# Lab 10. General CPU design 2

201810800 이혜인

# Lab 10-1

- 주어진 VHDL code(Page 4)를 사용해서 EC-2 microprocessor를 구현하고 "RTL view" 기능을 이용해서 구현된 결과를 분석하라.
  1. 해당 code를 분석해서 Control unit의 state diagram을 도출하고 손으로 그려라.
     - Note: Asynch RAM used in EC-2 is not supported by current generation FPGAs, for example Cyclone IV. Cyclone IV only supports Synchronous RAM, so the VHDL code is also modified.
  2. "program_EC2.mif" 에 명시된 프로그램 두 개를 simulation으로 검증하라. (The last content in program.mif is actually written to the memory.) You have to only perform "RTL simulation" since it is much easier for verification. (SDO 파일 없이, "gate_work" 대신 "work" library에서 entity load)
     1. Calculating the GCD of two numbers (You have to be able to enter external inputs, as in EC-2 in our lecture)
     2. Sum N down to 1. (You have to be careful for the "Enter" signal duration)

# Lab 10-1. ISA

| Instruction | Encoding | Operation | Comment |
|---|---|---|---|
| LOAD A, address | 000 aaaaa | $A \leftarrow M[\text{aaaaa}]$ | Load $A$ with content of memory location aaaaa |
| STORE A, address | 001 aaaaa | $M[\text{aaaaa}] \leftarrow A$ | Store $A$ into memory location aaaaa |
| ADD A, address | 010 aaaaa | $A \leftarrow A + M[\text{aaaaa}]$ | Add $A$ with $M[\text{aaaaa}]$ and store the result back into $A$ |
| SUB A, address | 011 aaaaa | $A \leftarrow A - M[\text{aaaaa}]$ | Subtract $A$ with $M[\text{aaaaa}]$ and store result back into $A$ |
| IN A | 100 ××××× | $A \leftarrow \textit{Input}$ | Input to $A$ |
| JZ address | 101 aaaaa | IF $(A = 0)$ THEN $PC = \text{aaaaa}$ | Jump to address if $A$ is zero |
| JPOS address | 110 aaaaa | IF $(A \geq 0)$ THEN $PC = \text{aaaaa}$ | Jump to address if $A$ is a positive number |
| HALT | 111 ××××× | Halt | Halt execution |

Notations:

$A$ = accumulator.
$M$ = memory.
$PC$ = program counter.
aaaaa = five bits for specifying a memory address.
× = don't-cares.

**Figure 12.10** Instruction set for the EC-2.

# Lab 10-1. Reference:EC-2

- Behavioral VHDL code for EC-2: mp_EC2.vhd
- Memory content: program_EC2.mif

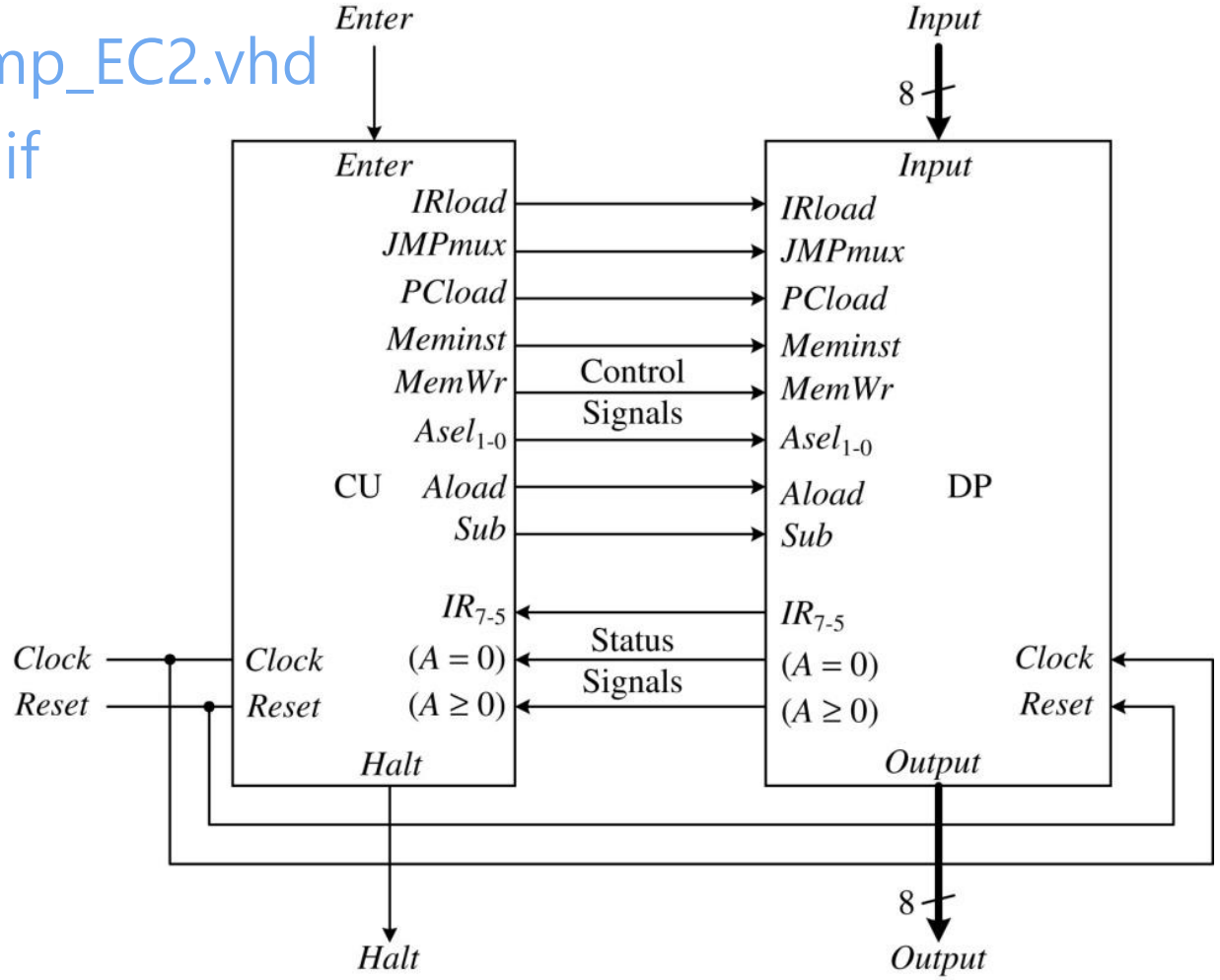



**Figure 12.13** Complete circuit for the EC-2 general-purpose microprocessor.

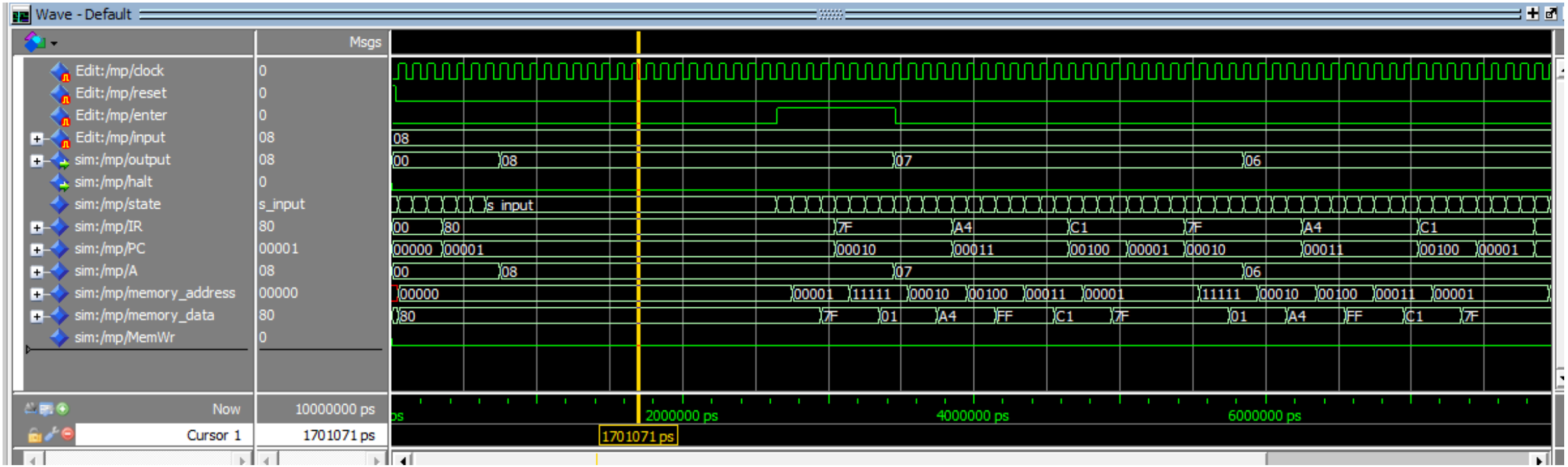
# Lab 10-2

- Write the assembly code & the binary code for countdown from N, store it as program_EC2.mif, run, verify with simulation.
- Lab 9에서 사용한 countdown 프로그램을 참고하라.

# Lab 10-1 & 10-2

- Result
  - Explain how the mp_EC2.vhd is different from the EC-2 in the textbook.
  - Explain the binary code you wrote for countdown. (for 10-2)
  - Simulation capture & detailed explanation. Does it behave as the program was intended?
  - etc.
- Discussion
  - the problems met during simulation & verification
  - how they have been solved
  - the problems remained unsolved

# Example simulation capture




- Input, output, 내부 signal 모두 관찰 필요

# Lab 10. General CPU design 2

## 10-1 : Result

# 1. VHDL Code – 달라진 부분 위주

```
-- EC-2 Behavioral FSMD description
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY mp_EC2 IS PORT (
   clock, reset: IN STD_LOGIC;
   enter: IN STD_LOGIC;
   -- data input
   input: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
   -- data output
   output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
   -- control outputs
   halt: OUT STD_LOGIC);
END mp_EC2;

ARCHITECTURE FSMD OF mp_EC2 IS
   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_sub,s_input,s_jz,s_jpos,s_halt);
   SIGNAL state: state_type;                    -- states
   SIGNAL IR: STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Instruction register
   SIGNAL PC: STD_LOGIC_VECTOR(4 DOWNTO 0);  -- Program counter
   SIGNAL A:  STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Accumulator
   SIGNAL memory_address:  STD_LOGIC_VECTOR(4 DOWNTO 0); -- memory address
   SIGNAL memory_data:  STD_LOGIC_VECTOR(7 DOWNTO 0); -- memory data input
   SIGNAL MemWr: STD_LOGIC;
BEGIN
   memory: lpm_ram_dq   -- 32 locations x 8 bits wide asynchronous memory
      GENERIC MAP (
         lpm_widthad => 5,
         lpm_outdata => "REGISTERED", -- jj
         lpm_indata  => "REGISTERED", -- jj
         lpm_numwords => 32,  -- jj
         lpm_address_control => "REGISTERED", -- jj
         lpm_file => "program_EC2.mif", -- fill ram with content of file program.mif
         lpm_width   => 8)
      PORT MAP (
         data     => A,
         address     => memory_address,
         inclock => clock,  -- added jj
         outclock => clock, -- jj
         we       => MemWr,
         q        => memory_data);

   PROCESS(clock,reset)
   BEGIN
      IF(reset = '1') THEN
         PC <= "00000";
         IR <= "00000000";
         A  <= "00000000";
         MemWr <= '0';
         halt <= '0';
         state <= s_start;
      ELSIF(clock'EVENT AND clock = '1') THEN
         CASE state IS
         WHEN s_start => -- reset, start
            memory_address <= PC;
            state <= s_start2; --jj
         WHEN s_start2 =>
            state <= s_start3;
         WHEN s_start3 =>
            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);    -- memory access using last 5 bits of IR
            state <= s_decode2;  -- jj
         WHEN s_decode2 =>
            state <= s_decode3;
         WHEN s_decode3 =>  -- jj
            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_sub;
               WHEN "100" => state <= s_input;
               WHEN "101" => state <= s_jz;
               WHEN "110" => state <= s_jpos;
               WHEN "111" => state <= s_halt;
               WHEN OTHERS => state <= s_start;
            END CASE;
         WHEN s_load =>          -- load A from memory
            A <= memory_data;
            state <= s_start;
         WHEN s_store =>            -- store A to memory
            MemWr <= '1';
            state <= s_store2;
         WHEN s_store2 =>        -- need an extra state to de-assert MemWr before changing the memory address
            MemWr <= '0';
            state <= s_start;
         WHEN s_add =>           -- add
            A <= A + memory_data;
            state <= s_start;
         WHEN s_sub =>           -- subtract
            A <= A - memory_data;
            state <= s_start;
         WHEN s_input =>
            A <= input;
            IF (Enter = '0') THEN   -- wait for Enter key
               state <= s_input;
            ELSE
               state <= s_start;
            END IF;
         WHEN s_jz =>
            IF (A = 0) THEN         -- jump if A is 0
               PC <= IR(4 DOWNTO 0);
            END IF;
            state <= s_start;
         WHEN s_jpos =>
            IF (A(7) = '0') THEN -- jump if MSB(A) is 0
               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;
      END IF;
   END PROCESS;

   output <= A;   -- send value of Accumulator to the output
END FSMD;
```

→ 다음은 주어진 VHDL Code이다.

# 1. VHDL Code – 달라진 부분 위주

```
1   -- EC-2 Behavioral FSMD description
2   LIBRARY IEEE;
3   USE IEEE.STD_LOGIC_1164.ALL;
4   USE IEEE.STD_LOGIC_ARITH.ALL;
5   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
6
7   LIBRARY lpm;
8   USE lpm.lpm_components.ALL;
9
10  ENTITY mp_EC2 IS PORT (
11      clock, reset: IN STD_LOGIC;
12      enter: IN STD_LOGIC;
13      -- data input
14      input: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
15      -- data output
16      output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
17      -- control outputs
18      halt: OUT STD_LOGIC);
19  END mp_EC2;
20
21  ARCHITECTURE FSMD OF mp_EC2 IS
22      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_sub,s_input,s_jz,s_jpos,s_halt);
23      SIGNAL state: state_type;               -- states
24      SIGNAL IR: STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Instruction register
25      SIGNAL PC: STD_LOGIC_VECTOR(4 DOWNTO 0);  -- Program counter
26      SIGNAL A:  STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Accumulator
27      SIGNAL memory_address:  STD_LOGIC_VECTOR(4 DOWNTO 0); -- memory address
28      SIGNAL memory_data:  STD_LOGIC_VECTOR(7 DOWNTO 0); -- memory data input
29      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.

# 1. VHDL Code – 달라진 부분 위주

```
1   -- EC-2 Behavioral FSMD description
2   LIBRARY IEEE;
3   USE IEEE.STD_LOGIC_1164.ALL;
4   USE IEEE.STD_LOGIC_ARITH.ALL;
5   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
6
7   LIBRARY lpm;
8   USE lpm.lpm_components.ALL;
9
10  ENTITY mp_EC2 IS PORT (
11      clock, reset: IN STD_LOGIC;
12      enter: IN STD_LOGIC;
13      -- data input
14      input: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
15      -- data output
16      output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
17      -- control outputs
18      halt: OUT STD_LOGIC);
19  END mp_EC2;
20
21  ARCHITECTURE FSMD OF mp_EC2 IS
22      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_sub,s_input,s_jz,s_jpos,s_halt);
23      SIGNAL state: state_type;              -- states
24      SIGNAL IR: STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Instruction register
25      SIGNAL PC: STD_LOGIC_VECTOR(4 DOWNTO 0);  -- Program counter
26      SIGNAL A:  STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Accumulator
27      SIGNAL memory_address:  STD_LOGIC_VECTOR(4 DOWNTO 0); -- memory address
28      SIGNAL memory_data:  STD_LOGIC_VECTOR(7 DOWNTO 0); -- memory data input
29      SIGNAL MemWr: STD_LOGIC;
```

→ Output으로는 output, halt를 가지고 있다. 이때 output은 연산 한 결과이고, halt는 IR의 상위 3bit가 111이 되면 halt가 1이 되면서 프로그램의 종료를 알리는 역할을 한다.

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

Lab 10 General CPU design 2

# 1. VHDL Code – 달라진 부분 위주

```
1   -- EC-2 Behavioral FSMD description
2   LIBRARY IEEE;
3   USE IEEE.STD_LOGIC_1164.ALL;
4   USE IEEE.STD_LOGIC_ARITH.ALL;
5   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
6
7   LIBRARY lpm;
8   USE lpm.lpm_components.ALL;
9
10  ENTITY mp_EC2 IS PORT (
11      clock, reset: IN STD_LOGIC;
12      enter: IN STD_LOGIC;
13      -- data input
14      input: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
15      -- data output
16      output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
17      -- control outputs
18      halt: OUT STD_LOGIC);
19  END mp_EC2;
20
21  ARCHITECTURE FSMD OF mp_EC2 IS
22      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_sub,s_input,s_jz,s_jpos,s_halt);
23      SIGNAL state: state_type;              -- states
24      SIGNAL IR: STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Instruction register
25      SIGNAL PC: STD_LOGIC_VECTOR(4 DOWNTO 0);  -- Program counter
26      SIGNAL A:  STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Accumulator
27      SIGNAL memory_address:  STD_LOGIC_VECTOR(4 DOWNTO 0); -- memory address
28      SIGNAL memory_data:  STD_LOGIC_VECTOR(7 DOWNTO 0); -- memory data input
29      SIGNAL MemWr: STD_LOGIC;
```

→ 다음 IR은 Instruction Register로 실행할 instruction을 명시해준다. PC는 Program Counter로 Memory의 특정 address를 가리킨다. A는 Accumulator로 연산 결과를 저장하고, memory_address와 memory_data은 각 이름 그대로 memory의 address와 data input을 받는다. MemWr은 Memory 저장 여부를 알려준다.

# 1. VHDL Code – 달라진 부분 위주

```
1   -- EC-2 Behavioral FSMD description
2   LIBRARY IEEE;
3   USE IEEE.STD_LOGIC_1164.ALL;
4   USE IEEE.STD_LOGIC_ARITH.ALL;
5   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
6
7   LIBRARY lpm;
8   USE lpm.lpm_components.ALL;
9
10  ENTITY mp_EC2 IS PORT (
11      clock, reset: IN STD_LOGIC;
12      enter: IN STD_LOGIC;
13      -- data input
14      input: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
15      -- data output
16      output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
17      -- control outputs
18      halt: OUT STD_LOGIC);
19  END mp_EC2;
20
21  ARCHITECTURE FSMD OF mp_EC2 IS
22      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_sub,s_input,s_jz,s_jpos,s_halt);
23      SIGNAL state: state_type;               -- states
24      SIGNAL IR: STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Instruction register
25      SIGNAL PC: STD_LOGIC_VECTOR(4 DOWNTO 0);  -- Program counter
26      SIGNAL A:  STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Accumulator
27      SIGNAL memory_address:  STD_LOGIC_VECTOR(4 DOWNTO 0); -- memory address
28      SIGNAL memory_data:  STD_LOGIC_VECTOR(7 DOWNTO 0); -- memory data input
29      SIGNAL MemWr: STD_LOGIC;
```

→ 해당 부분에서 강의자료에서 배운 EC-2와 다른 부분이 많이 나온다. 이는 Cyclone IV가 EC-2가 사용하는 Asynchronous RAM이 아닌 Synchronous RAM만을 지원해서 일부 코드가 수정되게 되었다.

Lab 10 General CPU design 2

# 1. VHDL Code – 달라진 부분 위주

```
31  BEGIN
32      memory: lpm_ram_dq   -- 32 locations x 8 bits wide asynchronous memory
33          GENERIC MAP (
34              lpm_widthad => 5,
35              lpm_outdata => "REGISTERED", -- jj
36              lpm_indata  => "REGISTERED", -- jj
37              lpm_numwords => 32,  -- jj
38              lpm_address_control => "REGISTERED", -- jj
39              lpm_file => "program_EC2.mif", -- fill ram with content of file program.mif
40              lpm_width   => 8)
41          PORT MAP (
42              data      => A,
43              address      => memory_address,
44              inclock => clock,  -- added jj
45              outclock => clock, -- jj
46              we        => MemWr,
47              q         => memory_data);
48
49      PROCESS(clock,reset)
50      BEGIN
51          IF(reset = '1') THEN
52              PC <= "00000";
53              IR <= "00000000";
54              A  <= "00000000";
55              MemWr <= '0';
56              halt <= '0';
57              state <= s_start;
58          ELSIF(clock'EVENT AND clock = '1') THEN
59              CASE state IS
60              WHEN s_start => -- reset, start
61                  memory_address <= PC;
62                  state <= s_start2; --jj
63              WHEN s_start2 =>
64                  state <= s_start3;
65              WHEN s_start3 =>
66                  state <= s_fetch;
67              WHEN s_fetch => -- fetch
68                  IR <= memory_data;
69                  PC <= PC + 1;
70                  state <= s_decode;
71              WHEN s_decode => -- decode
72                  memory_address <= IR(4 DOWNTO 0);    -- memory access using last 5 bits of IR
73                  state <= s_decode2;  -- jj
```

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

→ 각각의 signal끼리 올바르게 연결해주었다.

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

→ Clock의 rising edge에서 state가 s_start일 때, memory_address에 PC 값을 저장하고, s_start2로 이동한다. s_start2이면 바로 s_start3으로 가고, s_start3에서는 s_fetch 상태로 넘어간다.

# 1. VHDL Code – 달라진 부분 위주

```
31  BEGIN
32      memory: lpm_ram_dq   -- 32 locations x 8 bits wide asynchronous memory
33          GENERIC MAP (
34              lpm_widthad => 5,
35              lpm_outdata => "REGISTERED", -- jj
36              lpm_indata  => "REGISTERED", -- jj
37              lpm_numwords => 32,  -- jj
38              lpm_address_control => "REGISTERED", -- jj
39              lpm_file => "program_EC2.mif", -- fill ram with content of file program.mif
40              lpm_width   => 8)
41          PORT MAP (
42              data      => A,
43              address      => memory_address,
44              inclock => clock,  -- added jj
45              outclock => clock, -- jj
46              we        => MemWr,
47              q         => memory_data);
48
49      PROCESS(clock,reset)
50      BEGIN
51          IF(reset = '1') THEN
52              PC <= "00000";
53              IR <= "00000000";
54              A  <= "00000000";
55              MemWr <= '0';
56              halt <= '0';
57              state <= s_start;
58          ELSIF(clock'EVENT AND clock = '1') THEN
59              CASE state IS
60              WHEN s_start => -- reset, start
61                  memory_address <= PC;
62                  state <= s_start2; --jj
63              WHEN s_start2 =>
64                  state <= s_start3;
65              WHEN s_start3 =>
66                  state <= s_fetch;
67              WHEN s_fetch => -- fetch
68                  IR <= memory_data;
69                  PC <= PC + 1;
70                  state <= s_decode;
71              WHEN s_decode => -- decode
72                  memory_address <= IR(4 DOWNTO 0);    -- memory access using last 5 bits of IR
73                  state <= s_decode2;  -- jj
```

→ s_fetch 상태에서는 memory_data를 IR에 저장하고, PC는 1 증가시킨다. 그리고 state는 s_decode 상태로 넘어간다.

→ s_decode 상태에서는 IR의 하위 5bit가 memory_address에 저장되게 되고, state는 s_decode2로 넘어가게 된다.

Lab 10 General CPU design 2

# 1. VHDL Code – 달라진 부분 위주

```
74      WHEN s_decode2 =>
75         state <= s_decode3;
76      WHEN s_decode3 =>  -- jj
77         CASE IR(7 DOWNTO 5) IS
78            WHEN "000" => state <= s_load;
79            WHEN "001" => state <= s_store;
80            WHEN "010" => state <= s_add;
81            WHEN "011" => state <= s_sub;
82            WHEN "100" => state <= s_input;
83            WHEN "101" => state <= s_jz;
84            WHEN "110" => state <= s_jpos;
85            WHEN "111" => state <= s_halt;
86            WHEN OTHERS => state <= s_start;
87         END CASE;
88      WHEN s_load =>          -- load A from memory
89         A <= memory_data;
90         state <= s_start;
91      WHEN s_store =>            -- store A to memory
92         MemWr <= '1';
93         state <= s_store2;
94      WHEN s_store2 =>        -- need an extra state to de-assert MemWr before changing the memory address
95         MemWr <= '0';
96         state <= s_start;
97      WHEN s_add =>           -- add
98         A <= A + memory_data;
99         state <= s_start;
100     WHEN s_sub =>           -- subtract
101        A <= A - memory_data;
102        state <= s_start;
103     WHEN s_input =>
104        A <= input;
105        IF (Enter = '0') THEN   -- wait for Enter key
106           state <= s_input;
107        ELSE
108           state <= s_start;
109        END IF;
110     WHEN s_jz =>
111        IF (A = 0) THEN         -- jump if A is 0
112           PC <= IR(4 DOWNTO 0);
113        END IF;
114        state <= s_start;
```

→ 다음으로 s_decode2 상태에서는 바로 s_decode3로 넘어가고, s_decode3에서는 IR 상위 3 bit에 따라 각각에 맞는 Control Word로 넘어가게 된다.

→ 먼저 “000”이면, state가 s_load가 되어 memory_data를 A에 저장하여 data를 load한 다음, s_start 상태로 다시 되돌아간다.

→ “001”이면, state가 s_store가 되어 MemWr가 1이 되어 Momory에 저장되었음을 알리고, s_store2 상태로 넘어간다. s_store2 상태에서는 MemWr이 다시 0이 되고, s_start 상태로 다시 되돌아간다.

Lab 10 General CPU design 2

# 1. VHDL Code – 달라진 부분 위주

```
74      WHEN s_decode2 =>
75         state <= s_decode3;
76      WHEN s_decode3 =>  -- jj
77         CASE IR(7 DOWNTO 5) IS
78            WHEN "000" => state <= s_load;
79            WHEN "001" => state <= s_store;
80            WHEN "010" => state <= s_add;
81            WHEN "011" => state <= s_sub;
82            WHEN "100" => state <= s_input;
83            WHEN "101" => state <= s_jz;
84            WHEN "110" => state <= s_jpos;
85            WHEN "111" => state <= s_halt;
86            WHEN OTHERS => state <= s_start;
87         END CASE;
88      WHEN s_load =>          -- load A from memory
89         A <= memory_data;
90         state <= s_start;
91      WHEN s_store =>            -- store A to memory
92         MemWr <= '1';
93         state <= s_store2;
94      WHEN s_store2 =>        -- need an extra state to de-assert MemWr before changing the memory address
95         MemWr <= '0';
96         state <= s_start;
97      WHEN s_add =>           -- add
98         A <= A + memory_data;
99         state <= s_start;
100     WHEN s_sub =>           -- subtract
101        A <= A - memory_data;
102        state <= s_start;
103     WHEN s_input =>
104        A <= input;
105        IF (Enter = '0') THEN   -- wait for Enter key
106           state <= s_input;
107        ELSE
108           state <= s_start;
109        END IF;
110     WHEN s_jz =>
111        IF (A = 0) THEN         -- jump if A is 0
112           PC <= IR(4 DOWNTO 0);
113        END IF;
114        state <= s_start;
```

→ "010"이면, state가 s_add가 되어 A + memory_data가 되어 A에 저장되고, s_start 상태로 다시 되돌아간다.

→ "011"이면, state가 s_sub가 되어 A – memory_data가 되어 A에 저장되고, s_start 상태로 다시 되돌아간다.

→ "100"이면, state가 s_input이 되어 A에 외부 input 값이 저장되고, Enter가 0이면 s_input 상태가 계속 유지되고, Enter가 1이면 s_start 상태로 다시 되돌아가 프로그램이 계속해서 진행 될 수 있게 한다.

# 1. VHDL Code – 달라진 부분 위주

```
74          WHEN s_decode2 =>
75             state <= s_decode3;
76          WHEN s_decode3 =>  -- jj
77             CASE IR(7 DOWNTO 5) IS
78                WHEN "000" => state <= s_load;
79                WHEN "001" => state <= s_store;
80                WHEN "010" => state <= s_add;
81                WHEN "011" => state <= s_sub;
82                WHEN "100" => state <= s_input;
83                WHEN "101" => state <= s_jz;
84                WHEN "110" => state <= s_jpos;
85                WHEN "111" => state <= s_halt;
86                WHEN OTHERS => state <= s_start;
87             END CASE;
110         WHEN s_jz =>
111            IF (A = 0) THEN          -- jump if A is 0
112               PC <= IR(4 DOWNTO 0);
113            END IF;
114            state <= s_start;
115         WHEN s_jpos =>
116            IF (A(7) = '0') THEN -- jump if MSB(A) is 0
117               PC <= IR(4 DOWNTO 0);
118            END IF;
119            state <= s_start;
120         WHEN s_halt =>
121            halt <= '1';
122            state <= s_halt;
123         WHEN OTHERS =>
124            state <= s_halt;
125         END CASE;
126      END IF;
127   END PROCESS;
128
129   output <= A;    -- send value of Accumulator to the output
130END FSMD;
```

→ "101"이면, state가 s_jz가 되어 A가 0이면 PC에 IR 하위 5 bit를 저장하여 해당 address로 이동하고, s_start 상태로 다시 되돌아간다.

→ "110"이면, state가 s_jpos가 되어 A(7)이 0이면 즉 A의 MSB값이 0이면, PC에 IR 하위 5bit가 저장되어 해당 address로 이동하고, s_start 상태로 다시 되돌아간다.

→ "111"이면, state가 s_halt가 되어 halt가 1이 되고, s_halt state에 계속 머무르며 프로그램이 종료되었음을 알린다.

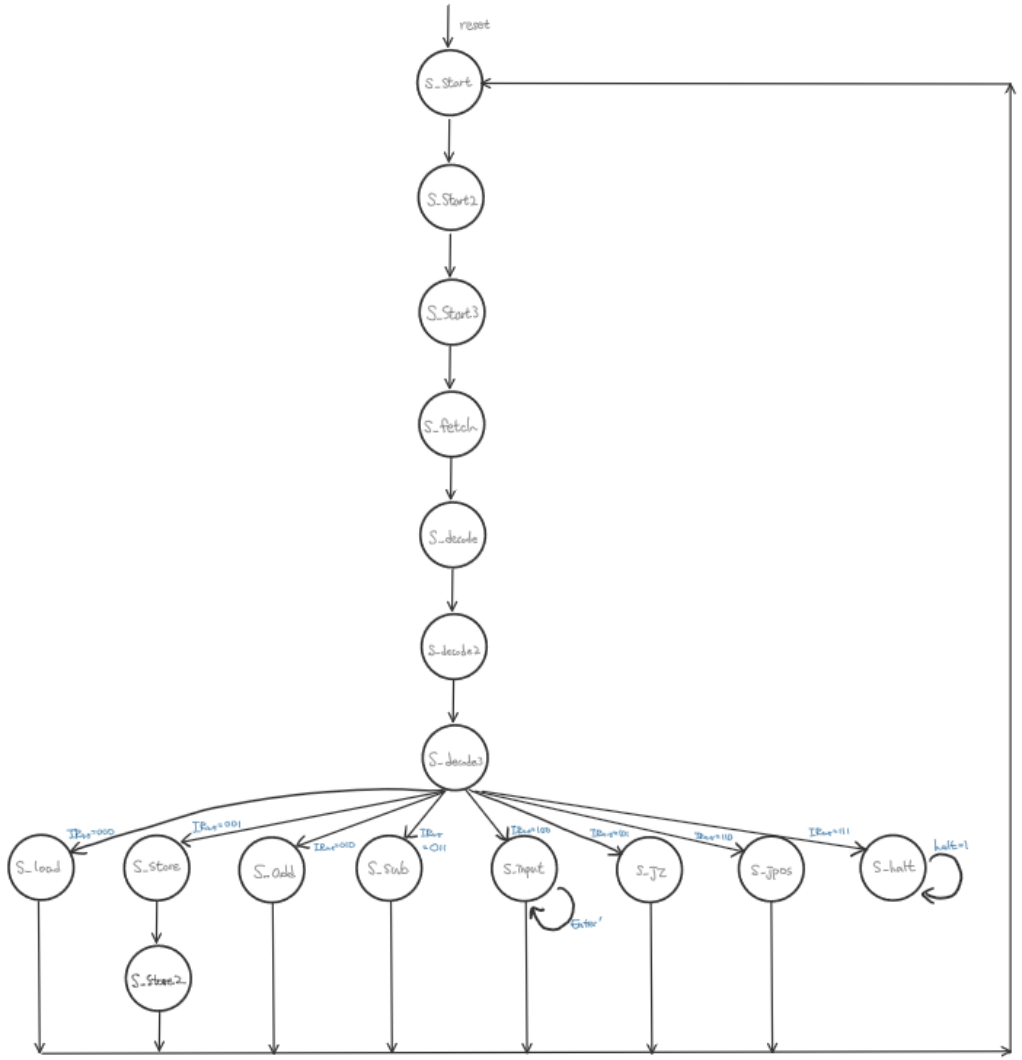
Lab 10 General CPU design 2

# 1. VHDL Code – 달라진 부분 위주

```
1   -- EC-2 Behavioral FSMD description
2   LIBRARY IEEE;
3   USE IEEE.STD_LOGIC_1164.ALL;
4   USE IEEE.STD_LOGIC_ARITH.ALL;
5   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
6
7   LIBRARY lpm;
8   USE lpm.lpm_components.ALL;
9
10  ENTITY mp_EC2 IS PORT (
11     clock, reset: IN STD_LOGIC;
12     enter: IN STD_LOGIC;
13     -- data input
14     input: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
15     -- data output
16     output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
17     -- control outputs
18     halt: OUT STD_LOGIC);
19  END mp_EC2;
20
21  ARCHITECTURE FSMD OF mp_EC2 IS
22     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_sub,s_input,s_jz,s_jpos,s_halt);
23     SIGNAL state: state_type;                    -- states
24     SIGNAL IR: STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Instruction register
25     SIGNAL PC: STD_LOGIC_VECTOR(4 DOWNTO 0);  -- Program counter
26     SIGNAL A:  STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Accumulator
27     SIGNAL memory_address:  STD_LOGIC_VECTOR(4 DOWNTO 0); -- memory address
28     SIGNAL memory_data:  STD_LOGIC_VECTOR(7 DOWNTO 0); -- memory data input
29     SIGNAL MemWr: STD_LOGIC;
30
31  BEGIN
32     memory: lpm_ram_dq   -- 32 locations x 8 bits wide asynchronous memory
33        GENERIC MAP (
34           lpm_widthad => 5,
35           lpm_outdata => "REGISTERED", -- jj
36           lpm_indata  => "REGISTERED", -- jj
37           lpm_numwords => 32,  -- jj
38           lpm_address_control => "REGISTERED", -- jj
39           lpm_file => "program_EC2.mif", -- fill ram with content of file program.mif
40           lpm_width   => 8)
41        PORT MAP (
42           data     => A,
43           address     => memory_address,
44           inclock => clock,  -- added jj
45           outclock => clock, -- jj
46           we       => MemWr,
47           q        => memory_data);
48
49     PROCESS(clock,reset)
50     BEGIN
51        IF(reset = '1') THEN
52           PC <= "00000";
53           IR <= "00000000";
54           A  <= "00000000";
55           MemWr <= '0';
56           halt <= '0';
57           state <= s_start;
58        ELSIF(clock'EVENT AND clock = '1') THEN
59           CASE state IS
60           WHEN s_start => -- reset, start
61              memory_address <= PC;
62              state <= s_start2; --jj
63           WHEN s_start2 =>
64              state <= s_start3;
65           WHEN s_start3 =>
66              state <= s_fetch;
67           WHEN s_fetch => -- fetch
68              IR <= memory_data;
69              PC <= PC + 1;
70              state <= s_decode;
71           WHEN s_decode => -- decode
72              memory_address <= IR(4 DOWNTO 0);    -- memory access using last 5 bits of IR
73              state <= s_decode2;  -- jj
74           WHEN s_decode2 =>
75              state <= s_decode3;
76           WHEN s_decode3 =>  -- jj
77              CASE IR(7 DOWNTO 5) IS
78                 WHEN "000" => state <= s_load;
79                 WHEN "001" => state <= s_store;
80                 WHEN "010" => state <= s_add;
81                 WHEN "011" => state <= s_sub;
82                 WHEN "100" => state <= s_input;
83                 WHEN "101" => state <= s_jz;
84                 WHEN "110" => state <= s_jpos;
85                 WHEN "111" => state <= s_halt;
86                 WHEN OTHERS => state <= s_start;
87              END CASE;
88           WHEN s_load =>          -- load A from memory
89              A <= memory_data;
90              state <= s_start;
91           WHEN s_store =>            -- store A to memory
92              MemWr <= '1';
93              state <= s_store2;
94           WHEN s_store2 =>        -- need an extra state to de-assert MemWr before changing the memory address
95              MemWr <= '0';
96              state <= s_start;
97           WHEN s_add =>           -- add
98              A <= A + memory_data;
99              state <= s_start;
100          WHEN s_sub =>           -- subtract
101             A <= A - memory_data;
102             state <= s_start;
103          WHEN s_input =>
104             A <= input;
105             IF (Enter = '0') THEN   -- wait for Enter key
106                state <= s_input;
107             ELSE
108                state <= s_start;
109             END IF;
110          WHEN s_jz =>
111             IF (A = 0) THEN         -- jump if A is 0
112                PC <= IR(4 DOWNTO 0);
113             END IF;
114             state <= s_start;
115          WHEN s_jpos =>
116             IF (A(7) = '0') THEN -- jump if MSB(A) is 0
117                PC <= IR(4 DOWNTO 0);
118             END IF;
119             state <= s_start;
120          WHEN s_halt =>
121             halt <= '1';
122             state <= s_halt;
123          WHEN OTHERS =>
124             state <= s_halt;
125          END CASE;
126       END IF;
127    END PROCESS;
128
129    output <= A;   -- send value of Accumulator to the output
130 END FSM;
```

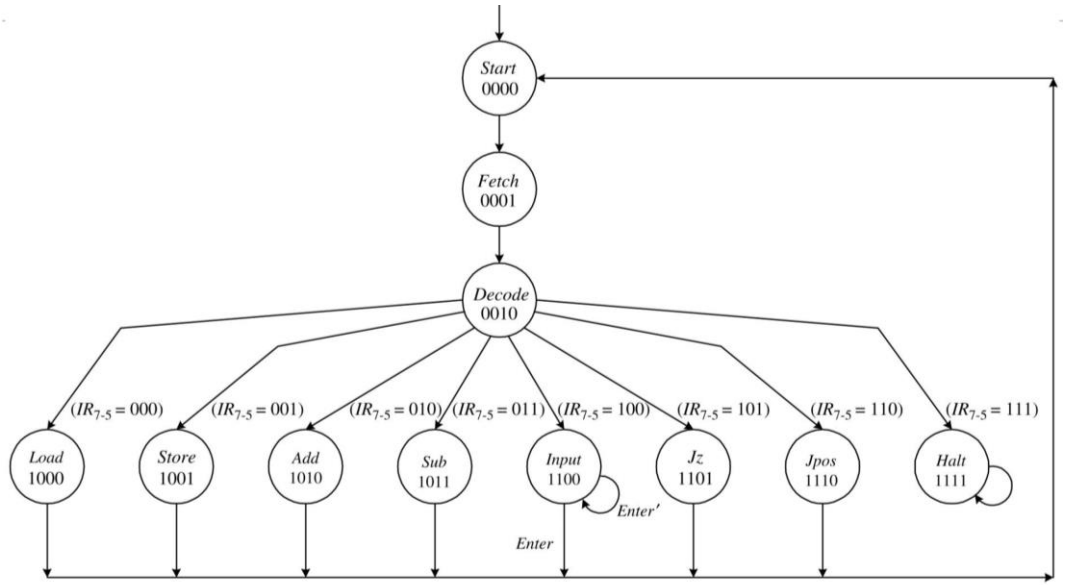
→ 해당 코드가 강의자료에 비해 달라진 부분은 더 많은 State가 존재해 해당 State에서 새롭게 연결이 이뤄진다는 것이다.

# 2. State Diagram

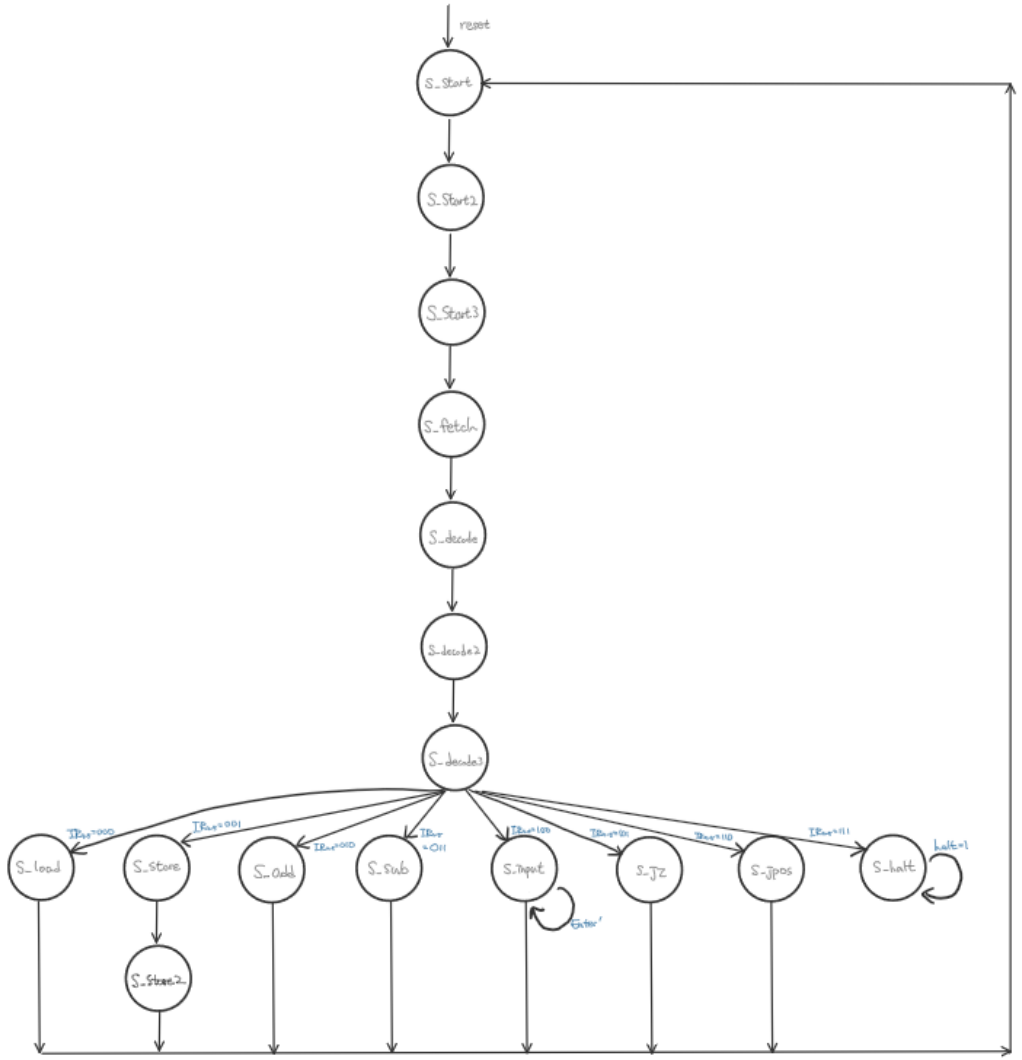



→ 이는 위에서 설명한 State를 State Diagram을 통해 나타낸 것이다.

→ 이는 기존 State Diagram과 다르게 s_start2, s_start3, s_decode2, s_decode3, s_store2 상태가 추가 되었다.



## 2. State Diagram

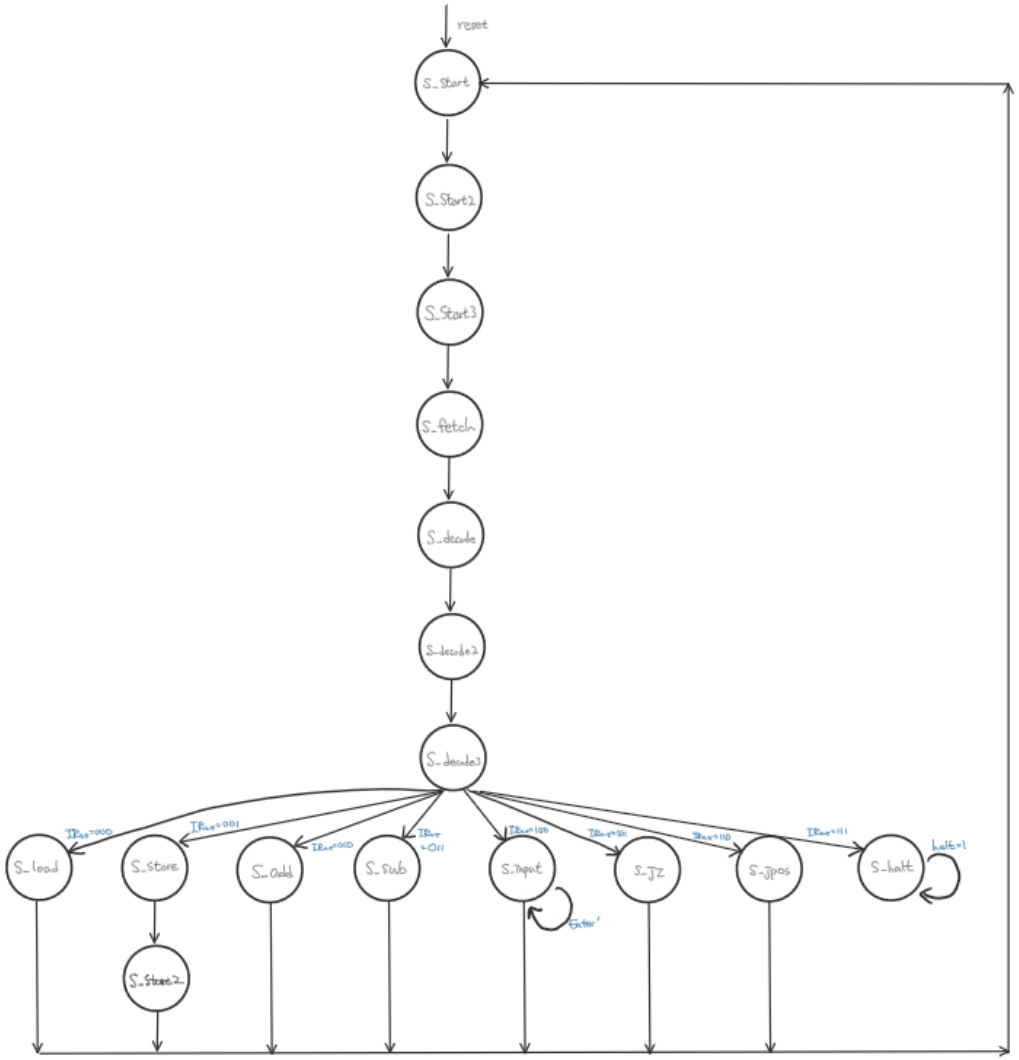



→ s_start상태에서 시작하여 s_start2, s_start3 상태로 가고, s_fetch로 가면서 IR에 memory data를 저장하고, PC를 1씩 증가해 다음 address로 이동한다.

→ s_decode 상태로 가서 memory_address에 IR 하위 5 bit(address)를 저장하고, s_decode2, s_decode3 상태로 이동한다.

→ s_decode3 상태에서는 각각 IR 상위 3bit에 따라 각각 맞는 Control Word(state)로 이동하게 된다.

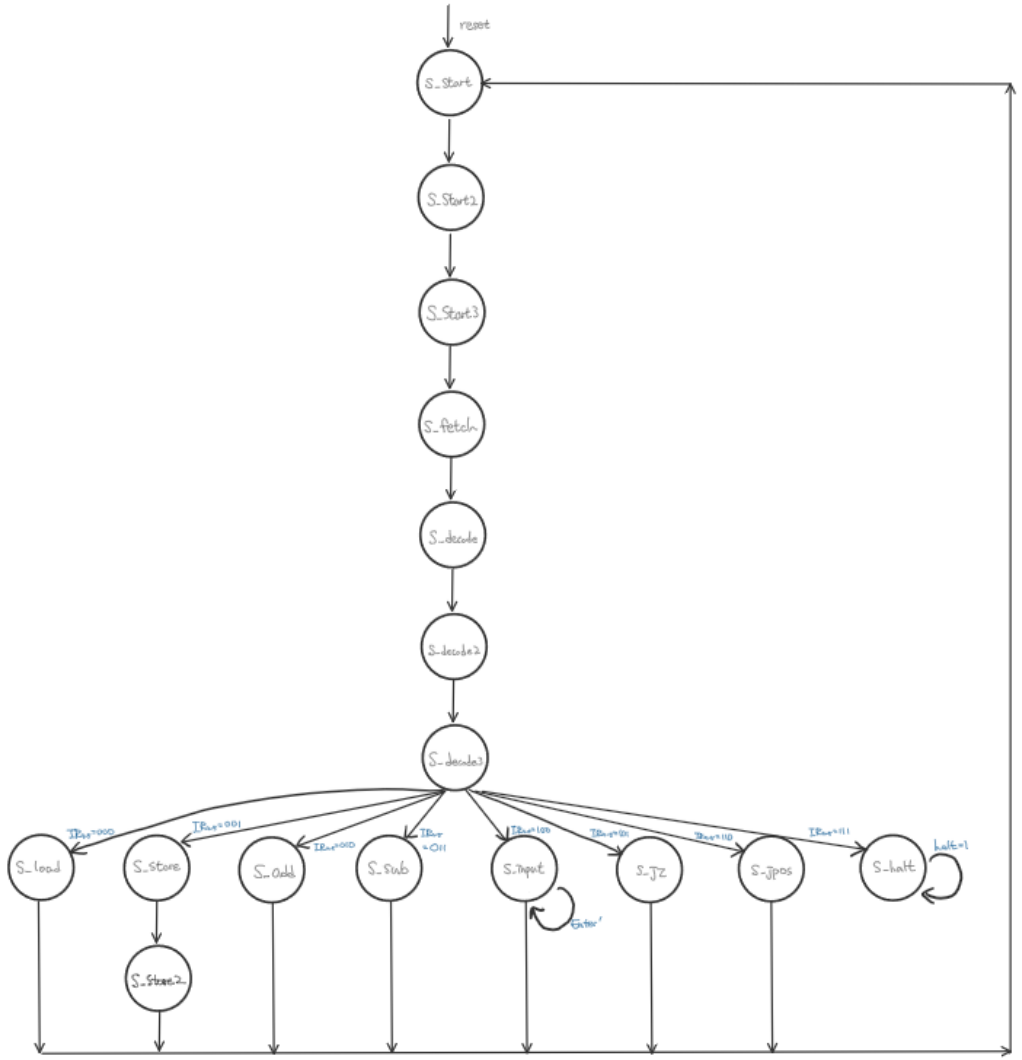
## 2. State Diagram




→ s_decode3에서 이동할 수 있는 state는 s_load, s_store, s_add, s_sub, s_input, s_jz, s_jpos, s_halt, 나머진 경우에는 s_start로 이동한다.

→ s_store의 경우에는 s_store2로 가서 MemWr을 다시 0으로 저장하는 state를 거친 다음 s_start로 가고, s_halt를 제외하고 모든 state는 다시 s_start로 되돌아간다.

## 2. State Diagram

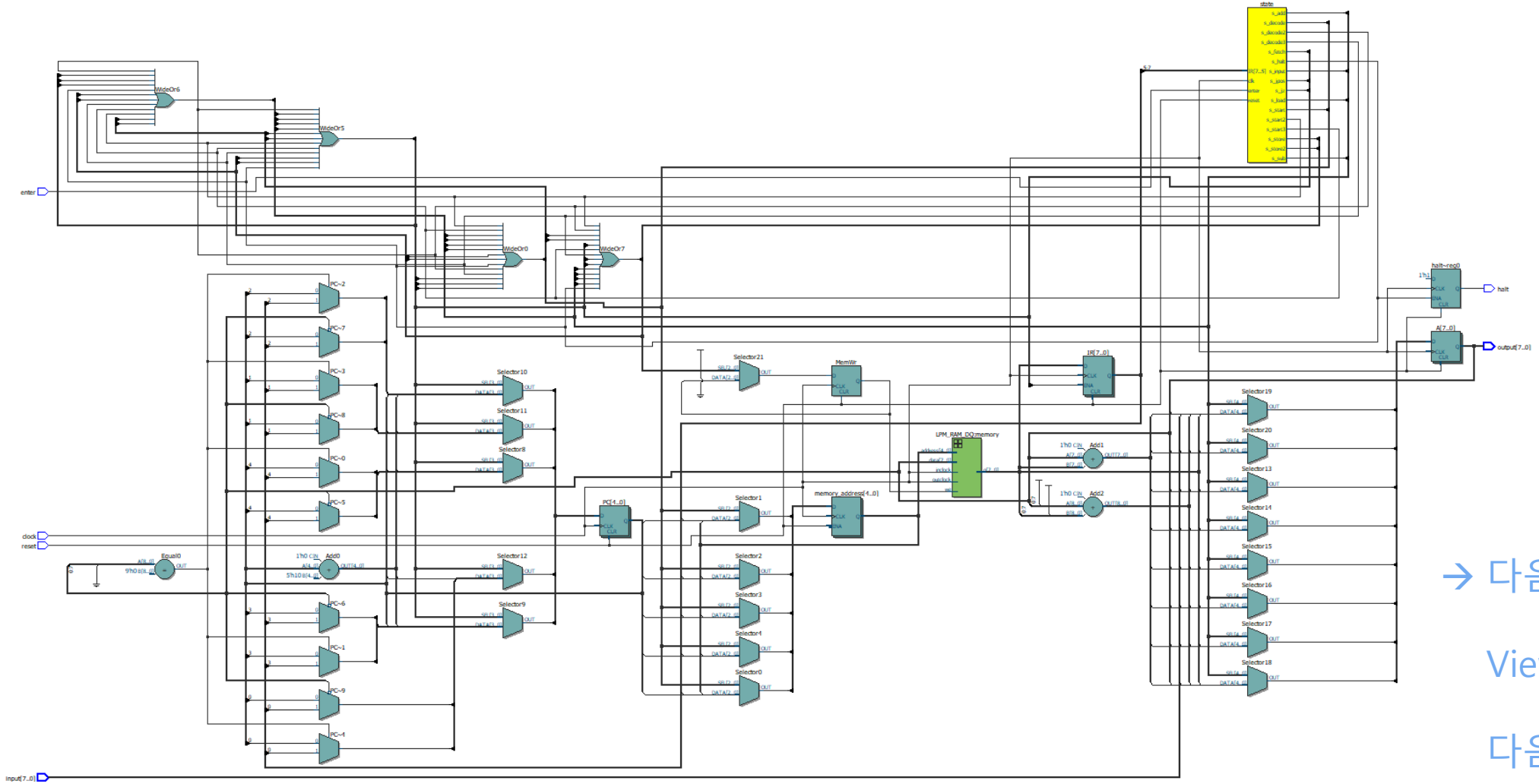



→ s_input의 경우에는 Enter 값이 0인 경우에는 s_input에 계속 머무르고, Enter 값이 1이면 s_start 상태로 되돌아간다.

→ s_halt는 프로그램 종료를 알리며, 계속해서 s_halt에 머무른다.

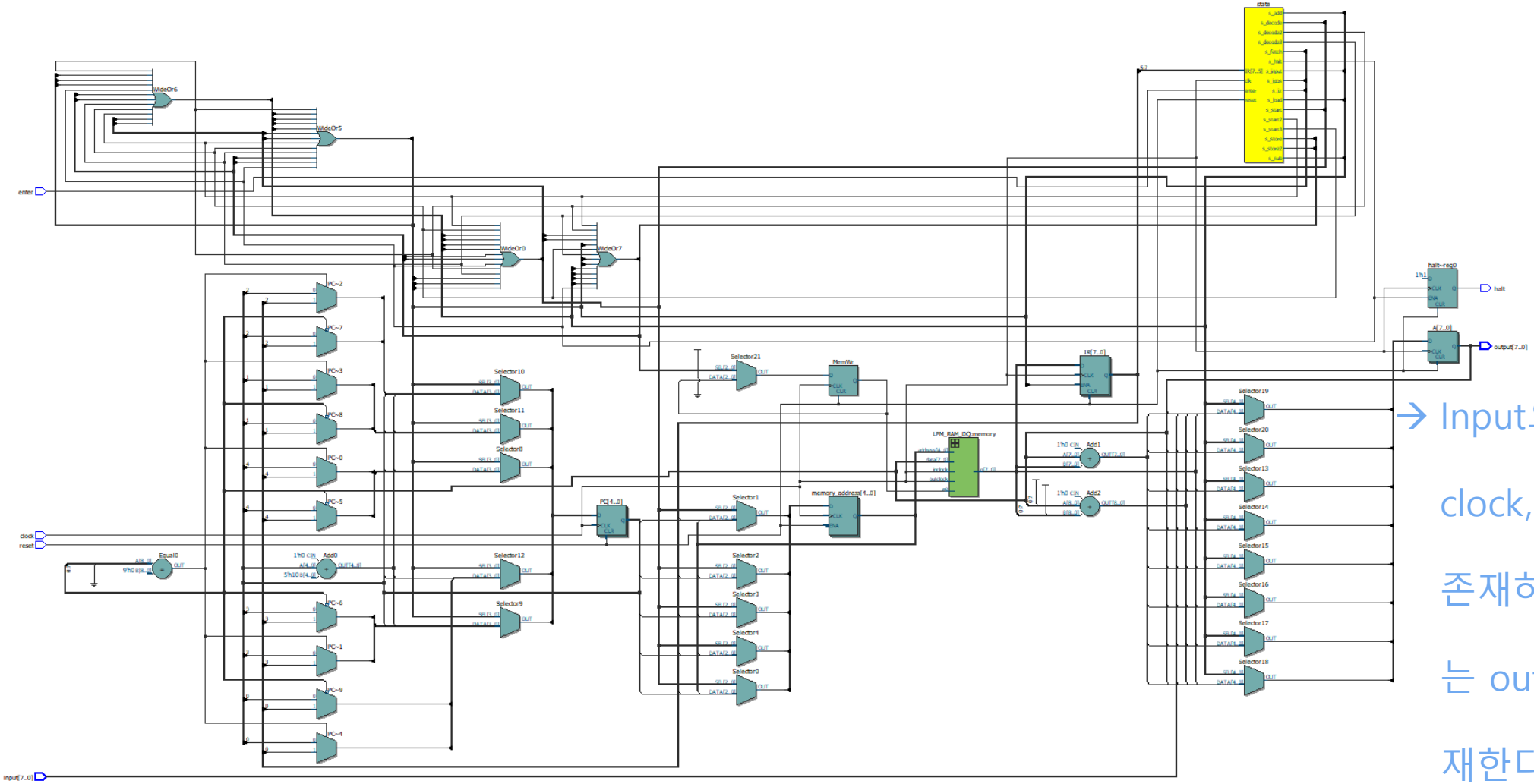
→ 따라서 다음 그림과 같이 State Diagram을 나타낼 수 있다.

# 3. RTL Viewer




→ 다음 Code를 RTL Viewer로 나타내면 다음과 같다.

# 3. RTL Viewer




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

# 4. Simulation – GCD Simulation

```
-- GCD
-- Program to calculate the GCD of two numbers
    00000 :  10000000;   -- input A
    00001 :  00111110;   -- store A,x
    00010 :  10000000;   -- input A
    00011 :  00111111;   -- store A,y

    00100 :  00011110;   -- loop: load A,x -- x = y?
    00101 :  01111111;   -- sub A,y
    00110 :  10110000;   -- jz out       -- x=y
    00111 :  11001100;   -- jp xgty      -- x>y

    01000 :  00011111;   -- load A,y     -- y>x
    01001 :  01111110;   -- sub A,x      -- y-x
    01010 :  00111111;   -- store A,y
    01011 :  11000100;   -- jp loop

    01100 :  00011110;   -- xgty: load A,x -- x>y
    01101 :  01111111;   -- sub A,y        -- x-y
    01110 :  00111110;   -- store A,x
    01111 :  11000100;   -- jp loop

    10000 :  00011110;   -- load A,x
    10001 :  11111111;   -- halt

    11110 :  00000000;   -- x
    11111 :  00000000;   -- y
```

→ 먼저 GCD의 assembly code & the binary code를 보면,

memory는 x, y 2개를 가진다. 외부 input 2개를 받아서 각각 x, y에 저장한다.

→ 다음으로 x에 저장된 값을 load하고, x에서 y를 뺀다. 이때, x와 y가 같은 지 확인하고, x가 y보다 큰지 판단하여 x가 더 크면 주어진 address(01100)로 이동한다. 해당 address(01100)는 x를 load하여 x-y를 진행하고, 이를 A에 저장해 x-y가 0이 될 때까지 이를 반복하는 loop이다.

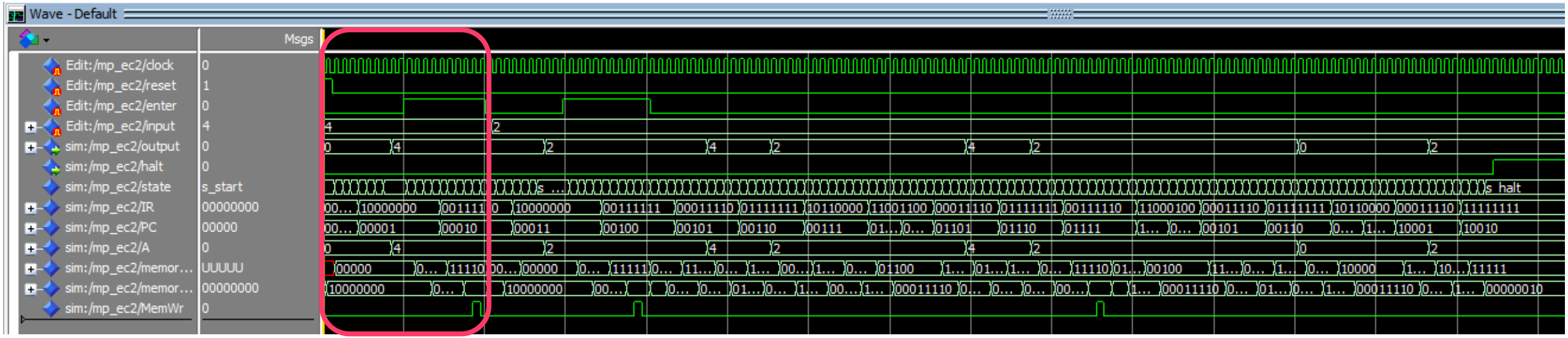
# 4. Simulation – GCD Simulation

```
-- GCD
-- Program to calculate the GCD of two numbers
    00000 :  10000000;   -- input A
    00001 :  00111110;   -- store A,x
    00010 :  10000000;   -- input A
    00011 :  00111111;   -- store A,y

    00100 :  00011110;   -- loop: load A,x -- x = y?
    00101 :  01111111;   -- sub A,y
    00110 :  10110000;   -- jz out       -- x=y
    00111 :  11001100;   -- jp xgty      -- x>y

    01000 :  00011111;   -- load A,y     -- y>x
    01001 :  01111110;   -- sub A,x      -- y-x
    01010 :  00111111;   -- store A,y
    01011 :  11000100;   -- jp loop

    01100 :  00011110;   -- xgty: load A,x -- x>y
    01101 :  01111111;   -- sub A,y        -- x-y
    01110 :  00111110;   -- store A,x
    01111 :  11000100;   -- jp loop

    10000 :  00011110;   -- load A,x
    10001 :  11111111;   -- halt

    11110 :  00000000;   -- x
    11111 :  00000000;   -- y
```

→ y가 값이 더 큰 경우에는 y를 load한 다음 y-x를 하고, 이를 A에 저장해 y-x가 0이 될 때까지 이를 반복한다.

→ 모든 것이 종료되면 GCD된 결과를 출력하고, halt로 이동하여 프로그램이 종료되었음을 알린다.

# 4. Simulation – GCD Simulation

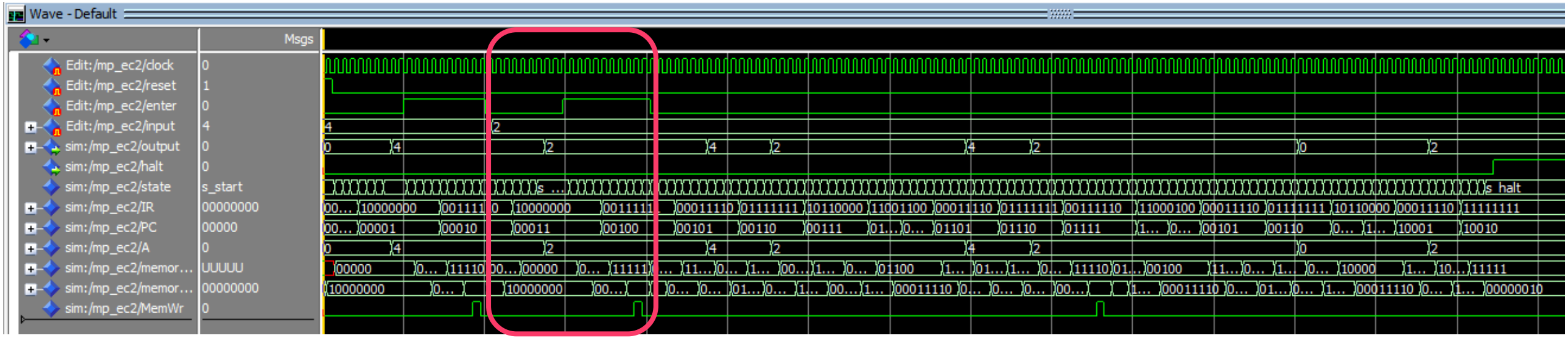



→ 다음은 GCD simulation을 진행한 결과이다.

→ 먼저 외부 input을 받아, 해당 input을 x에 저장한다. 이때, 저장을 했다는 의미에서 MemWr이 1이 되었다.

→ 또한, 해당 과정에서 output에 4가 나타나는 부분의 state는 s_input이고, 이때 Enter가 1이므로 다음 State로 진행될 수 있다. 그리고 s_input 이전까지는 s_start ~ s_decode3까지 순차적으로 진행되었다.

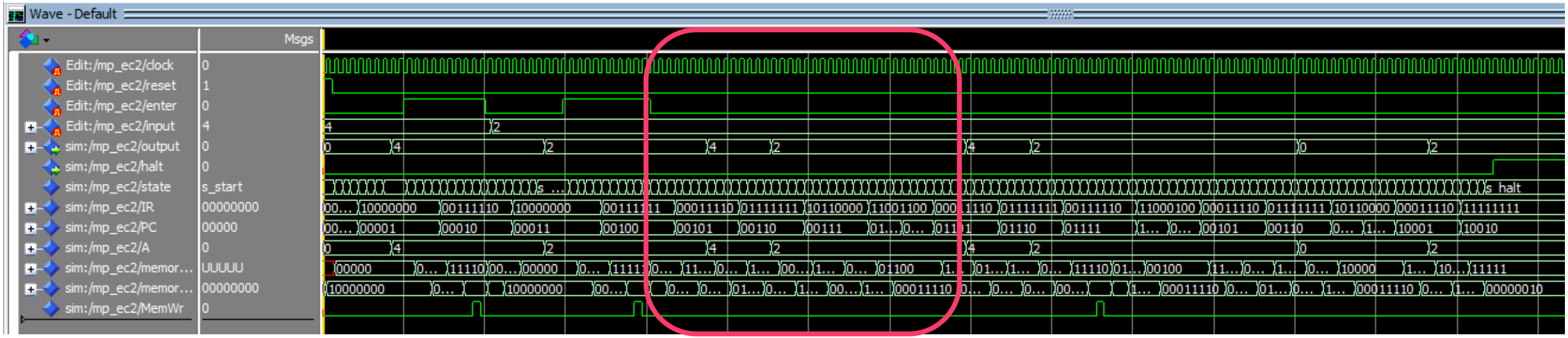
# 4. Simulation – GCD Simulation




→ 다음으로 외부 input을 받아, 해당 input을 y에 저장한다. 이때, 저장을 했다는 의미에서 MemWr이 1이 되었다.

→ 또한, 해당 과정에서 output에 2가 나타나는 부분의 state는 s_input이고, 이때 Enter가 1이므로 다음 State로 진행될 수 있다. 그리고 s_input 이전까지는 s_start ~ s_decode3까지 순차적으로 진행되었다.

# 4. Simulation – GCD Simulation



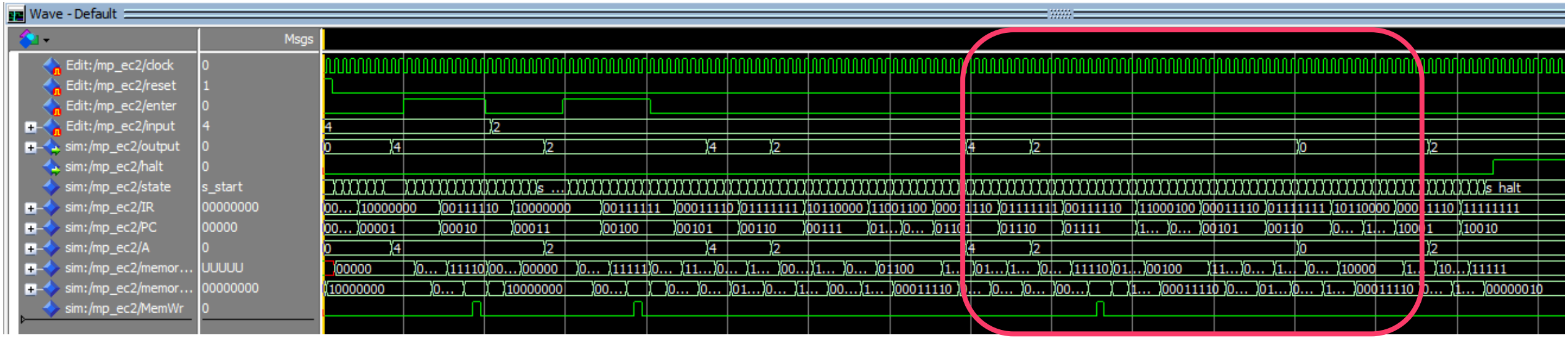

→ 다음으로 x와 y 중 어떤 것이 큰 지 비교하기 위해 x를 load하고, x에서 y를 뺐다.

→ 이를 통해 x값은 4이고, x-y는 2여서 x 값이 y 값보다 더 크다는 것을 알 수 있다.

→ 따라서 이는 x가 더 큰 경우로 다음 simulation이 진행되게 된다.

→ 해당 과정들 사이 State는 s_start ~ s_decode3가 반복해서 진행되게 된다.

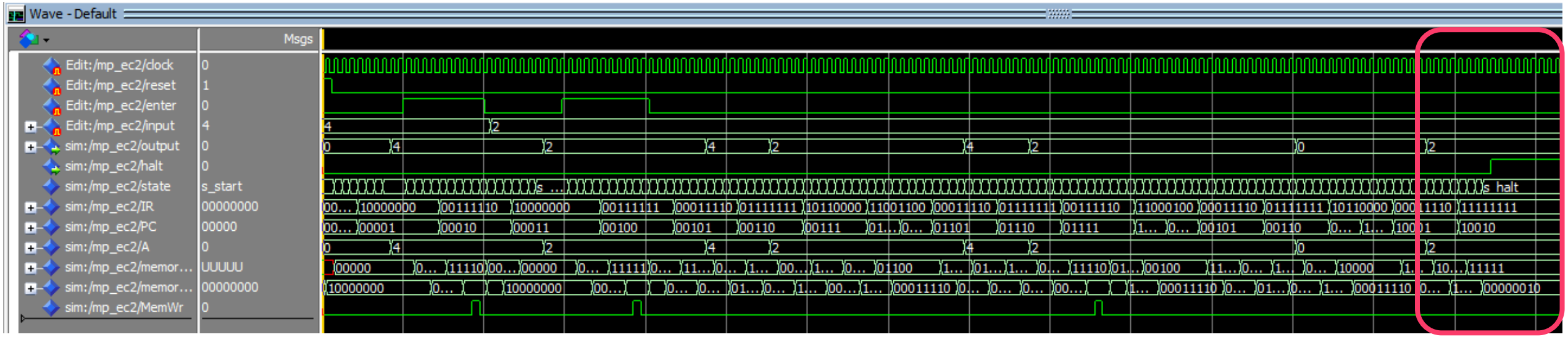
# 4. Simulation – GCD Simulation




→ x값이 더 크므로 x를 load한 다음 x-y를 진행한다. 그리고 해당 값이 0이 될 때까지 x-y를 진행한다. 해당 Simulation은 4에서 2를 빼는 것이므로 2번만 해당 과정을 진행하면 된다.

→ 2번 진행하면 0이 나오는 것을 다음을 통해 확인할 수 있다.

# 4. Simulation – GCD Simulation




→ 값이 0이 나와서, 이를 통해 계산한 GCD 결과인 2를 load하여 출력하였다.

→ 그리고 GCD 결과가 출력됐으므로 state가 s_halt로 가면서 halt가 1이 되고, 해당 프로그램은 종료하게 된다.

→ 이 Simulation을 통해 해당 Code가 올바르게 돌아간 것을 확인할 수 있다.

# 4. Simulation – Sum Sinulation

```
-- Sum
-- Program to sum N downto 1
   00000 :  00011101;   -- load A,one  -- zero sum by doing 1-1
   00001 :  01111101;   -- sub A,one
   00010 :  00111110;   -- store A,sum

   00011 :  10000000;   -- input A
   00100 :  00111111;   -- store A,n

   00101 :  00011111;   -- loop: load A,n -- n + sum
   00110 :  01011110;   -- add A,sum
   00111 :  00111110;   -- store A,sum

   01000 :  00011111;   -- load A,n       -- decrement A
   01001 :  01111101;   -- sub A,one
   01010 :  00111111;   -- store A,n

   01011 :  10101101;   -- jz out
   01100 :  11000101;   -- jp loop
   01101 :  00011110;   -- out: load A,sum
   01110 :  11111111;   -- halt

   11101 :  00000001;   -- one
   11110 :  00000000;   -- sum
   11111 :  00000000;   -- n
```

→ 먼저 Sum의 assembly code & the binary code를 보면, memory는 one, sum, n 3개를 가진다. 외부 input 1개를 받아서 이를 n에 저장한다.

→ 다음으로 one에 있는 1값을 load해서 여기서 1을 빼고, 이를 sum에 저장하면서 sum을 0으로 초기화한다.

→ Input A를 받아서 이를 n에 저장한다.

# 4. Simulation – Sum Sinulation

```
-- Sum
-- Program to sum N downto 1
   00000 :  00011101;   -- load A,one  -- zero sum by doing 1-1
   00001 :  01111101;   -- sub A,one
   00010 :  00111110;   -- store A,sum

   00011 :  10000000;   -- input A
   00100 :  00111111;   -- store A,n

   00101 :  00011111;   -- loop: load A,n -- n + sum
   00110 :  01011110;   -- add A,sum
   00111 :  00111110;   -- store A,sum

   01000 :  00011111;   -- load A,n       -- decrement A
   01001 :  01111101;   -- sub A,one
   01010 :  00111111;   -- store A,n

   01011 :  10101101;   -- jz out
   01100 :  11000101;   -- jp loop
   01101 :  00011110;   -- out: load A,sum
   01110 :  11111111;   -- halt

   11101 :  00000001;   -- one
   11110 :  00000000;   -- sum
   11111 :  00000000;   -- n
```

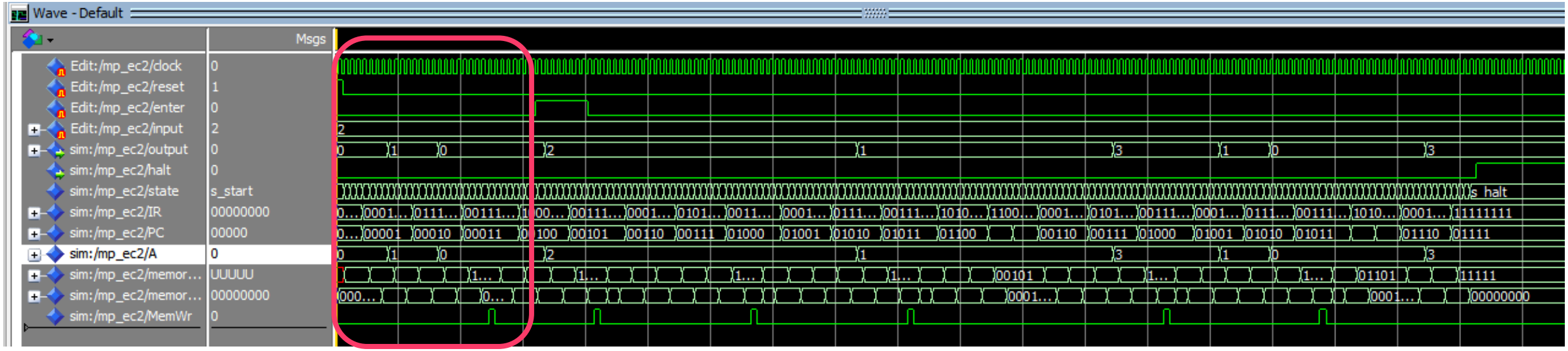
→ n에 있는 값을 load 해서, 여기에 sum에 있는 값을 더해 sum에 저장한다.

→ 다음으로 n에 있는 값에서 1을 빼서 다시 n에 저장한다.

→ 만약 n에 있는 값이 0이면, sum에 있는 값을 load 한 후 halt를 통해 프로그램을 종료한다.

→ 만약 0이 아니고 0보다 크면, 00101 address로 이동하여 해당 과정을 n이 0이 될 때까지 반복해준다.

# 4. Simulation – Sum Sinulation

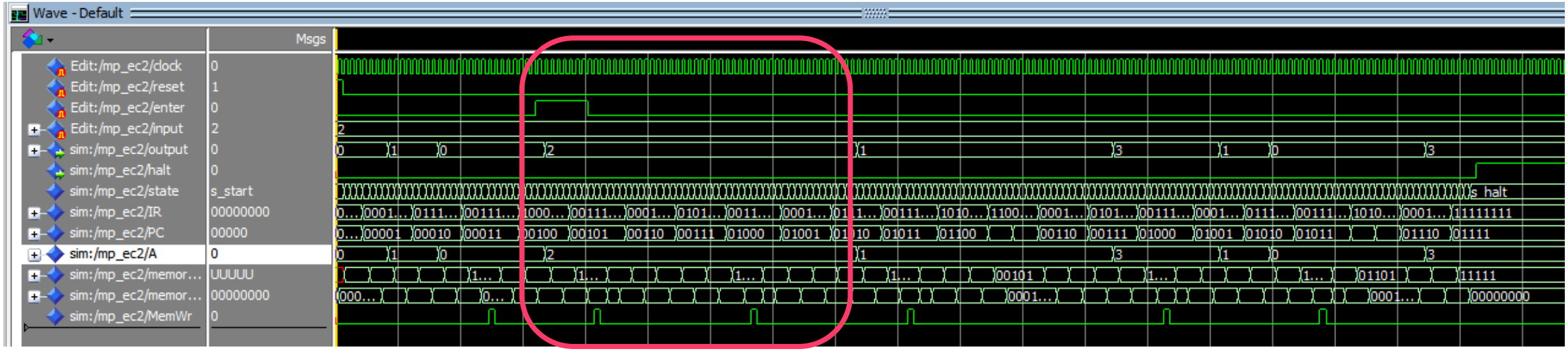



→ 다음은 Sum simulation을 진행한 결과이다.

→ 먼저 one에 있는 1값을 load해서 여기서 1을 빼고, 이를 sum에 저장하면서 sum을 0으로 초기화하는 과정을 거친다. 해당 과정들 사이 State는 s_start ~ s_decode3가 반복해서 진행되게 된다.

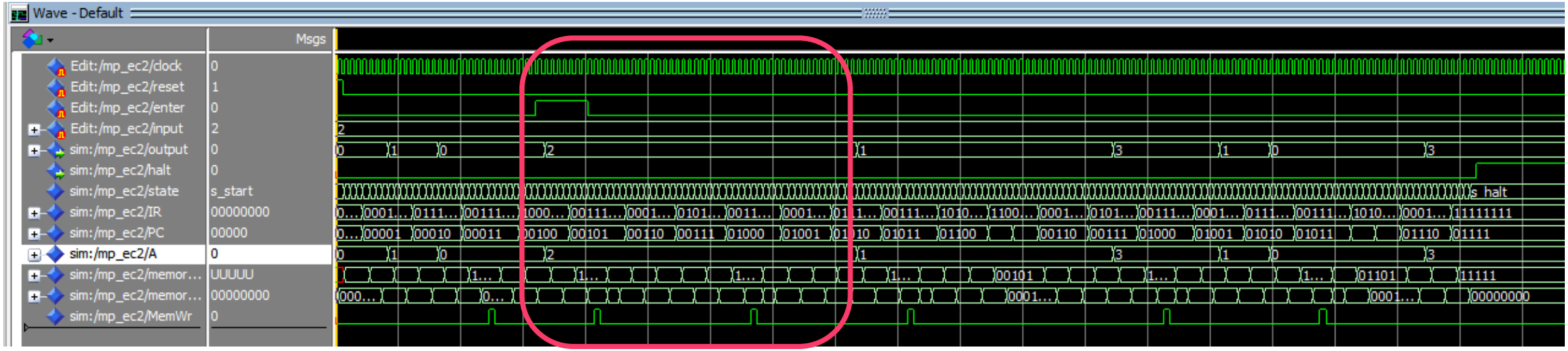
→ 이때, 저장을 했다는 의미에서 MemWr이 1이 되었다.

# 4. Simulation – Sum Sinulation




→ 다음으로 외부 input을 받아, 해당 input을 n에 저장한다. 또한, 해당 과정에서 output에 2가 나타나는 부분의 state는 s_input이고, 이때 Enter가 1이므로 다음 State로 진행될 수 있다. 그리고 s_input 이전까지는 s_start ~ s_decode3까지 순차적으로 진행되었다.

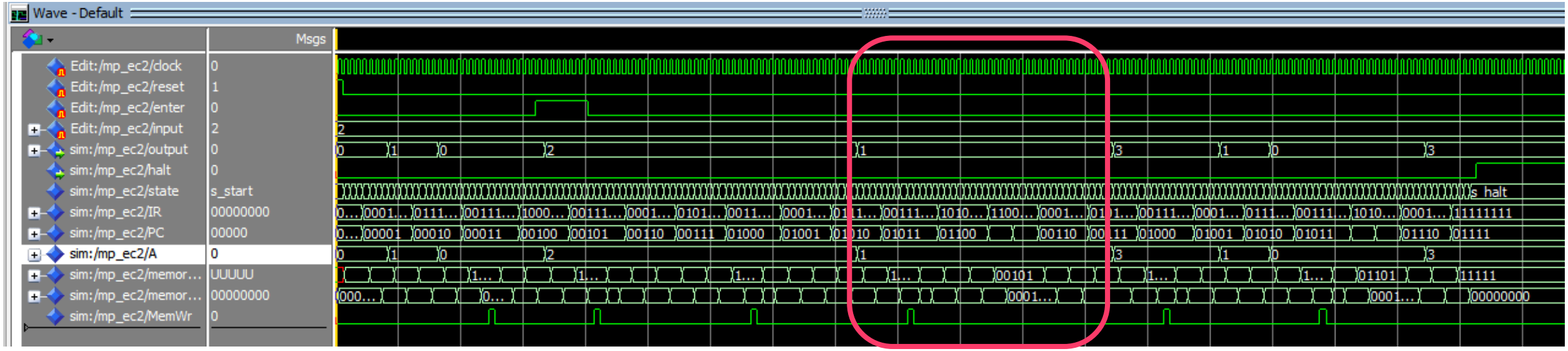
# 4. Simulation – Sum Sinulation




→ 2를 input으로 받고 저장한 후, 이를 load해서 sum에 있는 값과 더한 후 이를 sum에 저장한다. 이때, 저장을 했다는 의미에서 MemWr이 1이 되었다.

→ 해당 부분에서는 2를 n에 저장하는 부분과 sum에 더한 값을 저장하는 부분으로 저장하는 곳, 즉 MemWr이 1인 부분이 2번 존재한다.

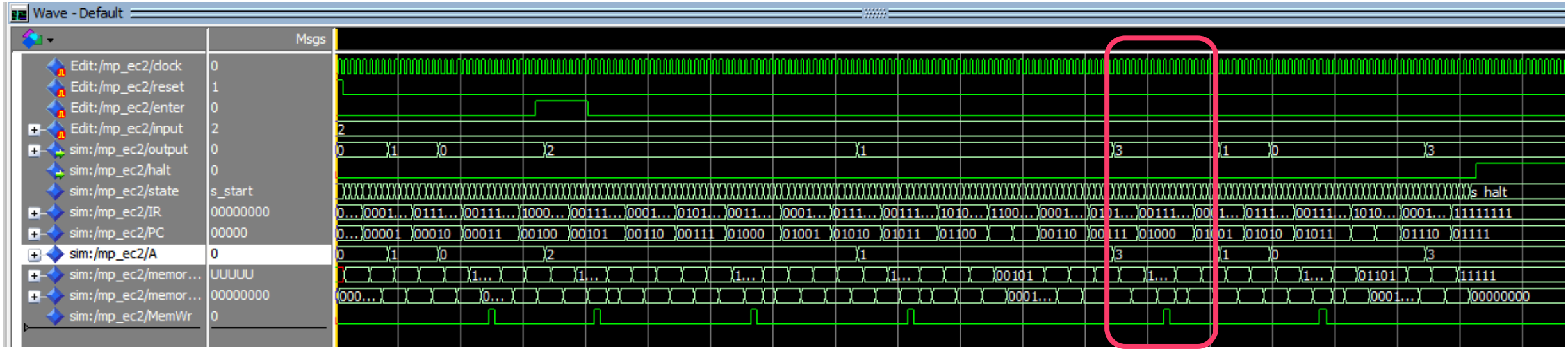
# 4. Simulation – Sum Sinulation




→ n에 있는 값에서 1을 빼서 이를 다시 n에 저장한다. 이때, 저장을 했다는 의미에서 MemWr이 1이 되었다.

→ 그리고 n에 있는 값이 0이 아니므로 값을 더해서 sum에 저장하고, n에서 1을 빼고 이를 다시 n에 저장하는 과정을 n이 0이 될 때까지 반복해준다.

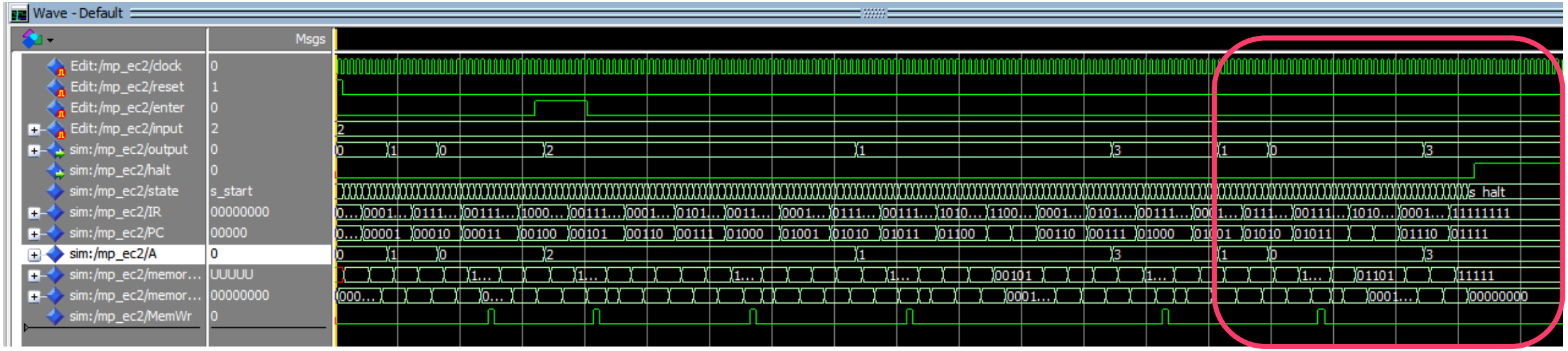
# 4. Simulation – Sum Sinulation




→ 해당 부분에서 앞서 n에 저장된 2에서 1을 뺀 값 1과 sum에 저장되어 있던 2를 더해서 3이라는 값이 나타나게 되었다. 해당 값을 sum에 저장한다.

→ 해당 과정들 사이 State는 s_start ~ s_decode3가 반복해서 진행되게 된다.

# 4. Simulation – Sum Sinulation




→ n에 있는 값을 load해서 다시 1을 빼고, 이를 n에 저장한다.

→ 이때 n에 있는 값이 0이 되게 되면서 sum에 있는 값인 3이 load되었고, 그 후 State가 s_halt가 되면서 halt의 값이 1이 되고, 해당 프로그램이 종료되게 된다.

→ 이 Simulation을 통해 해당 Code가 올바르게 돌아간 것을 확인할 수 있다.

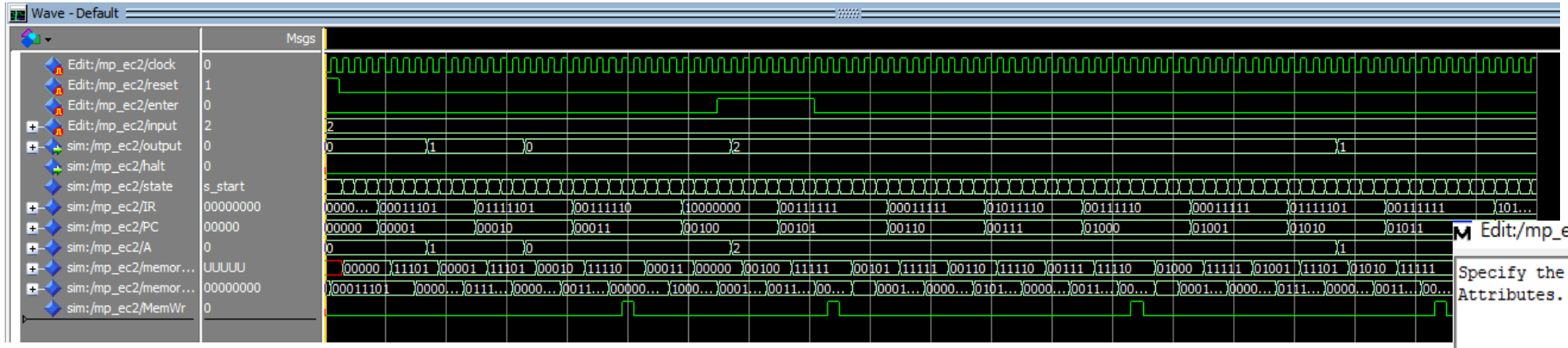
# Lab 10. General CPU design 2

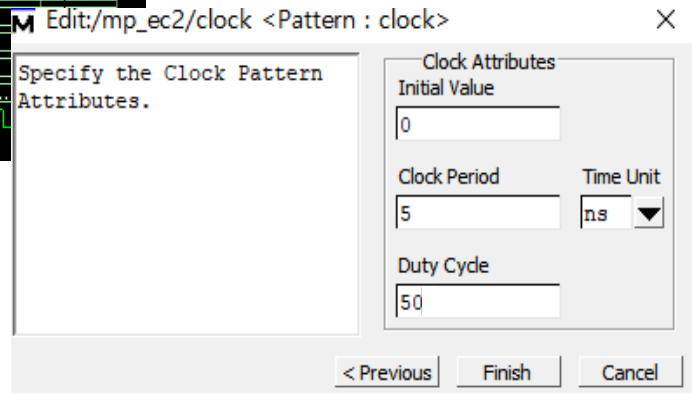
## 10-1 : Discussion

# Lab 10 General CPU design 2

- Discussion

→ 해당 Simulation을 진행하면서 여러 문제점이 있었는데, 가장 먼저 State의 개수가 많아 많은 Clock이 소요되면서 다음과 같이 쉽게 결과를 볼 수 없다는 문제점이 있었다. 이를 해결하기 위해 수를 작은 수로 결정하고, Clock Period를 다음과 같이 5로 줄이면서 해당 문제를 해결하였다.

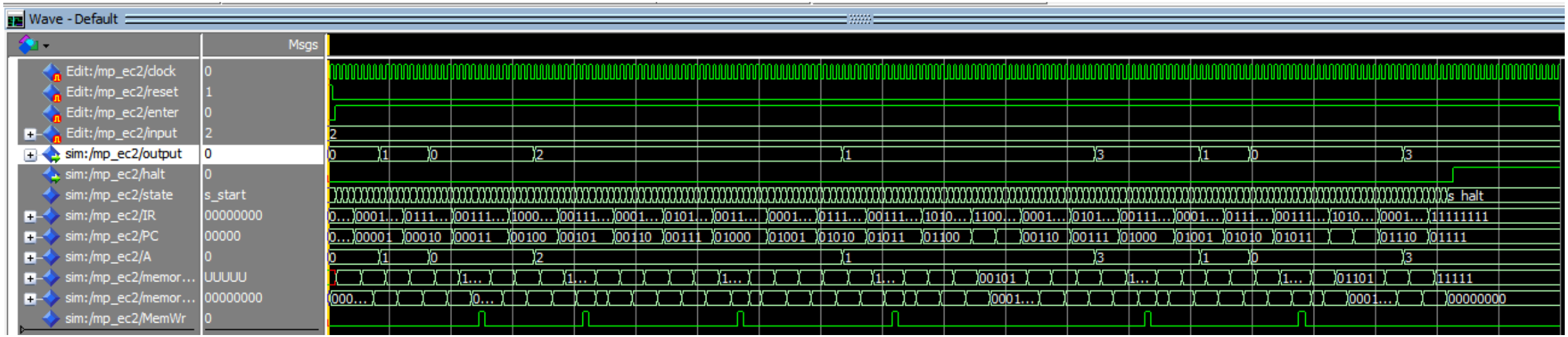


- Discussion

→ 또한, 많은 Clock이 있어 s_input state가 어디서 진행되는지 알기 어려웠다.

→ 따라서 처음에 simulatio을 할 때 Enter값을 모두 1로 해놓고 진행하여 어느 부분에만 Enter를 1로 해야 하는 지 알 수 있었다.



- Discussion

→ 이번 실습은 Lab 09번 실습과 마찬가지로 VHDL Code가 주어진 상태에서 내가 이해를 하고, Simulation을 진행하여 이를 분석하는 실습이라 비교적 어렵지 않았던 것 같다. 하지만, 이번 실습에서는 Lab 09와는 다르게 assembly code & the binary code를 이해하면서 실습을 진행해서 비교적 해당 구조에 대해 이해하기 좀 더 수월했던 것 같다.

→ 이번 실습에서 assembly code & the binary code의 구조를 이해하는데 시간이 많이 소요되었지만, 한 번 이해하면 다음 과정인 Simulation을 진행할 때는 보다 편하게 될 수 있었다.

# Lab 10. General CPU design 2

## 10-2 : Result

# 1. assembly code & the binary code

```
--Countdown
--countdown from N
   00000 :  10000000;    -- input A
   00001 :  00111111;    -- store A,n

   00010 :  00011111;    -- load A,n     -- decrement A
   00011 :  01111110;    -- sub A,one
   00100 :  00111111;    -- store A,n

   00101 :  10101101;    -- jz out
   00110 :  11000010;    -- jp loop
   01101 :  00011111;    -- out: load A,n
   01110 :  11111111;    -- halt

   11110 :  00000001;    -- one
   11111 :  00000000;    -- n
END;
```

→ Countdown 프로그램의 assembly code & the binary code를 보면, memory는 one, n 2개를 가진다. 외부 input 1개를 받아서 이를 n에 저장한다.

→ 외부 Input을 받아 이를 n에 저장한다.

→ n에 있는 값을 load해서, 이를 one에 저장된 1로 빼고, 다시 n에 뺀 값을 저장한다.

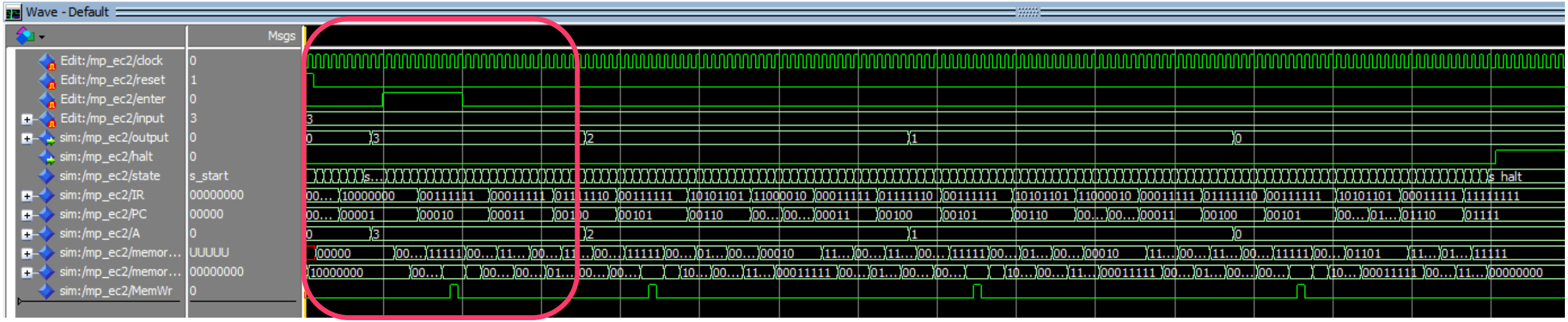
# 1. assembly code & the binary code

```
--Countdown
--countdown from N
   00000 :  10000000;    -- input A
   00001 :  00111111;    -- store A,n

   00010 :  00011111;    -- load A,n     -- decrement A
   00011 :  01111110;    -- sub A,one
   00100 :  00111111;    -- store A,n

   00101 :  10101101;    -- jz out
   00110 :  11000010;    -- jp loop
   01101 :  00011111;    -- out: load A,n
   01110 :  11111111;    -- halt

   11110 :  00000001;    -- one
   11111 :  00000000;    -- n
END;
```

→ 만약 n에 있는 값이 0이면, n에 있는 값을 load한 후 halt를 통해 프로그램을 종료한다.

→ 만약 0이 아니고 0보다 크면, 00010 address로 이동하여 해당 과정을 n이 0이 될 때까지 반복해준다.

# 2. Simulation

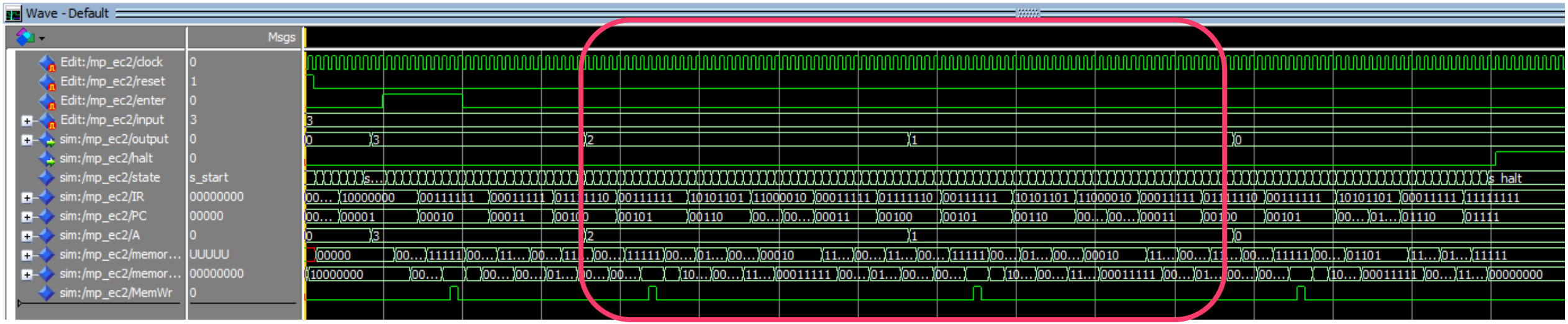



→ 다음은 Countdown simulation을 진행한 결과이다.

→ 먼저 외부 Input 3을 받아서 이를 n에 저장한다. 이때, 저장을 했다는 의미에서 MemWr이 1이 되었다.

→ 해당 과정들 사이 State는 s_start ~ s_decode3가 반복해서 진행되게 된다.

# 2. Simulation



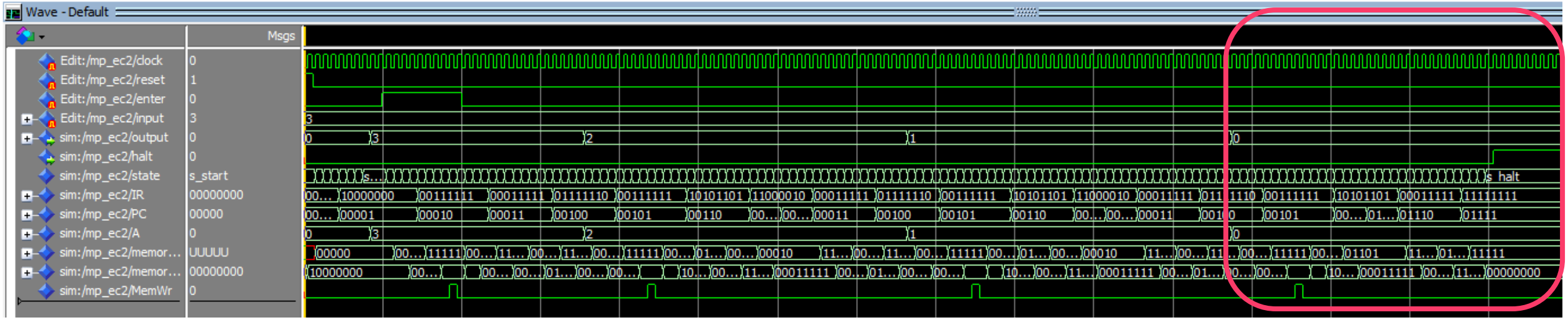

→ n에 있는 값을 load해서, 이를 one에 저장된 1로 빼고 해당 값을 n에 저장한다.

→ 이때, n에 있는 값이 0이 아니고 0보다 크므로 위의 과정을 n이 0이 될 때까지 반복해준다.

→ 해당 과정들 사이 State는 s_start ~ s_decode3가 반복해서 진행되게 되고, 중간중간 s_store, s_jz 등의 State가 존재한다.

Lab 10 General CPU design 2

# 2. Simulation




→ n에 있는 값을 load해서, 이를 one에 저장된 1로 빼고 해당 값을 n에 저장한다.

→ 그 후 State가 s_halt가 되면서 halt의 값이 1이 되고, 해당 프로그램이 종료되게 된다. 해당 과정들 사이 State는 s_start ~ s_decode3가 반복해서 진행되게 되고, 중간중간 s_store, s_jz 등의 State가 존재한다.

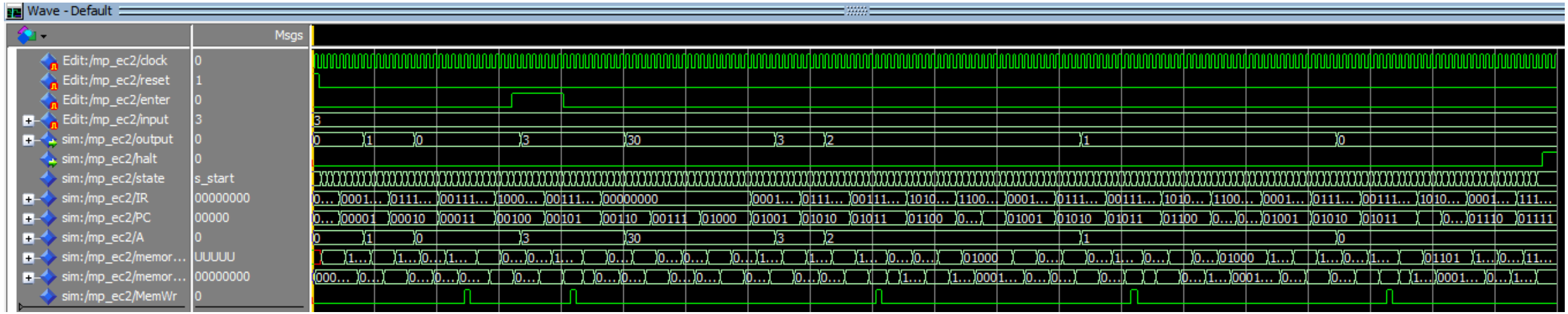
→ 이 Simulation을 통해 해당 Code가 올바르게 돌아간 것을 확인할 수 있다.

# Lab 10. General CPU design 2

## 10-2 : Discussion

- Discussion

→ 해당 Simulation을 진행하면서 한가지 문제점이 있었다. 해당 Simulation은 input 값을 받아 저장되고, 다시 n에 있는 값을 load해서 빼는 과정을 거쳐야 하는데, load가 될 때 3이 아닌 다른 값이 load가 되었다가 다시 3이 load되면서 해당 과정이 진행된다는 것이었다. 이를 해결하기 위해 assembly code & the binary code를 계속 수정해보았지만, 해결하는 것이 쉽지 않았다.



- Discussion

```
]-----------------------------------------------------
 --Countdown
--countdown from N
    00000 :  00011110;   -- load A,one  -- zero sum by doing 1-1
    00001 :  01111110;   -- sub A,one
    00010 :  00111111;   -- store A,n

    00011 :  10000000;   -- input A
    00100 :  00111111;   -- store A,n

    01000 :  00011111;   -- load A,n    -- decrement A
    01001 :  01111110;   -- sub A,one
    01010 :  00111111;   -- store A,n

    01011 :  10101101;   -- jz out
    01100 :  11001000;   -- jp loop
    01101 :  00011111;   -- out: load A,n
    01110 :  11111111;   -- halt

    11110 :  00000001;   -- one
    11111 :  00000000;   -- n

END;
```

→ 이는 해당 Simulation을 진행한 assembly code & the binary code이다. 이는 앞에 memory의 위치가 순차적으로 이루어지지 않았다. 따라서 해당 위치를 1씩 증가시켜 순차적으로 수정해주자 해당 문제가 해결되었다.

→ 해당 문제는 memory가 순차적으로 되지 않아 다른 부분에 있는 memory값을 불러온 것으로 예상된다.

- Discussion

→ 이번 실습은 assembly code & the binary code를 직접 구현해보는 과정을 거치면서 memory address 체계에 대해 확실하게 알 수 있었다. 이는 앞에 오류를 고치는 과정에서 더욱 잘 알 수 있었다. 해당 code가 진행되기 위해서는 어떤 과정을 거쳐야 하는지 하나하나 써보면서 진행하는 과정에서 assembly code & the binary code에 대해 잘 알아갈 수 있었다.

→ 또한, 이 실습 역시 직접 KIT를 통해 실습을 진행하면, 어떤 결과가 나오게 될지도 궁금해졌다.