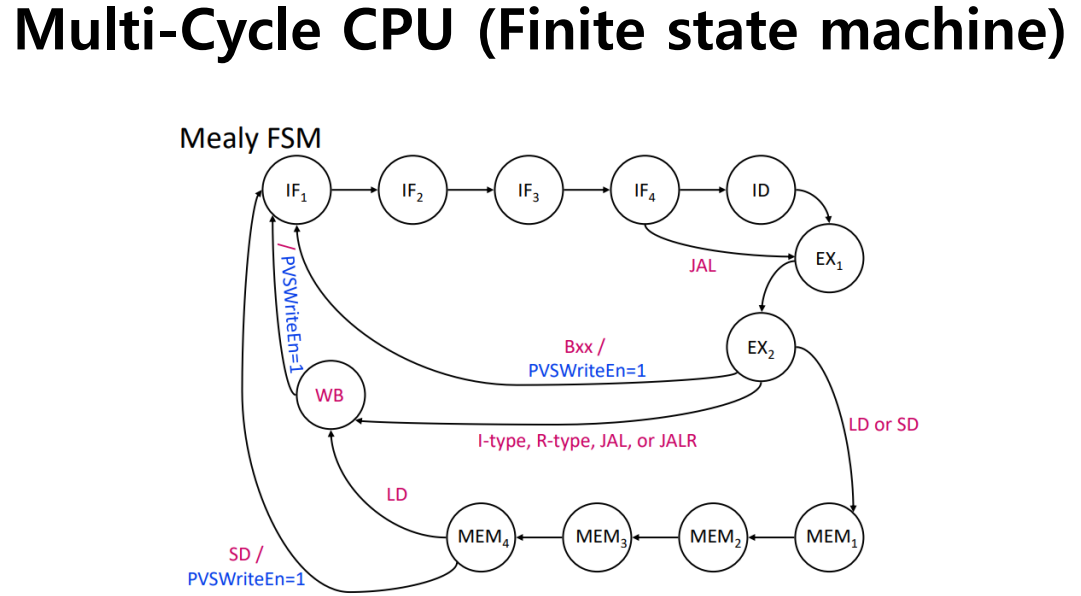
Lab 4. Mutli-Cycle CPU

20190065 강두경

20190316 유병호

# Introduction

이번 랩의 목표는 Verilog를 이용해 Multi-Cycle CPU를 구현하는 것이다. Single-Cycle CPU는 한 사이클 당 하나의 명령어를 실행하기 때문에, 사이클 시간은 명령어 가운데 가장 오래 걸리는 명령어의 시간보다 길어야 한다. RISC-V에서는 LD 명령어가 가장 오랜 지연 시간이 걸리는데, Single-Cycle CPU의 경우 LD 보다 짧은 지연 시간을 갖는 명령어여도 LD 명령어의 지연 시간 이후에 다음 instruction으로 넘어가기 때문에 그만큼의 utilization 손실을 겪는다. 이 문제를 해결하기 위해 도입된 개념이 Multi-Cycle CPU이다. Multi-Cycle CPU는 사이클을 조금 더 작은 시간 단위로 나누어 한 인스트럭션이 여러 사이클에 걸쳐 실행되도록 하는 것이다. 그렇게 되면 각 인스트럭션마다 걸리는 시간을 다르게 배치함으로써 효율성을 높이는 데에 그 의의가 있다.



Multi-Cycle CPU의 핵심은 Instruction을 작은 Instruction으로 쪼개는 것이다. 이는 Instruction에 따라 다르게 나타나며, 쪼개진 micro instruction에 맞게 control bit을 연결하여 이에 맞게 CPU가 작동하는 것이다. instruction이 PC를 state로 갖는 FSM으로부터 PC 주소로 메모리에 접근해서 받아온 후 연산을 통해 다음 PC State를 계산한다면, micro instruction은 주어진 instruction으로부터 다음 micro state를 계산하는 FSM으로 구성되어 있다. 실제로 이번 Lab의 핵심도 Micro State FSM을 Control Unit 내에 잘 구현하는 것이다.

# Design

## Control Unit

Control Unit은 CPU 회로 각 부분에서 어떤 데이터가 들어갈지, 특정 레지스터의 latch를 작동할지 등을 결정하는 Control Bit을 결정하는 핵심적인 역할을 한다. Multi-Cycle CPU는 Instruction이 작은 부분으로 쪼개져야 하는데 이 부분에 대한 FSM도 구현되어야 하므로 이번 lab의 핵심 모듈이라고 할 수 있다.

우선, Control Unit의 입력과 출력은 다음과 같다.

input reset\_n;

input [3:0] opcode;

input [5:0] func\_code;

reg [4:0] state\_reg;

reg [4:0] next\_state\_reg;

input clk;

output pc\_write\_cond, pc\_write, i\_or\_d, mem\_read, mem\_to\_reg, mem\_write, ir\_write, pc\_src;

output pc\_to\_reg, halt, wwd, new\_inst, pc\_temp\_write;

output [1:0] reg\_write, alu\_src\_A, alu\_src\_B;

output alu\_op;

input에는 우선 clk과 reset\_n을 받고, Instruction의 opcode와 func\_code를 받는다. microinstruction state가 opcode와 func\_code에 의존하여 다음 state를 결정하기 때문이다.

다음은 각 output control bit에 대한 설명이다.

Table 1

|  |  |
| --- | --- |
| Control Bit | 설명 |
| pc\_write\_cond | Branch Condition이 만족하면 PC Latch를 enable한다. |
| pc\_write | PC Latch를 enable 한다. |
| i\_or\_d | 1 : data address로 ALU의 Output을 대입한다. (LD, SD)  0 : data address로 PC를 대입한다. |
| mem\_read | 메모리 값을 읽어들인다. |
| mem\_to\_reg | 메모리에서 레지스터로 값을 저장한다. |
| mem\_write | 메모리에 값을 쓴다. |
| ir\_write | Instruction Register의 latch를 enable한다. |
| pc\_src | 1 : PC에 ALU Out Register의 값을 대입한다.  0 : PC에 ALU 의 출력값을 그대로 대입한다. |
| pc\_to\_reg | PC의 값을 reg에 대입한다. |
| halt | Halt 시그널을 보내고 프로그램을 종료한다. |
| wwd | Output Port에 주어진 값을 반환한다. |
| new\_inst | instruction이 끝났으며 num\_inst에 값을 추가한다. |
| pc\_temp\_write | PC Temp register의 latch를 enable한다. |
| reg\_write[1:0] | 01 : $2 레지스터에 값을 쓴다.  10: rd 레지스터에 값을 쓴다.  11 : rt 레지스터에 값을 쓴다. |
| alu\_src\_A | 1 : ALU의 A Input에 REG\_A register의 값을 대입한다.  0 : ALU 의 A Input에 PC의 값을 대입한다. |
| alu\_src\_B[1:0] | 00 : ALU의 B Input에 REG\_B의 값을 대입한다.  01 : ALU의 B Input에1을 대입한다.  10 : ALU의 B Input에 sign extended된 instr[7:0]의 값을 대입한다.  11 : ALU의 B Input에 Unsigned Extended된 instr[11:0]의 값을 대입한다. |
| alu\_op | ALU의 연산 방법을 결정한다.  0 : ALU의 연산을 +(adder)로 고정한다.  1 : ALU의 연산을 instruction에 의존하여 결정한다. |

각 control bit는 언제 1이 되어야 하고, 언제 0이 되어야 할까? 그 전에 각 instruction은 어떤 microinstruction으로 가게 될지를 알아볼 필요가 있다.

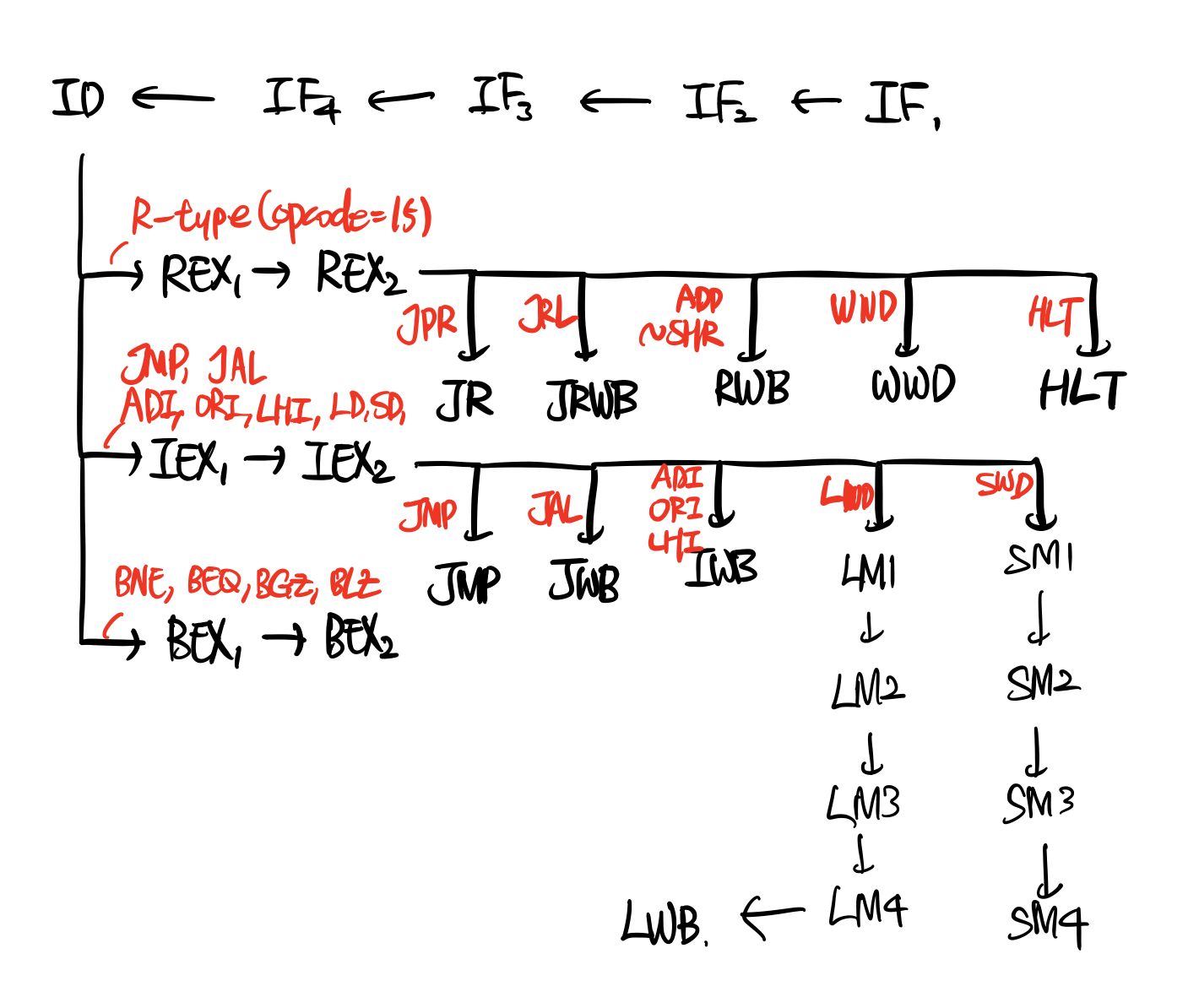
Instruction은 기본적으로 Instruction Fetch, Instruction Decode, Execution, Memory Access, Write Back의 다섯 단계로 이루어져 있다. 각 단계마다 소요되는 시간이 다르고, Instruction 에 따라 특정 단계를 건너뛸 수도 있다. 간단한 접근은 각 단계를 Microinstruction State를 놓는 것이다.

clk은 일정한 사이클 시간을 두고 계속 돌아가는데, 단계마다 각기 다른 시간을 가진 것을 해결하는 간단한 방법은 state를 여러 개로 쪼개는 것이다. 한 사이클 시간이 50ps고, Instruction Fetch가 총 200ps의 지연 시간을 소요한다면 IF1, IF2, IF3, IF4의 state를 놓을 수 있다. 같은 방법으로 IF는 200ps, ID는 50ps, EXE는 100ps, MEM은 200ps, WB는 50ps가 소요되는 점을 이용하면 다음과 같은 state를 놓을 수 있다.

IF1, IF2, IF3, IF4, ID, EX1, EX2, MEM1, MEM2, MEM3, MEM4, WB

그러나 이렇게 하면 state만 갖고 control bit를 결정할 수 없다. 가령 EX 단계에서 R type은 ALU B Input에 rt 레지스터의 값이 들어가지만 I type은 immediate 레지스터의 값이 들어가기에 ALUSrc\_B bit의 값을 다르게 해 주어야 한다. 즉 각 control bit가 state뿐만 아니라 instruction 값에 의존한다. 이렇게 디자인할 수도 있겠지만, control bit를 최대한 micro state 값에만 의존하게 하여 결합도를 낮추고 싶었다.

control bit가 micro state 값에만 의존할 수 있도록 하기 위해 state의 수를 세분화했다. 가령, 똑같은 EX 단계라 하더라도 R-type일 경우 REX1, REX2로 놓았고, I-type일 경우 IEX1, IEX2 타입으로 놓았으며, 끝으로 B-type일 경우 BEX1, BEX2로 놓았다. 이와 같은 방법으로30개의 state를 확정지었다. rst state의 경우 예외적으로 reset\_n상태일 때 시작하는 state이며, 켜지면 if1로 이동한다.



각 State별로 다음 State로 가는 Instruction은 다음과 같다. 위 그림에서 표시하지 않은 한 가지 예외는 JMP, JAL 명령어의 경우 IF4에서 ID를 거치지 않고 바로 IEX1로 이동한다는 것이다. 이 규칙을 바탕으로 microstate의 다음 state를 결정하는 코드를 작성할 수 있다. 또한 HLT를 제외한 나머지 instruction은 끝난 후 다시 IF1로 회귀한다.

끝으로, 각 microinstruction이 control\_bit을 어떻게 결정하는 지에 대한 진리표이다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| uInst | pcwc | pcw | i/d | mr | mw | m2r | irw | pcsrc | pc2r | newI | RW | asA | asB | tw | aOp |
| IF | 0 | 1(if4) | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0(if1) | 00 | 0 | 01 | 0 | 0 |
| ID | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00 | 0 | 10 | 10 | 0 |
| IEX | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00 | 1 | 1u | 1(ex1) | 1 |
| REX | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00 | 1 | 00 | 0 | 1 |
| BEX | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 00 | 1 | 00 | 0 | 1 |
| LM | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 00 | 1 | 10 | 0 | 1 |
| SM | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 00 | 1 | 10 | 0 | 1 |
| JMP | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 00 | 0 | 00 | 0 | 0 |
| JWB | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 01 | 0 | 00 | 0 | 0 |
| JR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 00 | 1 | 00 | 0 | 0 |
| JRWB | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 01 | 1 | 00 | 0 | 0 |
| RWB | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 00 | 0 | 0 |
| IWB | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 11 | 0 | 00 | 0 | 0 |
| WWD | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00 | 0 | 00 | 0 | 0 |
| LWB | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 11 | 0 | 00 | 0 | 0 |

* HLT일 경우 어떤 HLT Control Bit만 1이면 프로그램 작동을 완전히 종료하므로 다른 control bit와는 무관하다.
* 값에 무관하게 잘 작동하는 don’t-care 항도 많지만, (latch를 결정하는 bit의 경우 0, 1이 정확해야 함) 우선 모두 0으로 두었다. Gate level SImplification을 진행할 때 don’t care 항을 고려하면 될 것 같다.
* alusrcB 열의 IEX의 경우 opcode가 9 or 10일 경우 값은 11이 된다. 이 이유는 J type일 경우 instruction의 11:0까지의 값을 읽어오는데, I-type일 때 9:0의 값을 sign extend 시키는 immediate 값과 구조적으로 다르기 때문이다. 따라서 이 부분에서 예외적으로 control bit를 state 뿐만 아니라 opcode에 의존하도록 하는 회로를 추가했다. 물론 새로운 State(예를 들어 JEX)를 넣어 의존성을 해결 가능하다.

## Register File

Single Cycle CPU에서 사용한 register와 유사하게 4개의 16bit Register을 저장하는 역할의 모듈이다. read1, read2, write\_reg을 받아 읽을 두 개의 레지스터 번호와 쓸 레지스터의 번호를 가져오고, write\_data를 받아 write\_reg 레지스터에 값을 대입한다. output은 read, read1 레지스터의 값을 반환한다.

## Memory

주어진 메모리를 변경하지 않고 그대로 사용했다. posedge clk 조건마다 read\_m이 1이면 target memory에서 data 값을 읽어와 output으로 주고, write\_m이 1이면 input으로 가져온 data를 target memory에 저장시켜 준다.

if(read\_m)output\_data <= memory[address];

if(write\_m)memory[address] <= data;

## ALU Control Unit

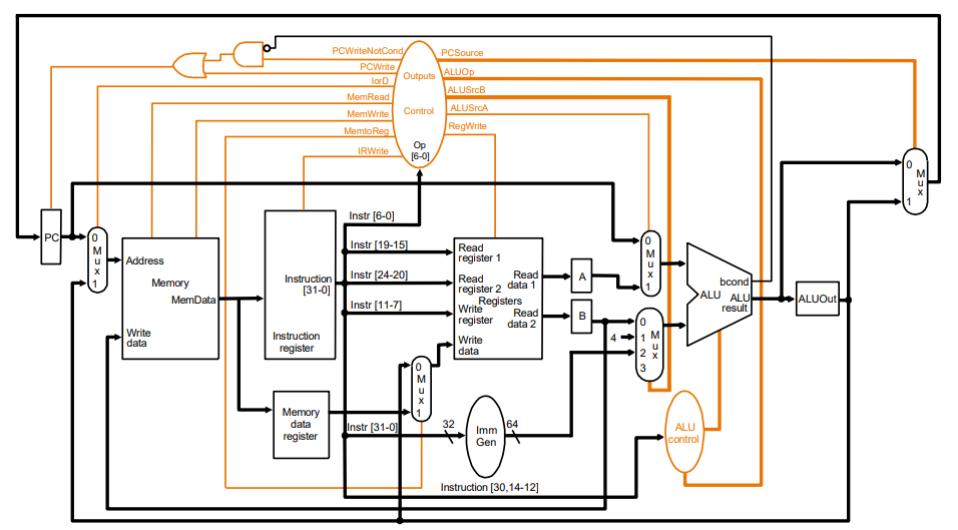
Instruction의 opcode, funcCode와 ALUOp를 받아 ALU 에서 할 연산의 종류를 결정한다. 현재 instruction이 branch일 경우에는 각 branch마다 branch type을 계산하여 이를 alu에서 사용한다. 이를 구분해주는 이유는 branch의 경우 alu의 input data들에 대해 계산을 수행하여 새로운 data를 만들어 내는 대신에, input data에 대한 어떤 comparison을 통해서 branch condition bit를 만들기 때문이다.

## ALU

Input으로 받은 func\_code와 branch\_type에 해당하는 연산을 받은 두 개의 데이터를 이용해 연산한 후 결과를 반환한다. 일반적으로는 instruction의 규칙을 따르는 연산을 하되 형식적으로 EX 과정을 거치면서 그대로 반환되는 몇 가지가 있다. branch일 경우 branch\_type에 따라 bncond의 값을 결정한다.

## CPU

CPU는 아래 이미지의 mutli cycle CPU Logic을 이용하여 구현되었다.



한 가지 차이점은 JRL에서 $2에 이전 pc+1의 값을 저장하는데, 동시에 pc에는 레지스터의 값을 넣어주어야 하기 때문에 $2에 이전 pc+1의 값이 아닌 새로운 pc가 대입되는 문제가 생겨 임시로 이전 pc의 값을 저장해주는 레지스터를 하나 더 만들었으며, control\_pc\_to\_reg 컨트롤 비트가 1일 경우 reg\_write 값에 그 임시 pc register 값을 대입하도록 하는 로직을 만들었다.

Instruction의 다섯 단계(IF, ID, EX, MEM, WB)에 따라 CPU의 작동 방식을 나타내면 다음과 같다.

|  |  |
| --- | --- |
| Stage | Behavior |
| IF | memory에 접근할 때 pc or alu\_out를 결정하는 mux에서 pc를 결정하여 memory에서 instruction을 읽어온다. |
| ID | instruction에서 필요한 register의 data들을 register\_module에서 읽어온다. |
| EX | mux를 통해 alu의 input1과 input2를 결정하고, instruction의 연산을 alu에서 수행해준다. |
| MEM | alu에서 계산된 memory의 address를 pc or alu\_out를 결정하는 mux에서 alu\_out을 결정하고, memory에 접근한다. store instuction일 경우, write data를 target memory에 저장해주고, load instruction일 경우 이 data를 WB에서 사용한다. |
| WB | load instruction일 때는 memory에서 가져온 data를, 일반적인 alu instruction일 경우에는 alu\_out을 mux를 통해 결정하여 register에 저장해준다. |

CPU는 많은 레지스터를 포함하는데, 레지스터는 posedge clk 조건시마다 다음 값으로 업데이트해주는 sequential logic을 구현해야 한다. 각 레지스터마다 latch 조건이 있다.

# Implementation

## Control Unit

현재 micro instruction의 state에서 opcode로부터 다음 state로 이동하는 FSM를 combinational logic으로 구현한 코드는 다음과 같다.

always @(\*) begin

    case (state\_reg)

      `ms\_rst: next\_state\_reg = `ms\_if1;

      `ms\_if1: next\_state\_reg = `ms\_if2;

      `ms\_if2: next\_state\_reg = `ms\_if3;

      `ms\_if3: next\_state\_reg = `ms\_if4;

      `ms\_if4: next\_state\_reg =

        opcode == 4'd9 || opcode == 4'd10 ? `ms\_iex1 :

        `ms\_id;

      `ms\_id: next\_state\_reg =

        opcode == 4'd15 ? `ms\_rex1 :

        opcode[3:2] == 2'b00 ? `ms\_bex1 :

        `ms\_iex1;

      `ms\_iex1: next\_state\_reg = `ms\_iex2;

      `ms\_iex2: next\_state\_reg =

        opcode == 9 ? `ms\_jmp :

        opcode == 10 ? `ms\_jwb :

        opcode == 7 ? `ms\_lm1 :

        opcode == 8 ? `ms\_sm1 :

        `ms\_iwb;

      `ms\_rex1: next\_state\_reg = `ms\_rex2;

      `ms\_rex2: next\_state\_reg =

        func\_code == 25 ? `ms\_jr :

        func\_code == 26 ? `ms\_jrwb :

        func\_code == 28 ? `ms\_wwd :

        func\_code == 29 ? `ms\_hlt :

        `ms\_rwb;

      `ms\_bex1: next\_state\_reg = `ms\_bex2;

      `ms\_bex2: next\_state\_reg = `ms\_if1;

      `ms\_lm1: next\_state\_reg = `ms\_lm2;

      `ms\_lm2: next\_state\_reg = `ms\_lm3;

      `ms\_lm3: next\_state\_reg = `ms\_lm4;

      `ms\_lm4: next\_state\_reg = `ms\_lwb;

      `ms\_lwb: next\_state\_reg = `ms\_if1;

      `ms\_sm1: next\_state\_reg = `ms\_sm2;

      `ms\_sm2: next\_state\_reg = `ms\_sm3;

      `ms\_sm3: next\_state\_reg = `ms\_sm4;

      `ms\_sm4: next\_state\_reg = `ms\_if1;

      `ms\_iwb: next\_state\_reg = `ms\_if1;

      `ms\_jmp: next\_state\_reg = `ms\_if1;

      `ms\_jwb: next\_state\_reg = `ms\_if1;

      `ms\_jr: next\_state\_reg = `ms\_if1;

      `ms\_jrwb: next\_state\_reg = `ms\_if1;

      `ms\_rwb: next\_state\_reg = `ms\_if1;

      `ms\_wwd: next\_state\_reg = `ms\_if1;

      `ms\_hlt: next\_state\_reg = `ms\_hlt;

      default: next\_state\_reg = `ms\_hlt;

    endcase

  end

다음은, microinstruction에 의해 나타나는 control bit의 진리표를 코드로 구현한 것이다.

자주 중복되는 항을 묶은 후 각 control bit에 or 연산을 통해 알맞은 값을 대입하였다. 작성한 코드는 다음과 같다.

assign isBex = state\_reg == `ms\_bex1 || state\_reg == `ms\_bex2;

  assign isRex = state\_reg == `ms\_rex1 || state\_reg == `ms\_rex2;

  assign isIex = state\_reg == `ms\_iex1 || state\_reg == `ms\_iex2;

  assign isIf = state\_reg == `ms\_if1 || state\_reg == `ms\_if2 || state\_reg == `ms\_if3 || state\_reg == `ms\_if4;

  assign isLm = state\_reg == `ms\_lm1 || state\_reg == `ms\_lm2 || state\_reg == `ms\_lm3 || state\_reg == `ms\_lm4;

  assign isSm = state\_reg == `ms\_sm1 || state\_reg == `ms\_sm2 || state\_reg == `ms\_sm3 || state\_reg == `ms\_sm4;

  assign pc\_write\_cond = isBex;

  assign pc\_temp\_write = state\_reg == `ms\_id || state\_reg == `ms\_iex1;

  assign pc\_write = state\_reg == `ms\_if4 || state\_reg == `ms\_jmp || state\_reg == `ms\_jwb || state\_reg == `ms\_jr || state\_reg == `ms\_jrwb;

  assign i\_or\_d = isLm || isSm;

  assign mem\_read = isIf || isLm;

  assign mem\_write = isSm;

  assign mem\_to\_reg = state\_reg == `ms\_lwb;

  assign ir\_write = state\_reg == `ms\_if1 || state\_reg == `ms\_if2 || state\_reg == `ms\_if3;

  assign pc\_src = isBex || state\_reg == `ms\_jmp || state\_reg == `ms\_jwb || state\_reg == `ms\_jr || state\_reg == `ms\_jrwb;

  assign pc\_to\_reg = state\_reg == `ms\_jwb || state\_reg == `ms\_jrwb;

  assign new\_inst = state\_reg == `ms\_if1;

  assign reg\_write[1] = state\_reg == `ms\_rwb || state\_reg == `ms\_iwb || state\_reg == `ms\_lwb;

  assign reg\_write[0] = state\_reg == `ms\_jwb || state\_reg == `ms\_jrwb || state\_reg == `ms\_iwb || state\_reg == `ms\_lwb;

  assign alu\_src\_A = isRex || isIex || isBex || isSm || isLm || state\_reg == `ms\_jr || state\_reg == `ms\_jrwb;

  assign alu\_src\_B[1] = state\_reg == `ms\_id || isIex || isSm || isLm;

  assign alu\_src\_B[0] = isIf || (isIex && (opcode == 9 || opcode == 10));

  assign alu\_op = isRex || isIex || isBex || isLm || isSm;

  assign halt = state\_reg == `ms\_hlt;

  assign wwd = state\_reg == `ms\_wwd;

끝으로, next state는 combinational logic에 의해 대입되고, stage는 clk이 posedge가 될 때마다 synchronous하게 업데이트된다.

always @(posedge clk) begin

    if (!reset\_n)

      state\_reg <= `ms\_rst;

    else

      state\_reg <= next\_state\_reg;

  end

## Register

Register의 Implementation은 Single Cycle 때와 같다.

initial begin

        register[0] <= 16'b0;

        register[1] <= 16'b0;

        register[2] <= 16'b0;

        register[3] <= 16'b0;

    end

    assign read\_out1 = register[read1];

    assign read\_out2 = register[read2];

    always @(posedge clk) begin

        if (reg\_write == 1)

            register[write\_reg] <= write\_data;

    end

reg\_write bit가 1이면 clk이 posedge에 register[write\_reg]에 대입한다. output은 read1, read2번째 레지스터를 알맞은 단의 출력에 항상 반환한다.

## ALU Control Unit

ALUOp가 0이면 연산은 반드시 adder이고, 그렇지 않으면 instruction에 의존하여 결정한다. opcode가 0~3이면 branch이므로 funcCode는 branch가 되고 branchType은 이에 맞춰서 ne, eq, gz, lz 중 하나로 결정된다. opcode == 4~10은 I-type 연산이며, opcode == 15일 때는 R-type 연산이며 이 때는 instruction의 funcCode에 따라 결과를 결정한다. 작성한 코드는 다음과 같다.

 always @(\*) begin

    if (ALUOp == 0) begin

      funcCode = `ac\_add;

    end

    else begin

      case (opcode)

        0: begin

          funcCode = `ac\_branch;

          branchType = `bt\_ne;

        end

        1: begin

          funcCode = `ac\_branch;

          branchType = `bt\_eq;

        end

        2: begin

          funcCode = `ac\_branch;

          branchType = `bt\_gz;

        end

        3: begin

          funcCode = `ac\_branch;

          branchType = `bt\_lz;

        end

        4: funcCode = `ac\_add;

        5: funcCode = `ac\_orr;

        6: funcCode = `ac\_lhi;

        7: funcCode = `ac\_add;

        8: funcCode = `ac\_add;

        9: funcCode = `ac\_jmp;

        10: funcCode = `ac\_jmp;

        15: begin

          case (funct)

            0: funcCode = `ac\_add;

            1: funcCode = `ac\_sub;

            2: funcCode = `ac\_and;

            3: funcCode = `ac\_orr;

            4: funcCode = `ac\_not;

            5: funcCode = `ac\_tcp;

            6: funcCode = `ac\_shl;

            7: funcCode = `ac\_shr;

            25: funcCode = `ac\_jr;

            26: funcCode = `ac\_jr;

            28: funcCode = `ac\_wwd;

            default: funcCode = `ac\_add;

          endcase

        end

        default: funcCode = `ac\_add;

      endcase

    end

## ALU

ALU는 기본적으로 func\_code에 따라 알맞은 값을 연산하여 C에 combinational logic으로 대입한다. brnach의 경우에는 조건에 맞게 bcond의 값을 바꾼다. wwd, jr일 경우에는 A(rs)의 값을 그대로, jmp일 경우에는 B(imm)의 값을 그대로 반환한다. 끝으로 overflow flag는 ac\_add이거나 ac\_sub일 때 켜지는데 입력값의 MSB와 출력값의 MSB를 비교하여 출력하였다.

always @(\*) begin

      case (func\_code)

         `ac\_add: C = A + B;

         `ac\_and: C = A & B;

         `ac\_orr: C = A | B;

         `ac\_sub: C = A - B;

         `ac\_not: C = ~A;

         `ac\_tcp: C = ~A + 1;

         `ac\_shl: C = A << 1;

         `ac\_shr: C = A >> 1;

         `ac\_lhi: C = B << 8;

         `ac\_wwd: C = A;

         `ac\_jmp: C = B;

         `ac\_jr: C = A;

         `ac\_branch:

            case (branch\_type)

               `bt\_ne: bcond = (A != B);

               `bt\_eq: bcond = (A == B);

               `bt\_gz: bcond = (A[15] != 1'b1) && ( A != 16'd0);

               `bt\_lz: bcond = (A[15] == 1'b1);

            endcase

      endcase

      overflow\_flag =

         func\_code == `ac\_add ? (A[15] && B[15] && ~C[15] || ~A[15] && ~B[15] && C[15]) :

         func\_code == `ac\_sub ? (A[15] && ~B[15] && ~C[15] || ~A[15] && B[15] && C[15]) : 0;

   end

## CPU

CPU는 가장 큰 모듈인 만큼 부분부분 나눠서 설명하겠다.

### Immediate

assign immediate\_j[11:0] = INSTR[11:0];

assign immediate\_j[15:12] = 4'b0000;

assign immediate\_extended[7:0] = INSTR[7:0];

assign immediate\_extended[15:8] = INSTR[7] ? 8'b11111111 : 8'b0;

Immediate에는 두 가지 종류가 있다. j-type instruction일 경우에 instruction에서 하위 12비트를 immediate로 사용하고, I-type일 경우에는 하위 8비트를 sign extend하여 immediate로 사용한다.

### Control Bit : pc\_write, pc\_write\_cond

PC register을 latch하는 조건은 두 가지 경우가 있다. 둘 중 하나가 충족되면 PC는 latch되어야 한다.

assign pc\_update = (control\_pc\_write\_cond & bcond) | control\_pc\_write & reset\_n;

1. bcond가 1인 동시에 control\_pc\_write\_cond가 1일 때 (Branch)
2. control\_pc\_write가 1일 때 (IF, Jump)

### Control Bit : reg\_write

control\_reg\_write는 어떤 레지스터에 write할지를 결정하는 모듈이다. R type이면 rd에, I type이면 rt에, jal이나 jrl이면 $2에 저장한다. 구현은 다음과 같다. MUX로 구현할 수 있겠으나 주어진 MUX가 16비트 MUX이기 때문에 2bit만 있으면 되는 이 경우는 assign 문으로 대체했다.

assign reg\_write\_port =

        control\_reg\_write == 2'b10 ? INSTR[7:6] :

        control\_reg\_write == 2'b01 ? 2'b10 :

        control\_reg\_write == 2'b11 ? INSTR[9:8] : 2'b00;

### Control Bit : mem\_read, mem\_write, halt

이 세 개의 control bit들은 외부 output으로 연결해 준다. read\_m, write\_m은 testbench에서 memory와 연결되어 데이터를 읽거나 쓰는 데에 사용되어, halt가 켜지면 testbench에서 해당 인스트럭션이 끝난 후 즉시 프로그램을 종료한다.

assign is\_halted = control\_halt;

    assign read\_m = control\_mem\_read;

    assign write\_m = control\_mem\_write;

### Control Bit : i\_or\_d

i\_or\_d control bit은 data address로 PC를 넣을지(IF) ALU\_OUT 값을 넣을지(LD MEM, SD MEM)를 결정하는 컨트롤 비트이다. 주어진 MUX를 연결하여 구현한다.

mux2\_1 m0(

        .sel(control\_i\_or\_d),

        .i1(PC),

        .i2(ALU\_OUT),

        .o(address)

    );

### Control Bit : alu\_src\_A, alu\_src\_B

Table 1의 alu\_src\_A, alu\_src\_B의 정의에 맞게 MUX를 통해 ALU에 알맞은 값을 대입한다.

|  |  |
| --- | --- |
| alu\_src\_A | 1 : ALU의 A Input에 REG\_A register의 값을 대입한다.  0 : ALU 의 A Input에 PC의 값을 대입한다. |
| alu\_src\_B[1:0] | 00 : ALU의 B Input에 REG\_B의 값을 대입한다.  01 : ALU의 B Input에1을 대입한다.  10 : ALU의 B Input에 sign extended된 instr[7:0]의 값을 대입한다.  11 : ALU의 B Input에 Unsigned Extended된 instr[11:0]의 값을 대입한다. |

mux2\_1 m1(

        .sel(control\_alu\_src\_A),

        .i1(PC),

        .i2(REG\_A),

        .o(alu\_input1)

    );

    mux4\_1 m2(

        .sel(control\_alu\_src\_B),

        .i1(REG\_B),

        .i2(16'd1),

        .i3(immediate\_extended),

        .i4(immediate\_j),

        .o(alu\_input2)

    );

### Control Bit : mem\_to\_reg, pc\_to\_reg

mem\_to\_reg와 pc\_to\_reg는 reg의 write data에 어떤 값이 들어갈지를 결정한다. 둘 다 0이면 ALU의 output이 들어가며, mem\_to\_reg가 1이면 메모리에서 읽은 값이, pc\_to\_reg가 1이면 미리 저장해 놓은 PC (temp register)의 값이 들어간다. 둘 다 1인 경우는 존재하지 않지만, 처음에는 pc\_to\_reg 없이 구현하려고 시도했어서 m3의 .sel에 || control\_pc\_to\_reg 가 남아 있다.

mux2\_1 m3(

        .sel(control\_mem\_to\_reg || control\_pc\_to\_reg),

        .i1(ALU\_OUT),

        .i2(MEMDATA),

        .o(reg\_write\_middle)

    );

    mux2\_1 m5(

        .sel(control\_pc\_to\_reg),

        .i1(reg\_write\_middle),

        .i2(REG\_PC\_TEMP),

        .o(reg\_write\_data)

    );

### Control Bit : pc\_src

pc\_src는 PC에 alu에서 바로 출력된 값을 넣을지, 혹은 ALU\_OUT의 값을 넣을지를 결정하는 레지스터이다. 이 레지스터가 필요한 이유는 Branch와 Jump 때문이다.

mux2\_1 m4(

        .sel(control\_pc\_src),

        .i1(alu\_output),

        .i2(ALU\_OUT),

        .o(next\_pc)

    );

Branch에서 branch 조건인지 ex 단계에서 검출 후 별도의 사이클 소모 없이 바로 pc를 이동시킨다. 즉, pc에 imm을 더해 다음 pc의 값을 반환하는 사이클이 별도로 없다. 이 문제를 ID 단계에서 미리 연산을 함으로써 해결할 수가 있는데, Branch의 Execution의 경우 alu output의 값을 바꾸지 않으므로 ALU\_OUT register의 값도 ID 단계에 미리 계산해 두었던 저장된 값을 가지기 때문이다. Jump의 경우도 EX 단계에서 미리 계산해 놓은 결과로 이동하기 위해 ALU\_OUT에 값을 저장한다.

### Control Unit, Register File, ALU Control, ALU 연결

위에 구현해 두었던 4개의 모듈을 모두 연결한다.

control\_unit u2(

        .reset\_n(reset\_n),

        .opcode(INSTR[15:12]),

        .func\_code(INSTR[5:0]),

        .clk(clk),

        .pc\_temp\_write(control\_pc\_temp\_write),

        .pc\_write\_cond(control\_pc\_write\_cond),

        .pc\_write(control\_pc\_write),

        .i\_or\_d(control\_i\_or\_d),

        .mem\_read(control\_mem\_read),

        .mem\_to\_reg(control\_mem\_to\_reg),

        .mem\_write(control\_mem\_write),

        .ir\_write(control\_ir\_write),

        .pc\_to\_reg(control\_pc\_to\_reg),

        .pc\_src(control\_pc\_src),

        .halt(control\_halt),

        .wwd(control\_wwd),

        .new\_inst(control\_new\_inst),

        .reg\_write(control\_reg\_write),

        .alu\_src\_A(control\_alu\_src\_A),

        .alu\_src\_B(control\_alu\_src\_B),

        .alu\_op(control\_alu\_op)

    );

    register\_file u3(

        .read\_out1(reg\_out1),

        .read\_out2(reg\_out2),

        .read1(INSTR[11:10]),

        .read2(INSTR[9:8]),

        .write\_reg(reg\_write\_port),

        .write\_data(reg\_write\_data),

        .reg\_write(control\_reg\_write[1] | control\_reg\_write[0]),

        .clk(clk)

    );

    alu\_control\_unit u4(

        .funct(INSTR[5:0]),

        .opcode(INSTR[15:12]),

        .ALUOp(control\_alu\_op),

        .clk(clk),

        .funcCode(alu\_funcCode),

        .branchType(alu\_branchtype)

    );

    alu u5(

        .A(alu\_input1),

        .B(alu\_input2),

        .func\_code(alu\_funcCode),

        .branch\_type(alu\_branchtype),

        .C(alu\_output),

        .overflow\_flag(overflow\_flag),

        .bcond(bcond)

    );

### Sequential Logic

CPU에는 많은 양의 레지스터가 있는데, 이를 모두 posedge clk 조건에서 업데이트해 주어야 한다.

우선 위에서 계산한 pc\_update가 1이면 pc latch를 enable 하고, ir\_write가 1이면 instruction latch를 enable한다. 인스트럭션이 끝나 new\_inst 비트가 1이면 num\_inst에 1을 더한다. wwd 비트가 1이면 OUTPUT\_PORT를 latch하여 테스트벤치에서 결과를 확인할 수 있도록 한다. pc\_temp\_write가 1이면 reg\_pc\_temp에 pc를 대입한다.

initial begin

        REG\_A <= 0;

        REG\_B <= 0;

        ALU\_OUT <= 0;

    end

    always @(posedge clk) begin

        if (!reset\_n)

        begin

            PC <= 0;

            NUM\_INST <= 0;

        end

        else if (pc\_update)

            PC <= next\_pc;

        if (control\_ir\_write)

            INSTR <= data;

        if (control\_new\_inst)

            NUM\_INST <= NUM\_INST + 1;

        if (control\_wwd)

            OUTPUT\_PORT <= ALU\_OUT;

        MEMDATA <= data;

        REG\_A <= reg\_out1;

        REG\_B <= reg\_out2;

        ALU\_OUT <= alu\_output;

        if (control\_pc\_temp\_write)

            REG\_PC\_TEMP <= PC;

    end

# Discussion

1. Control Unit을 구현하면서, State의 수를 늘려 overhead를 증가시키는 한이 있더라도 control bit가 state의 값에만 의존하도록 설계했다. (alu src B의 경우 제외). control bit가 state 값 뿐만 아니라 opcode에도 의존하도록 했다면 state의 수와 코드의 길이를 짧게 줄일 수 있었을 것이다. 처음에 설계한 것이 크게 장점과 효용이 없어 보인다는 생각이 들었다.
2. 가장 힘들었던 부분은 jlr 부분이다. 이전 레지스터의 값을 pc에 넣으면서 동시에 레지스터 2에는 이전 pc의 값을 넣어야 한다. 문제는 53번에서 jlr $2를 호출하는데, 이러면 pc에는 기존의 $2의 값이 들어가야 하고 $2에는 기존의 pc 값이 들어가야 하기 때문에 상충된다. 다만 모든 대입을 한 사이클 안에 끝내고 싶어 여러 가지의 대안을 찾다가 PC의 값을 임시로 저장하는 레지스터를 하나 더 만드는 것이 가장 원칙을 잘 지키는 방법이라고 생각했다. 다른 사람들은 이 문제를 어떻게 해결했을 지 궁금하다.
3. 나는 wire로 처리가 가능한 것은 combinational logic에서 always 구문 안에 레지스터에 값을 집어넣는 대신 assign 구문을 통해 바로 선을 연결하여 똑같은 기능을 구현하는 것을 선호한다. 코드를 보면 삼항 연산자, 대등 연산자의 사용이 과한 것을 알 수 있는데 동작이 제대로 된다면 Q-M Method를 통해 gate level simplification을 하여 비트별로 대입하면 비약적인 성능 향상을 기대할 수 있을 것이다.
4. 이번 랩은 state를 어떻게 디자인했는지가 전반적인 디자인과 구현에 가장 큰 영향을 끼치는 것 같다. 이 lab 과제의 모범 답안이나 다른 사람들의 답안을 확인하고 우리 조가 만든 답과 비교하면서 부족한 점을 보완하고 싶다.

# Conclusion

하나의 instruction을 작은 단위로 쪼개어 하나의 인스트럭션당 서로 다른 사이클 시간이 걸리게끔 설계하여 전체적인 시간을 줄이고 효과를 높이는 Multi-Cycle CPU를 구현해 보았다. 테스트는 56/56으로 모두 통과했다. register, state 수가 많아져서 쉽지 않았지만 복잡한 fsm에 대해서 이해하고 나아가 이를 직접 구상해 보는 의미 있는 경험을 할 수 있었따.