# Lab 11 : Simple CPU design

휴먼지능정보공학과
201810756 김대환

# Lab 11 : Instruction

- 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 |


2

# Lab 11 : Instruction

- 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

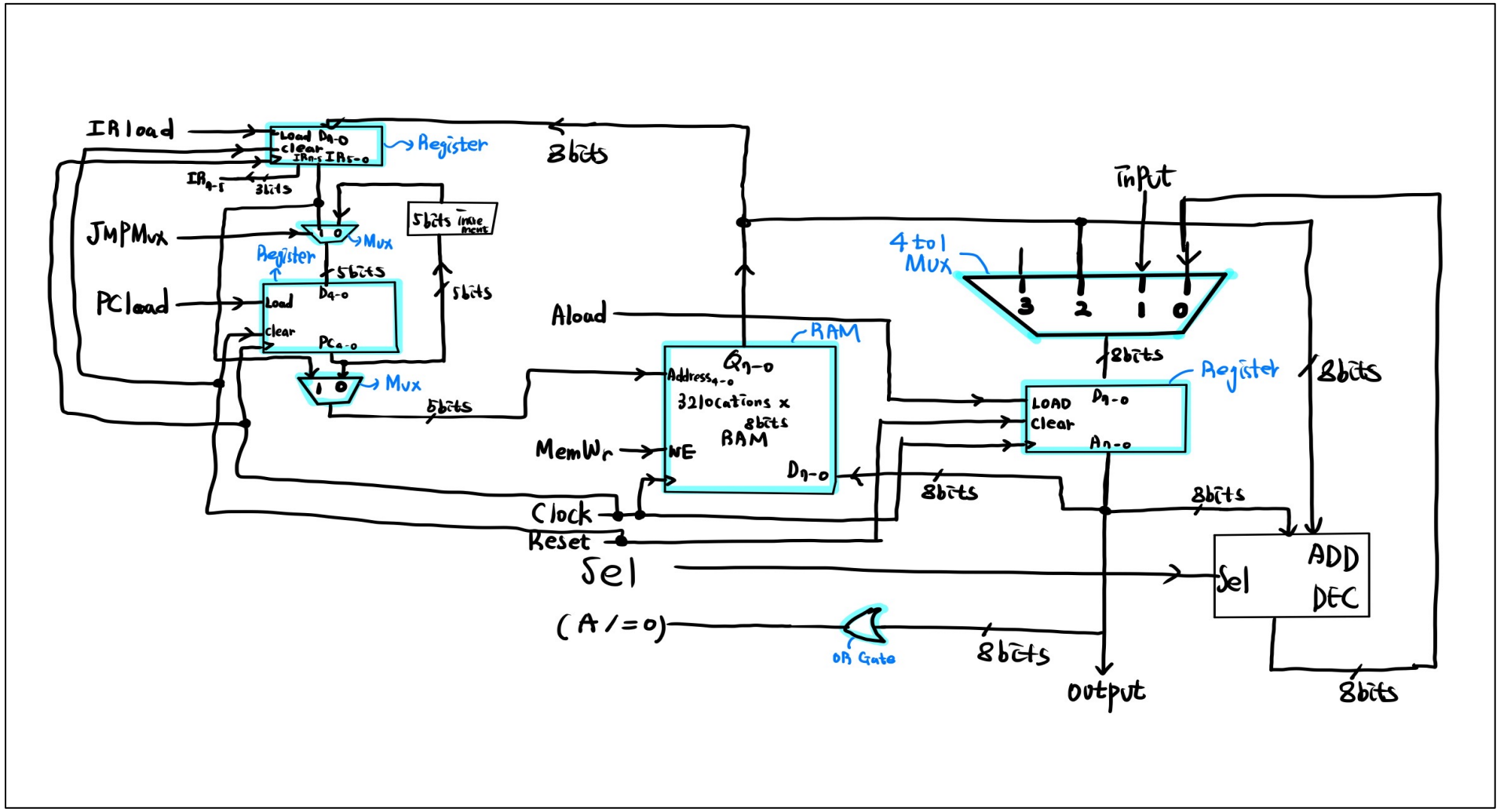
# Lab 11

| 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 |

- 해당 CPU는 7개의 instruction을 가진다.
  1. 'LOAD' ➔ 메모리의 주소 "aaaaa(IR LSB 하위 5bits)" 에 저장되어 있는 data를 불러옴
  2. 'STORE' ➔ data를 메모리의 주소 "aaaaa"에 저장함
  3. 'ADD' ➔ 메모리의 주소 "aaaaa"에 저장되어 있는 값과 A 값을 연산하여 저장
  4. 'DEC' ➔ A 값에 1을 빼는 연산
  5. 'IN' ➔ 외부 input 값을 A로
  6. 'OUT' ➔ A 값을 외부 outpu으로
  7. 'JMP' ➔ "aaaaa" 주소로 JUMP


4

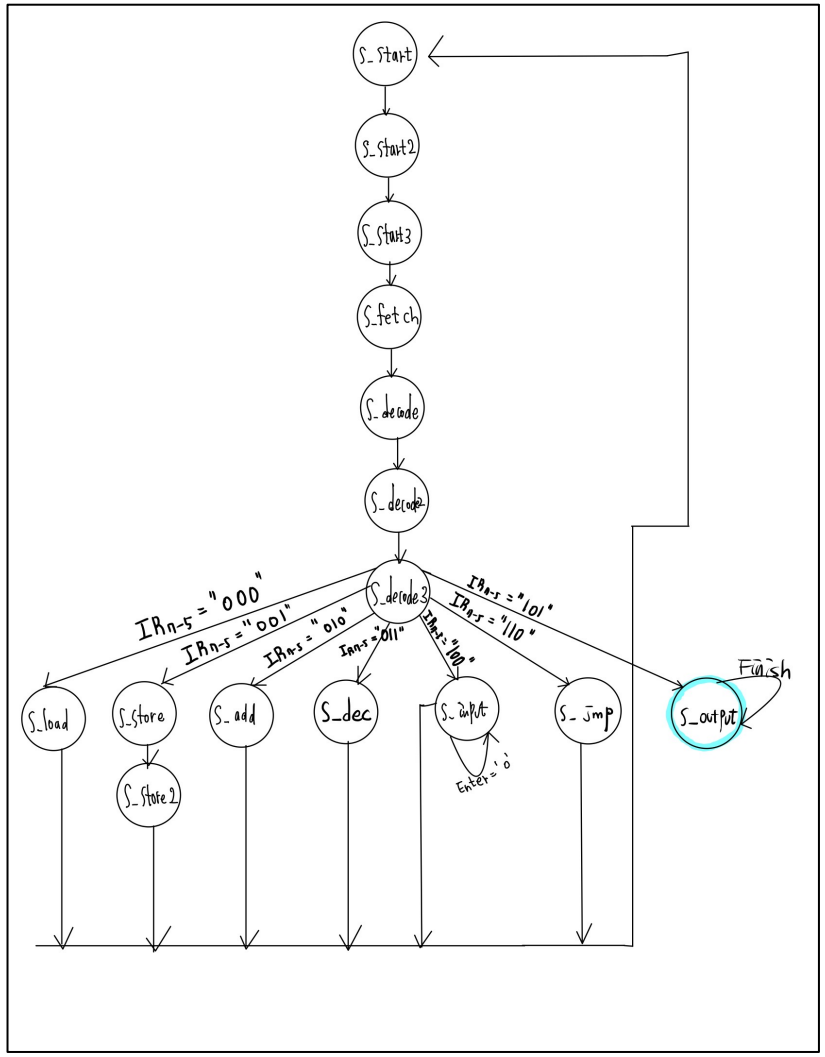
# Lab 11 : (1) Datapath




- 기존 Datapath 참고


5

# Lab 11 : (2) State Diagram




[State]

- s_start : PC register로부터 메모리 주소 공간 저장
- s_start2, s_start2 : clock timing을 위한 state
- s_fetch : 해당 메모리 주소에 저장되어 있는 데이터 IR 저장
  / PC increment 진행
- s_decode : JMP의 address 즉, IR의 하위 LSB 5bits 저장
- s_decode2 : clock timing을 위한 state
- s_decode3 : IR 상위 MSB 3bits에 따라 각각의 control unit state로 분기
- s_load : 해당 메모리 주소의 data를 A에 저장
- s_store, s_store2 : MemWr signal 조작을 통해 데이터 A RAM에 저장
- s_add : A와 메모리 공간으로부터 불러온 데이터 '+' 연산
- s_dec : A 1만큼 감소
- s_input : 외부 input을 A에 저장
- s_jmp : A 값 0인지 아닌지 확인. 아니라면 IR 하위 LSB 4bits를 통해 loop 문
- s_output : A값을 외부 output으로 출력(원하는 결과 이후, 따라서 s_output state 고정)


6

# Lab 11 : (3) VHDL code

- Lab10 – mp_ec2.vhdl 코드를 참고하여 작성하였다.
- 아래는 필요한 input, output과 내부 signal들을 선언해준 것이다.
  Input : clock, reset, input(external input)
  Ouptut : output(external output) _ 이번에는 Halt output이 없는 것을 알 수 있다.

```
entity final_lab is port(
   clock, reset : in std_logic;
   enter : in std_logic;
   input : in std_logic_vector(7 downto 0);
   output : out std_logic_vector(7 downto 0));
end final_lab;

architecture FSMD of final_lab 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_dec, s_input, s_output, s_jmp);
   signal state : state_type;
   signal IR : std_logic_vector(7 downto 0);
   signal PC : std_logic_vector(4 downto 0);
   signal A : std_logic_vector(7 downto 0); -- external input data
   signal memory_address : std_logic_vector(4 downto 0);
   signal memory_data : std_logic_vector(7 downto 0);
   signal MemWr : std_logic;
```

- 32개의 주소와 8bits 데이터를 저장할 수 있는 RAM을 사용하였다.

```
memory : lpm_ram_dq
   generic map(
      lpm_widthad => 5,
      lpm_outdata => "REGISTERED",
      lpm_indata => "REGISTERED",
      lpm_numwords => 32,
      lpm_address_control => "REGISTERED",
      lpm_file => "final_lab_ram.mif",
      lpm_width => 8)
   port map(
      data => A,
      address => memory_address,
      inclock => clock,
      outclock => clock,
      we => MemWr,
      q => memory_data);
```


7

# Lab 11 : (3) VHDL code

```
case state is
when s_start =>
   memory_address <= PC;
   state <= s_start2;
when s_start2 =>
   state <= s_start3;
when s_start3 =>
   state <= s_fetch;
when s_fetch =>
   IR <= memory_data;
   PC <= PC + 1;
   state <= s_decode;
when s_decode =>
   memory_address <= IR(4 downto 0);
   state <= s_decode2;
when s_decode2 =>
   state <= s_decode3;
when s_decode3 =>
   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 others => state <= s_start;
   end case;
when s_load =>
   A <= memory_data;
   state <= s_start;
when s_store =>
   MemWr <= '1';
   state <= s_store2;
when s_store2 =>
   MemWr <= '0';
   state <= s_start;
```

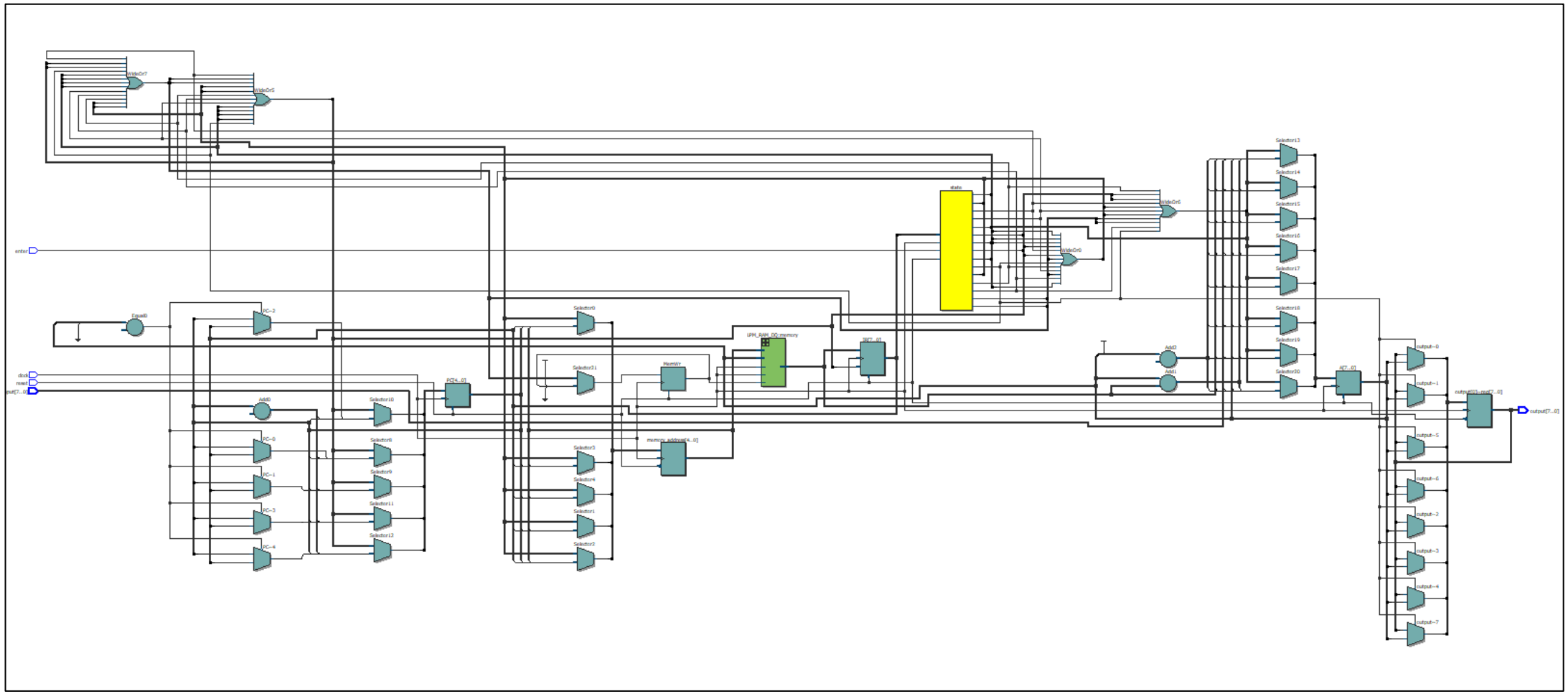
```
when s_add =>
   A <= A + memory_data;
   state <= s_start;
when s_dec =>
   A <= A - 1;
   state <= s_start;
when s_input =>
   A <= input;
   if (enter = '0') then
      state <= s_input;
   else
      state <= s_start;
   end if;
when s_output =>
   output <= A;
   state <= s_output;
when s_jmp =>
   if (A /= 0) then
      PC <= IR(4 downto 0);
   end if;
   state <= s_start;
when others =>
   state <= s_start;
```

- State 들을 다음과 같이 선언해주었고 여기서
  s_output state에 대해서 조금 설명하자면
  이번 실습 모델에서는 Halt state / 즉 , 프로그램의 종료를 알리는 state가 존재하지 않기 때문에
  s_output state에서 최종 결과를 외부 output으로 출력해주고
  계속해서 s_output state에 고정되도록 설정하였다.


8

# Lab 11 : (4) RTL view

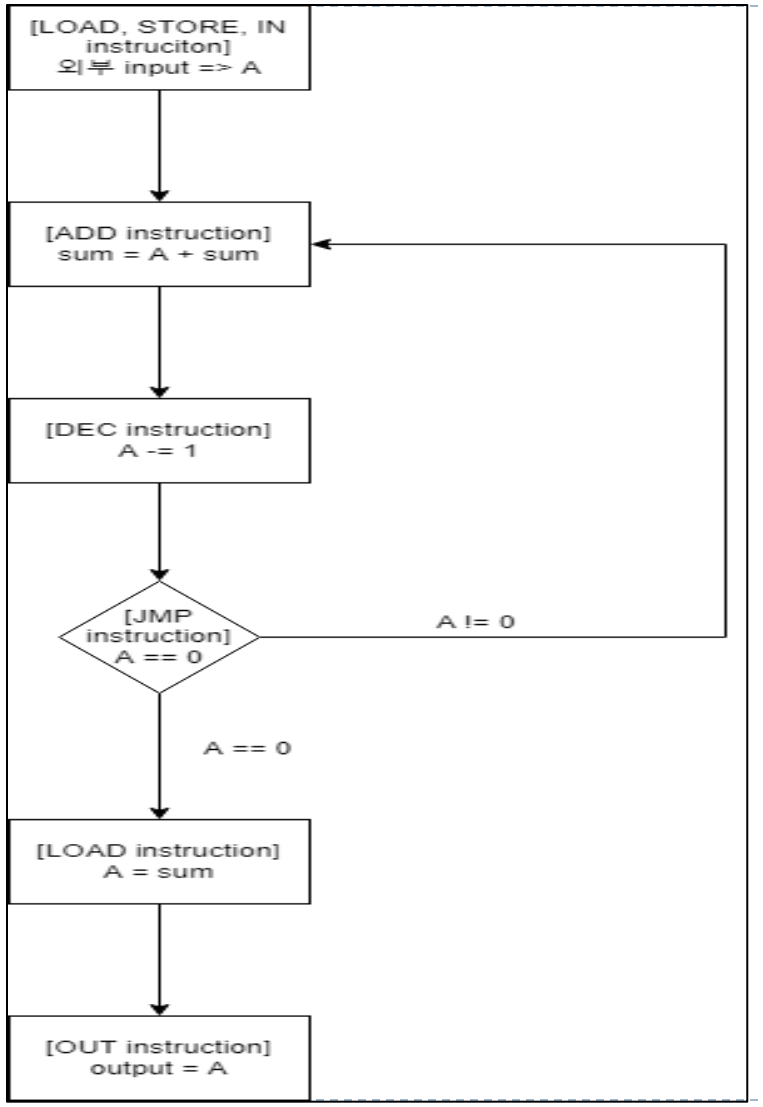
- 작성한 VHDL 코드를 통해 얻은 RTL view의 결과는 다음과 같다





9

# Lab 11 : (5) Test Algorithm




- 실습 모델을 테스트하기 위한 알고리즘으로 lab10에서 사용한 sum 알고리즘을 참고하였다.
- 이번 모델이 가져야할 7개의 instruction을 모두 한번씩 사용하고 확인하기 위해서는 sum 알고리즘이 최적화 되어 있다고 생각했기 때문이다.
- 하지만 기존 sum 알고리즘이 Halt signal을 갖고 있는데 반해 현재 모델은 프로그램의 종료를 알리는 Halt instruction 이 존재하지 않기 때문에 output instruction을 활용하여 이를 구현해보고자 한다.

- [알고리즘 설명]
  1. 외부 input A를 받아와서 메모리 공간에 store 한다.
  2. 이를 load 해온 뒤 sum 과 add 하여 sum에 저장한다.
  3. A는 dec한 뒤 A에 저장한다.
  4. A의 값이 0이 될 때 까지 (2)~ 반복한다.
  5. A가 0이라면, 저장되어 있는 sum 값을 A로 load 해온다.
  6. A를 output으로 출력해주고 프로그램의 종료를 알린다.


10

# Lab 11 : (6) Assembly & Binary Code

- 테스트 알고리즘 수행을 위한 Assembly & binary code는 아래와 같다.

```
CONTENT
        BEGIN
        [00000..11111]      :       00000000;       -- Initialize locations range 00-FF to 0000

        00000 : 10000000; -- IN A
        00001 : 00111111; -- STORE A, "11111" / store external input A to "11111"
        00010 : 00011111; -- LOOP : LOAD A, "11111" / load memory data in "11111" to A
        00011 : 01011110; -- ADD A, "11110" / add A and data in memory address "11110"
        00100 : 00111110; -- STORE A, "11110" / store add result in "11110"
        00101 : 00011111; -- LOAD A, "11111" / load memory data in "11111" to A
        00110 : 01111111; -- DEC A / decline A
        00111 : 00111111; -- STORE A, "11111" / store dec result in "11111"
        01000 : 11000010; -- JMP / jump to "00010" until A==0
        01001 : 00011110; -- LOAD A, "11110" / load data in memory address "11110" to A
        01010 : 10111110; -- OUT A / show result data A
        11110 : 00000000; -- memory for sum(initial value = 0) (=> sum)
        11111 : 00000000; -- memory for External input (=> n)
END;
```

- Loop 구문이다. A = 0이 될 때 까지 A를 dec하면서 sum과 더해간다. (Sum countdown from N)
- Output을 정확히 원할 때만 보기 위한 과정이다. A에는 이미 '0'이 저장되어 있기 때문에 Output으로 출력하기 전에 sum 데이터가 저장되어 있는 공간에서 load 해올 필요가 있다. 따라서 해당 data를 load 한 이후에 이를 output으로 출력한다.
- 기본 메모리 공간은 2개를 사용한다. Sum data를 저장할 공간과 input data를 저장하는 공간으로 나뉜다.

# Lab 11 : (7) Simulation(by Test Bench)

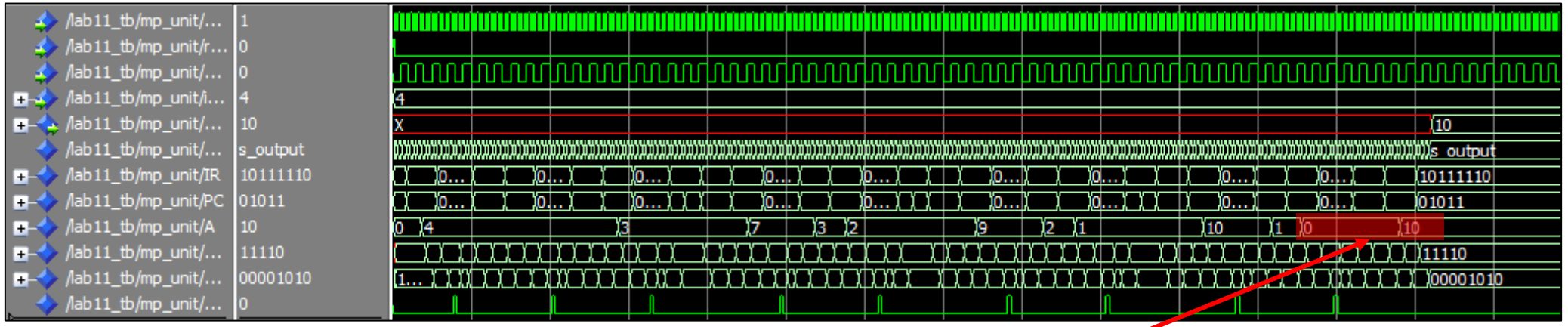
- Test Bench code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lab11_tb is
end lab11_tb;

architecture arch of lab11_tb is
   constant T : time:=50 ns;
   constant num_of_clocks : integer:= 100;
   signal i : integer :=0;
   signal clock : std_logic := '0';
   signal reset: STD_LOGIC;
   signal enter: STD_LOGIC;
   signal input: STD_LOGIC_VECTOR(7 DOWNTO 0);
   signal output: STD_LOGIC_VECTOR(7 DOWNTO 0);
   signal finished : std_logic := '0';
begin
   mp_unit : entity work.final_lab
      port map(clock=>clock, reset=>reset, enter=>enter, input=>input, output=>output);
   clock <= not clock after T/2 when finished /='1' else '0';
   reset<='1', '0' after T/2;
   -- for GCD : input <= "00001000", "00000100" after 400 ns;
   input <= "00000100"; -- input = '4'
   process
      begin
         enter<='0';
         wait for T*2;
         enter<='1';
         wait for T*2;
         if(i=num_of_clocks) then
            finished <= '1';
            wait;
         else
            i<=i+1;
         end if;
      end process;
end arch;
```

- Lab10 에서 사용한 test bench 코드를 일부만 수정한 뒤 그대로 사용하였다.


12

# Lab 11 : (7) Simulation Result




- Simulation 결과 확인을 통해 올바르게 원하는 모델이 작동하는 것을 확인할 수 있다.
- 외부 input으로 "00000100(4)" 를 넣어주었더니 s_output state에서 countdown sum 결과인 '10'을 출력하고 있다.

- A의 값을 확인해보더라도 반복적으로 dec 되어 최종적으로 '0' 값을 가진 후 JMP loop를 통과하여 그 다음 instruciton인 sum 값을 load 한 것을 확인할 수 있다.

```
CONTENT
        BEGIN
        [00000..11111]  :       00000000;       -- Initialize locations range 00-FF to 0000

        00000 : 10000000; -- IN A
        00001 : 00111111; -- STORE A, "11111" / store external input A to "11111"
        00010 : 00011111; -- LOOP : LOAD A, "11111" / load memory data in "11111" to A
        00011 : 01011110; -- ADD A, "11110" / add A and data in memory address "11110"
        00100 : 00111110; -- STORE A, "11110" / store add result in "11110"
        00101 : 00011111; -- LOAD A, "11111" / load memory data in "11111" to A
        00110 : 01111111; -- DEC A / decline A
        00111 : 00111111; -- STORE A, "11111" / store dec result in "11111"
        01000 : 11000010; -- JMP / jump to "00010" until A==0
        01001 : 00011110; -- LOAD A, "11110" / load data in memory address "11110" to A
        01010 : 10111110; -- OUT A / show result data A
        11110 : 00000000; -- memory for sum(initial value = 0) (=> sum)
        11111 : 00000000; -- memory for External input (=> n)
END;
```

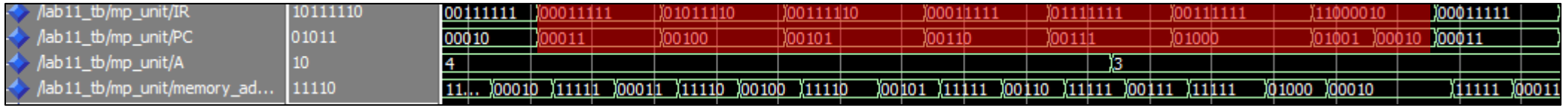
- 이전 실습과는 달리 일부러 내부 과정 속에서 발생하는 값들은 output으로 출력하지 않고 최종 결과만을 Output으로 가져왔다.
  ➔ EC2에서는 Output instruction이 따로 존재하지 않았기 때문에 내부 결과를 계속 출력한 것으로 생각하고 별도의 instruction을 둔 이유가 이와 같이 최종 결과만을 state를 통해서 출력하기 위함이라고 생각하였기 때문에 이렇게 구현하였다.


13

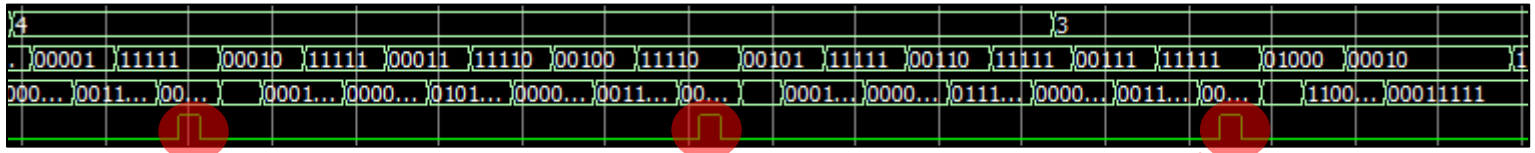
# Lab 11 : (7) Simulation Result



- 위처럼 state 들도 원하는 대로 작동하고 있는 것을 확인할 수 있다.

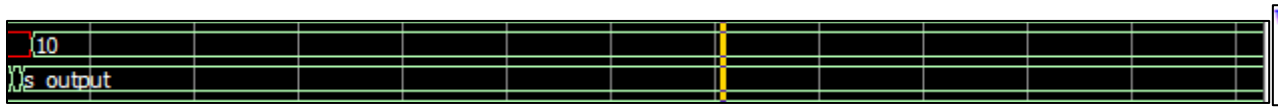



- 한번의 JMP loop cycle에 해당하는 wave다.
- A의 값이 '0' 이 될 때까지 해당 cycle을 반복하며 binary code 대로 올바르게 작동하는 것을 확인할 수 있다.




- MemWr = '1' 이 될 때 data를 저장하는데
  처음에는 input ='4' 를 저장하고 / sum과 add 연산을 하는데 sum이 '0'이므로 A 값이 변화하지 않은 채 sum에 저장되는 것을 확인할 수 있다.

  그 다음으로 A 값의 dec 연산 이후 해당 값을 A에 저장하는 것도 확인할 수 있다.




```
when s_output =>
   output <= A;
   state <= s_output;
```

- 원하는 결과 이후 state도 더 이상 변화하지 않고 s_output 에 fix 되어 있는 것을 확인할 수 있다.


14

# Lab_11 : (8) Discussion

- 최종적으로 코드 작성부터 모든 과정을 직접 진행하면서 CPU를 design 해 보았다.
- 기본적으로 instruction이 복잡하지 않고 기본적인 것들만 포함하고 있기 때문에 초반에 코드 작성은 크게 어렵지 않았다.
- 또한, EC2라는 참고할 수 있는 코드가 있어 일부 수정하는 정도로만 진행하였기 때문에 큰 오류 없이 완성할 수 있었다.
- 처음에 JMP를 어떤 조건으로써 처리할 것인지 고민하였지만 후에 테스트 알고리즘을 작성하는 과정에서 data 값이 '0' 인지 아닌지 확인할 필요가 있다고 생각하여 JUMP not zero 연산을 수행하였다.
- 또한 별도로 output instruction 이 존재하였기 때문에 이를 좀 더 효율적으로 사용하고 싶어 원하는 대로 구현하려고 노력하였다. 이 과정에서 몇번의 시행착오를 경험하였다. 우선 기존의 lab10에서 진행한 실습에서는 외부로 출력되는 data 값이 중간 과정을 모두 포함하고 있어 Halt의 도움 없이는 원하는 결과를 확인하는 것이 시각적으로 깔끔하지 않았다고 생각한다. 더군다나 이번 실습에서는 Halt 기능이 없기 때문에 만약 중간 과정의 data 값을 모두 output으로 출력한다면 simulation 결과를 확인하는 입장에서 불편할 수 있겠다고 생각하여 연산을 종료하면 종료된 data 만 출력하도록 하기 위해서 코드를 작성하였다. 바로 성공하지는 못하였고 단순히 생각하다가 A 값을 그냥 출력해보니 '0' 값만 출력되는 오류를 범했다. 당연한 결과이지만 코드 작성할 당시에는 이 오류를 디버깅하지 못해 꽤 헤맸다. 생각해보면 A 데이터는 이미 반복적인 dec 연산으로 '0' 이 된 이후였고 확인하고 싶은 결과는 sum이 저장되어 있는 memory address에 있다는 것을 뒤늦게 인지하고 이를 load하고 output으로 출력하기 위해서 기존의 binary code를 수정하였다.

```
01001 : 00011110; -- LOAD A, "11110" / load data in memory address "11110" to A
01010 : 10111110; -- OUT A / show result data A
```

위가 그 기능을 수행하기 위한 byte code에 해당한다. 간단한 생각이었지만 당시에는 꽤 오래 헤매 오류를 찾은 이후에는 조금 허망하기도 하였다.

- 기존의 실습 코드를 참고하긴 하였지만 기본적으로 처음부터 주도적으로 실습을 진행한 것은 처음이기도 하다. 주도적으로 오류를 찾아가고 원하는 모델을 구현하기 위해서 노력했더니 CPU design 을 한단계 더 이해할 수 있는 시간이 되었다.
- Instruction 들을 구현해두고, RAM의 binary code의 수정만을 통해서 완전히 다른 결과를 얻을 수 있고 만들어 낼 수 있다는 것이 CPU design의 매력인 것 같다.