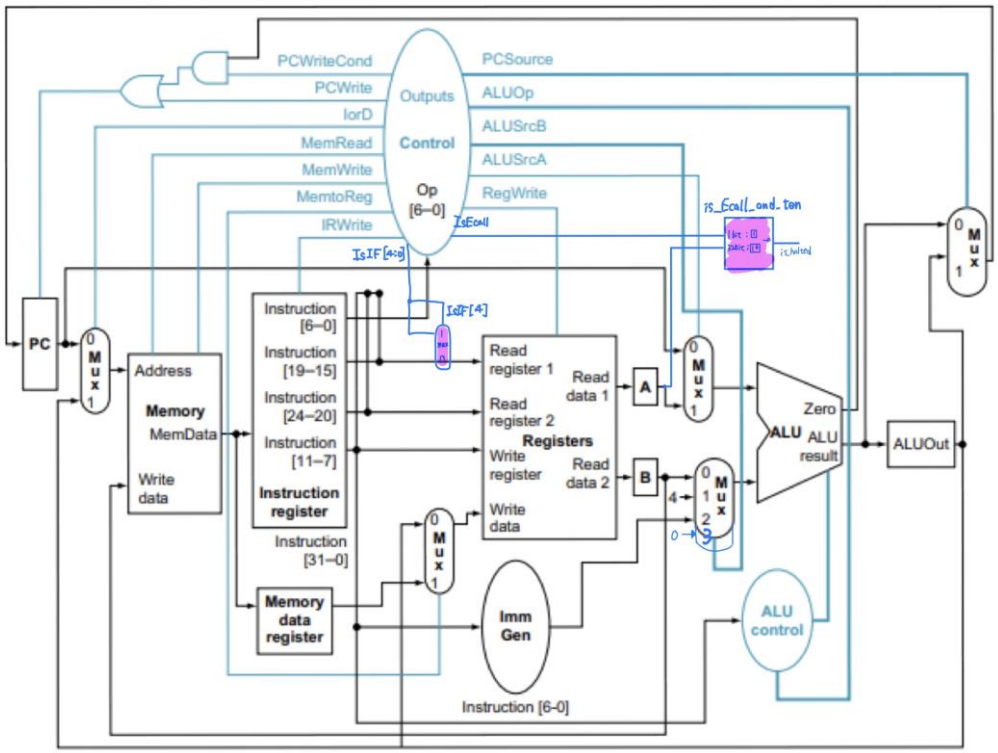
multi-cycle CPU

20210115 이세규, 20210706 이지원

1. introduction

이번 과제의 목표는 Vivado를 통해 multi-cycle RISC-V CPU를 구현하며 RISC-V의 구조와 state에 따른 control의 변화를 정확히 이해하고 구현하여 multi-cycle CPU가 어떻게 동작하는지 알아보는 것이다. 주어진 스켈레톤 코드와 LAB2의 single-cycle CPU의 구조도를 기반으로 Modularization을 중심으로 구현하였다.

2. design



A. datapath and control

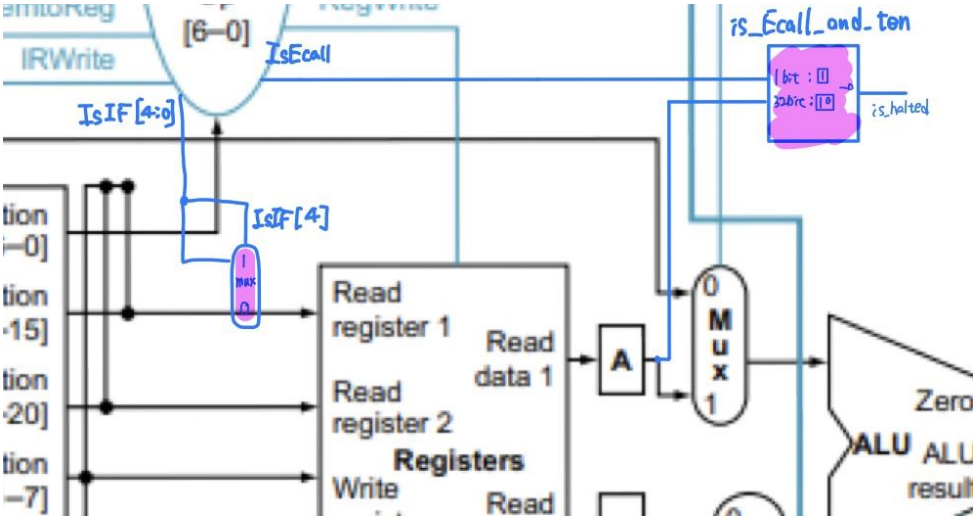
전체적으로 흐름은 single cycle CPU와 유사하나 memory를 통합하여 하나의 메모리만을 사용하고, ALU를 각 stage별로 연산을 나누어 사용하며 각 stage에서 올바른 값을 사용할 수 있도록 레지스터를 이용하는 것이 특징이다. IF stage에서는 PC의 값을 읽어 memory에서 연산해야 할 instruction을 찾아 instruction register에 저장한다. 이때 ALU에서는 PC+4의 연산을 수행하여 바로 다음에 수행될 instruction을 위해 PC값에 저장한다. 이러한 구현상 특징은 이후 implementation장에서 알아보겠다. ID stage에서는 register file에서 각 register를 읽어와 A, B register에 저장을 하고, 동시에 B type, JAL의 다음 PC값 연산을 진행하여 ALUOut register에 저장

을 한다. EX stage에서는 각 명령어 별 ALU에서 진행되어야 할 연산을 control에 따라 ALU에서 진행한 뒤 이를 ALUOut에 저장한다. 이후 Load와 Store는 MEM stage를 거친다. 각 instructions는 필요한 경우 WB stage를 거친 뒤 다음 명령어의 IF stage가 진행되며 넘어간다. WB하는 값은 memory에서 가지고 온 데이터가 될 수도 있고 이전 클락에서 진행한 ALUOut이 될 수도 있다.

Multi-cycle CPU의 경우 각 stage별로 CPU 내부의 사용되는 부분도 다르고 목적도 달라지기에 이를 Control로 적절히 통제해주어야 한다. stage에 따라 Mux의 아웃풋을 조절해주기 위해 알맞은 시그널을 출력해야 하며 PVS도 적절히 변경되어야 하기에 이를 시그널로 통제한다. 결국 control은 불필요한 연산을 무시하고 PVS가 임의로 변경되는 일이 없도록 해주는 역할을 한다.

B. ISIF 방법에 관하여

이번 과제의 조건에는 'register file을 변형하지 말 것' 이라는 조건이 있었다. 이를 지키기 위해서는 저번 single cycle CPU와 같은 구현은 불가능하다. 따라서 또 다른 방식의 접근이 필요했고 우리 팀은 multi cycle CPU의 '자원 재사용'이 가능한 특징에 착안하여 IF단계에서 사용되지 않고 있는 registers 모듈에 접근하여 17번째 레지스터 값을 읽기로 하였다. 그 값을 A 레지스터에 저장하고 다음 단계인 decode 단계에서 control unit에서 나온 IsEcall signal과 17번째 레지스터 값이 10인지를 확인하여 유효한 Ecall 연산임을 파악하여 CPU작동을 멈추는 방식을 구현하였다. 아래 확대한 그림을 보며 구체적인 사항을 확인해보면, IsIF는 IF단계에서 항상 17의 출력을 하고 그 밖의 단계에서는 0을 출력하여 기존의 구현과 독립적이되, 원하는 기능인 '17번째 레지스터 값'을 읽을 수 있는 기능을 구현하였다. MUX의 select signal로는 IsIF의 4번째 비트를 입력하는 식으로 select를 위한 signal을 간소화하였다. 그리고 is_Ecall_and_ten 모듈에서는 IF단계에서 register에 접근하여 읽어온 값이 저장된 곳인 A에 있는 값이 10인지와 IsEcall인지를 확인하는 signal을 가지고 is_halted signal을 판단한다.



3. implementation

A. 자원 재사용에 따른 control unit의 변화

아래 4. discussion에서 multi cycle가 single cycle CPU에 비해 자원을 적게 사용할 수 있는 구조가 마련되어 자원을 재사용 한다는 것에 대한 배경을 살펴볼 수 있다. 이를 우선 읽는 것을 추천한다. 이 장에서는 자원 재사용에 따른 control unit의 변화에 대해 살펴볼 것이다.

기존 single cycle CPU에서는 control unit은 '연산당' 한번만 정해지면 충분하였다. datapath가 연산에 따라 유일하게 결정되고 그러한 datapath를 signal이 결정해주는 식이었다. single cycle CPU에서는 한 클락당 하나의 연산을 진행하고 자원이 충분하기 때문에 결정된 datapath는 동일한 클락 내에서 변화될 일이 없었다.

하지만 multi cycle cpu에서는 다소 차이가 있다. datapath가 연산에 따라 유일하게 결정(결정되게 implemnet하였으니, 실제 작동하는 cpu에서는 유일하게 결정되는 것임.)되기는 하지만, 각 단계별로 클락을 하나씩 사용하므로 유일하게 결정된 datapath가 하나의 클락에서 모두 실행될 수는 없다는 차이점이 존재한다. 이 차이는 꽤나 큰 변화를 불러 일으킨다. 바로 control unit의 출력이 같은 연산이라고 할지라도 클락에 따라 달라져야한다는 점이다. 이 때문에 control unit은 단순한 combinational logic이 아닌 state를 가진 fsm이 되어야한다. control unit을 fsm으로 구성하는 식의 방법 이외에도 동일한 기능을 single cycle CPU에 단순히 PVSWriteEn이라는 신호를 출력해주는 fsm을 추가함으로써 만들 수도 있지만, 이번 과제의 목표는 control unit을 microcode controller로 구현하는 것에 있으므로 microcode controller에 대해 우리가 구현한 내용을 바탕으로 알아보자.

B. microcode controller에 관하여

아래는 control unit을 구성하는 code들이다.

```
module ControlUnit(
    input clk,
    input reset,
    input [6:0] instr,

    output  PCWriteCond,
    output  PCWrite,
    output  IorD,
    output  MemRead,
    output  MemWrite,
    output  MemtoReg,
    output  IRWrite,
    output  PCSource,
    output  [1:0] ALUOp,
    output  [1:0] ALUSrcB,
    output  ALUSrcA,
    output  RegWrite,
    output  IsEcall,
    output  [4:0] Islf
    );

    wire [2:0] AddrCtl;
    wire [3:0] state;

    Address_select_logic_unit ASLU(
     .clk(clk),
     .reset(reset),
     .instr(instr),
     .AddrCtl(AddrCtl),
     .state(state)
     );

    ControlUnit_PLA CU_PLA(
    .state(state),
    .PCWriteCond(PCWriteCond),
    .PCWrite(PCWrite),
    .IorD(IorD),
    .MemRead(MemRead),
    .MemWrite(MemWrite),
    .MemtoReg(MemtoReg),
    .IRWrite(IRWrite),
    .PCSource(PCSource),
    .ALUOp(ALUOp),
    .ALUSrcB(ALUSrcB),
    .ALUSrcA(ALUSrcA),
    .RegWrite(RegWrite),
    .AddrCtl(AddrCtl),

    .IsEcall(IsEcall),
    .Islf(Islf)
    );

endmodule
```

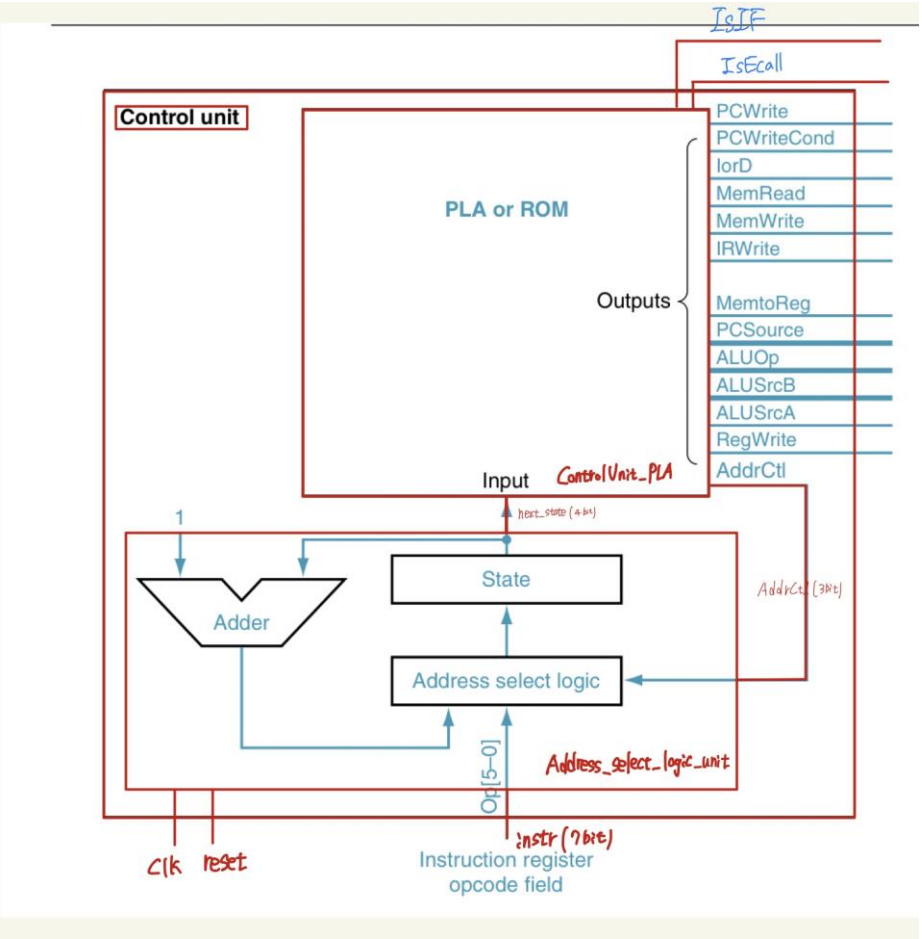
- control unit code

```
`timescale 1ns / 1ps

module Address_select_logic_unit(
    input clk,
    input reset,
    input [6:0] instr,
    input [2:0] AddrCtl,
    output reg [3:0] state
    );

     wire[3:0] R1,R2,A2M,out;

    MUX3 muxMC(
      .A(4'd0),
      .B(R1),
      .C(R2),
      .D(A2M),
      .E(4'd11),
      .F(4'd7),
      .select(AddrCtl),
      .OUT(out)
    );

    bit4adder adder(
      .A(state),    // input
      .B(4'd1),    // input
      .ALUresult(A2M)
    );

    ROM1 r1(
      .part_of_inst(instr),
      .state(R1)
    );

    ROM2 r2(
      .part_of_inst(instr),
      .state(R2)
    );



    always @(posedge clk) begin
        if(reset == 1'd1)
        begin
            state <=4'd0;
        end
        else if(reset == 1'd0)
        begin
            state<=out;
        end
    end


endmodule
```

- Address select logic unit code

```
`timescale 1ns / 1ps
module ControlUnit_PLA(
    input wire [3:0] state,  // input

    output reg PCWriteCond,
    output reg PCWrite,
    output reg IorD,
    output reg MemRead,
    output reg MemWrite,
    output reg MemtoReg,
    output reg IRWrite,
    output reg PCSource,
    output reg [1:0] ALUOp,
    output reg [1:0] ALUSrcB,
    output reg ALUSrcA,
    output reg RegWrite,
    output reg [3:0] AddrCtl,

    output reg IsEcall,
    output reg [4:0] Islf

    );


 always @(*) begin

        if(state==4'd0)begin
            PCWriteCond=1'b0;
            PCWrite=1'b1;
            IorD=1'b0;
            MemRead=1'b1;
            MemWrite=1'b0;
            MemtoReg=1'b0;
            IRWrite=1'b1;
            PCSource=1'b0;
            ALUOp=2'b00;
            ALUSrcB=2'b01;
            ALUSrcA=1'b0;
            RegWrite=1'b0;
            AddrCtl=3'b011;
            IsEcall=1'b0;
            Islf=5'b10001;
        end

        else if(state==4'd1)begin
            PCWriteCond=1'b0;
            PCWrite=1'b0;
            IorD=1'b0;
            MemRead=1'b0;
            MemWrite=1'b0;
            MemtoReg=1'b0;
            IRWrite=1'b0;
            PCSource=1'b0;
            ALUOp=2'b11;
            ALUSrcB=2'b10;
            ALUSrcA=1'b0;
            RegWrite=1'b0;
            AddrCtl=3'b001;
            IsEcall=1'b0;
            Islf=5'b00000;
        end
```

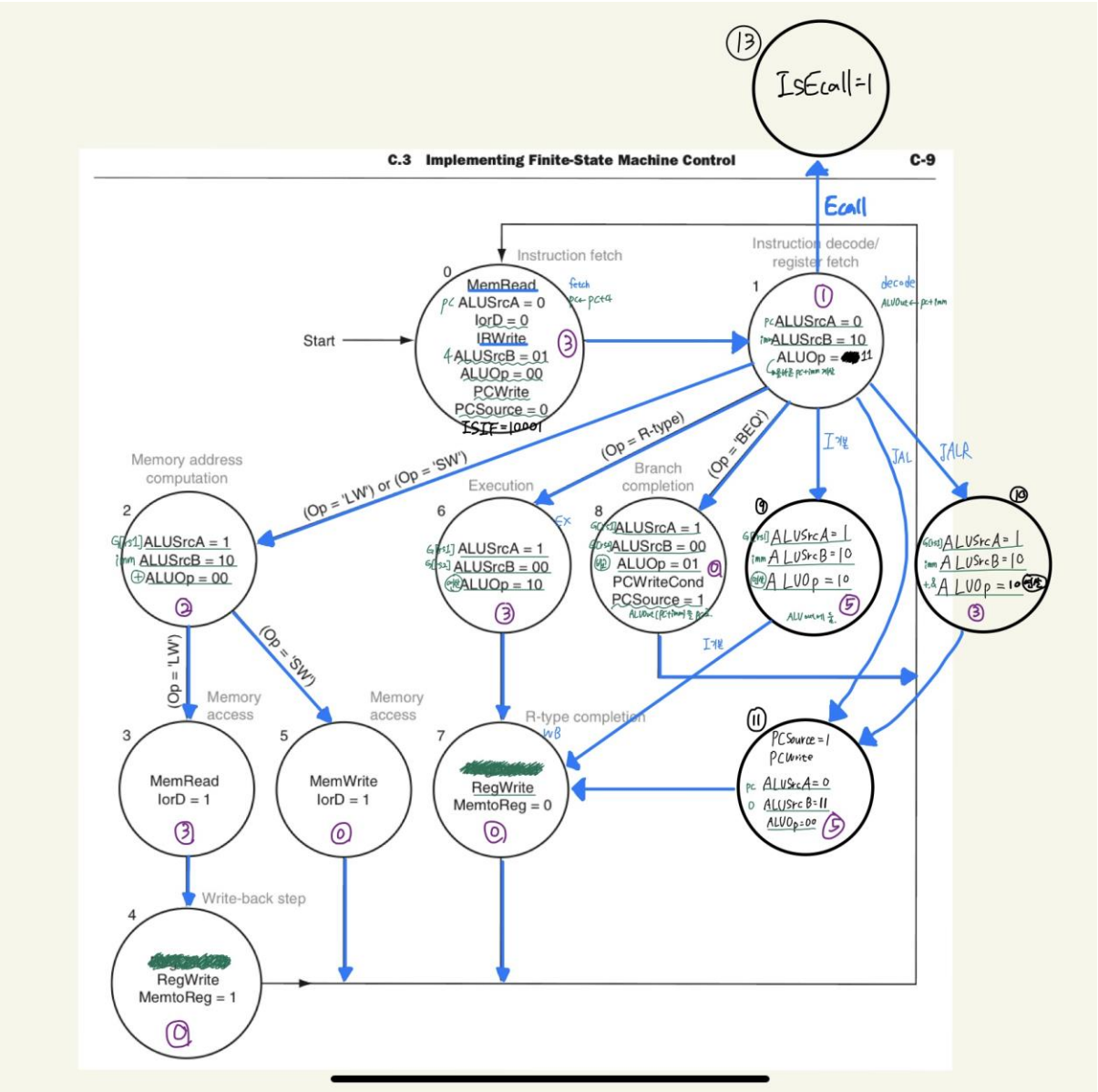
- PLA code: 예시로 2개의 state에 대한 구현만을 첨부하였습니다. 총 13개의 state에 대해 위와 같이 signal이 출력되도록 구현하였습니다.



위 그림이 최종적으로 우리가 구현해야 할 microcode controller이다. Control unit으로 빨간색 박스를 쳐 둔 곳이 위 CPU 도식에서 control 부분에 들어갈 것이다. control unit은 크게 두 가지 부분으로 구성된다. 하나는 CPU에서 사용할 signal을 출력하는 'PLA(programmable logic array) 또는 ROM(read only memory)' 이고 또 하나는PLA의 입력을 결정하는, 즉 control unit의 state를 결정하는 'address select logic unit'이다.

1. PLA

각 state의 output을 결정하는 것을 제외한 PLA의 구현 자체는 일반적인 fsm 구현보다 간단하다. sequential logic 인 fsm과 달리 PLA는 입력에 따라 출력이 정해져 있는 combinational logic이기 때문이다.



위 그림은 우리 팀이 구현한 PLA의 state diagram이다. state 즉, input은 총 13개가 있으며 (state 12는 없음.)각각의 state는 의미에 맞는 output을 가진다. 각 state에서 보라색 숫자는 addrctl값을 의미한다

state diagram에서 중요하게 보아야 할 부분 몇 개를 살펴보자.

a. 0 state의 기본 기능은 instruction fetch이다. 즉, MemRead값과 IRWrite 값을 인가해 줌으로써 Instruction register에 값을 저장하는 것이다. 이때, PC+4값을 PC에 저장하는데, 이는 fetch 단계에서 사용하지 않는 alu를 '사용'하는 것이다. 즉, 자원을 재활용하는 것에 대한 반향으로 자원을 더욱 알뜰하게 사용하는 것으로 해석하면 된다. 만약 pc+4를 이 state에서 하지 않으면, branch 연산을 처리할 때 taken과 nottaken인 경우의 필요 클락 사이클 수가 달라지게 되는, 또는 불필요한 스테이지가 더 필요하게 되는, 또는 state가 더 늘어나는 등의 곤란한 상황이 연출된다.

b. 1 state에서는 다음 단계로 갈 수 있는 선택지가 다양하다. 총 7개의 state가 다음 state로 결정될 수 있는데 이 결정에 dispatch rom을 사용한다. 세부 내용은 address select logic unit에서 알아보도록 하고 1state에서는 opcode에 따라 다음 상태가 달라질 수 있음을 기억하자. 그리고 이 상태는 decode를 위한 signal을 보내는 단계로 구성하였는데, decode단계에서는 아무런 control signal이 필요 없다. 따라서 이 단계에서는 pc+imm을 계산하여 aluout에 저장해둔다. 이때, 0state에서 pc값이 pc+4가 되었으므로 pc+imm을 할 때 4를 빼 주는 것이 필요하다. 이러한 특별한 연산은 ALU에서 간단히 새로운 연산 기능을 추가함으로써 구현할 수 있다.

c. 11 state에서는 jal의 wb이전 단계로, pc+4를 destination register에 저장하기 위한 구성을 해야한다. 그런데 1 state에서 pc+4가 구성되었으므로 기존의 pc값을 그대로 사용하면 된다. 따라서 0 과 더해준다. alu의 2번째 입력값으로 0 이될 수 있게 mux의 선택지에 4, imm, GPR[rs2] 이외에 0을 추가하였다.

이를 제외한 나머지 state는 각 연산의 sementic을 떠올리면 쉽게 생각해낼 수 있다. 자원을 재활용하는 과정에서 생기는 부수효과를 고려한 설계가 필요한 점을 명심해야한다. 그리고 각 state를 만드는 기본 생각은 연산 sementic의 추상화이다. 어떤 연산이 어떤 단계를 거치는지 파악하고 conflict에 대한 분석과 함께 각 연산의 경우마다 state를 따라가보며 적절한 신호를 출력하는지 확인하는 과정이 필요하다. 이 단계에서 겹치는 경우를 동일한 state로 구성하는 것 또한 필요하다.

aluop를 간략히 짚고 넘어가자면, alu control unit에서 이 2비트 값을 받아서 alu가 어떤 연산을 할 지 결정하는 4비트 값을 결정하는데, aluop가 00인 경우 항상 + 연산을 하면 된다. 즉, add 명령을 의미하는 4비트를 출력한다. 01의 경우는 branch연산에 해당하는 경우로 bcond값을 결정한다. 10의 경우는 R type이나 I type에서 필요한 연산을 출력해야 한다는 것을 알려주고 alu control unit에서는 instruction 32bit 중 일부를 보고 적절한 값을 출력한다. 마지막으로 11의 경우는 이 state diagram에서 필요한 두 값을 더하고 4를 빼는 연산을 출력하도록한다.

2. address select logic unit

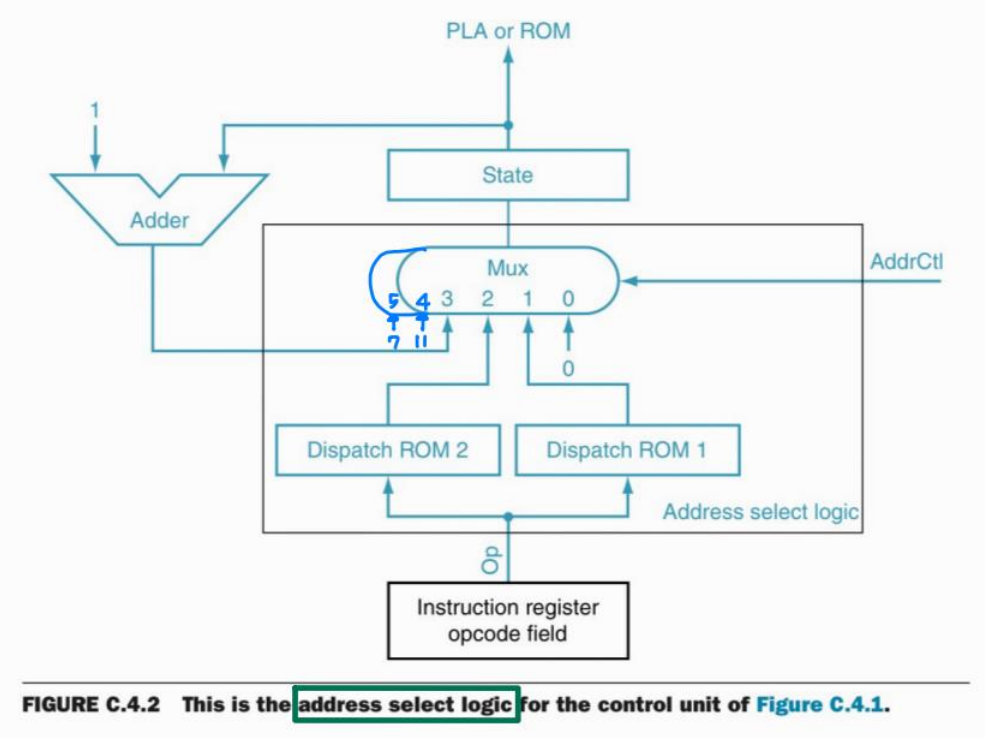


FIGURE C.4.2 This is the address select logic for the control unit of Figure C.4.1.

이는 address select logic의 세부적인 설계도이다. 각 state는 몇 번째 stage인지와 현재 instruction의 opcode가 무엇인지에 따라 다음 stage가 결정된다. 우선 다음 stage가 무엇인지를 결정하는 signal은 AddrCtl이다. AddrCtl은 PLA에서 결정되며 0인 경우 다음 stage가 0이고, 1과 2는 각각 ROM1과 ROM2에 의해 결정된다. 3의 경우 n stage의 다음 stage가 n+1 stage이며 이를 Adder로 더한 값을 사용하여 구현하였다. 4의 경우 구현하다가 필요가 사라져 쓰이지 않게 됐지만 11 stage로 향하며 마지막으로 5인 경우 7 stage로 향하게 된다. 위의 그림의 보라색 글씨로 써진 숫자는 각 stage별 AddrCtl이다.

Dispatch ROM 1

Op	Opcode name	Value
0110011	R-format	0110
1100011	~~beq~~ B type	1000
0000011	~~lw~~ LW	0010
0100011	~~sw~~ SW	0010
1101111	JAL	1011
0010011	I-계열	1001
1100111	JALR	1010
1110011	ECALL	1101

Dispatch ROM 2

Op	Opcode name	Value
0000011	lw	0011
0100011	sw	0101

이때 ROM1과 ROM2는 instruction의 opcode에 따라 다음 stage가 바뀌어야 하기에 구현된 부분으로 PLA처럼 combinational logic으로 구현되며 각 opcode에 따른 다음 stage를 출력하는 간단한 모듈이다. 이렇게 정해진 AddrCtl signal로 6가지의 input에 대해

output을 결정하는 Mux를 사용하여 다음 stage를 정하는 것이 address select logic unit의 주된 역할이라고 할 수 있다. 이를 통해 PLA에 있는 state diagram의 1 state와 2 state에서 다음 단계를 선택할 수 있게 된다.

C. datapath, control unit, storage unit와 clock-synchronous

multi-cycle CPU는 각 클럭에 따라 바뀌는 control에 의해 세심하게 진행된다. 기존 single-cycle CPU에서 control unit이 클럭에 상관없이 instruction에 따라서만 값을 변경하는 asynchronous한 회로였다면 이번 과제의 control unit은 clock-synchronous하다. 이는 control unit 내부의 address select logic 내부에 상태를 저장하는 state register가 있는 것을 통해 알 수도 있다. 또한 memory와 register file을 비롯하여 IR, MDR, A, B, ALUOut은 모두 클럭에 따라 그 값이 변경되기에 clock-synchronous하다. datapath가 클럭에 따라 진행되는 multi-cycle은 각 모듈이 모두 clock-synchronous하게 구현되었다고 말할 수 있다. immgen 모듈과 alucontrol 모듈, mux 모듈들은 combinational logic이지만 이의 출력을 결정하는 입력이 모두 레지스터임을 알아 둘 필요도 있다.

4. discussion

A. multi-cycle CPU와 single-cycle CPU의 비교를 통해 알아보는 multi-cycle CPU가 더 좋은 이유 – 자원 재사용과 시간 이득을 중심으로

single cycle CPU에서는 총 3개의 alu용된다. 하지만 multi cycle CPU에서는 하나의 alu로 가능하다. (이는 아래에서 살펴볼 '각 stage별로 각각의 클락에 나누어 연산하는 multi cycle CPU의 특성'을 이해하면 이유를 알 수 있다.) 따라서 리소스의 사용량 관점에서 보았을 때 multi cycle CPU가 훨씬 경제적이다. 비슷한 맥락에서 memory를 instruction memory와 data memory로 나누어 구현하지 않아도 된다는 점에서 multi cycle cpu의 자원의 경제성을 엿볼 수 있다.

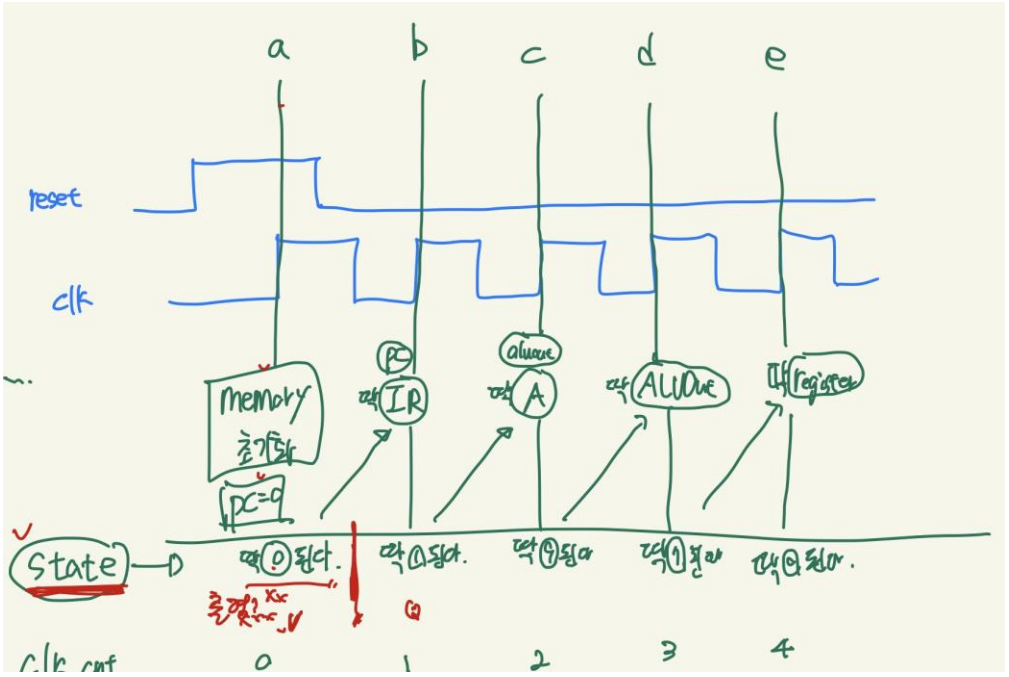
multi cycle CPU가 single cycle CPU에 비해 좋은 점은 이뿐만이 아니다. 어떠한 연산 타입이 오든 single cycle CPU는 항상 CPU가 다루는 어떤 연산 타입 중 필요한 최대 시간만큼을 소요하였다. single cycle CPU의 경우 R type 연산에서 mem 단계를 거치지 않지만 I type에서 mem 단계를 거쳐야 할 일이 있으므로 해당 작업에 필요한 시간만큼을 아무 일을 하지 않고 기다리는 식으로 시간적인 낭비가 생긴다. 이러한 문제점을 해결할 수 있는 것이 multi cycle CPU이다. multi cycle CPU에서는 필요한 단계만을 실행하는 방식을 취한다. 따라서 클락 사이클의 수만을 보면은 single cycle에 비해 크지만, 전체 걸리는 시간은 더 짧아지는 효과를 기대할 수 있

다. 여기서 의문이 생길 수 있다. 클락 사이클의 수가 증가하였는데 전체 프로그램을 끝마치는 것에 필요한 시간이 줄어들 수 있는가? 이는 클락 주기, 즉 클락 진동수의 변화를 줄 수 있기 때문에 가능하다. single cycle CPU에서는 특정 연산 전체가 끝나는 시간 중 가장 긴 시간을 하한으로 하였던 것과 달리, multi cycle CPU에서는 클락 사이클은 가장 긴 stage에서 필요한 시간을 하한으로 하여 클락의 주기가 상당히 짧아진다.(=클락의 진동수가 높아진다.) 이러한 multi cycle CPU의 특성으로 인해 연산에 따라 필요한 스테이지 수가 달라진다. 이는 위에서 제시하였던 예시인 R type과 I type의 경우를 비교하면 R type은 필요 단계가 I type에 비해 하나 적음을 알 수 있다.

multi cycle CPU의 스테이지 당 클락이 하나 할당되는 구조를 이해하면, ALU의 개수를 줄일 수 있게 된다는 것을 알 수 있다. 물론 ALU를 더 추가하여 더 좋은 multi cycle cpu, 가령 pc+4와 pc+imm을 동시에 할 수 있는, 을 만들 수는 있겠지만 요점은 필수적으로 alu가 3개 필요했던, data memory와 instruction memory가 나누어져 있어야만 했던, single cycle CPU와는 달리, multi cycle CPU는 하나의 연산을 위해 여러 클락을 사용할 수 있으므로 하나의 연산에서도 자원에 '다시' 접근할 수 있어 자원을 '재' 사용할 수 있고 따라서 물리적인 자원의 수를 줄일 수 있다는 것이다. single cycle cpu에서는 단일 클락에 모든 단계가 포함되므로 불가능했던, 자원에의 재접근이 가능해진다는 점이 multi cycle의 가장 큰 특징 중 하나인 것이다.

B. reset에 관한 고찰 + console 출력 vs simulation 출력

기본적으로 cpu 구현에 클락의 경계에서 어떤 일이 일어나는지 파악하는 것은 구현에서 중요한 요소이다. 우리 팀은 'cpu가 잘 작동 중인 상황'에 대한 고려만을 하여 multi cycle CPU를 design하여 초기에 대한 분석이 부족하였다. 따라서 이번 부분에서는 우리 팀이 구현에서 애를 먹었던 reset에 대해 알아보며 cpu의 동작 과정을 한 번 더 살펴보겠다.



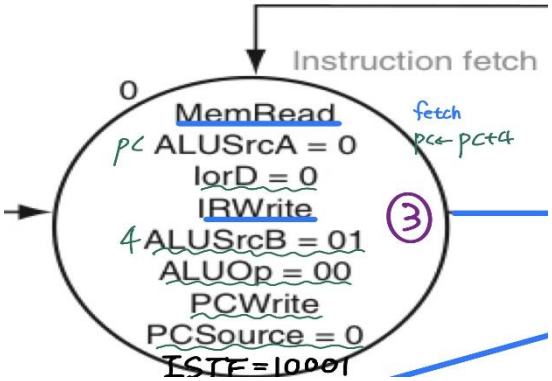
우선 위의 그림을 살펴보자. CPU에 있는 레지스터는 클락이 올라가는 시점에 값이 변화하기 때문에 그림상의 a, b, c, d, e에서만 값이 변화한다. reset이 1일때, 클락이 올라가면 CPU구동을 위한 초기 작업을 한다. 이러한 초기 작업에는,

```
always @(posedge clk) begin
  // Initialize data memory (do not touch)
  if (reset) begin
    for (i = 0; i < MEM_DEPTH; i = i + 1)
      mem[i] = 32'b0;
    // Provide path of the file including instructio
    $readmemh("C:/non-controlflow_mem.txt", mem);
  end
```

```
always @(posedge clk) begin
    if(reset==1'd1)  current_pc <= 0;
    else
```

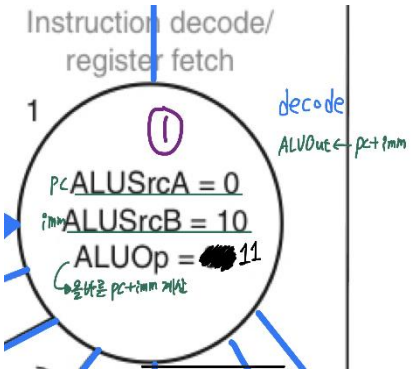
```
always @(posedge clk) begin
    if(reset == 1'd1)
    begin
        state <=4'd0;
    end
```

memory 모듈에서 볼 수 있듯이 모든 값을 0으로 만들기가 있으며, pc 모듈에서 볼 수 있듯이 pc값을 0으로 만들기가 있다. 그리고 정확히 a 시점에서 control unit의 control state 또한 0이 된다. 이를 $display로 출력하여 보면 a 시점에 state는 X값이 나오게 되는데 console창에 출력하는 것인 $display에서는 'a시점 직전'의 값을 출력하기 때문이다. 반면, 파형을 확인할 수 있는 창에서 a 시점에서



state값을 확인해보면 0값이 나오게 되는데 이는 정확히 그 시점에서 어떤 값이 '되는지'를 출력하는 console 창의 특징 때문이다. 비바도에서 사용할 수 있는 두 출력의 특성을 혼동하면, 구현에서 어려움을 겪을 수 있고 그것을 겪었다. console 창에서의 출력은 단순히 c언

어의 printf와 같은 비바도의 display를 활용하여 만들 수 있지만 시뮬레이션 파형을 보는 창에서의 출력을 simulation을 수행하는 모듈에 직접 output을 만들어야 하므로 확인하기 위한 추가작업이 필요하다. 즉, 모듈로부터 값을 출력하도록 선 연결을 해야한다는 것이다. 이 경우 top모듈 하위에 cpu 모듈이 있고 이 모듈 하위에 또 ctrl_unit이 있는데 따라서 총 3개의 모듈의 변경해야 확인할 수 있어 번거롭다. 하지만 더 정확한 정보를 취할 수 있으므로 명확한 분석이 필요할 때 이방법을 사용하면 좋다. 다시 본론으로 돌아가자. reset이 된 경우, 즉 a 시점에서는 메모리 값이 초기화되고 pc값이 초기화된다. 가장 중요한 것은 control unit의 내부 state가 0이 된다는 것이다. 이는 곧 0 state에 해당하는 출력이 생김을 의미하는데 이 출력은 다음 b시점에 활용된다.



0 state에서는 위 그림과 같은 signal을 출력하기 때문에 b시점에서 Instruction register에 값이 write되며, pc값이 pc+4값으로 업데이트된다. 그리고 b 시점이 되는 순간 control unit의 내부 state는 구현한 microcode controller로 인해 1이 된다. 그렇게 되면 control unit은 출력을 위 그림과 같이 1 state에 따른 것으로 바꿀 것이다. 이러한 control signal을 바탕으로 CPU의 다음 상태를 업데이트 할 것이다. 이 장을 작성한 까닭은 control signal이 정확히 어떤 시점에 변화하는지와 어떤 영향을 주는 지 알아보기 위해서이다. a 시점과 b 시점을 통한 설명이 충분했다고 생각하지만 깊은 이해를 위해 첨언하겠다.

1. control signal을 microcode controller로 구현하였기 때문에 내부에 state를 기억하는 register가 존재하고, 그리고 이 state값에 따라 출력이 결정되는 rom 형태를 지닌 모듈도 존재한다. 결국 control unit의 출력은 내부의 state에 전적으로 의존한다. 따라서 레지스터가 클락이 올라가는 시점에서만 바뀔 수 있으므로 control signal은 클락이 올라가는 시점에만 바뀔 수 있다.

2. a 시점에 결정된 state로 인한 control unit의 출력(control signal)이 b시점의 CPU안의 레지스터 값의 변화에 영향을 주었다.(PC<- PC+4와 Instruction Register에 값 저장) 즉, b 시점에 PVS와 그 밖의 구현을 위한 register들이 업데이트 되므로 control

signal의 state 또한 변화한다. control signal의 state 또한 변화는 곧 다음 CPU연산에 또 다른 영향을 줌을 의미한다.

3. b시점에서 결정된 control signal은 c 시점의 CPU 연산에 영향을 주고 이때 control unit의 signal 또한 바뀐다. 이때 바뀐 signal은 d시점 CPU연산에 영향을 줄 것이다.

5. conclusion

single-cycle CPU와의 클락수 비교.

먼저 우리가 구현한 multi-cycle CPU는 이전에 구현했던 single-cycle CPU와 register의 값 변화가 일치하였다. 즉, 우리의 multi cycle CPU는 정상적으로 잘 구현이 되었다. 먼저 basic_mem.txt를 실행했을 때 결과이다. 기존 single-cycle CPU의 경우 28사이클에 실행되었고 multi-cycle CPU는 117사이클에 실행되었다. non-controlflow_mem.txt은 기존 single-cycle CPU의 경우 39사이클에 실행되었고 multi-cycle CPU는 158사이클에 실행되었다. loop_mem.txt은 기존 single-cycle CPU의 경우 222사이클에 실행되었고 multi-cycle CPU는 960사이클에 실행되었다. ifelse_mem.txt은 기존 single-cycle CPU의 경우 34사이클에 실행되었고 multi-cycle CPU는 140사이클에 실행되었다. recursive_mem.txt은 기존 single-cycle CPU의 경우 896사이클에 실행되었고 multi-cycle CPU는 3811사이클에 실행되었다.

각각의 사이클은 multi-cycle CPU가 single-cycle CPU의 사이클 수의 4~5배 정도 되는 것을 확인할 수 있다. 각 instruction이 타입에 따라 다르지만 대체로 4개에서 5개 정도의 state를 거쳐 실행되는 것을 고려하면 사이클 수가 4~5배임을 확인할 수 있다. multi-cycle CPU의 클럭 속도가 single-cycle CPU의 클럭 속도보다 작은 것을 고려하면 전체적인 시간 이득이 있을 것임을 생각해볼 수 있다.

각 stage별로 어떤 시그널이 어떻게 변화해야 하는 지를 직접 설계해가며 구현하면서 개념과 설계의 중요성을 깊게 익힐 수 있었다. 각 instruction에 의해 결정된 data path를 익힐 수 있었고 multi-cycle CPU에서 자원이 어떤 식으로 재활용되는 지를 배울 수 있었다.