lab 5. pipeline – cpu

20190316 유병호

20190065 강두경

Introduction

이번 랩의 목표는 Verilog를 이용해 pipeline-cpu를 구현하는 것이다. 이번에 구현해야할 pipeline-cpu는 총 5개의 isntruction이 pipeline되어있는 cpu이다.

파이프라인을 구현하기 위해, cpu의 동작은 IF, ID, EX, MEM, WB 5개의 stage로 나뉘어진다. 그리고 PC, IF/ID, ID/EX, EX/MEM, MEM/WB 5개의 REGSITER가 추가되어 CPU 내의 각각 PIPELINE되어 있는 서로 다른 instruction들이 다음 단계에서 진행하는데 필요한 데이터 값들을 저장해놓는다.

이렇게 PIPELINE되어 있는 instruction 사이에는 각자의 데이터들이 서로 연관되어 있어 코드가 의도한 대로 instruction이 수행되지 않는 문제점이 발생할 수 있다.

data dependency에는 총 3가지가 있지만, 그중 data hazard를 일으키는 data dependency는 RAW dependency이다. 이 RAW dependency를 해결해주기 위해서 우리는 data forwarding을 수행해주었다. 하지만 data forwarding을 수행하더라도 load instruction에 대한 RAW dependency는 완전히 해결하는 것이 불가능하다. 이 경우에는 한번의 stall과 dataforwarding 을 통해 hazard를 해결하는 것이 가능하다.

jmp, jal, jpr, jrl 과 같은 jump instuction들은 다음 pc에 대한 계산을 ID 단계에서 수행하도록 코드를 작성했다. 그렇기 때문에 jump instuction일 때는 모두 한번씩 stall 된 뒤에, jump instuction의 ID 단계에서 계산된 다음 instuction에 대한 pc로 cpu의 IF 단계를 수행한다. 이러한 jump instuction들도 이전 instuction들과 data hazard가 발생할 수 있는데, 이 data hazard에 대해서는 data forwarding 대신 stall을 시키는 것으로 문제를 해결했다. (수정 될 수 있음)

BRANCHHHHHHHHHHHHHHHHHHHH BRANCH PREDICTORRRRRRRRRRRRRRRRRRRRRR

이렇게 구현된 PIEPELINE의 기본적인 구조는 다음과 같다

IMAGEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE

DESIGN

CONTROL UNIT

CONTROL UNIT은 이전의 SINGLE CYCLE CPU와 MULTI CYCLE CPU에서 만들었던 CONTROL UNIT들과 굉장히 유사하다. CONTROL\_UNIT의 INPUT과 OUTPUT은 다음과 같다.

input [3:0] opcode;

input [5:0] func\_code;

input clk;

input reset\_n;

output branch, alu\_src, mem\_write, mem\_read, mem\_to\_reg;

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

output [1:0] reg\_dst, pc\_src;

output [3:0] alu\_op;

현재 IF 단계의 instuction의 opcode와 func\_code, clk와 reset\_n를 input으로 받는다.

다음은 각 output control bit에 대한 설명이다.

branch : 현재 IF 단계의 instuction이 brach type instruction일 경우 1.  
alu\_src : 0 일때 alu의 두번째 input은 레지스터의 값, 1일 때 immediate.  
mem\_write : 현재 IF 단계의 instuction이 store type일 경우 1. memory에 값 작성을 enable.  
mem\_read : 현재 IF 단계의 instuction이 load type일 경우 1. memory에서 값을 읽어올 수 있도록 한다.  
mem\_to\_reg : 현재 IF 단계의 instuction이 load type일 경우 1. memory의 값을 register에 작성하도록 한다.  
pc\_to\_reg : 현재 IF 단계의 instuction이 jal, jrl 일 경우 1. 현재 pc의 값을 2번 register에 저장하도록 한다.  
halt : 현재 IF 단계의 instuction이 HLT 일 경우.  
wwd : 현재 IF 단계의 instuction이 WWD 일 경우.  
new\_inst : new\_instruction이 수행되었다는 signal bit.  
reg\_write : register에 data를 저장하는 instuction일 때 1.  
alu : alu를 사용하는 instuction일 때 1.  
reg\_dest : 현재 instuction의 destination register를 표현하는 2 bit signal.  
 00 : rd , 01 : rt , 10 : 2  
pc\_src : 현재 instuction이 다음 pc를 결정하는 경우를 나눈 2bit signal.  
 00 : pc+1 , 01 : pc+1+imm (branch) , 10 : imm (jmp, jal) , 11 : rs (jpr, jrl)  
alu\_op : alu에서 수행해야할 operation을 정하는 control.

위 컨트롤 비트들을 살펴보면 직전 lab에서 만들었던 multi-cycle cpu보다는 single-cycle cpu의 control과 흡사함을 알 수 있다. 그 이유는 이번 pipeline cpu에서는 모듈을 재사용을 하지 않기 때문이다.

REGISTER FILE

register file은 4개의 16비트 register를 갖고 있다. register에 대한 wirte는 negedge clk에 일어난다

input clk, reset\_n;

input [1:0] read1;

input [1:0] read2;

input [1:0] dest;

input reg\_write;

input [`WORD\_SIZE-1:0] write\_data;

output [`WORD\_SIZE-1:0] read\_out1;

output [`WORD\_SIZE-1:0] read\_out2;

<input>   
clk : register file의 update timing을 결정한다.  
reset\_n : reset\_n 이 0 일때 register의 모든 값을 0으로 초기화한다.  
read1, read2 : 데이터를 읽어올 target reigster의 index.  
dest : reigster에 data를 작성할 때의 dest\_register (target register)  
reg\_write : 1일 때 dest\_register에 data를 write하는 것을 enable 시킨다.  
write\_data : reg\_write가 1일 때 dest\_register에 작성할 data

<output>  
read\_out1, read\_out2 : register[read1], register[read2] 의 data값.

FORWARDING

data hazard를 일으키는 RAW dependency일 때, 필요한 데이터가 레지스터에 작성되기 전에 사용할 수 있도록 해주는 module이다.

input [1:0] IDEX\_rs, IDEX\_rt, EXMEM\_rdest, MEMWB\_rdest;

input EXMEMC\_regwrite, MEMWBC\_regwrite;

output [1:0] ALU1\_sel, ALU2\_sel;

<input>  
IDEX\_rx, IDEX\_rt : IDEX register에 저장되어 있는 instruction의 rs 와 rt 의 index.  
EXMEM\_rdest : EXMEM register에 저장되어 있는 instruction의 rdest 의 index.  
MEMWB\_rdest : MEMWB register에 저장되어 있는 instruction의 rdest 의 index.  
EXMEMC\_regwrite : EXMEM register에 저장되어 있는 instructon의 regwrite control bit 값.  
MEMWBC\_regwrite : MEMWBC register에 저장되어 있는 instructon의 regwrite control bit 값.  
ALU1\_sel : alu의 input 1으로 들어오게 될 값을 결정하는 2비트 signal.  
ALU2\_sel : alu의 input 1으로 들어오게 될 값을 결정하는 2비트 signal.

HAZARD DETECT

위 forwarding을 통해서 RAW dependency에 의한 data hazard는 대부분 해결되었지만, 아직 load instruction의 경우에는 data를 MEM stage에서 생성할 수 있기 때문에 이부분에 대해서는 해결이 되지 않았다. 따라서 load instruction과 다음 pipeline의 instruction간의 RAW dependency가 있을 경우헤는 한번 stall을 해주어야 한다.

IMPLEMENTATION

CONTROL UNIT

wire br, alui, lwd, swd, jmp, jal, jpr, jrl, rtype;

assign rtype = opcode == 15;

assign branch = opcode == 0 || opcode == 1 || opcode == 2 || opcode == 3;

assign alu = rtype && ~func\_code[5] && ~func\_code[4] && ~func\_code[3]; (alu사용)

assign alui = opcode == 4 || opcode == 5 || opcode == 6; (immediate로 alu 사용)

assign lwd = opcode == 7;

assign swd = opcode == 8;

assign jmp = opcode == 9;

assign jal = opcode == 10;

assign jpr = rtype && func\_code == 25;

assign jrl = rtype && func\_code == 26;

먼저 opcode와 func\_code를 이용해 instruction들을 type에 따라서 분리해주었다. 이를 통해 control\_bit들을 계산할 때 코드의 직관성을 높였다.

    assign reg\_dst[1] = jal || jrl;

    assign reg\_dst[0] = lwd || alui;

    // 00 -> rd, 01 -> rt, 10 -> 2

    assign alu\_src = ~rtype;

    assign mem\_write = swd;

    assign mem\_read = lwd;

    assign mem\_to\_reg = lwd;

    assign pc\_to\_reg = jal || jrl;

    assign pc\_src[1] = jmp || jal || jpr || jrl;

    assign pc\_src[0] = branch || jpr || jrl;

    // 00 : pc+1

    // 01 : pc+1+imm(branch)

    // 10 : imm (jmp, jal)

    // 11 : rs (jpr, jrl)

  assign wwd = rtype && func\_code == 28;

    assign halt = rtype && func\_code == 29;

    assign new\_inst = 1;

    assign reg\_write = alu || alui || lwd || jal || jrl;

    assign alu\_op = alu ? func\_code[2:0] :

                    (opcode == 5) ? 4'd3 :

                    (opcode == 6) ? 4'd8 :

                    (wwd || jpr || jrl) ? 4'd9 : 4'd0;

그리고 위에서 계산된 instuction의 type에 따라서 control bit 들을 계산해준다.

REGISTER FILE

    assign read\_out1 = register[read1];

    assign read\_out2 = register[read2];

reigster에서 data를 읽어오는 것은 assign을 통해 구현했다.

    always @(negedge clk)begin

        if (!reset\_n) begin

            register[0] <= 16'b0;

            register[1] <= 16'b0;

            register[2] <= 16'b0;

            register[3] <= 16'b0;

        end

        else begin

            if (reg\_write == 1)

                register[dest] <= write\_data;

        end

    end

reigster에 대한 write와 update는 negedge clk에서 수행해준다. 그 이유는, cpu의 datapath내부에서 다음 stage로 update가 되는 순간이 posedge clk이기 때문에 한 clk 내에서 register에 대한 write를 수행해 주기 위해선 negedge clk 가 사용되어야 한다.

FORWARDING

FORWARDING MODULE에서는 먼저 instruction 간의 data dependency 가 존재하는지 확인하고, data dependency의 거리에 따라 alu의 input으로 들어가는 data의 forwarding을 결정하는 signal을 생성해준다.

forwarding의 case는 총 6가지 경우가 있다.  
1) MEM의 instruction이 register에 writeback 시키고, MEM의 rdest 와 ID의 rs가 같을 때,  
2) WB의 instruction이 register에 writeback 시키고, WB의 rdest 와 ID의 rs가 같을 때,  
3) register의 internal forwarding (rs에 대한)  
4) MEM의 instruction이 register에 writeback 시키고, MEM의 rdest 와 ID의 rt가 같을 때,  
5) WB의 instruction이 register에 writeback 시키고, WB의 rdest 와 ID의 rt가 같을 때 이다.  
6) register의 internal forwarding (rt에 대한)

    assign ALU1\_sel[1] = EXMEMC\_regwrite && IDEX\_rs == EXMEM\_rdest;

    assign ALU1\_sel[0] = MEMWBC\_regwrite && IDEX\_rs == MEMWB\_rdest;

    assign ALU2\_sel[1] = EXMEMC\_regwrite && IDEX\_rt == EXMEM\_rdest;

    assign ALU2\_sel[0] = MEMWBC\_regwrite && IDEX\_rt == MEMWB\_rdest;

ALU1\_sel 과 ALU2\_sel은 1) 2) case와 4) 5) case의 datapath내에서 dataforwarding을 수행해줄 때 어떤 data 값을 이용해서 data forwarding을 해줄지 결정해는 signal이다.

< datapath 내의 dataforwarding 구현 방식 >

assign ex\_alu\_input1 = ex\_alu\_sel1 == 2'b00 ? IDEX\_REG1 :

                       ex\_alu\_sel1 == 2'b01 ? wb\_writedata :

                       ex\_alu\_sel1 == 2'b10 ? EXMEM\_ALUOUT : IDEX\_REG1;

assign ex\_alu\_input2 = ex\_alu\_temp\_sel2 == 2'b00 ? ex\_alu2\_temp :

                       ex\_alu\_temp\_sel2 == 2'b01 ? wb\_writedata :

                       ex\_alu\_temp\_sel2 == 2'b10 ? EXMEM\_ALUOUT : ex\_alu2\_temp;

datapath에서 구현된 dataforwarding은 다음과 같다.

3) 6) case의 dataforwarding에서 수행해주어야 하는 register internal dataforwarding은 stage의 update timing과 register의 update timing의 차이로 자연스럽게 수행된다. negedge clk에서 update된 register의 데이터는 다음 stage로 posedge clk에서 update되기 전에 combinational logic에 의해서 readout 1과 readout 2로 정확히 전달된다.

HAZARD DETECT

위 forwarding을 통해서