Lab 11. Simple CPU design

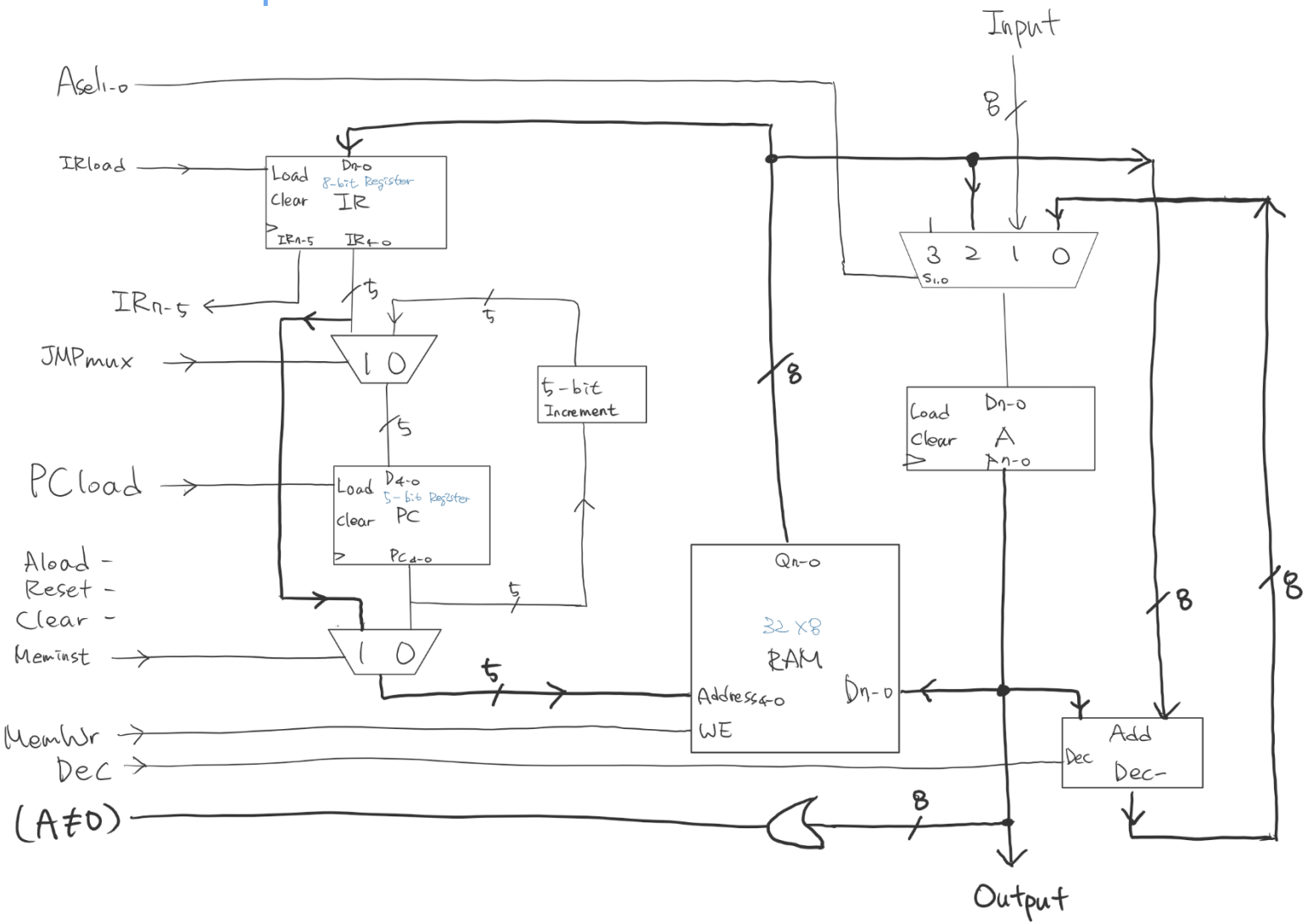
201810800 이혜인

- CPU Specifications
  - The CPU must be able to access 32 8-bit words of RAM.
  - The CPU must have an 8-bit Register (A)
  - The CPU may include additional internal components, such as an instruction register (IR) and program counter (PC), depending on each student's design.
  - The CPU must be capable of fetching, decoding and executing the instructions shown in the table. In this table, aaaaa specifies a 5-bit address.
  - The CPU must be able to successfully read instructions and data from memory.

Instruction	Opcode	Operation
LOAD	000aaaaa	A ← M[aaaaa]
STORE	001aaaaa	M[aaaaa] ← A
ADD	010aaaaa	A ← A+ M[aaaaa]
DEC	011aaaaa	A ← A - 1
IN	100xxxxx	A ← Input (external)
OUT	101xxxxx	Output (external) ← A
JMP	110aaaaa	PC ← aaaaa

- 앞 page에서 정의한 spec에 맞게 CPU를 설계하라. 가능한 가장 간단하게 설계하라.
  - Datapath를 그려라.
  - Control unit의 State diagram을 그려라.
  - Behavioral VHDL description으로 완성하라.
- 7개의 instruction을 모두 사용하는, 적당한 test용 알고리즘과 이에 해당하는 Assembly code, Binary(machine) code를 작성해서 simulation하고 검증하라.
- 보고서에 다음 사항을 포함하라.
  - Datapath 그림과 CU의 State diagram 그림
  - VHDL code
  - RTL view capture
  - 각자의 Algorithm
  - Assembly code
  - Binary code
  - Simulation capture
  - 분석 및 discussion

1. Datapath

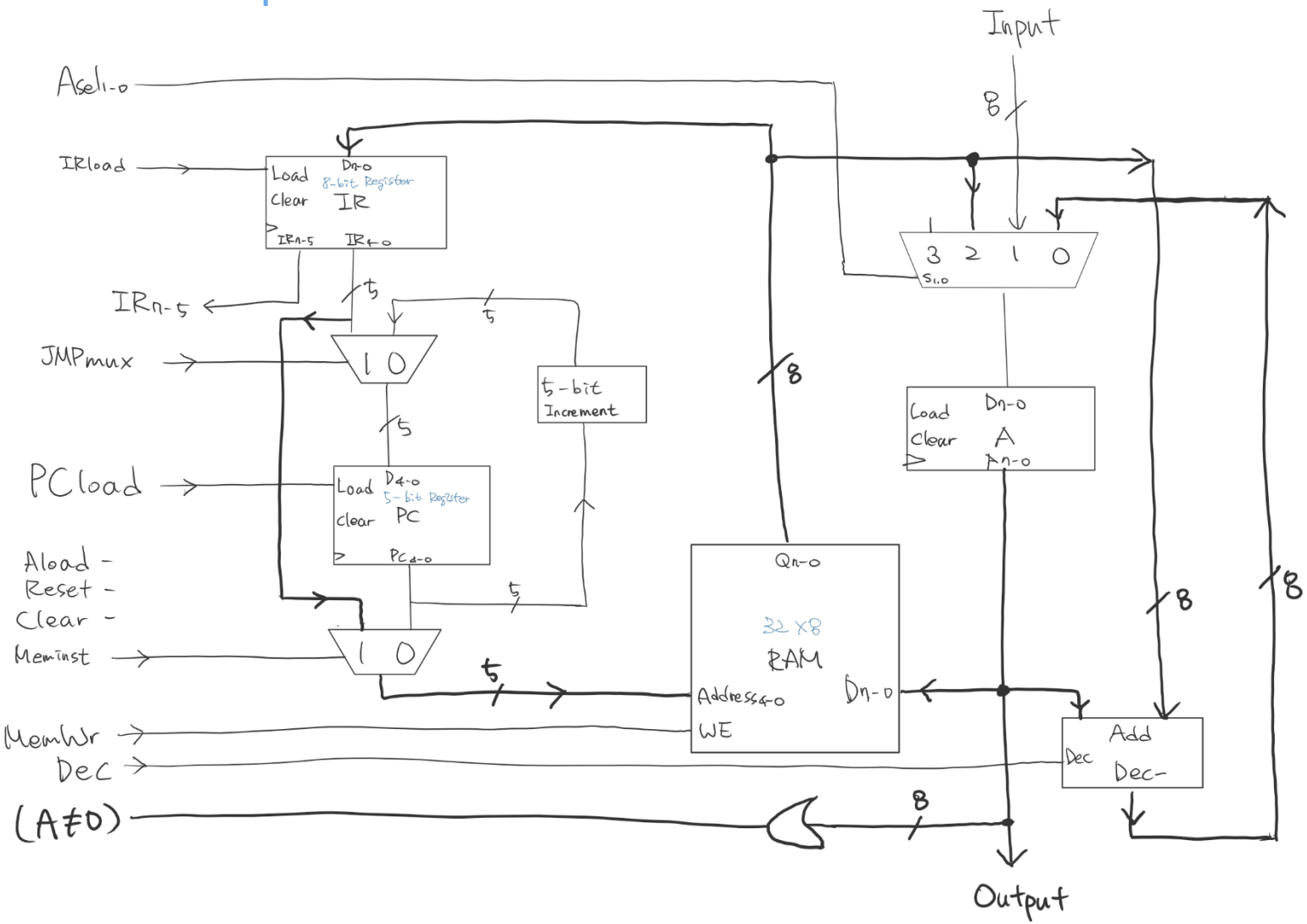


→ 해당 CPU의 Datapath는 다음과 같이 그릴 수 있다.

→ RAM에서 나가는 Q_{7-0}은 Memory에서 A로 Load하는 것이고, Meminst와 MemWr은 Store할 때 1이 된다.

→ 외부 Input과 Output이 존재하고, Input은 Memory에 있는 값이나 dec, add된 값이 들어올 수도 있으므로 Asel을 통해 정해준다.

1. Datapath

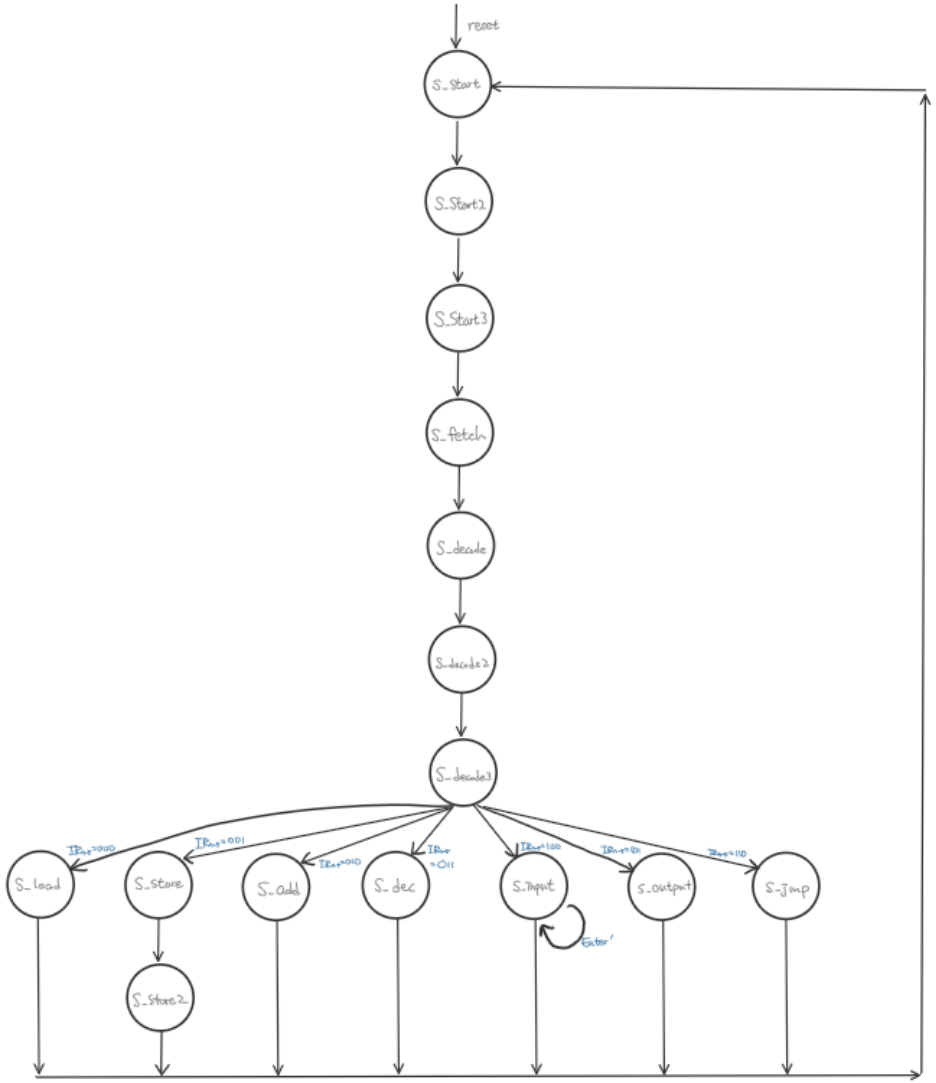


→ Dec가 1이면 Dec Instruction을 진행해주고, 0이면 ADD를 진행한다.

→ 만약에 A가 0이 아니면 해당 address로 Jump한다. 이는 A값을 받아온 후, JMPmux를 통해 JMP 여부를 결정해준다.

Lab 11. Simple CPU design

2. Control Unit의 State Diagram

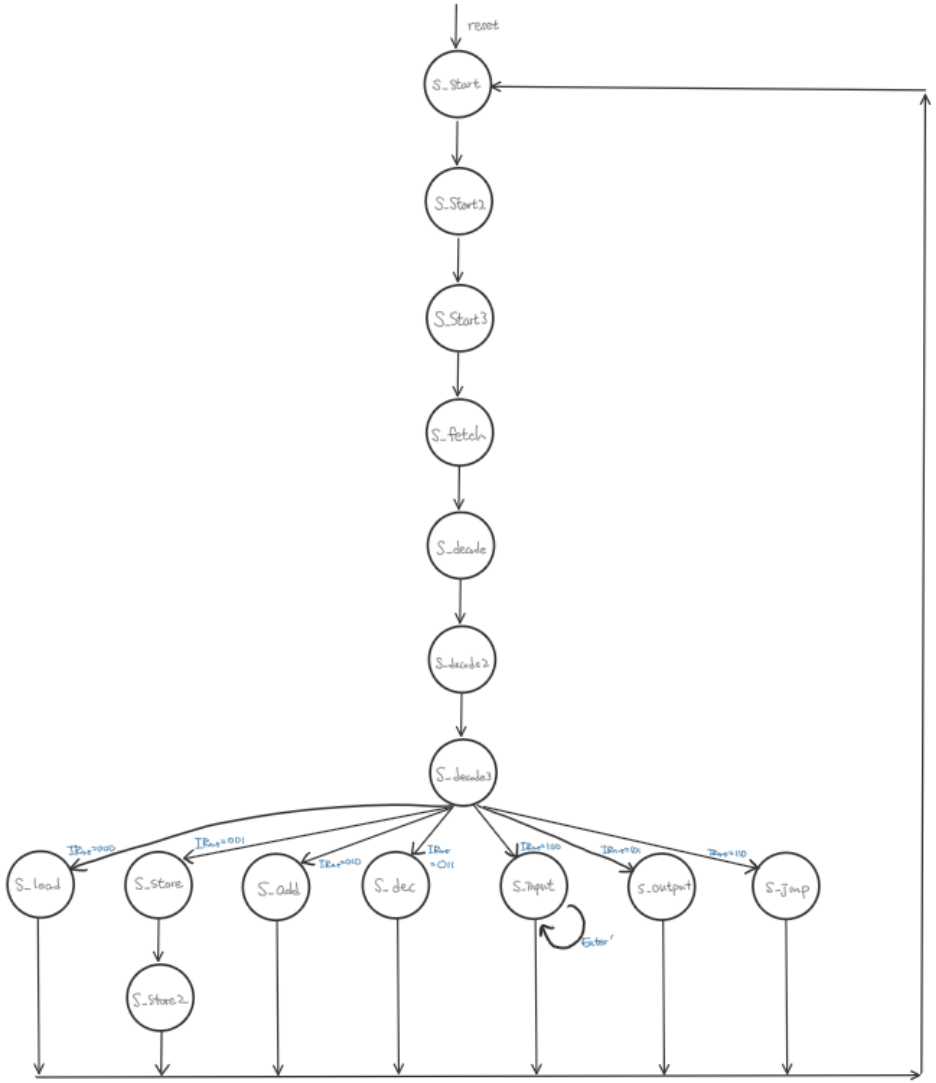


→ State Diagram은 다음과 같이 그릴 수 있다.

→ 먼저 state s_start에서 momory_address에 PC값을 넣고, state s_start2로 간다. 그리고 s_start2에서 s_start3으로 가고, s_Start3에서는 s_fetch로 간다.

→ 해당 과정이 필요한 이유는 충분한 clock이 있어야 모든 과정이 원활하게 일어날 수 있기 때문이다.

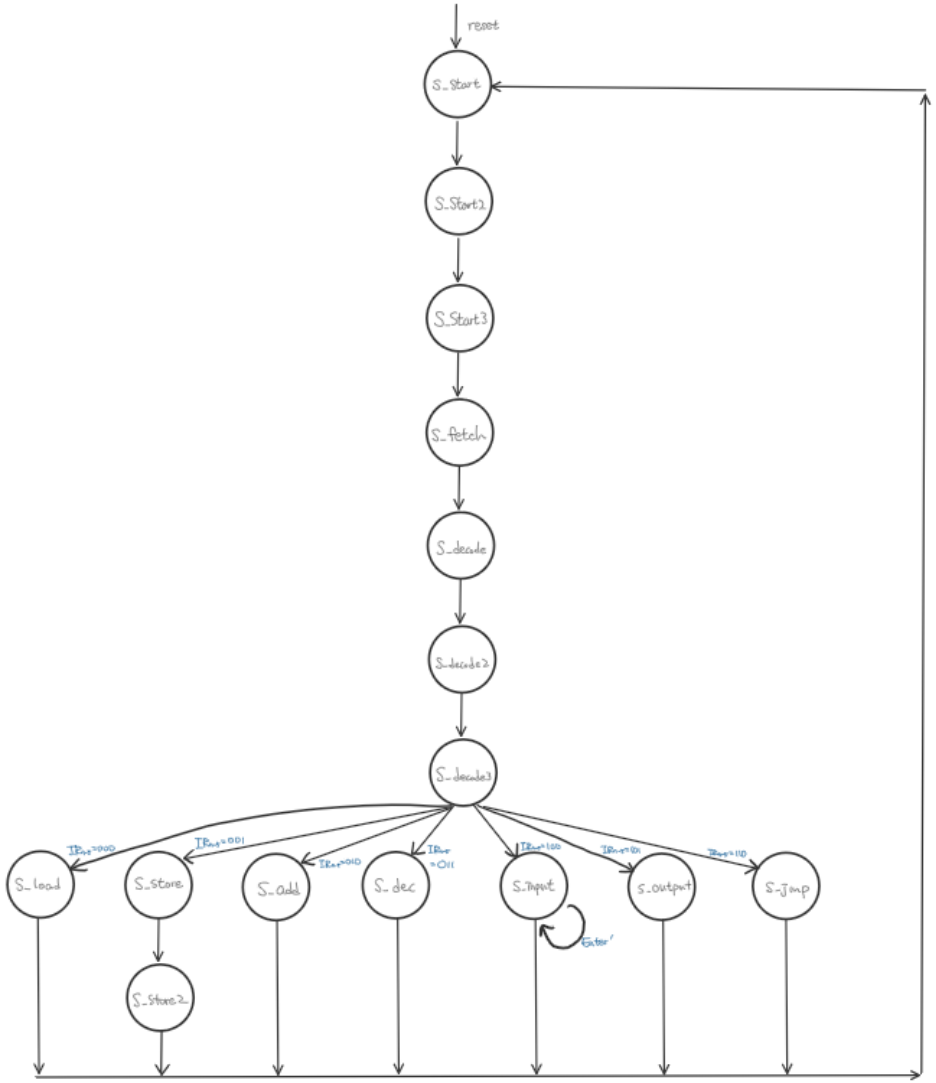
2. Control Unit의 State Diagram



→ state s_fetch에서 memory_data를 IR에 저장하고, PC 값을 하나 증가시킨다. 그리고 state s_decode로 넘어간다.

→ s_decode에서 memory_address에 IR_{4-0}을 저장하고 state는 s_decode2로 간다. s_decode2에서는 s_decode3으로 넘어간다.

2. Control Unit의 State Diagram



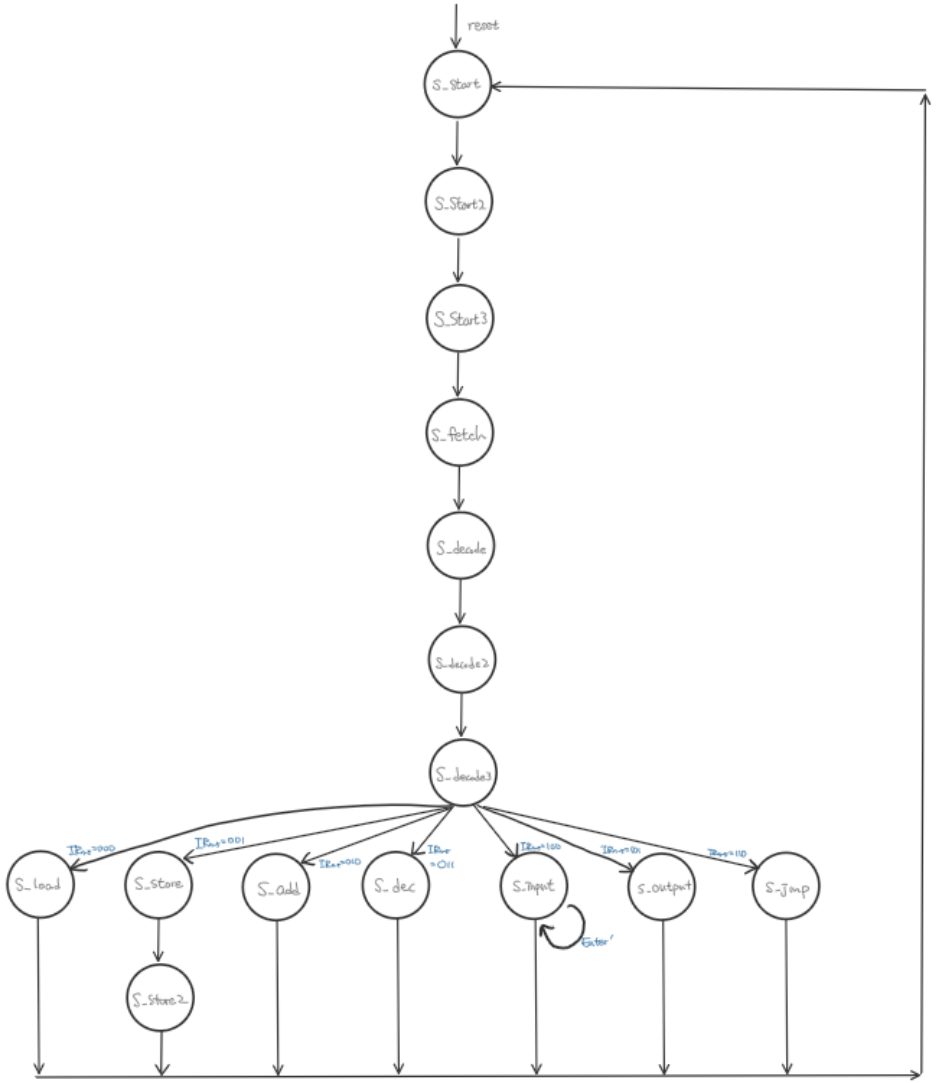
→ S_decode3에서는 각 IR7-5에 따라 다른 state로 간다.

→ "000"일 때는 s_load로 가서 A에 memory_data를 저장하고, s_start로 간다.

→ "001"일 때는 s_store로 가서 MemWr를 1로 해 저장이 되었음을 알리고, s_store2로 간다. S_store2에서는 MemWr을 다시 0으로 바꾸고 s_start로 간다.

→ "010"일 때는 s_add로 가서 A에 memory_data를 더한 다음 저장하고, s_start로 간다.

2. Control Unit의 State Diagram



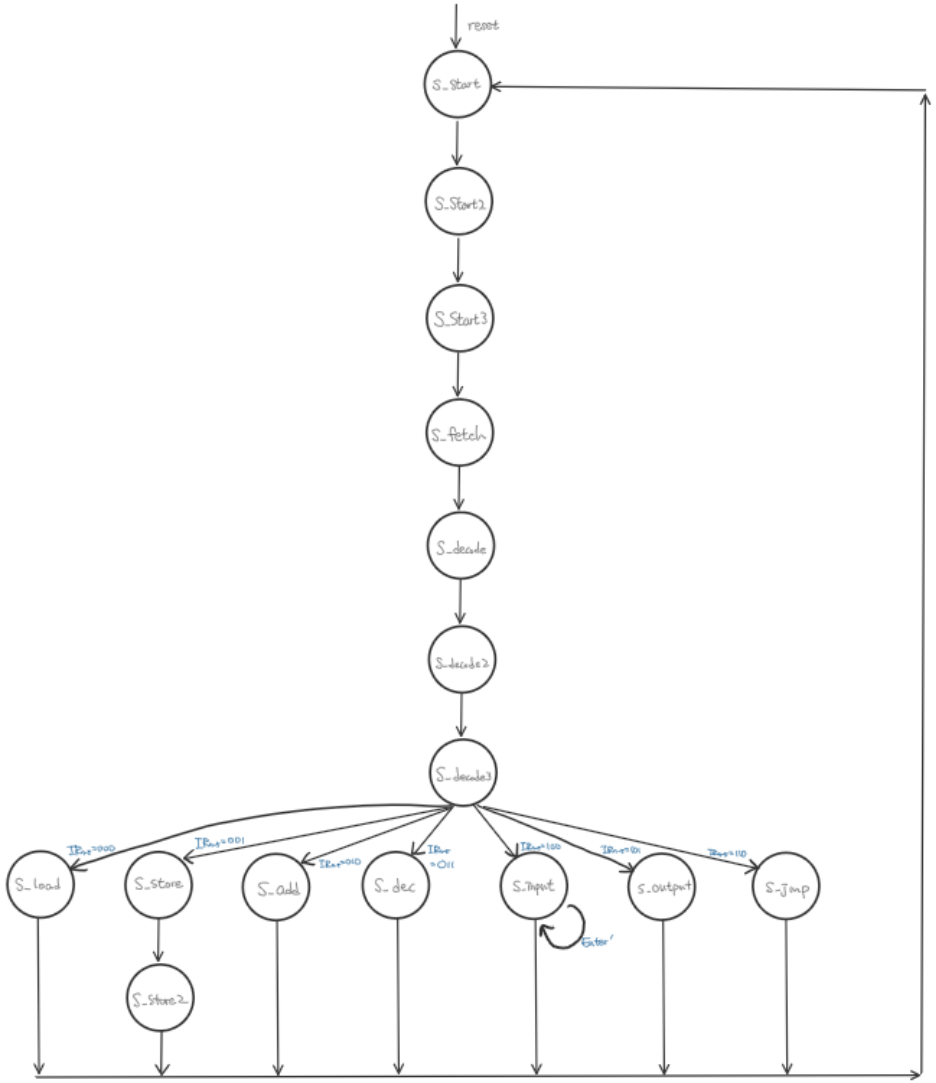
→ "011"일 때는 s_dec로 가서 A값을 1 감소시킨 다음 저장하고, s_start로 간다.

→ "100"일 때는 s_input로 가서 A에 input을 받고, Enter에 따라 Enter가 1이면 s_input에 있고, 0이면 s_start로 간다.

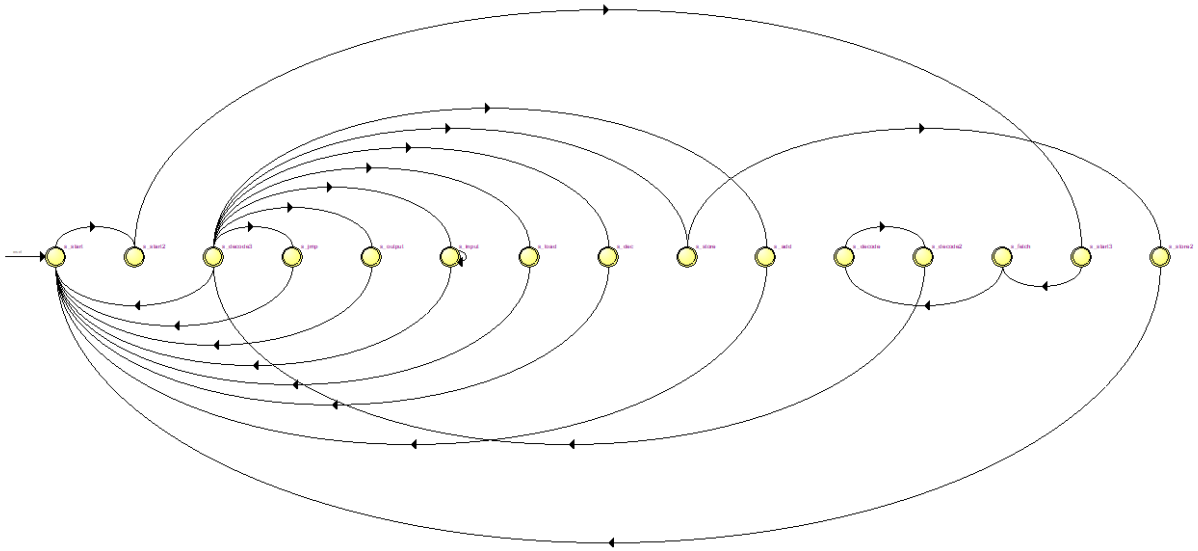
→ "101"일 때는 s_output로 가서 s_start로 간다.

→ "110"일 때는 s_jmp로 가서 A가 0이 아니면 PC에 IR4-0값을 넣고, s_start로 간다.

2. Control Unit의 State Diagram



→ 이는 아래 State Diagram과 동일함으로 올바르게 VHDL Code를 짠 것을 알 수 있다.



3. VHDL Code

```
1   LIBRARY IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3   USE IEEE.STD_LOGIC_ARITH.ALL;
4   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6   LIBRARY lpm;
7   USE lpm.lpm_components.ALL;
8
9   ENTITY lab_11 IS PORT (
10     clock, reset: IN STD_LOGIC;
11     enter: IN STD_LOGIC;
12     -- data input
13     input: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
14     -- data output
15     output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
16     -- control outputs;
17  END lab_11;
18
19  ARCHITECTURE SCD OF lab_11 IS
20     TYPE state_type IS (s_start,s_start2,s_start3,s_fetch,s_decode,s_decode2,s_decode3,s_load,s_store,s_store2,s_add,s_dec,s_input,s_output,s_jmp);
21     SIGNAL state: state_type;                -- states
22     SIGNAL IR: STD_LOGIC_VECTOR(7 DOWNTO 0);   -- Instruction register
23     SIGNAL PC: STD_LOGIC_VECTOR(4 DOWNTO 0);   -- Program counter
24     SIGNAL A:  STD_LOGIC_VECTOR(7 DOWNTO 0);   -- Accumulator
25     SIGNAL memory_address:  STD_LOGIC_VECTOR(4 DOWNTO 0);   -- memory address
26     SIGNAL memory_data:  STD_LOGIC_VECTOR(7 DOWNTO 0);   -- memory data input
27     SIGNAL MemWr: STD_LOGIC;
29  BEGIN
30     memory: lpm_ram_dq   -- 32 locations x 8 bits wide asynchronous memory
31        GENERIC MAP (
32           lpm_widthad   => 5,
33           lpm_outdata   => "REGISTERED",
34           lpm_indata    => "REGISTERED",
35           lpm_numwords => 32,
36           lpm_address_control => "REGISTERED",
37           lpm_file    => "program_lab11.mif", -- fill ram with content of file program.mif
38           lpm_width    => 8)
39        PORT MAP (
40           data          => A,
41           address       => memory_address,
42           inclock        => clock,
43           outclock     => clock,
44           we             => MemWr,
45           q              => memory_data);
```

```
47     PROCESS(clock,reset)
48     BEGIN
49        IF(reset = '1') THEN
50           PC <= "00000";
51           IR <= "00000000";
52           A  <= "00000000";
53           MemWr <= '0';
54           state <= s_start;
55        ELSIF(clock'EVENT AND clock = '1') THEN
56           CASE state IS
57           WHEN s_start => -- reset, start
58              memory_address <= PC;
59              state <= s_start2;
60           WHEN s_start2 =>
61              state <= s_start3;
62           WHEN s_start3 =>
63              state <= s_fetch;
64           WHEN s_fetch => -- fetch
65              IR <= memory_data;
66              PC <= PC + 1;
67              state <= s_decode;
68           WHEN s_decode => -- decode
69              memory_address <= IR(4 DOWNTO 0);   -- memory access using last 5 bits of IR
70              state <= s_decode2;
71           WHEN s_decode2 =>
72              state <= s_decode3;
73           WHEN s_decode3 =>
74              CASE IR(7 DOWNTO 5) IS
75                 WHEN "000" => state <= s_load;
76                 WHEN "001" => state <= s_store;
77                 WHEN "010" => state <= s_add;
78                 WHEN "011" => state <= s_dec;
79                 WHEN "100" => state <= s_input;
80                 WHEN "101" => state <= s_output;
81                 WHEN "110" => state <= s_jmp;
82                 WHEN OTHERS => state <= s_start;
83              END CASE;
84           WHEN s_load =>            -- load A from memory
85              A <= memory_data;
86              state <= s_start;
87           WHEN s_store =>            -- store A to memory
88              MemWr <= '1';
89              state <= s_store2;
90           WHEN s_store2 =>          -- need an extra state to de-assert MemWr before changing the memory address
91              MemWr <= '0';
92              state <= s_start;
93           WHEN s_add =>             -- add
94              A <= A + memory_data;
95              state <= s_start;
96           WHEN s_dec =>             -- subtract
97              A <= A - 1;
98              state <= s_start;
99           WHEN s_input =>
100             A <= input;
101             IF (Enter = '0') THEN   -- wait for Enter key
102                state <= s_input;
103             ELSE
104                state <= s_start;
105             END IF;
106          WHEN s_output =>
107             state <= s_start;
108          WHEN s_jmp =>
109             IF (A /= 0) THEN         -- jump if A is not 0
110                PC <= IR(4 DOWNTO 0);
111             END IF;
112             state <= s_start;
113          END CASE;
114       END IF;
115    END PROCESS;
116    output <= A;   -- send value of Accumulator to the output
117 END SCD;
```

→ 다음은 위의 Datapath와 Control Unit의 State Diagram을 반영하여 짠 VHDL Code이다. 해당 코드는 EC2와 유사하므로 달라진 부분 위주로 설명할 것이다.

3. VHDL Code

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  USE IEEE.STD_LOGIC_ARITH.ALL;
4  USE IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  LIBRARY lpm;
7  USE lpm.lpm_components.ALL;
8
9  ENTITY lab_11 IS PORT (
10    clock, reset: IN STD_LOGIC;
11    enter: IN STD_LOGIC;
12    -- data input
13    input: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
14    -- data output
15    output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
16    -- control outputs;
17 END lab_11;
18
19 ARCHITECTURE SCD OF lab_11 IS
20    TYPE state_type IS (s_start,s_start2,s_start3,s_fetch,s_decode,s_decode2,s_decode3,s_load,s_store,s_store2,s_add,s_dec,s_input,s_output,s_jmp);
21    SIGNAL state: state_type;              -- states
22    SIGNAL IR: STD_LOGIC_VECTOR(7 DOWNTO 0);   -- Instruction register
23    SIGNAL PC: STD_LOGIC_VECTOR(4 DOWNTO 0);   -- Program counter
24    SIGNAL A:  STD_LOGIC_VECTOR(7 DOWNTO 0);   -- Accumulator
25    SIGNAL memory_address:  STD_LOGIC_VECTOR(4 DOWNTO 0);   -- memory address
26    SIGNAL memory_data:  STD_LOGIC_VECTOR(7 DOWNTO 0);   -- memory data input
27    SIGNAL MemWr: STD_LOGIC;
```

→ Input으로는 Clock, reset, enter, 그리고 input을 가지고 있다. 이때 input은 연산 할 값을 입력 받고, enter는 외부 signal이 있는 경우, 즉 외부에서 input이 들어오는 경우 이를 받고, 1이 되면 다음 State로 넘어가는 역할을 한다.

3. VHDL Code

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  USE IEEE.STD_LOGIC_ARITH.ALL;
4  USE IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  LIBRARY lpm;
7  USE lpm.lpm_components.ALL;
8
9  ENTITY lab_11 IS PORT (
10    clock, reset: IN STD_LOGIC;
11    enter: IN STD_LOGIC;
12    -- data input
13    input: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
14    -- data output
15    output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
16    -- control outputs;
17 END lab_11;
18
19 ARCHITECTURE SCD OF lab_11 IS
20    TYPE state_type IS (s_start,s_start2,s_start3,s_fetch,s_decode,s_decode2,s_decode3,s_load,s_store,s_store2,s_add,s_dec,s_input,s_output,s_jmp);
21    SIGNAL state: state_type;              -- states
22    SIGNAL IR: STD_LOGIC_VECTOR(7 DOWNTO 0);   -- Instruction register
23    SIGNAL PC: STD_LOGIC_VECTOR(4 DOWNTO 0);   -- Program counter
24    SIGNAL A:  STD_LOGIC_VECTOR(7 DOWNTO 0);   -- Accumulator
25    SIGNAL memory_address:  STD_LOGIC_VECTOR(4 DOWNTO 0);   -- memory address
26    SIGNAL memory_data:  STD_LOGIC_VECTOR(7 DOWNTO 0);   -- memory data input
27    SIGNAL MemWr: STD_LOGIC;
```

→ Output으로는 output을 가지고 있다. 이때 output은 연산 한 결과이다. 기존 EC2와는 다르게 halt를 output으로 가지지 않는다.

→ 내부 Signal로는 State와, IR, PC, A, memory_address, memory_data, MemWr가 있다.

3. VHDL Code

```
29  BEGIN
30      memory: lpm_ram_dq   -- 32 locations x 8 bits wide asynchronous memory
31          GENERIC MAP (
32              lpm_widthad   => 5,
33              lpm_outdata   => "REGISTERED",
34              lpm_indata    => "REGISTERED",
35              lpm_numwords => 32,
36              lpm_address_control => "REGISTERED",
37              lpm_file    => "program_labl1.mif", -- fill ram with content of file program.mif
38              lpm_width    => 8)
39          PORT MAP (
40              data          => A,
41              address       => memory_address,
42              inclock        => clock,
43              outclock     => clock,
44              we             => MemWr,
45              q              => memory_data);
```

→ Memory RAM은 32X8로 8bit word가 32개 있다.

→ 각각의 signal끼리 올바르게 연결해주었다.

3. VHDL Code

```
47  PROCESS(clock,reset)
48  BEGIN
49     IF(reset = '1') THEN
50        PC <= "00000";
51        IR <= "00000000";
52        A  <= "00000000";
53        MemWr <= '0';
54        state <= s_start;
55     ELSIF(clock'EVENT AND clock = '1') THEN
56        CASE state IS
57        WHEN s_start => -- reset, start
58           memory_address <= PC;
59           state <= s_start2;
60        WHEN s_start2 =>
61           state <= s_start3;
62        WHEN s_start3 =>
63           state <= s_fetch;
64        WHEN s_fetch => -- fetch
65           IR <= memory_data;
66           PC <= PC + 1;
67           state <= s_decode;
68        WHEN s_decode => -- decode
69           memory_address <= IR(4 DOWNTO 0);    -- memory access using last 5 bits of IR
70           state <= s_decode2;
71        WHEN s_decode2 =>
72           state <= s_decode3;
73        WHEN s_decode3 =>
74           CASE IR(7 DOWNTO 5) IS
75              WHEN "000" => state <= s_load;
76              WHEN "001" => state <= s_store;
77              WHEN "010" => state <= s_add;
78              WHEN "011" => state <= s_dec;
79              WHEN "100" => state <= s_input;
80              WHEN "101" => state <= s_output;
81              WHEN "110" => state <= s_jmp;
82              WHEN OTHERS => state <= s_start;
83           END CASE;
```

→ 만약 reset이 1이면 PC, IR, A, MemWr을 각각 0으로 초기화해주고, state는 초기 상태인 s_start로 간다.

→ Clock의 rising edge에서 다음과 같이 state가 이동된다. 이는 앞에 있는 State Diagram과 동일하다.

3. VHDL Code

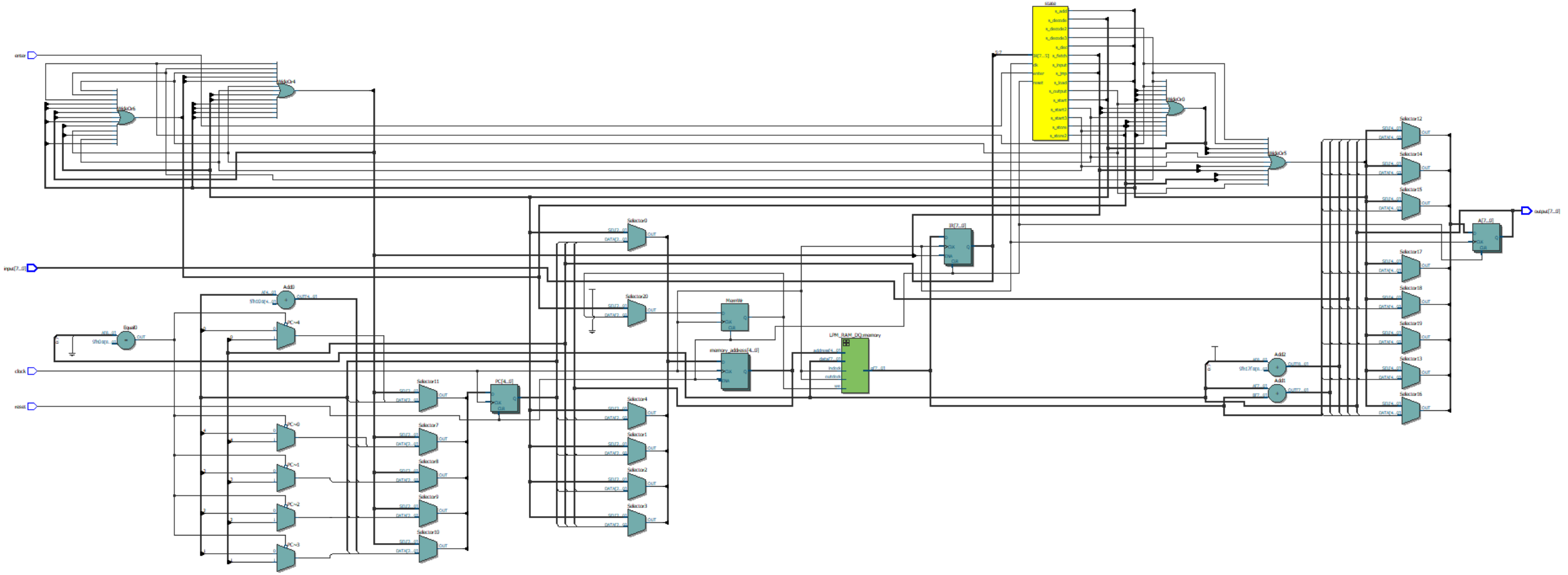
```
84          WHEN s_load =>            -- load A from memory
85             A <= memory_data;
86             state <= s_start;
87          WHEN s_store =>           -- store A to memory
88             MemWr <= '1';
89             state <= s_store2;
90          WHEN s_store2 =>          -- need an extra state to de-assert MemWr before changing the memory address
91             MemWr <= '0';
92             state <= s_start;
93          WHEN s_add =>             -- add
94             A <= A + memory_data;
95             state <= s_start;
96          WHEN s_dec =>             -- subtract
97             A <= A - 1;
98             state <= s_start;
99          WHEN s_input =>
100            A <= input;
101            IF (Enter = '0') THEN   -- wait for Enter key
102               state <= s_input;
103            ELSE
104               state <= s_start;
105            END IF;
106         WHEN s_output =>
107            state <= s_start;
108         WHEN s_jmp =>
109            IF (A /= 0) THEN        -- jump if A is not 0
110               PC <= IR(4 DOWNTO 0);
111            END IF;
112            state <= s_start;
113         END CASE;
114      END IF;
115   END PROCESS;
116   output <= A;   -- send value of Accumulator to the output
117 END SCD;
```

→ 해당 과정 역시 앞에 있는 State Diagram과 동일하다.

→ 최종적으로 Output에 A를 넣어 출력해주면 해당 Process가 마무리 된다.

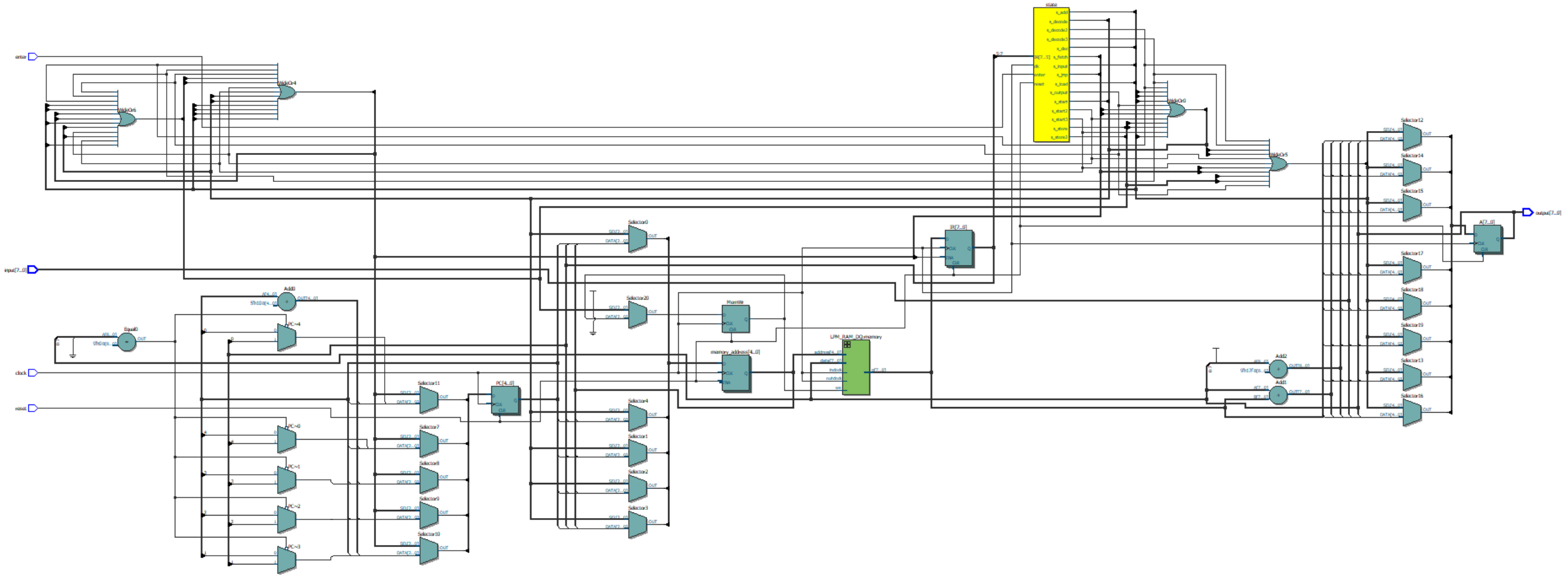
→ 해당 Code는 EC2와 다르게 jmp부분이 하나라는 점이 다르고, 또한, output으로 halt를 갖지 않는다는 점이 다르다.

4. RTL Viewer



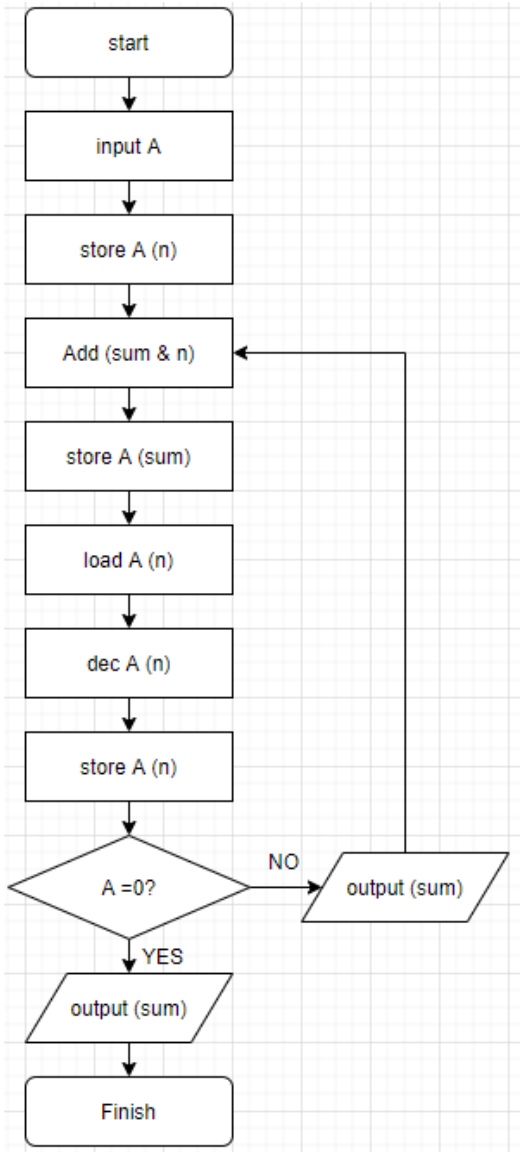
→ 다음 Code를 RTL Viewer로 나타내면 다음과 같다.

4. RTL Viewer



→ Input으로는 Enter, clock, reset, input이 존재하고, Output으로는 output이 존재한다.

5. Algorithm



→ Test용 알고리즘은 다음과 같다.

→ 먼저 input을 받고, 이를 address n에 저장한다.

→ 그리고 address n에 있는 값과 sum에 있는 값을 더해 해당 값을 sum에 저장한다.

→ 다시 address n에 있는 A를 가져와 dec해서 다시 address n에 저장한다.

→ 만약 A값이 0이면 sum에 있는 값을 출력하고 종료한다.

→ A값이 1이면 sum에 있는 값을 출력한 후, Add부터 다시 진행한다.

6. Assembly code & Binary Code

```
-- Content of the RAM memory in the file PROGRAM.MIF

DEPTH = 32;     -- Depth of memory: 5-bit address
WIDTH = 8;      -- Width of memory: 8-bit data

ADDRESS_RADIX = BIN; -- All values in binary (HEX, DEC, OCT, BIN)
DATA_RADIX = BIN;

-- Opcodes for the EC-2
--    000 = load A,aaaaa
--    001 = store A,aaaaa
--    010 = add A,aaaaa
--    011 = dec A,aaaaa
--    100 = in A
--    101 = out A
--    110 = jmp aaaaa

-- Specify the memory content.
-- Format of each memory location is
--     address : data
```

```
CONTENT
   BEGIN
   [00000..11111] :  00000000;   -- Initialize locations range 00-FF to 0000
-------------------------------------------------------------
-- Sum
-- Program to sum N downto 1
   00000 :  10000000;   -- input A
   00001 :  00111111;   -- store A,n

   00010 :  00011111;   -- loop: load A,n -- n + sum
   00011 :  01011110;   -- add A,sum
   00100 :  00111110;   -- store A,sum

   00101 :  00011111;   -- load A,n       -- decrement A
   00110 :  01111111;   -- dec A,n
   00111 :  00111111;   -- store A,n

   01000 :  11000010;   -- jmp
   01001 :  00011110;   -- out: load A,sum

   11110 :  00000000;   -- sum
   11111 :  00000000;   -- n
END;
```

→ 다음은 Assembly Code와 Binary Code이다.

→ Memory는 32 * 8 bit이고, Address Radix는 Binary이고, Data Radix 역시 Binary이다.

→ 각각 $IR_{7\text{-}5}$의 값에 따른 state를 알 수 있다.

6. Assembly code & Binary Code

```
-- Content of the RAM memory in the file PROGRAM.MIF

DEPTH = 32;     -- Depth of memory: 5-bit address
WIDTH = 8;      -- Width of memory: 8-bit data

ADDRESS_RADIX = BIN; -- All values in binary (HEX, DEC, OCT, BIN)
DATA_RADIX = BIN;

-- Opcodes for the EC-2
--     000 = load A,aaaaa
--     001 = store A,aaaaa
--     010 = add A,aaaaa
--     011 = dec A,aaaaa
--     100 = in A
--     101 = out A
--     110 = jmp aaaaa

-- Specify the memory content.
-- Format of each memory location is
--      address : data
```

```
CONTENT
   BEGIN
   [00000..11111] :  00000000;   -- Initialize locations range 00-FF to 0000
-------------------------------------------------------------
-- Sum
-- Program to sum N downto 1
   00000 :  10000000;   -- input A
   00001 :  00111111;   -- store A,n

   00010 :  00011111;   -- loop: load A,n -- n + sum
   00011 :  01011110;   -- add A,sum
   00100 :  00111110;   -- store A,sum

   00101 :  00011111;   -- load A,n       -- decrement A
   00110 :  01111111;   -- dec A,n
   00111 :  00111111;   -- store A,n

   01000 :  11000010;   -- jmp
   01001 :  00011110;   -- out: load A,sum

   11110 :  00000000;   -- sum
   11111 :  00000000;   -- n
END;
```

→ Binary Code에서 Sum을 하는데, 이는 앞에 Algorithm을 기반으로 동작한다.

→ 먼저 memory는 sum, n 2개를 가진다. 외부 input 1개를 받아서 이를 n에 저장한다.

→ 다음으로 sum에 있는 값과 n에 있는 값을 더해서 이를 sum에 저장한다.

6. Assembly code & Binary Code

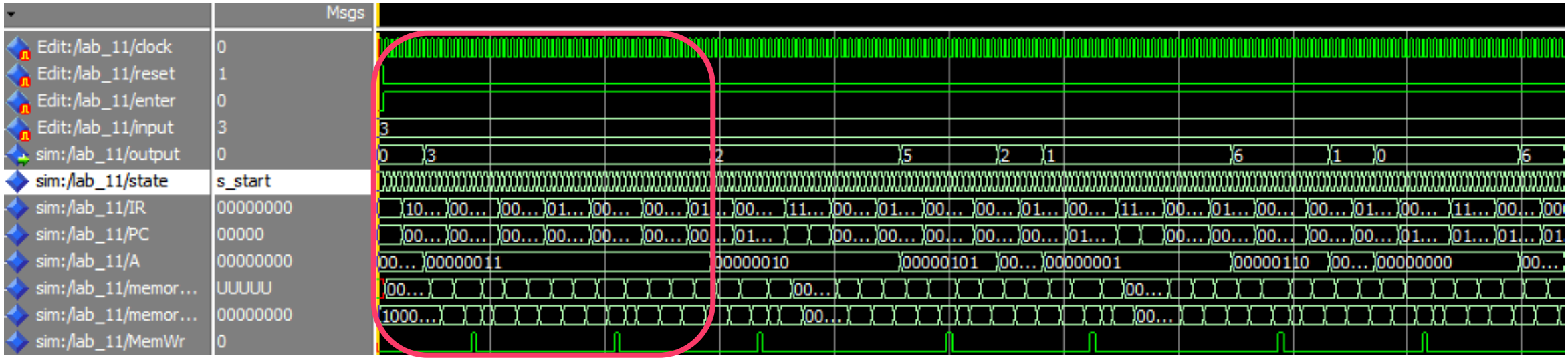
```
-- Content of the RAM memory in the file PROGRAM.MIF

DEPTH = 32;     -- Depth of memory: 5-bit address
WIDTH = 8;      -- Width of memory: 8-bit data

ADDRESS_RADIX = BIN; -- All values in binary (HEX, DEC, OCT, BIN)
DATA_RADIX = BIN;

-- Opcodes for the EC-2
--    000 = load A,aaaaa
--    001 = store A,aaaaa
--    010 = add A,aaaaa
--    011 = dec A,aaaaa
--    100 = in A
--    101 = out A
--    110 = jmp aaaaa

-- Specify the memory content.
-- Format of each memory location is
--     address : data
```

```
CONTENT
   BEGIN
   [00000..11111] :  00000000;   -- Initialize locations range 00-FF to 0000
-------------------------------------------------------------
-- Sum
-- Program to sum N downto 1
   00000 :  10000000;   -- input A
   00001 :  00111111;   -- store A,n

   00010 :  00011111;   -- loop: load A,n -- n + sum
   00011 :  01011110;   -- add A,sum
   00100 :  00111110;   -- store A,sum

   00101 :  00011111;   -- load A,n       -- decrement A
   00110 :  01111111;   -- dec A,n
   00111 :  00111111;   -- store A,n

   01000 :  11000010;   -- jmp
   01001 :  00011110;   -- out: load A,sum

   11110 :  00000000;   -- sum
   11111 :  00000000;   -- n
END;
```

→ n에 있는 값을 load해서 dec시키고, 다시 n에 저장한다.

→ 만약 A가 0이 아니면 '00010' address로 이동하여 해당 과정을 반복한다. 그리고 sum에 있는 값을 출력해준다.

7. Simulation

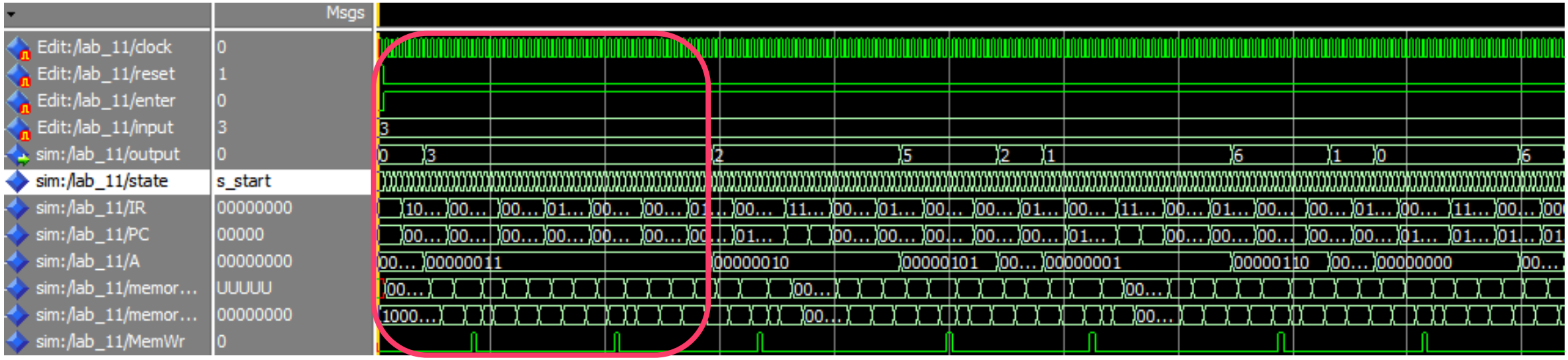


→ 다음은 Simulation을 진행한 결과이다.

→ 해당 Simulation은 load, store, add, dec, in, out, jmp와 같이 모든 Instruction이 포함된 것을 확인할 수 있다.

→ 먼저 n에 input받은 값을 store한다. 이때, 저장을 했다는 의미에서 MemWr이 1이 되었다.

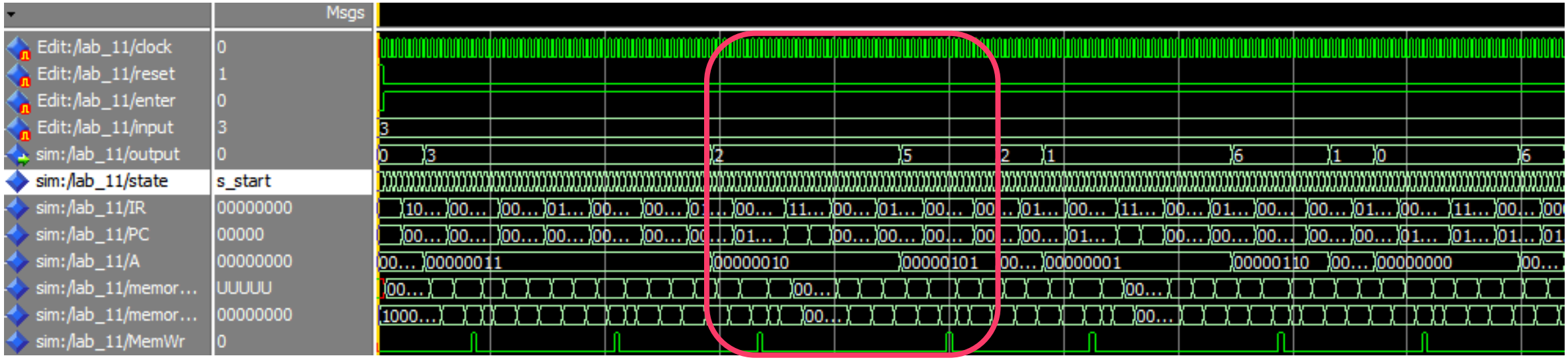
7. Simulation



→ 또한, 해당 과정에서 output에 3이 나타나는 부분의 state는 s_input이고, 이때 Enter가 1이므로 다음 State로 진행될 수 있다. 그리고 s_input 이전까지는 s_start ~ s_decode3까지 순차적으로 진행되었다.

→ 그리고 n에 있는 값과 sum에 있는 값을 더해서 이를 sum에 저장한다. 이때 역시 저장을 했다는 의미에서 MemWr이 1이 되었다.

7. Simulation

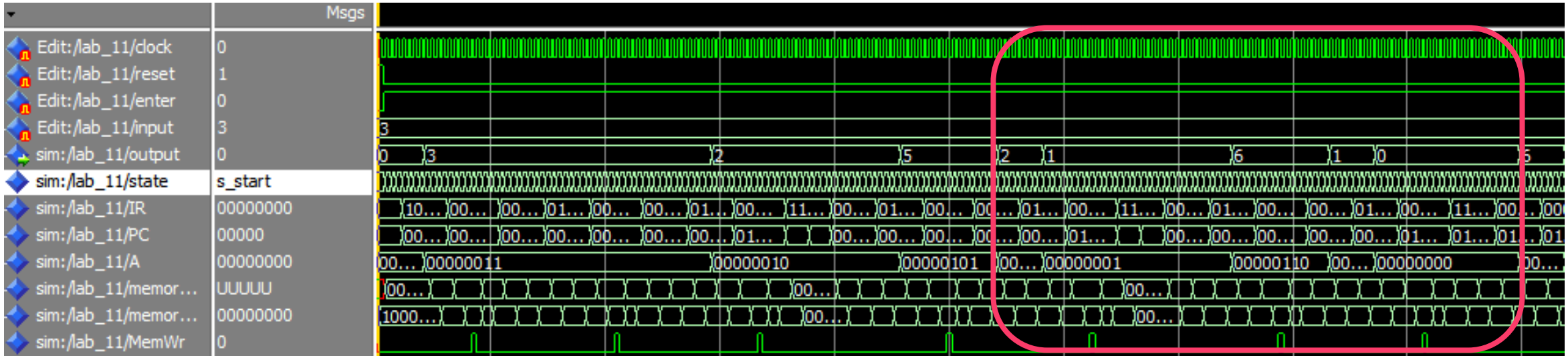


→ 다음으로 n에 있는 A를 load해서 dec를 진행시킨다. 그럼 2가 n에 저장이 된다.

→ 그리고 A가 0이 아니므로 다시 Add를 진행하러 가고, 이때 sum에 있는 값인 5가 출력되어 저장되었다.

→ 이때, 저장을 했다는 의미에서 각각 MemWr이 1이 되었다.

7. Simulation



→ 다음으로 반복해서 n에 있는 A를 load해서 dec를 진행시킨다. 그럼 2이 n에 저장이 된다. 이 때, A가 0이 아니므로 다시 Add를 진행하러 가고, 이때 sum에 있는 값인 6이 출력되어 저장되었다.

→ 그리고 A가 1 다음엔 dec 되어 0이 되므로 해당 과정이 종료된다.

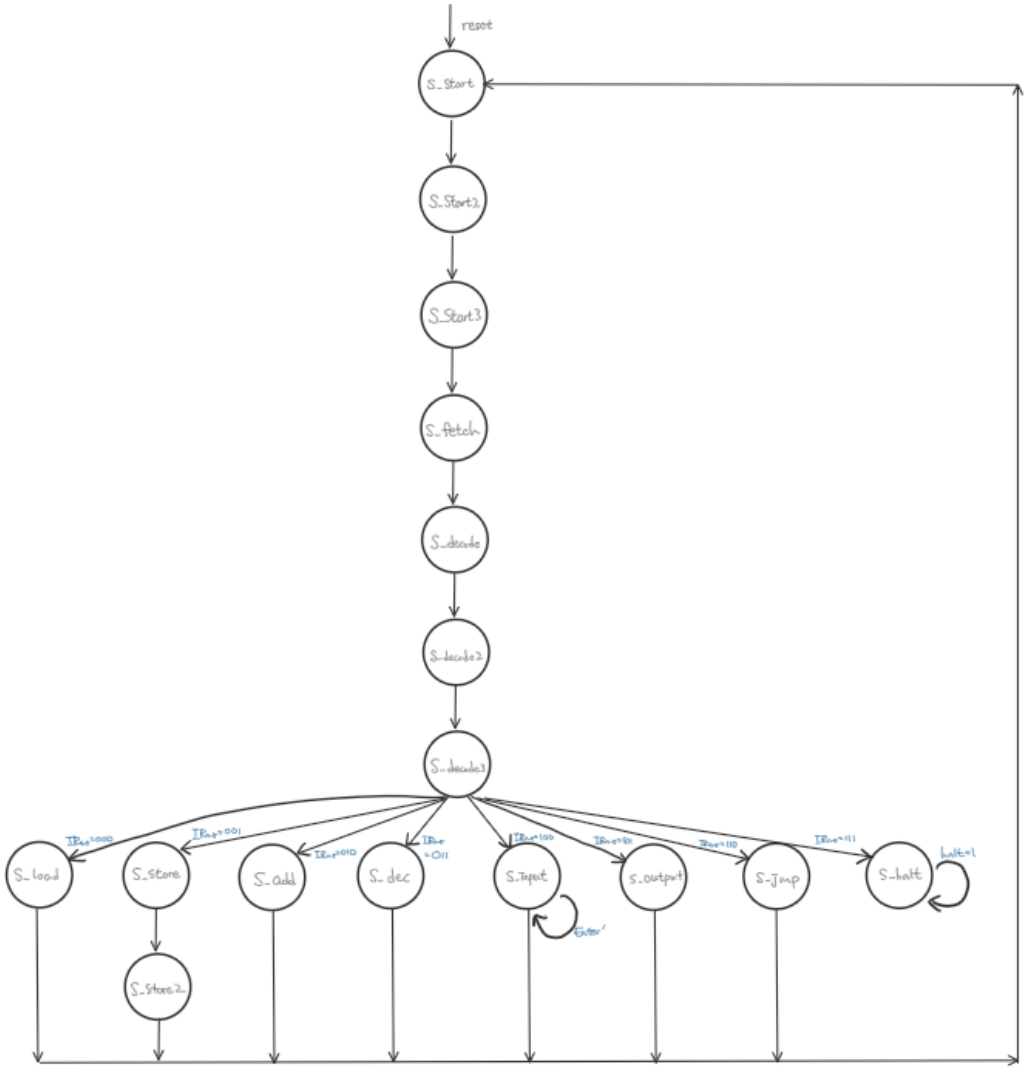
Lab 11. Simple CPU design

Halt 추가

→ 종료되는 시점을 명확히 알기 위해

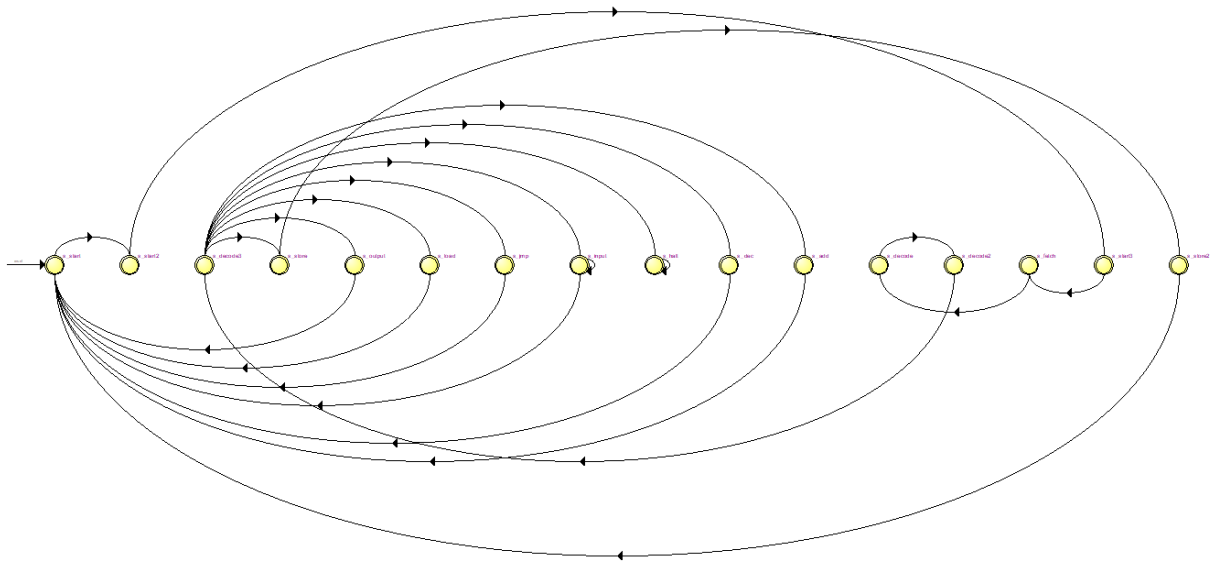
Halt를 추가해서 진행하였다.

2. Control Unit의 State Diagram



→ State Diagram은 앞에 있는 State Diagram에 s_halt 만 추가하였다.

→ 이는 아래 State Diagram과 동일함으로 올바르게 VHDL Code를 수정한 것을 알 수 있다.



3. VHDL Code

```
1   LIBRARY IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3   USE IEEE.STD_LOGIC_ARITH.ALL;
4   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6   LIBRARY lpm;
7   USE lpm.lpm_components.ALL;
8
9   ENTITY lab_11 IS PORT (
10     clock, reset: IN STD_LOGIC;
11     enter: IN STD_LOGIC;
12     -- data input
13     input: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
14     -- data output
15     output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
16     -- control outputs:
17     halt: OUT STD_LOGIC);
18  END lab_11;
19
20  ARCHITECTURE SCD OF lab_11 IS
21     TYPE state_type IS (s_start,s_start2,s_start3,s_fetch,s_decode,s_decode2,s_decode3,s_load,s_store,s_store2,s_add,s_dec,s_input,s_output,s_jmp,s_halt);
22     SIGNAL state: state_type;              -- states
23     SIGNAL IR: STD_LOGIC_VECTOR(7 DOWNTO 0);   -- Instruction register
24     SIGNAL PC: STD_LOGIC_VECTOR(4 DOWNTO 0);   -- Program counter
25     SIGNAL A:  STD_LOGIC_VECTOR(7 DOWNTO 0);   -- Accumulator
26     SIGNAL memory_address:  STD_LOGIC_VECTOR(4 DOWNTO 0);   -- memory address
27     SIGNAL memory_data:  STD_LOGIC_VECTOR(7 DOWNTO 0);   -- memory data input
28     SIGNAL MemWr: STD_LOGIC;
30  BEGIN
31     memory: lpm_ram_dq   -- 32 locations x 8 bits wide asynchronous memory
32        GENERIC MAP (
33           lpm_widthad   => 5,
34           lpm_outdata   => "REGISTERED",
35           lpm_indata    => "REGISTERED",
36           lpm_numwords => 32,
37           lpm_address_control => "REGISTERED",
38           lpm_file    => "program_lab11.mif", -- fill ram with content of file program.mif
39           lpm_width   => 8)
40        PORT MAP (
41           data         => A,
42           address      => memory_address,
43           inclock      => clock,
44           outclock     => clock,
45           we           => MemWr,
46           q            => memory_data);
```

```
48     PROCESS(clock,reset)
49     BEGIN
50        IF(reset = '1') THEN
51           PC <= "00000";
52           IR <= "00000000";
53           A  <= "00000000";
54           MemWr <= '0';
55           halt <= '0';
56           state <= s_start;
57        ELSIF(clock'EVENT AND clock = '1') THEN
58           CASE state IS
59           WHEN s_start => -- reset, start
60              memory_address <= PC;
61              state <= s_start2;
62           WHEN s_start2 =>
63              state <= s_start3;
64           WHEN s_start3 =>
65              state <= s_fetch;
66           WHEN s_fetch => -- fetch
67              IR <= memory_data;
68              PC <= PC + 1;
69              state <= s_decode;
70           WHEN s_decode => -- decode
71              memory_address <= IR(4 DOWNTO 0);   -- memory access using last 5 bits of IR
72              state <= s_decode2;
73           WHEN s_decode2 =>
74              state <= s_decode3;
75           WHEN s_decode3 =>
76              CASE IR(7 DOWNTO 5) IS
77                 WHEN "000" => state <= s_load;
78                 WHEN "001" => state <= s_store;
79                 WHEN "010" => state <= s_add;
80                 WHEN "011" => state <= s_dec;
81                 WHEN "100" => state <= s_input;
82                 WHEN "101" => state <= s_output;
83                 WHEN "110" => state <= s_jmp;
84                 WHEN "111" => state <= s_halt;
85                 WHEN OTHERS => state <= s_start;
86              END CASE;
87           WHEN s_load =>            -- load A from memory
88              A <= memory_data;
89              state <= s_start;
90           WHEN s_store =>           -- store A to memory
91              MemWr <= '1';
92              state <= s_store2;
93           WHEN s_store2 =>          -- need an extra state to de-assert MemWr before changing the memory address
94              MemWr <= '0';
95              state <= s_start;
96           WHEN s_add =>             -- add
97              A <= A + memory_data;
98              state <= s_start;
99           WHEN s_dec =>             -- subtract
100             A <= A - 1;
101             state <= s_start;
102          WHEN s_input =>
103             A <= input;
104             IF (Enter = '0') THEN   -- wait for Enter key
105                state <= s_input;
106             ELSE
107                state <= s_start;
108             END IF;
109          WHEN s_output =>
110             state <= s_start;
111          WHEN s_jmp =>
112             IF (A /= 0) THEN          -- jump if A is 0
113                PC <= IR(4 DOWNTO 0);
114             END IF;
115             state <= s_start;
116          WHEN s_halt =>
117             halt <= '1';
118             state <= s_halt;
119          WHEN OTHERS =>
120             state <= s_halt;
121          END CASE;
122       END IF;
123    END PROCESS;
124    output <= A;   -- send value of Accumulator to the output
125 END SCD;
```

→ 다음은 위의 Datapath와 Control Unit의 State Diagram을 반영하여 짠 VHDL Code이다. 위의 코드에서 halt부분만 추가되었다.

3. VHDL Code

```
87          WHEN s_load =>            -- load A from memory
88             A <= memory_data;
89             state <= s_start;
90          WHEN s_store =>            -- store A to memory
91             MemWr <= '1';
92             state <= s_store2;
93          WHEN s_store2 =>          -- need an extra state to de-assert MemWr before changing the memory address
94             MemWr <= '0';
95             state <= s_start;
96          WHEN s_add =>             -- add
97             A <= A + memory_data;
98             state <= s_start;
99          WHEN s_dec =>             -- subtract
100            A <= A - 1;
101            state <= s_start;
102         WHEN s_input =>
103            A <= input;
104            IF (Enter = '0') THEN    -- wait for Enter key
105               state <= s_input;
106            ELSE
107               state <= s_start;
108            END IF;
109         WHEN s_output =>
110            state <= s_start;
111         WHEN s_jmp =>
112            IF (A /= 0) THEN          -- jump if A is 0
113               PC <= IR(4 DOWNTO 0);
114            END IF;
115            state <= s_start;
116         WHEN s_halt =>
117            halt <= '1';
118            state <= s_halt;
119         WHEN OTHERS =>
120            state <= s_halt;
121      END CASE;
122   END IF;
123 END PROCESS;
124 output <= A;    -- send value of Accumulator to the output
125 END SCD;
```

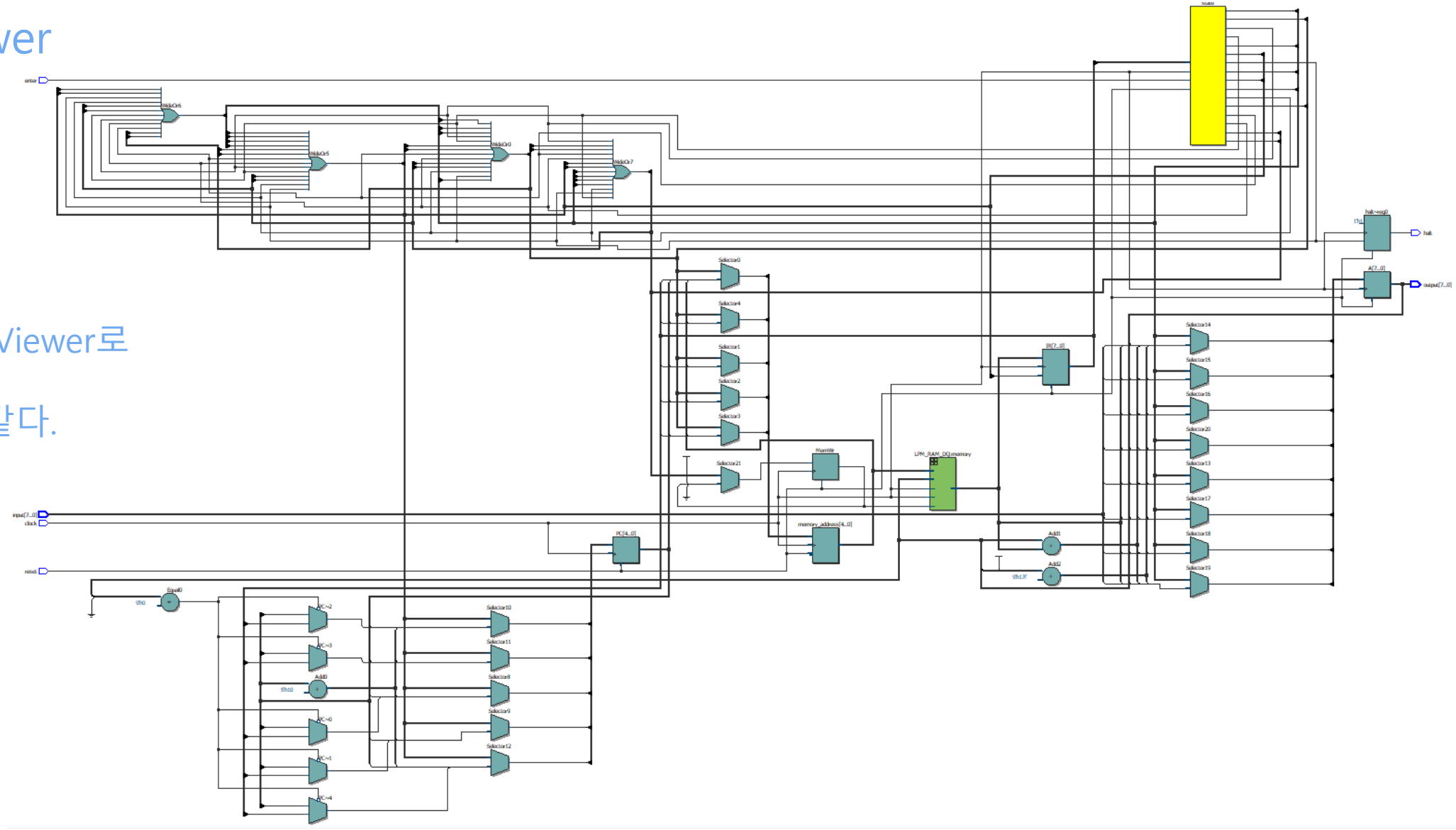
→ 다른 과정은 모두 앞에 있는 State Diagram및 Code와 동일하다.

→ 앞선 Code와 다르게 s_halt일 때, halt가 1이 되면서 state가 계속 s_halt에 머무른다는 것을 알 수 있다.

4. RTL Viewer

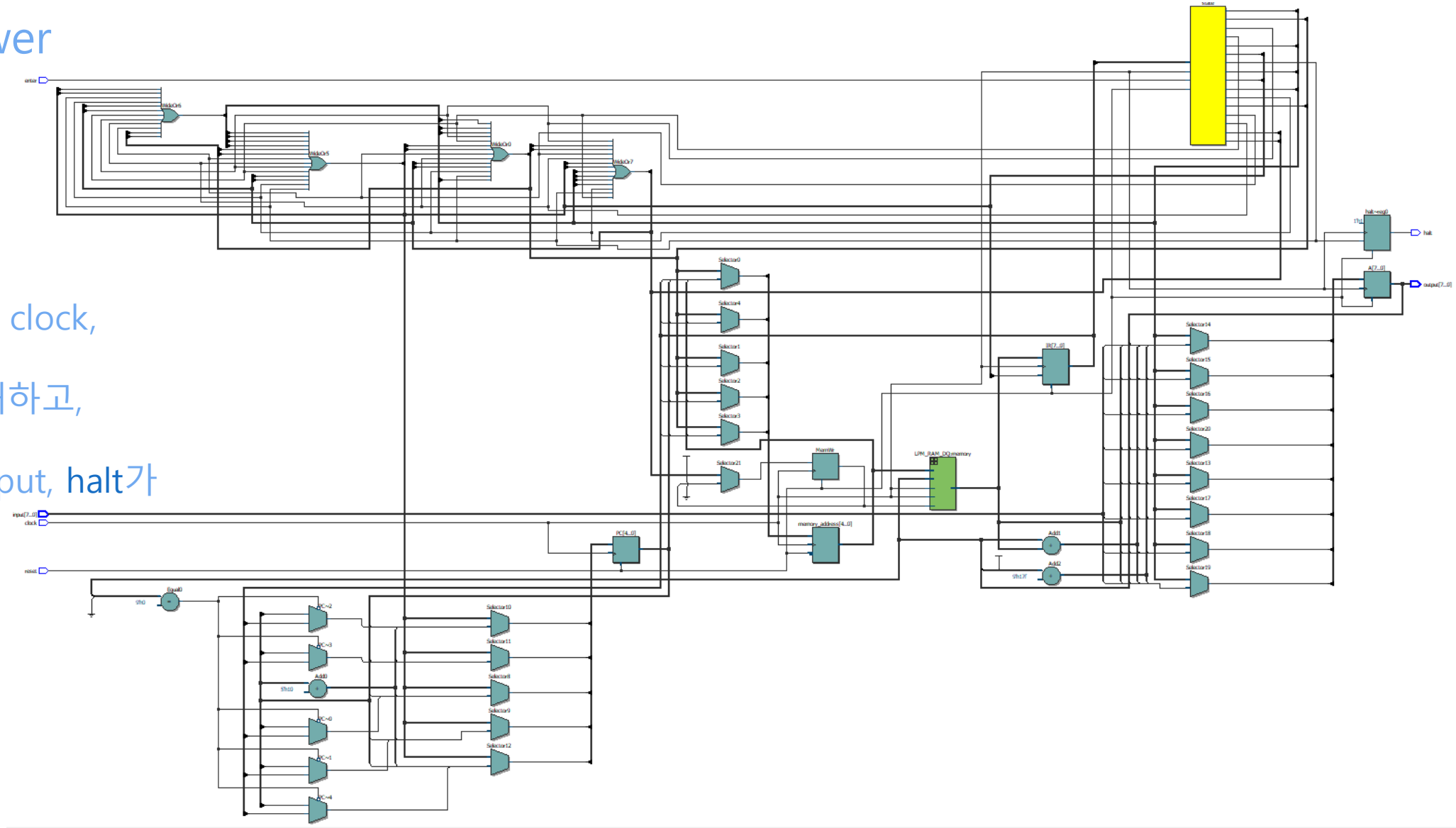
→ 다음 Code를 RTL Viewer로

나타내면 다음과 같다.

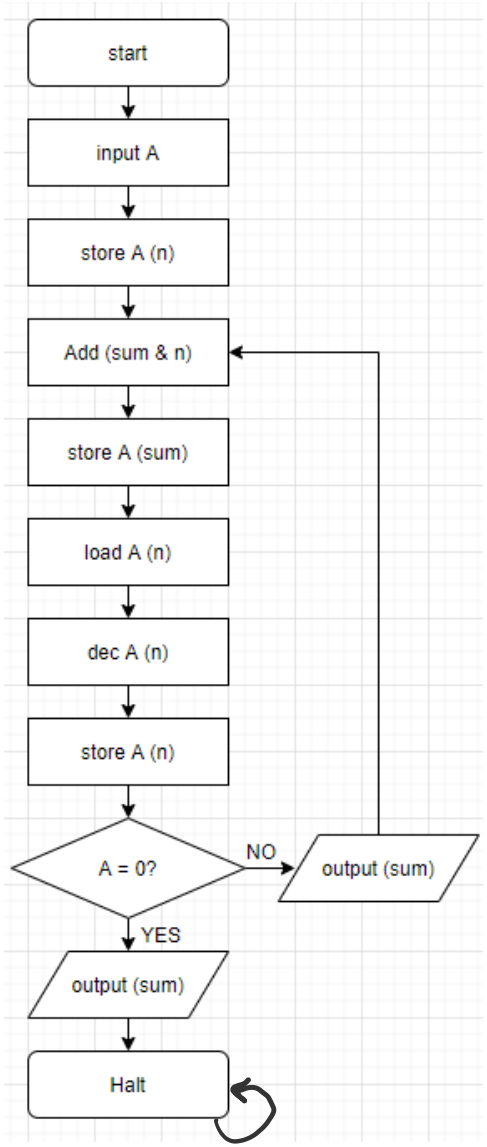


4. RTL Viewer

→ Input으로는 Enter, clock, reset, input이 존재하고, Output으로는 output, halt가 존재한다.



5. Algorithm



→ Test용 알고리즘은 다음과 같다.

→ 다른 부분은 앞에 있는 알고리즘과 모두 동일하다. 하지만 마지막 부분에 halt가 추가된 점이 다르다는 것을 알 수 있다.

6. Assembly code & Binary Code

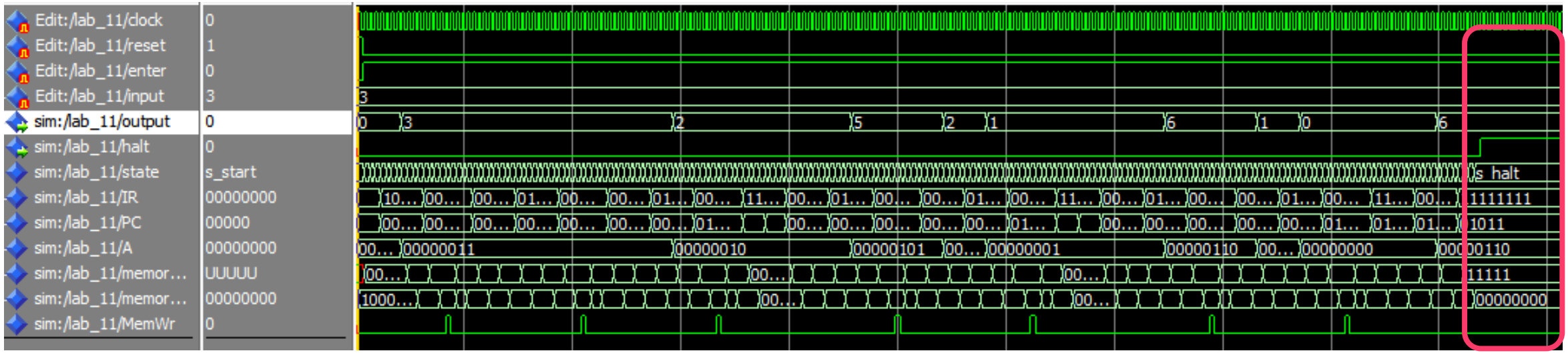
```
-- Content of the RAM memory in the file PROGRAM.MIF

DEPTH = 32;     -- Depth of memory: 5-bit address
WIDTH = 8;      -- Width of memory: 8-bit data

ADDRESS_RADIX = BIN; -- All values in binary (HEX, DEC, OCT, BIN)
DATA_RADIX = BIN;

-- Opcodes for the EC-2
--     000 = load A,aaaaa
--     001 = store A,aaaaa
--     010 = add A,aaaaa
--     011 = dec A,aaaaa
--     100 = in A
--     101 = out A
--     110 = jmp aaaaa
--     111 = halt

-- Specify the memory content.
-- Format of each memory location is
--      address : data
```

```
CONTENT
   BEGIN
   [00000..11111] :  00000000;   -- Initialize locations range 00-FF to 0000
-------------------------------------------------------------
-- Sum
-- Program to sum N downto 1
   00000 :  10000000;   -- input A
   00001 :  00111111;   -- store A,n

   00010 :  00011111;   -- loop: load A,n -- n + sum
   00011 :  01011110;   -- add A,sum
   00100 :  00111110;   -- store A,sum

   00101 :  00011111;   -- load A,n       -- decrement A
   00110 :  01111111;   -- dec A,n
   00111 :  00111111;   -- store A,n

   01000 :  11000010;   -- jmp
   01001 :  00011110;   -- out: load A,sum
   01010 :  11111111;   -- halt

   11110 :  00000000;   -- sum
   11111 :  00000000;   -- n
END;
```

→ 다음은 Assembly Code와 Binary Code이다.

→ 이 역시 마찬가지로 앞선 Binary Code에서 halt만 추가해주었다.

7. Simulation



→ 다음은 Simulation을 진행한 결과이다.

→ 해당 Simulation은 앞에 진행한 Simulation과 모든 것이 동일한 것을 확인할 수 있다. 다만 마지막에 sum이 6이 되고, A가 0이 되면서 더 이상 loop를 진행하지 않아도 되어서 halt가 1이 된 것을 확인할 수 있다.

8. Discussion

→ 이번 실습은 처음부터 CPU를 직접 설계하면서 Datapath와 State Diagram도 그리고, test용 Algorithm 도 설계해서 assembly code & the binary code를 직접 구현해보는 과정을 거치면서 CPU에 대해서 더 잘 알 수 있었다.

→ 해당 과정에서 여러 오류를 고치기도 했고, 궁금한 점을 해결하기 위해 다양한 Simulation을 해보면서 해당 구조에 대해 더 잘 알 수 있었다.

→ 다음은 여러 오류 및 궁금한 점을 하나하나 Simulation해본 결과이다.

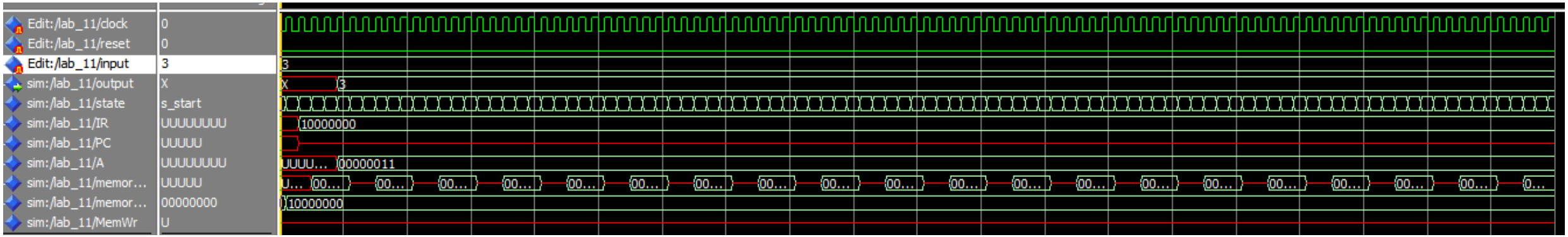
8. Discussion

```
CASE state IS
WHEN s_start => -- reset, start
   memory_address <= PC;
   state <= s_fetch;
WHEN s_fetch => -- fetch
   IR <= memory_data;
   PC <= PC + 1;
   state <= s_decode;
WHEN s_decode => -- decode
   memory_address <= IR(4 DOWNTO 0);
   state <= s_decode2;
WHEN s_decode2 =>
   CASE IR(7 DOWNTO 5) IS
      WHEN "000" => state <= s_load;
      WHEN "001" => state <= s_store;
      WHEN "010" => state <= s_add;
      WHEN "011" => state <= s_dec;
      WHEN "100" => state <= s_input;
      WHEN "101" => state <= s_output;
      WHEN "110" => state <= s_jmp;
      WHEN "111" => state <= s_halt;
      WHEN OTHERS => state <= s_start;
   END CASE;
WHEN s_load =>             -- load A fro
   A <= memory_data;
   state <= s_start;
WHEN s_store =>             -- store A t
   MemWr <= '1';
   state <= s_store2;
WHEN s_store2 =>          -- need an ext
   MemWr <= '0';
   state <= s_start;
```

```
WHEN s_add =>              -- add
   A <= A + memory_data;
   state <= s_start;
WHEN s_dec =>              -- subtract
   A <= A - 1;
   state <= s_start;
WHEN s_input =>
   A <= input;
   state <= s_start;
WHEN s_output =>
   --output <= A;
   state <= s_start;
WHEN s_jmp =>
   IF (A /= 0) THEN          -- jump if
      PC <= IR(4 DOWNTO 0);
   END IF;
   state <= s_start;
WHEN s_halt =>
   halt <= '1';
   state <= s_halt;
WHEN OTHERS =>
   state <= s_halt;
END CASE;
```

→ 다음은 State 수를 임의로 줄여서 Simulaion을 진행해보았다. 하지만, 하나의 State라도 없으면 Clock이 부족해 s_input을 제외하고 다른 State로 넘어가지 못하는 현상이 발생하였다. 이로 인해 해당 State들은 모두 필요한 것임을 알 수 있었다.

8. Discussion

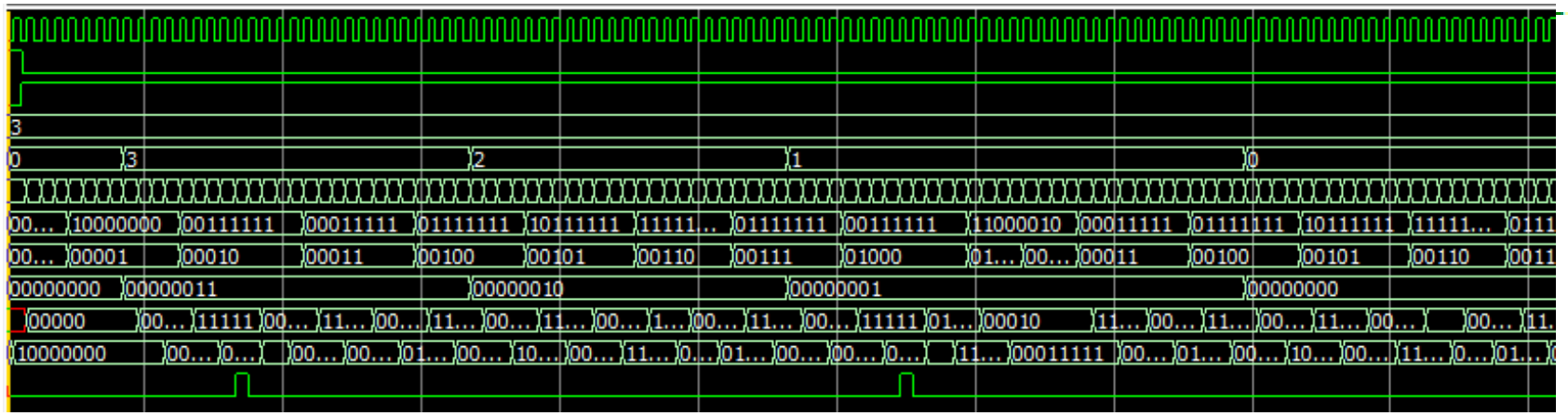


→ 이는 앞선 Code와 옆에 있는 Binary Code를 실행시킨 Simulation 결과이다. 이를 통해 해당 Simulation이 제대로 진행되지 않았음을 알 수 있다.

→ 이를 통해 State는 필요하지 않은 것 없이 존재한다는 것을 알 수 있다.

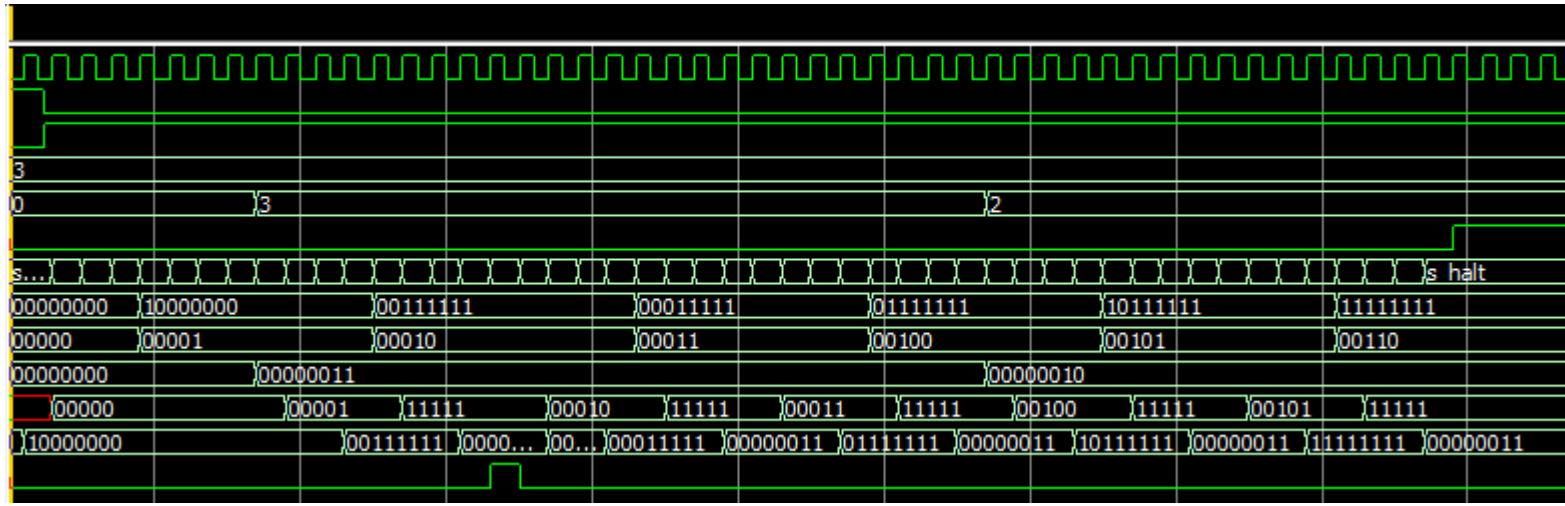
```
-- Sum
-- Program to sum N downto 1
   00000 :  10000000;   -- input A
   00001 :  00111111;   -- store A,n

   00010 :  00011111;   -- loop: load A,n -- n + sum
   00011 :  01011110;   -- add A,sum
   00100 :  00111110;   -- store A,sum

   00101 :  00011111;   -- load A,n        -- decrement A
   00110 :  01111111;   -- dec A,n
   00111 :  00111111;   -- store A,n

   01000 :  11000010;   -- jmp
   01001 :  00011110;   -- out: load A,sum
   01010 :  11111111;   -- halt

   11110 :  00000000;   -- sum
   11111 :  00000000;   -- n
END;
```

8. Discussion



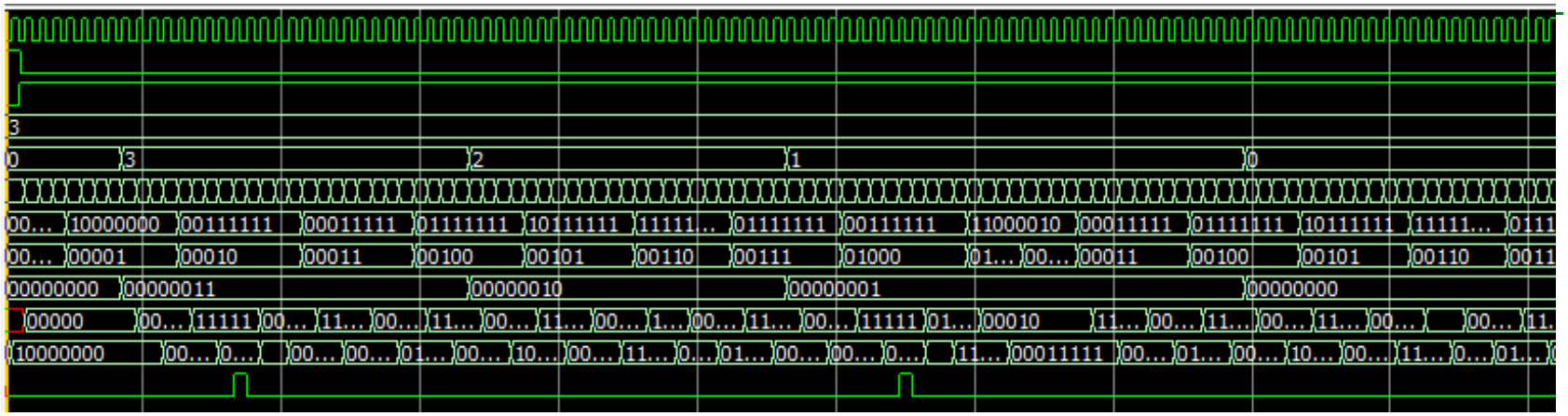
```
--decrement A
   00000 :   10000000;    -- input A
   00001 :   00111111;    -- store A,n

   00010 :   00011111;    -- load A, n
   00011 :   01111111;    -- dec A, n
   00100 :   10111111;    -- out A, n
   00101 :   11111111;    -- halt

   11111 :   00000000;    -- n
```

→ 또한, dec의 기능이 제대로 작동하는지 판단하기 위해 해당 Binary Code와 같이 작성하여 실행해보았다.



→ 위는 halt가 없는 경우 오른쪽은 halt가 있는 경우이다.

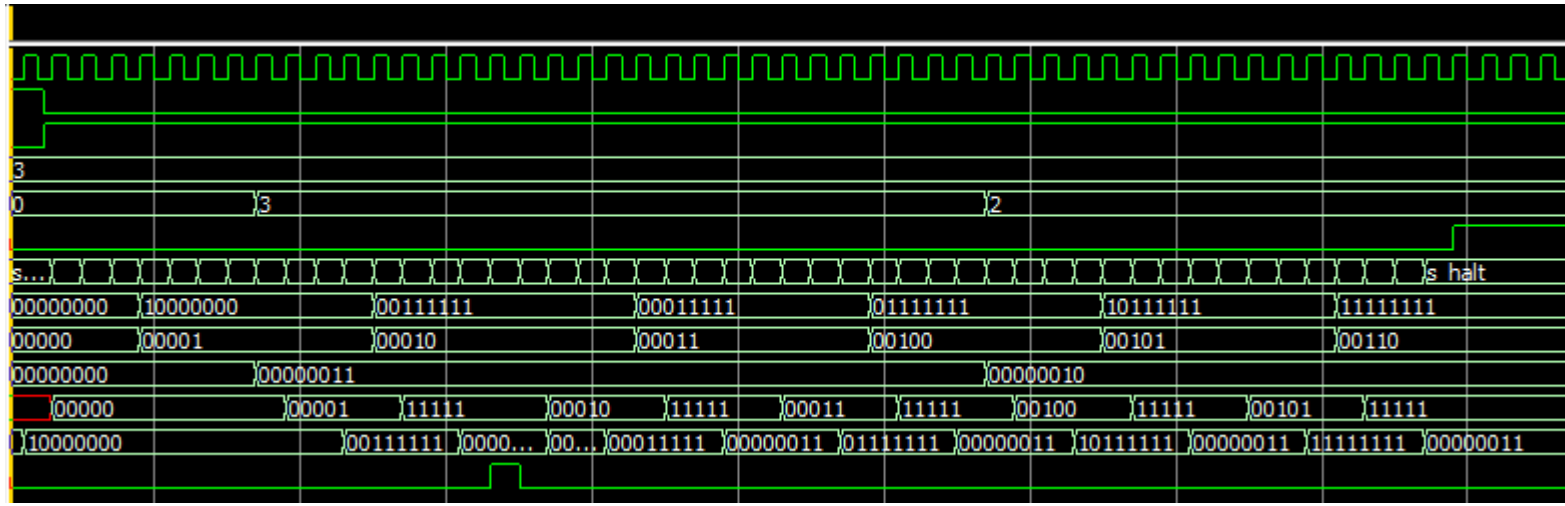
8. Discussion



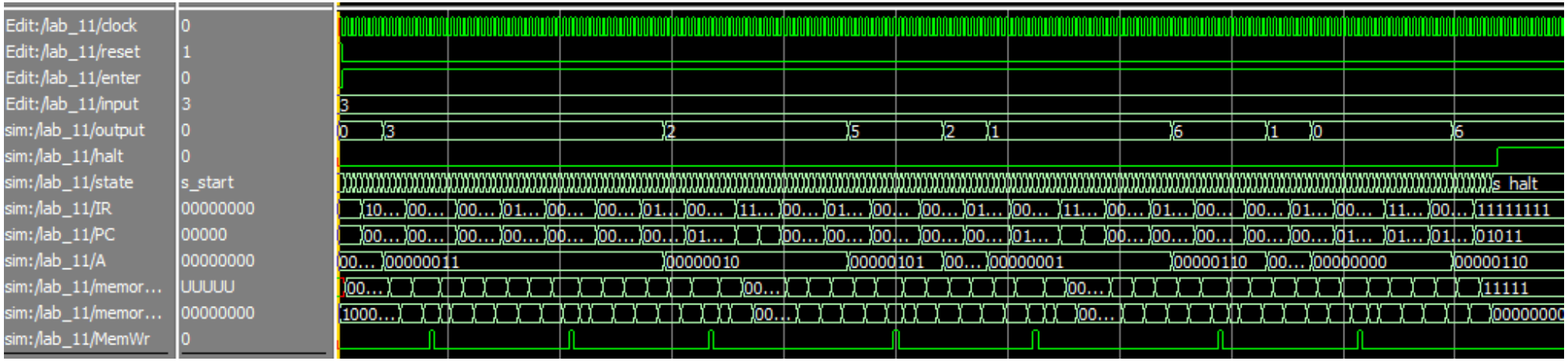
```
--decrement A
   00000 :   10000000;    -- input A
   00001 :   00111111;    -- store A,n

   00010 :   00011111;    -- load A, n
   00011 :   01111111;    -- dec A, n
   00100 :   10111111;    -- out A, n
   00101 :   11111111;    -- halt

   11111 :   00000000;    -- n
```

→ 이를 통해 dec Simulation이 잘 진행되는 것을 확인할 수 있다.

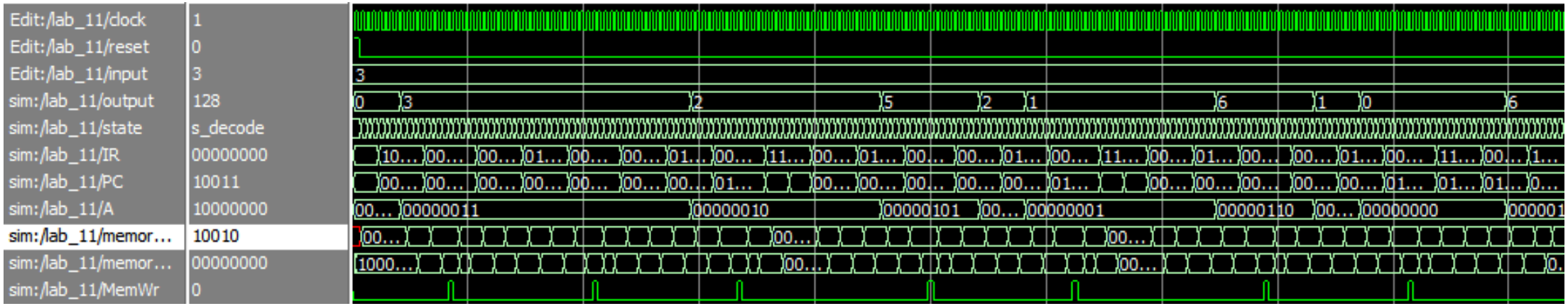
→ 또한, halt를 한 경우에는 loop를 실행시키지 않아 하나만 감소하고 종료되는 것을 알 수 있다.



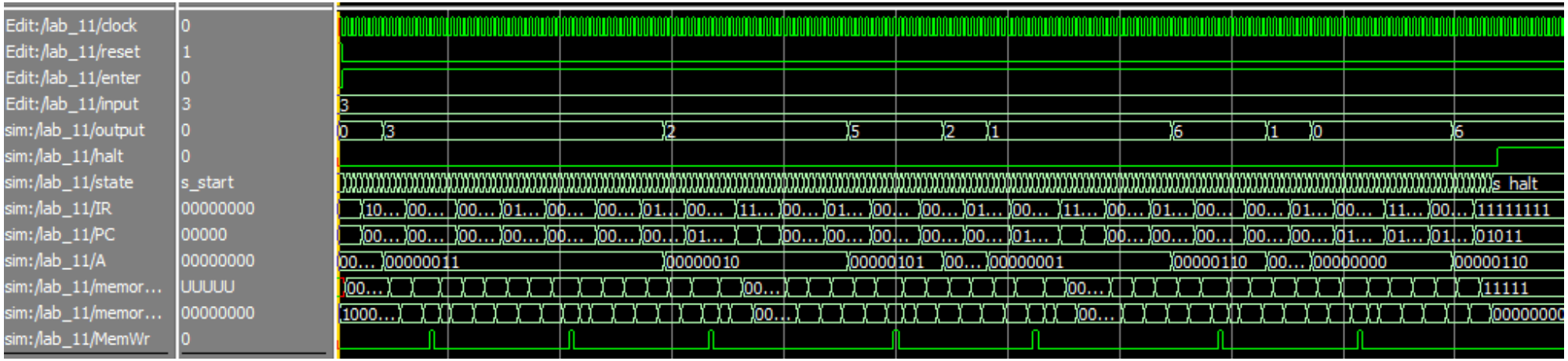
8. Discussion



→ 또한, Input이 맞는 Signal임을 알려주는 Enter의 존재가 굳이 필요하지 않다고 판단하여 이를 비교하는 Simulation을 진행하였다. 위에는 Enter가 있는 Simulation, 아래는 없는 Simulation이다.

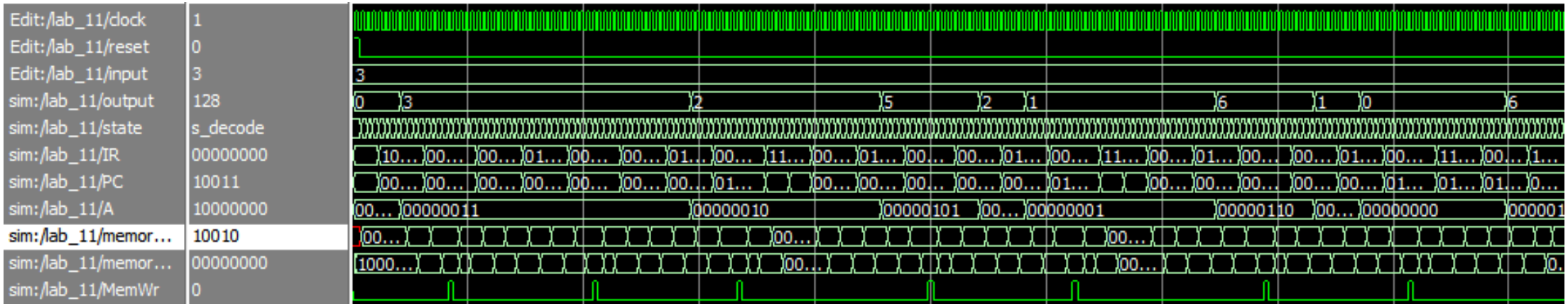


8. Discussion



→ 두 Simulation을 비교해본 결과 Enter가 존재하지 않아도

Simulation이 정상적으로 진행된다는 것을 알 수 있었다.



8. Discussion

→ 다음과 같이 여러가지 실험을 해본 결과 Clock 수를 충분히 주어져 다음 State로 무사히 넘어갈 수 있고, Enter와 Halt의 존재 여부와 상관없이 동일한 결과가 나온다는 것이다. 다만 Halt가 없을 때는 종료가 되는 시점을 명확하게 알 수 없다는 단점이 있다.

→ 또한, Binary Code를 수정함으로 전혀 다른 Simulation 결과를 볼 수 있다는 것을 알 수 있었고, Binary Code를 통해 VHDL Code가 잘 설계되었는지도 검증할 수 있다는 사실을 알 수 있었다.