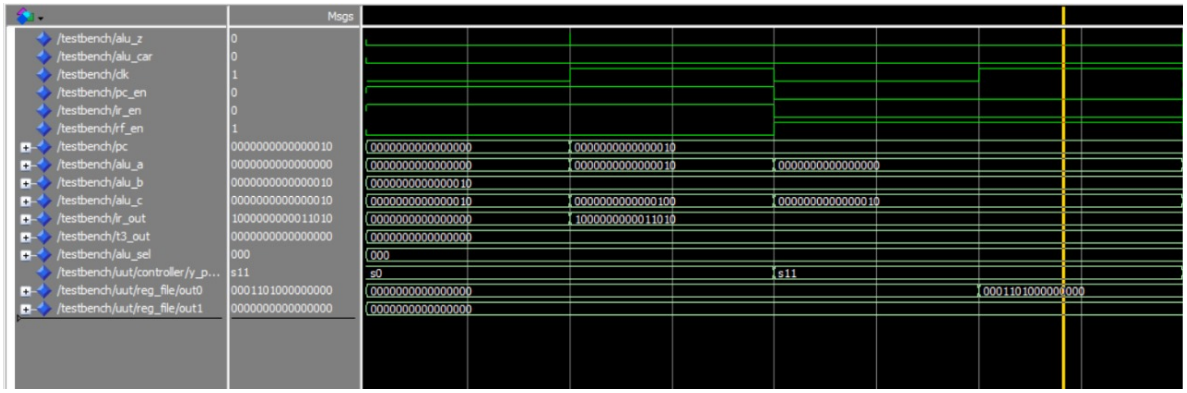



Figure 8.8: Execution of LHI instruction

### Explanation

The state transitions observed in the picture are :
$s_0 \rightarrow s_{11}$
The state transitions occur according to the control logic of our FSM. We can see that the value of 00011010 is loaded in reg A (000) with 8 0's padded to the right and only the output associated with this register changes and not for other registers like reg B (001).

## LLI

### Instruction

```
signal Mem : memory_array := (
    0 => "10010000",
    1 => "00001000",      --Load binary 8 in Reg A
    others => (others => '0'));
```

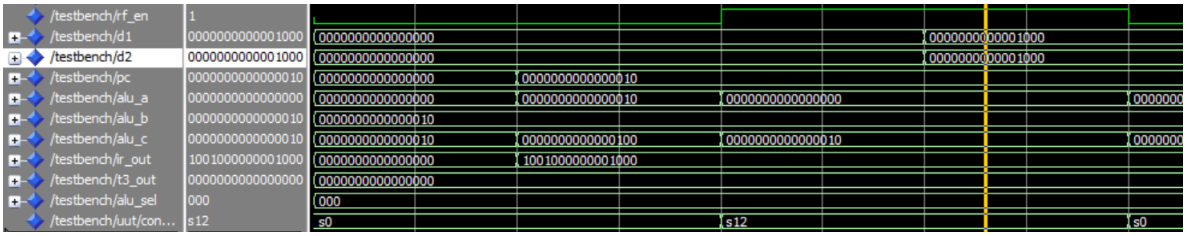



Figure 8.9: Execution of LLI instruction

### Explanation

The state transitions observed in the picture are :
$s_0 \rightarrow s_{12}$
The state transitions occur according to the control logic of our FSM. We can see that the value of 00001000 is loaded in reg A (000) with 8 0's padded to the left and only the output associated with this register changes and not for other registers like reg B (001).

## LW

### Instruction

```
signal Mem : memory_array := (
```

```
    0 => "10100000",
    1 => "01010000",      --Load Reg B + 16 th memory
                          --address onto Reg A
    16 => "10010010",
    17 => "00000011",
    others => (others => '0'));
```

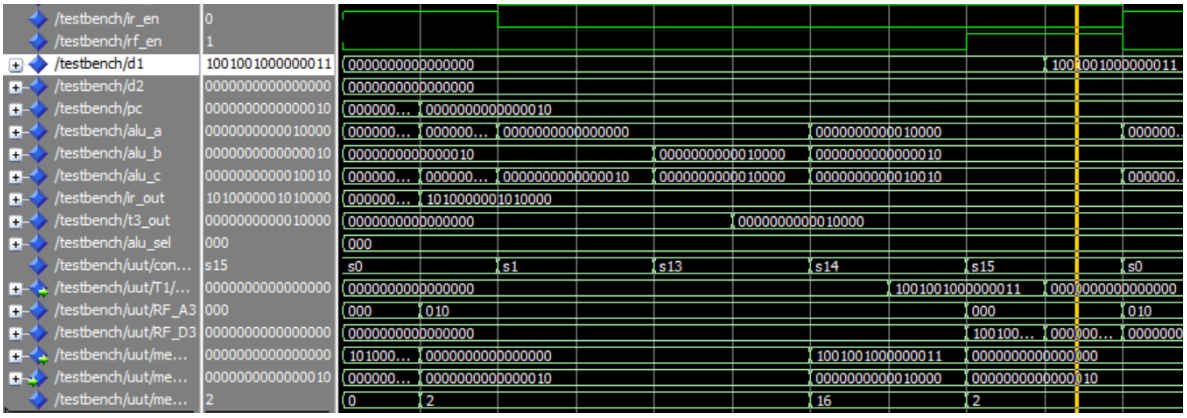



Figure 8.10: Execution of LW instruction

### Explanation

The state transitions observed in the picture are :
$s_0 \rightarrow s_1 \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{15}$
16th and 17th places in memory together hold the value 1001001000000011. First, we access reg B, add 16 to it by immediate addressing. Since reg B contains 0 for now, so we are directed to the 16th memory position. This returns the right value and is stored in reg A.

## SW

### Instruction

```
signal Mem : memory_array := (
    0 => "10010000",
    1 => "00011010",      -- Load binary 26 in Reg A
    2 => "10010010",
    3 => "00100010",      -- Load binary 34 in Reg 1
    4 => "10110000",
    5 => "01001100",      -- Load value of reg A into
```

```
                        -- Mem Address with Imm 12
    others => (others => '0'));
```

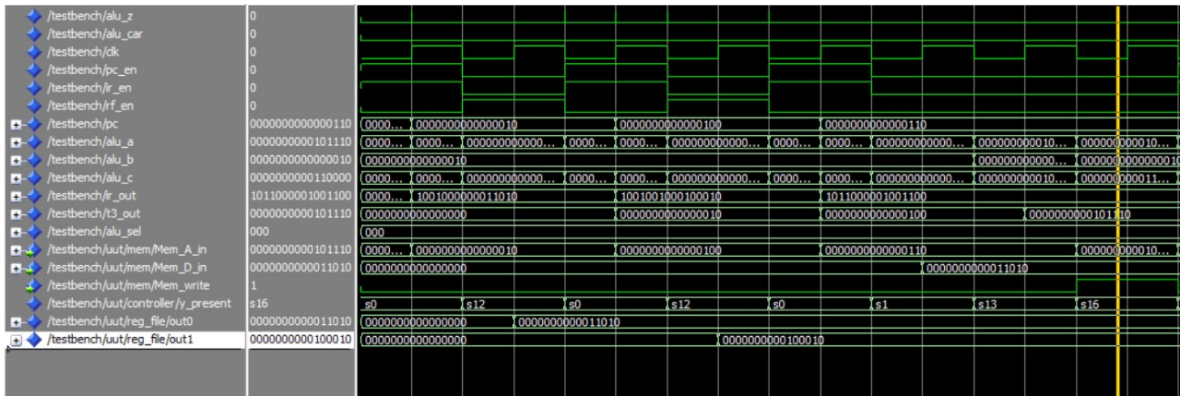



Figure 8.11: Execution of SW instruction

### Explanation

The state transitions observed in the picture are :
$s_0 \rightarrow s_1 \rightarrow s_{13} \rightarrow s_{16}$
The state transitions occur according to the control logic of our FSM. We can see that the value of 11010 is loaded in reg A (000) and the value 100010 is loaded in reg B (001). The immediate value in the instruction is 1100, and this is added to the value at reg B to give 101110. We can see that the address of the memory write location is 101110 while the data entered is 11010, which is the value at reg A.

## BEQ

### Instruction

```
signal Mem : memory_array := (
    .
    .
    .
    11 => "00001000",     -- Load binary 8 in Reg A
    12 => "10010010",
    13 => "00000010",     -- Load binary 2 in Reg B
    14 => "11000000",
    15 => "01000110",  -- Branch to 6 instructions ahead
                          -- if content of RegA = RegB
```

```
16 => "10010000",
17 => "00001000",      -- Load binary 8 in Reg A
18 => "10010010",
19 => "00001000",      -- Load binary 8 in Reg B
20 => "11000000",
21 => "01000110",  -- Branch to 6 instructions ahead
                        -- if content of RegA = RegB
.
.
.
others => (others => '0'));
```

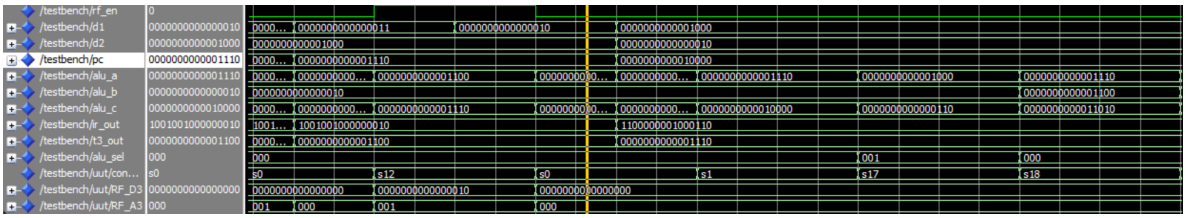



Figure 8.12: Execution of BEQ instruction (where contents of $A \neq B$), showing initial PC

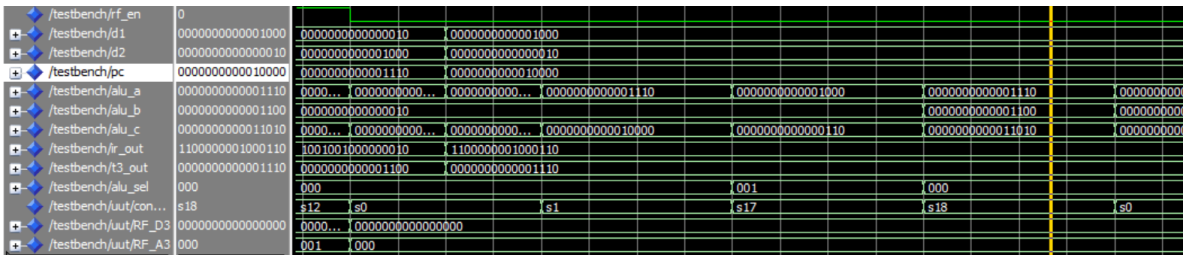



Figure 8.13: Execution of BEQ instruction (where contents of $A \neq B$), showing final PC

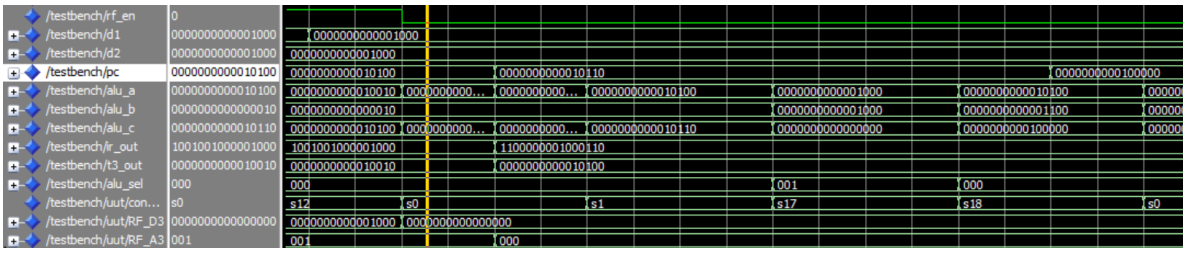



Figure 8.14: Execution of BEQ instruction (where contents of $A = B$), showing initial PC

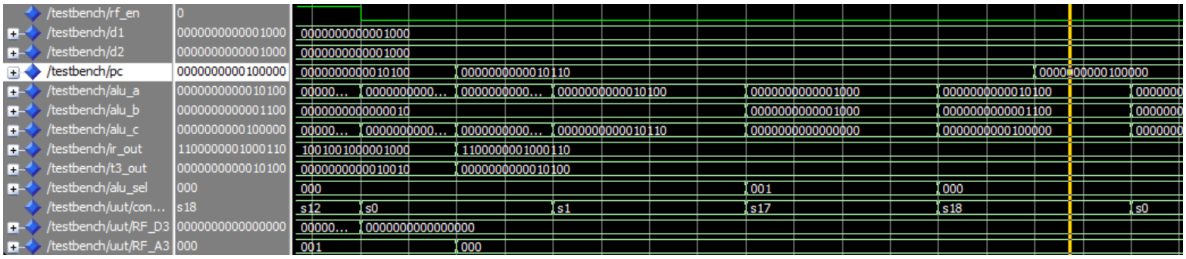



Figure 8.15: Execution of BEQ instruction (where contents of $A = B$), showing final PC

### Explanation

The state transitions observed in the picture are :
$s_0 \rightarrow s_1 \rightarrow s_{17} \rightarrow s_{18}$
We ignore the state transitions involved in loading the data using the LLI command. The state transitions occur according to the FSM we have defined in our instruction table.

In the first case, we have A storing 8 and B storing 2. Since $8 \neq 2$, we'll not observe a jump. Initial value of PC was 14 and final value is 16, which is the normal update done in updation of PC in $s_0$.

But, in the next case, we have A and B both storing 8. So, PC is observed to jump from 20 initially to 32 finally.

## JAL

### Instruction

```
-- JAL Instruction
    signal Mem : memory_array := (
        0 => "00000000",
        1 => "00000000",    -- put as a dummy instruction to
                            -- increment PC for more specificity
        2 => "11010110",
        3 => "00101011",    -- Stores address of PC into
                            -- Register A (011) and
                            -- jumps to (PC + 2*43)
        others => (others => '0'));
```

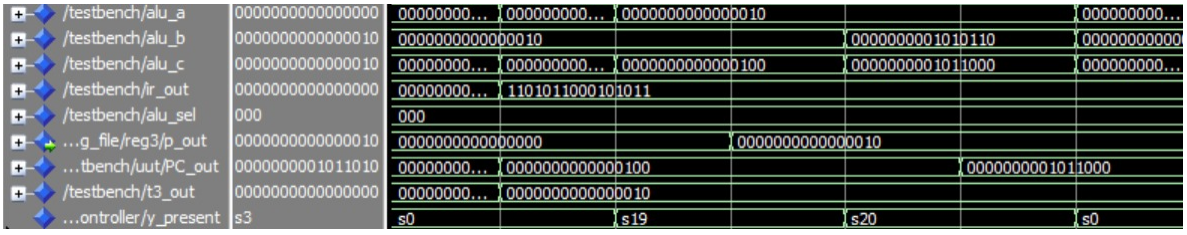



Figure 8.16: Execution of JAL instruction

### Explanation

The state transitions observed in the picture are :
$s_0 \rightarrow s_{19} \rightarrow s_{20}$
For testing this instruction, initially we have put a dummy instruction to simply increment the program counter. This is done to ensure more specificity. The state transitions occur according to the control logic of the FSM. We can observe that the address of the PC (0000000000000010) gets stored into the reg A (011) and the value stored in the PC changes to (2 + 2*43). And after this the PC starts executing from there.

## JLR

### Instruction

```
signal Mem : memory_array := (
    0 => "10010000",
    1 => "00011010",      --Load binary 26 in Reg A
    2 => "10010010",
    3 => "00100010",      --Load binary 34 in Reg 1
    4 => "11110000",
    5 => "01000000",      --Branch to address in reg B
    others => (others => '0'));
```

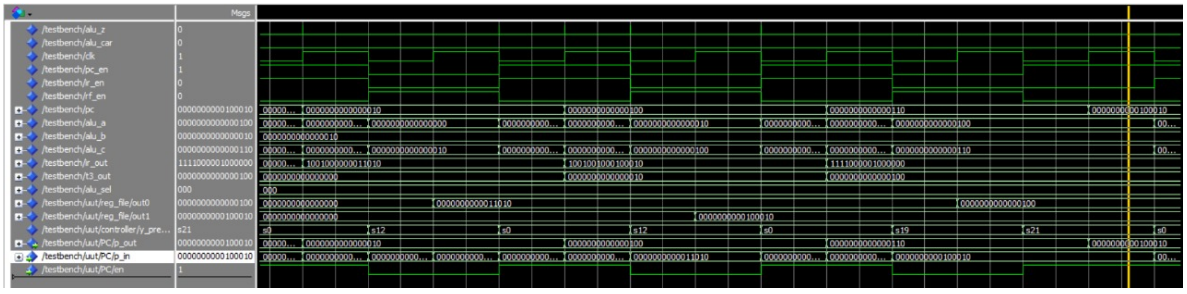


Figure 8.17: Execution of JLR instruction

### Explanation

The state transitions observed in the picture are :
$s_0 \rightarrow s_{19} \rightarrow s_{21}$
The state transitions occur according to the control logic of our FSM. We can see that the value of 11010 is loaded in reg A (000) and the value 100010 is loaded in reg B (001). After the instruction, the value of the JLR instruction, i.e. 100, is stored in reg A, while the input value of PC becomes the value stored in reg B. After this state, PC starts executing from the value at reg B.

## J

### Instruction

```
signal Mem : memory_array := (
    .
    .
    .
    32 => "11100000",
    33 => "00001100",  -- Branches unconditionally
                       -- 12 instructions ahead
    others => (others => '0'));
```

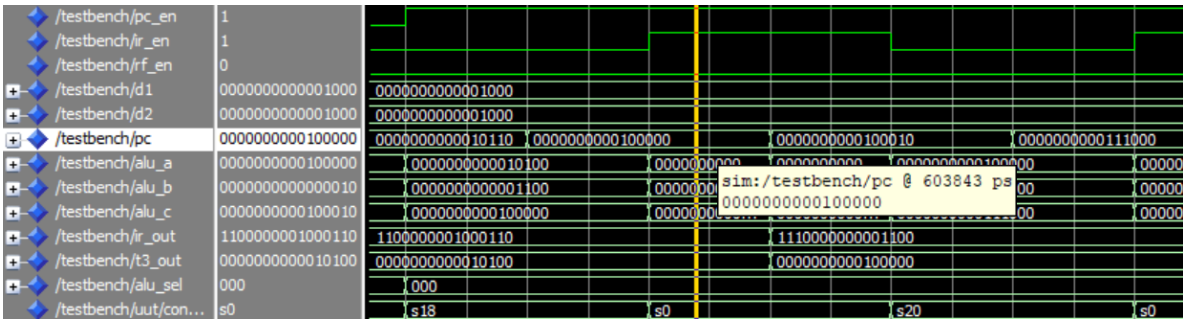



Figure 8.18: Execution of J instruction, showing initial PC

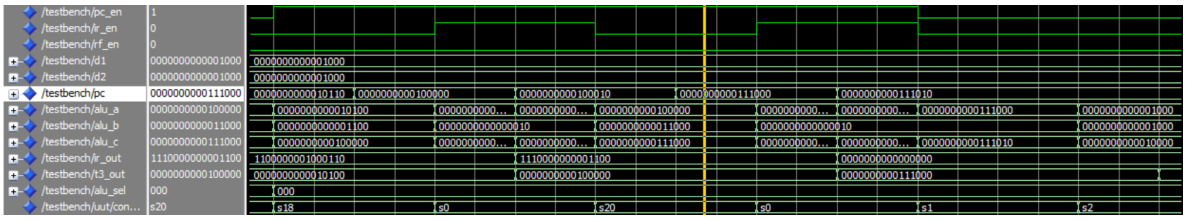



Figure 8.19: Execution of J instruction, showing final PC

### Explanation

The state transitions observed in the picture are :
$s_0 \rightarrow s_{20}$
PC is initially at 32. We give the instruction to make it jump 12 instructions ahead. This happens unconditionally, so we observe the final PC is at 44.

# Python Code for Binary Instruction Generation

To facilitate the testing of all instructions and improve the user interface for the CPU we developed a python script for loading instructions into memory. This script provides an interface that accepts assembly language instructions as input, converts them into their corresponding binary representations, and stores them in a simulated memory while initializing the simulation.

The memory is simulated as a 512-byte structure, divided into two equal parts:

- **Instruction Memory:** The first 256 bytes are reserved for storing assembly instructions. This allows for a maximum of 127 instructions (each instruction takes 2 bytes).
- **Data Memory:** The remaining 256 bytes are used to store 16-bit data values for testing purposes.

The script includes the following features:

- **Support for Multiple Line Inputs:** Users can input assembly programs line by line, enabling the testing of complex programs.
- **Memory Segregation:** The 128th instruction in the memory ensures that the program counter (PC) jumps back to address `0`. This prevents the CPU from interpreting data values in memory as instructions.

- **Memory Padding:** If the user enters fewer than 127 instructions, the remaining memory locations are filled with the instruction (`ADI 000 000 000000`). This ensures that no changes are made to register values after the program ends.
- **Data Initialization:** The script prompts the user to input 16-bit data values, which are stored in the data memory starting from byte 257. This allows for testing programs that interact with data in memory.

The figure below shows an example of a program being written using this interface.

```
Enter the Assembly Code (type 'done' to finish):
LLI 000 00100001
DONE
['LLI', '000', '00100001']
Enter a 16-bit binary data value to input in the memory from byte 257: (type 'done' to finish):
1000101101001111
DONE
```

Figure 9.1: Snippet of a program being written onto the interface.

# Fibonacci Sequence Generator using the CPU

The goal here is to generate the sequence of Fibonacci numbers in one of the registers in the register file. In our case, the numbers are generated sequentially at Register 4. The following logic achieves it's implementation:

### Explanation

We want to first keep a counter for the number of times that we want the loop to execute. To do this, we load that value in register 1 (001). The variable for counter is stored in register 0 (000). In registers 2 (010) and 3 (011), we load the starting values of A = 1 and B = 1. We will need to come to this point of execution again and again, so we execute the JAL instruction and store the current program counter in register 6 (110).

Register 4 (100) holds the value of C = A + B. After that, we increment the counter at register 0 using ADI by 1. We also update the registers 2 and 3 as A = B + 0 and B = C + 0. After that, we check if the value of register 1 equals that of register 0. On doing that, we skip two instructions and exit the loop, otherwise we perform the JLR instruction and link back to the value at register 6. This ends the structure of the loop.

## Pseudo-code

The assembly line code that implements the above description is:

```
LLI 000 00000000
LLI 001 00010101
LLI 010 00000001
LLI 011 00000001
JAL 110 000000001
ADD 010 011 100
ADI 000 000 000001
ADI 011 010 000000
ADI 100 010 000000
BEQ 000 001 000010
JLR 111 110 000000
```

## Simulation

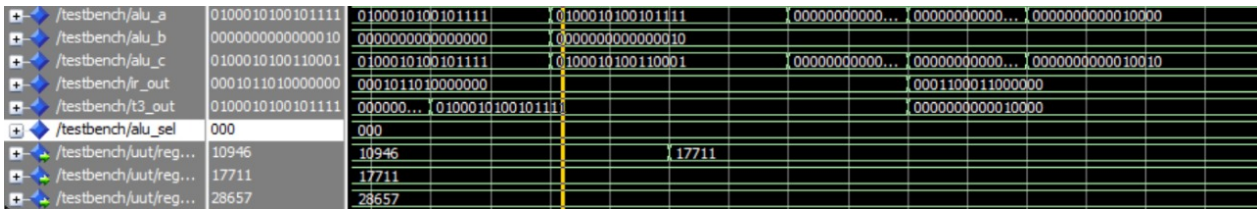



Figure 10.1: The values stored in the Register file.

Note that in the above image, 10946, 17711 and 28657 are the 21$^{st}$, 22$^{nd}$ and 23$^{rd}$ Fibonacci numbers respectively.