LAB 5. PIPELINED CPU

20190316 유병호

20190065 강두경

INTRODUCTION

이번 랩의 목표는 Verilog 를 이용해 pipelined-cpu 를 구현하는 것이다. 이번에 구현해야 할 pipeline-cpu 는 총 5 개의 isntruction 이 pipelined 되어있는 cpu 이다.

파이프라인을 구현하기 위해, cpu 의 동작은 IF, ID, EX, MEM, WB 5 개의 stage 로 나뉘어진다. 그리고 IF 단계에서 instruction 을 가져오기 위한 주소를 저장하는 PC 레지스터와 IF/ID, ID/EX, EX/MEM, MEM/WB 4 개의 Pipeline Register 가 추가되어 CPU 내의 각각 PIPELINED 되어 있는 서로 다른 instruction 들이 다음 단계에서 진행하는데 필요한 데이터 값들을 저장해놓는다.

이렇게 PIPELINE 되어 있는 instruction 사이에는 각자의 데이터들이 서로 연관되어 있어 코드가 의도한 대로 instruction 이 수행되지 않는 문제점이 발생할 수 있다.

data dependency 에는 총 3 가지가 있지만, 그중 data hazard 를 일으키는 data dependency 는 RAW dependency 이다. 이 RAW dependency 를 해결해주기 위해서 우리는 data forwarding 을 수행해주었다. 하지만 data forwarding 을 수행하더라도 load instruction 에 대한 RAW dependency 는 완전히 해결하는 것이 불가능하다. 이 경우에는 한번의 stall 과 dataforwarding 을 통해 hazard 를 해결하는 것이 가능하다.

jmp, jal, jpr, jrl 과 같은 jump instuction 들은 다음 pc 에 대한 계산을 ID 단계에서 수행하도록 코드를 작성했다. 뿐만 아니라 Branch 또한 branch condition 을 ID 단계에서 계산하도록 설계하였다. 그렇기 때문에 jump/branch instuction 일 때는 모두 한번씩 stall 된 뒤에, jump instuction 의 ID 단계에서 계산된 다음 instuction 에 대한 pc 로 cpu 의 IF 단계를 수행한다. EX 단계에서 발생하는 data hazard 와 같이 ID 단계에서도 data hazard 가 발생할 수 있는데, (Branch, Jump Register 명령어의 경우) 이 경우에도 별도로 data forwarding 을 시키는 구조를 구현하였다.

Branch Predictor 의 경우 Global 2-bit saturation predictor 를 사용하였다. 총 16 비트의 PC 중 하위 8 비트를 index 로 갖고, 상위 8 비트를 tag 로 갖는 size = 256 인 BTB 를 사용하였다.

DESIGN

CONTROL UNIT

CONTROL UNIT 은 이전의 SINGLE CYCLE CPU 와 MULTI CYCLE CPU 에서 만들었던 CONTROL UNIT 들과 굉장히 유사하다. CONTROL_UNIT 의 INPUT 과 OUTPUT 은 다음과 같다.

```
module control_unit (opcode, func_code, is_available, clk, reset_n, branch, re
g_dst, alu_op, alu_src, mem_write, mem_read, mem_to_reg, pc_src, pc_to_reg, ha
lt, wwd, reg_write, alu,jr, use_rs, use_rt, id_use_rs, id_use_rt);
input [3:0] opcode;
    input [5:0] func_code;
    input is_available;
    input clk;
    input reset_n;

    output branch, alu_src, mem_write, mem_read, mem_to_reg;
    output pc_to_reg, halt, wwd, reg_write, alu, jr, use_rs, use_rt, id_use_rs
, id_use_rt;
    output [1:0] reg_dst, pc_src;
    output [3:0] alu_op;
```

Control Unit 을 가져오는 것은 ID 단계에서 진행된다. 따라서 IFID 레지스터에 있는 instruction 으로부터 opcode 와 func_code 를 받아와 다양한 control bit 들을 발생시킨다. clk 와 reset_n 는 input 으로 받지만, register 을 사용하지 않으므로 사용하지 않는다.

Instruction = 0x0000 은 TSC 상에서 BNE $0, $0, 0 을 의미하는데 $0 과 $0 은 항상 같으므로 분기가 성립할 수 없으므로 자명하게 어떤 behavior 도 가지지 않는 no-op 로 볼 수 있다. 본 Pipelined CPU 구현에서는 bubblify 를 할 때마다 instruction 을 0x0000 으로 강제했으며, 이 때는 is_available 을 0 으로 두어 일부 control bit 을 0 으로 강제하도록 한다.

다음은 각종 발생시키는 control bit 들에 대한 설명이다.

Control Bit	설명	대상 Instruction
branch	branch 명령어이다.	BNE, BEQ, BGZ, BLZ
reg_dst	값을 쓸 레지스터를 결정한다. 10 : $2 에 작성 01 : rt 에 작성 00 : rd 에 작성	10 : JRL, JAL 01 : LWD, ADI, ORI, LHI 00 : ALU*
alu_op	ALU 의 연산 방식을 결정한다.	
alu_src	ALU 의 2 번째 input 에 register 가 들어갈지, imm 값이 들어갈지를 결정한다.	ADI, ORI, LHI, LWD, SWD, BRANCH*, JMP, JAL
mem_write	메모리에 값을 작성한다.	SWD
mem_read	메모리에 값을 읽어들인다.	LWD
mem_to_reg	메모리의 값을 읽어들여 레지스터에 저장한다.	LWD
pc_src	다음 PC 값을 어떤 값으로 업데이트할지 결정한다. 00 : PC + 1 01 : PC + imm + 1 (branch) 10 : TARGET (jmp, jal) 11 : RS (jpr, jrl)	
pc_to_reg	pc 값을 register 에 저장할 지를 결정한다.	JAL, JRL
halt	프로그램을 멈춘다.	HLT
reg_write	Register 에 값을 Write-back 한다.	ALU*, ADI, ORI, LHI, LWD, JAL, JRL
alu	opcode == 15 이고, func_code == 0 ~ 8 인 RS, RT 값을 이용해 RD 값을 저장하는 "평범한" instruction 을 의미한다.	ADD, SUB, AND, ORR, NOT, TCP, SHL, SHR
jr	레지스터 값으로 점프한다.	JPR, JRL
use_rs	EX 단계에서 rs 레지스터를 사용한다.	ALU*, ADI, ORI, WWD, LWD, SWD, BRANCH*, JPR, JRL
use_rt	EX 단계에서 rd 레지스터를 사용한다.	ADD, SUB, AND, ORR, SWD, BNE, BEQ
id_use_rs	ID 단계에서 rs 레지스터를 사용한다.	BRANCH*, JPR, JRL
id_use_rt	ID 단계에서 rd 레지스터를 사용한다.	BNE, BEQ

위 컨트롤 비트들을 살펴보면 직전 lab 에서 만들었던 multi-cycle cpu 보다는 single-cycle cpu 의 control 과 흡사함을 알 수 있다. 그 이유는 이번 pipeline cpu 에서는 모듈을 재사용을 하지 않기 때문이다.

REGISTER FILE

register file 은 4 개의 16 비트 register 를 갖고 있다. register 에 대한 wirte 는 negedge clk 에 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 file 의 update timing 을 결정한다.

reset_n : reset_n 이 0 일때 register 의 모든 값을 0 으로 초기화한다.

read1, read2 : 데이터를 읽어올 target reigster 의 index.

dest : reigster 에 data 를 작성할 때의 dest_register (target register)

reg_write : 1 일 때 dest_register 에 data 를 write 하는 것을 enable 시킨다.

write_data : reg_write 가 1 일 때 dest_register 에 작성할 data

<output>

read_out1, read_out2 : register[read1], register[read2] 의 data 값.

FORWARDING

data hazard 를 일으키는 RAW dependency 일 때, 필요한 데이터가 레지스터에 작성되기 전에 사용할 수 있도록 해주는 module 이다.

```
module forwarding(IDEX_rs, IDEX_rt, id_rs, id_rt, id_use_rs, id_use_rt, EXMEM_
rdest, MEMWB_rdest, EXMEMC_regwrite, MEMWBC_regwrite, ALU1_sel, ALU2_sel, id1_
sel, id2_sel);
    input [1:0] IDEX_rs, IDEX_rt, id_rs, id_rt, EXMEM_rdest, MEMWB_rdest;
    input id_use_rs, id_use_rt;
    input EXMEMC_regwrite, MEMWBC_regwrite;
    output [1:0] ALU1_sel, ALU2_sel;
    output [1:0] id1_sel, id2_sel;
```

Forwarding 조건은 EX/MEM 이나 MEM/WB pipeline register 에서 작성하려고 하는 레지스터의 위치가 ID 혹은 EX 에서 Read 하려고 하는 레지스터의 위치와 같을 때 Forwarding 을 시켜줄 수 있다. 또한 EX/MEM 이나 MEM/WB 에서는 레지스터를 작성해야 하고, (regwrite control bit 가 1 이어야 한다) 레지스터를 사용하려는 ID/EX 나 EX/MEM 에서는 그 레지스터를 읽어서 아용해야 한다. (use_rs, use_rt 비트를 사용한다.)

alu1_sel, alu2_sel, id1_sel, id2_sel 이렇게 4 개의 2bit wire output 으로 연결되어 있다. 이 output 에 따라 ID 와 ALU 에서 forwarding 을 할지를 결정한다.

<input>

IDEX_rs, IDEX_rt : IDEX register 에 저장되어 있는 instruction 의 rs 와 rt 의 index.

id_rs, id_rt : ID 단계에서 수행하고 있는 instruction 의 rs 와 rt 의 index

EXMEM_rdest : EXMEM register 에 저장되어 있는 instruction 의 rdest 의 index.

MEMWB_rdest : MEMWB register 에 저장되어 있는 instruction 의 rdest 의 index.

id_use_rs, id_use_rt : ID 단계에서 해당 instruction 을 사용하는 지를 결정.

EXMEMC_regwrite : EXMEM register 에 저장되어 있는 instructon 의 regwrite control bit 값.

MEMWBC_regwrite : MEMWBC register 에 저장되어 있는 instructon 의 regwrite control bit 값.

<output>

ALU1_sel : alu 의 input 1 으로 들어오게 될 값을 결정하는 2 비트 signal.

ALU2_sel : alu 의 input 1 으로 들어오게 될 값을 결정하는 2 비트 signal.

id1_sel : ID 단계에서 연산하는 instruction 에 대해 input 1 로 들어오게 될 값을 결정하는 2bit signal

id2_sel : ID 단계에서 연산하는 instruction 에 대해 input 2 로 들어오게 될 값을 결정하는 2bit signal

ID 단계의 경우 branch 와 jrl, jpr instruction 만이 값을 생성하므로 이 경우에만 forwarding 을 고려해주면 된다. 따라서 id_use_rs, id_use_rt 비트를 별도로 마련하여 이 경우에만 forwarding 이 진행되도록 한다. 반면 EX 단계의 경우 대부분의 instruction 이 rs 를 사용하고, rt 의 경우 forwarding unit 외부에서 rt 레지스터 사용 여부를 처리한다. 또한 ALU 에서 출력된 값은 instruction 에 따라 reg 나 memory 에 write 할 지가 결정되기 때문에, 꼭 use_rs, use_rt 비트를 사용하지 않아도 원치 않는 data forwarding 이 일어나지 않는다.

HAZARD DETECT

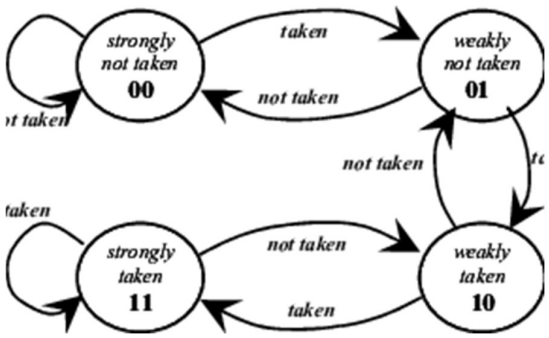
위 forwarding 을 통해서 RAW dependency 에 의한 data hazard 는 대부분 해결되었지만, 아직 load instruction 의 경우에는 data 를 MEM stage 에서 생성할 수 있기 때문에 이부분에 대해서는 해결이 되지 않았다. 따라서 load instruction 과 다음 pipeline 의 instruction 간의 RAW dependency 가 있을 경우에는 한번 stall 을 해주어야 한다.

또한, jpr, jrl, branch 와 같이 ID 단계에서 연산을 수행하는 경우 distance 가 1 일 때 반드시 hazard 가 발생하며, distance 가 2 이면 forwarding 을 통해 hazard 를 해결할 수 있으나 EX/MEM register 의 memread 가 1 이면 이 레지스터로부터 바로 값을 얻어올 수 없으므로 hazard 가 일어난다.

ex 단계의 data hazard 와 id 단계의 data hazard 를 나누어 bit 를 반환한다.

Branch Prediction

Branch 와 Jump 의 경우 분기 예측을 통해 원치 않는 flush 를 줄일 수 있다. IF 단계에서 PC 를 갖고, PC 의 하위 비트로 이루어진 캐시와 유사한 'BTB' 버퍼를 이용해 다음 PC 를 불러온다. 본 Lab 에서는 BTB 로 총 16 비트 PC 의 하위 8 비트를 index 로 사용하였으며 상위 8 비트를 tag 로 사용하였다. 또한 Global 2-bit saturation predictor 을 적용하였다.



그런데 만약 분기 예측이 틀리게 되면 PC 는 ID 단계에서 계산된 실제 목적지 PC 로 이동해야 한다. 따라서 이 때는 BTB 에서 가져온 값이 아닌 input 으로 받은 실제 목적지 PC 를 반환해야 한다. 또한 PC 의 tag 가 BTB 의 PC index 에 해당하는 tag 와 일치하면, (cache hit 과 유사) jump instruction 이거나 Saturation 이 taken 을 가리킬 때 B2B 의 목적지 값을 반환한다.

Branch Predictor 에서 초기에 instruction 의 branch / jump 목적지를 저장하는 것은 소프트웨어 방법으로 해결하거나, reset_n 상태일 때 직접 메모리 주소의 target 을 가져와 직접 다 대입하는 방식으로 해결할 수 있으나 이 부분은 구현하지 않았다. 따라서 처음

시작할 때에는 모든 index 의 tag 는 0000000 이며 next PC 는 PC + 1 을 가리킨다. 하지만 각 instruction 을 만나면서 instruction 이 Branch or Jump type 이면 BTB 에 해당 instruction 과 실제 target 값이 기록된다. 따라서 다음에 같은 instruction 을 만났을 때 BTB tag 와 일치하게 된다면 분기 예측을 성공적으로 진행할 수 있게 된다.

Datapath

본 pipelined cpu 의 핵심 기능을 담당하고 있는 본체 모듈이다. 이 모듈은 control unit, register file, branch predictor, hazard detection unit, forwarding unit, alu unit 모듈을 포함하고 있으며 또한 4 개의 파이프라인 레지스터도 포함하고 있다. Pipelined CPU 이기 때문에 모든 instruction 은 IF, ID, EX, MEM, WB 의 다섯 단계로 이루어지며 각 단계에 따라 분리된 행동을 취한다.

- IF (Instruction Fetch)
  - PC 값을 index 로 하여 memory 에 접근하여 instruction 을 받아 온다.
  - PC 값이 Branch Predictor 로 입력되어 다음 PC 를 전달받아 저장한다.
- ID (Instruction Decode)
  - Instruction 을 rs index, rt index, rd index, immediate, jump target, opcode, func_code 등으로 분리한다.
  - Register File 에 접근하여 rs, rt, rd 레지스터의 값을 구한다.
  - Instruction 의 opcode 와 func_code 로부터 control bit 들을 계산한다.
  - Branch instruction 의 경우 분기가 taken 되었는지를 검사한다.
  - Branch, jmp, jpr 을 포함하여 모든 instruction 에 대해 실제로 다음 이동할 PC 값을 구한다.
- EX (Execution)
  - RS, RT, Immediate 값을 토대로 ALU 에서 메인 연산을 수행한다.
- MEM (Memory)
  - LWD 의 경우 메모리에서 값을 읽어 온다.
  - SWD 의 경우 register 의 값을 메모리에 저장한다.

- WB (Write-Back)
  - 레지스터에 EX 단계에서 구한 값을 저장한다.
  - JAL, JPR 의 경우 EX 단계에서 구한 값 대신 PC + 1 값을 $2 레지스터에 저장한다.

ALU & BRANCH_ALU

```
module alu (A, B, func_code, alu_out);
    input [`WORD_SIZE-1:0] A;
    input [`WORD_SIZE-1:0] B;
    input [3:0] func_code;
```

ALU 는 연산을 처리하는 EX 단계의 모듈이다. 이 모듈은 control unit 으로부터 alu_op(func_code) 값을 받아 그 값에 따라 서로 다른 연산을 취해준다. TSC 에 존재하는 모든 연산을 10 가지로 압축하였다. alu 의 output 은 combinational logic 에 의해 계산되어 연결된다.

Branch ALU 는 ID 단계에서 Branch 의 진위 여부를 판별하기 위한 소형 ALU 이다. opcode 에 따라 A, B 가 같은지 혹은 다른지, A 가 0 보다 큰지 작은지를 판별하여 bcond 를 구한다.

IMPLEMENTATION

CONTROL UNIT

```
assign rtype = opcode == 15;
    assign branch = is_available && opcode == 0 || opcode == 1 || opcode == 2
|| opcode == 3;
    assign alu = rtype && ~func_code[5] && ~func_code[4] && ~func_code[3];
    assign alui = opcode == 4 || opcode == 5 || opcode == 6;
    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;
    assign jimm = jmp || jal;
    assign jr = jpr || jrl;
```

먼저 opcode 와 func_code 를 이용해 instruction 들을 type 에 따라서 분리해주었다. 비슷한 것끼리 묶는 작업을 통해 control_bit 들을 계산할 때 코드의 직관성과 가독성을 높이고 자주 사용되는 bit 을 재사용함으로써 simplification 효과를 얻을 수 있다.

```
    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] = jimm || jr;
    assign pc_src[0] = is_available && branch || jr; //PC + 1, branch, jmp, jr
    assign wwd = rtype && func_code == 28;
    assign halt = rtype && func_code == 29;
    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 || jr) ? 4'd9 : 4'd0;
    assign id_use_rs = branch || jr;
    assign id_use_rt = is_available && (opcode == 0 || opcode == 1); //bne, be
q
    assign use_rs = !(opcode == 6 || jmp || jal || halt);
    assign use_rt = rtype && !func_code[3] && !func_code[2] || mem_write || id
_use_rt;
```

그리고 위에서 계산된 instuction 의 type 에 따라서 control bit 들을 계산해준다. Design 에서 설명한 각 control unit 의 해당 instruction 과 동일한 instruction 에 대해서만 각 control bit 가 1 이 되도록 한다. is_available 의 경우 instruction 이 0x000 일 때만 0 을 반환하는데, 이 경우 완전히 '빈 명령어'로 약속하기에 원래였으면 1 이 되었을 c_branch 나 id_use_rs, id_use_rt 도 모두 0 이 된다.

REGISTER FILE

```
    assign read_out1 = register[read1];
    assign read_out2 = register[read2];
```

reigster 에서 data 를 읽어오는 것은 assign 을 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
```

reigster 에 대한 write 와 update 는 negedge clk 에서 수행해준다. 그 이유는, cpu 의 datapath 내부에서 다음 stage 로 update 가 되는 순간이 posedge clk 이기 때문에 한 clk 내에서 register 에 대한 write 를 수행해 주기 위해선 negedge clk 가 사용되어야 한다.

FORWARDING

FORWARDING MODULE 에서는 먼저 instruction 간의 data dependency 가 존재하는지 확인하고, data dependency 의 거리에 따라 alu 의 input 으로 들어가는 data 의 forwarding 을 결정하는 signal 을 생성해준다. Forwarding 은 다시 EX 단계의 Forwarding 과 ID 단계의 Forwarding 으로 나뉜다.

EX 단계의 forwarding 의 case 는 총 6 가지 경우가 있다.

1) MEM 의 instruction 이 register 에 writeback 시키고, MEM 의 rdest 와 ID 의 rs 가 같을 때,

2) WB 의 instruction 이 register 에 writeback 시키고, WB 의 rdest 와 ID 의 rs 가 같을 때,

3) register 의 internal forwarding (rs 에 대한)

4) MEM 의 instruction 이 register 에 writeback 시키고, MEM 의 rdest 와 ID 의 rt 가 같을 때,

5) WB 의 instruction 이 register 에 writeback 시키고, WB 의 rdest 와 ID 의 rt 가 같을 때 이다.

6) register 의 internal forwarding (rt 에 대한)

```
assign ALU1_sel[1] = ex_use_rs && EXMEMC_regwrite && IDEX_rs == EXMEM_rdest;
assign ALU1_sel[0] = ex_use_rs && MEMWBC_regwrite && IDEX_rs == MEMWB_rdest;
assign ALU2_sel[1] = ex_use_rt && EXMEMC_regwrite && IDEX_rt == EXMEM_rdest;
assign ALU2_sel[0] = ex_use_rt && MEMWBC_regwrite && IDEX_rt == MEMWB_rdest;
```

ALU1_sel 과 ALU2_sel 은 1) 2) case 와 4) 5) case 의 datapath 내에서 dataforwarding 을 수행해줄 때 어떤 data 값을 이용해서 data forwarding 을 해줄지 결정해는 signal 이다.

```
assign ex_alu2_temp = IDEXC_ALUSRC ? IDEX_IMM : IDEX_REG2;
assign ex_alu_input1 = ex_alu_sel1 == 2'b01 ? wb_writedata :
                       ex_alu_sel1 == 2'b10 ? ex_writedata : IDEX_REG1;
assign ex_alu_input2 = ex_alu_sel2 == 2'b01 ? wb_writedata :
                       ex_alu_sel2 == 2'b10 ? ex_writedata : ex_alu2_temp;
```

datapath 에서는 이렇게 alu_sel 값을 통해 ALU 의 input 에 RF 에서 받아온 값을 넣을지, EXMEM 에서 write 할 값을 넣을 지, MEMWB 에서 write 할 값을 넣을 지를 결정한다. 끝으로

6) case 의 dataforwarding 에서 수행해주어야 하는 register internal dataforwarding 은 stage 의 update timing 과 register 의 update timing 의 차이로 자연스럽게 수행된다. negedge clk 에서 update 된 register 의 데이터는 다음 stage 로 posedge clk 에서 update 되기 전에 combinational logic 에 의해서 readout 1 과 readout 2 로 정확히 전달된다.

ID 단계의 forwarding 도 위와 매우 유사하다. 레지스터에 작성될 결과 값은 EX 단계에서 생성되기 때문에 마찬가지로 EX/MEM 레지스터에서 값을 forwarding 하거나, MEM/WB 레지스터에서 값을 forwarding 하거나, Register File 에서 값을 Internal Forwarding 애 주어야 한다. 따라서 EX 단계의 forwarding 과 똑같은 6 개의 단계로 나눌 수 있다.

```
assign id1_sel[1] = id_use_rs && EXMEMC_regwrite && id_rs == EXMEM_rdest;
assign id1_sel[0] = id_use_rs && MEMWBC_regwrite && id_rs == MEMWB_rdest;
assign id2_sel[1] = id_use_rt && EXMEMC_regwrite && id_rt == EXMEM_rdest;
assign id2_sel[0] = id_use_rt && MEMWBC_regwrite && id_rt == MEMWB_rdest;
```

ID 단계의 data forwarding 은 다음과 같다.

```
assign id_bj_input1 = id1_sel == 2'b01 ? wb_writedata :
                      id1_sel == 2'b10 ? ex_writedata : id_reg1;
assign id_bj_input2 = id2_sel == 2'b01 ? wb_writedata :
                      id2_sel == 2'b10 ? ex_writedata : id_reg2;
```

ex_writedata 와 wb_writedata 는 각각 다음의 값으로 assign 된다.

```
assign ex_writedata = EXMEMC_PCTOREG ? EXMEM_PC + 1 : EXMEM_ALUOUT;
assign wb_writedata = MEMWBC_PCTOREG ? MEMWB_PC + 1 : MEMWB_OUT;
```

HAZARD DETECT

많은 data hazard 는 forwarding 으로 처리할 수 있지만, 그럼에도 불구하고 hazard 가 일어난다.

EX 단계에서의 hazard 는 바로 앞선 명령어가 RAW dependency 를 일으키는 LWD instruction 일 때 발생한다. 이 경우 결과값은 MEM 단계에서 produce 되기 때문에 EXMEM register 에서는 forwarding 할 수 없다. 이 경우에는 어쩔 수 없이 한 단계를 stall 하게 된다.

ID 단계에서의 hazard 는 앞선 명령어와 RAW dependency 를 일으킬 때나, 두 번 앞선 명령어가 RAW dependency 를 일으키는 LWD instruction 일 때 발생한다. EX 단계보다 한 단계 더 간다고 간주할 수 있다. 마찬가지로 이 경우 한 단계를 stall 하게 된다.

output 으로는 EX 단계의 hazard 와 ID 단계의 hazard 를 따로 출력한다.

```
input [1:0] id_rs, id_rt;
input id_users, id_usert, id_jpr, id_branch, idex_memread, exmem_memread;
input [1:0] idex_rd, exmem_rd;
input idex_rw, exmem_rw;

output ex_datahazard;
output id_datahazard;
wire ieh, emh;

assign ieh = (idex_rd == id_rs && id_users || idex_rd == id_rt && id_usert) && idex_rw;
assign emh = (exmem_rd == id_rs && id_users || exmem_rd == id_rt && id_usert) && exmem_rw;
assign ex_datahazard = idex_memread && ieh;
assign id_datahazard = (id_jpr || id_branch) && (ieh || emh && exmem_memread);
```

BRANCH PREDICTION

PC 의 하위 8 비트를 Index 를, 상위 8 비트를 Tag 로 사용하는 BTB 를 구현하였다. reset_n 이 0 일 때 이 BTB 를 초기화한다. 처음에는 일단 모든 entry 의 target 을 PC + 1 로 결정한다. 당연하겠지만, 이 경우 처음에 tag 가 일치하는 branch 나 jump 를 만나게 되면 taken 으로 예측을 해도 분기에 실패하게 된다. 하지만 분기 실패 여부는 PC 값의 차이로 검사하기 때문에 전체 behavior 에는 아무 지장이 없으며, 이후에는 branch 나 jump 에 대해 tag 와 target 값이 latch 되기 때문에 해당 index 에 대해서는 제대로 BTB 를 사용할 수 있다. 예외적으로 PC = 0 은 35 번째 instruction 으로 이동하는 jump 문이기 때문에 TARGET[0] = 35 로 설정한다.

```
if (!reset_n) begin
            for (i = 1; i < 256; i = i + 1) begin
                TAG[i] <= 0;
                TARGET[i][15:8] <= 0;
                TARGET[i][7:0] <= i + 1;
            end
            TAG[0] <= 0;
            TARGET[0] <= 35;
            SATURATION <= 2;
        end
```

또한 Global 2-bit Saturation Predictor 을 사용하는데, 이는 branch 문일 때만 값이 변화한다. TAG 와 TARGET 의 경우 branch, jump 문일 때 모두 값이 변화한다. 단 data hazard 상태일 경우에는 업데이트하지 않는다. branch 가 data hazard 문에 걸려 한 stage 이상 stall 하는

경우 bcond 값이 제대로 계산되지 않았음에도 불구하고 saturation predictor 가 움직이기 때문에 이 상황을 방지하기 위함이다.

```
else begin
    if ((is_branch || is_jump) && !datahazard) begin
        if (is_branch) begin
            if (taken)
                SATURATION <= SATURATION == 3 ? 3 : SATURATION + 1;
            else
                SATURATION <= SATURATION == 0 ? 0 : SATURATION - 1;
        end
        TAG[actual_pc_idx] <= actual_pc_tag;
        TARGET[actual_pc_idx] <= actual_next_PC;
    end
end
```

ALU

ALU 는 alu_op 값에 의존하여 값을 계산한다. 이 행동은 다음과 같다.

```
always @(*) begin
    case (func_code)
        0 : alu_out = A + B;
        1 : alu_out = A - B;
        2 : alu_out = A & B;
        3 : alu_out = A | B;
        4 : alu_out = ~A;
        5 : alu_out = -A;
        6 : alu_out = A << 1;
        7 : alu_out = A >> 1;
        8 : alu_out = B << 8;
        9 : alu_out = A;
        default: alu_out = A + B;
    endcase
end
```

ALU_OP 는 control_unit 에서 결정한다.

```
assign alu_op = alu ? func_code[2:0] :
                (opcode == 5) ? 4'd3 :
                (opcode == 6) ? 4'd8 :
                (wwd || jr) ? 4'd9 : 4'd0;
```

우선, opcode 가 15 이고 func_code 가 0~7 인 명령어의 경우 각 명령어에 맞는 연산 방법을 사용한다. opcode = 5 는 ORI 명령어로 OR 연산을 사용하며, 이는 opcode == 15 이고 func_code == 3 인 ORR 연산과 행동이 동일하다. opcode = 6 은 LHI 명령어로 imm << 8 연산을 수행한다. wwd 와 jr 의 경우 rs 의 값이 그대로 들어간다. (jr 의 경우 id 단계에서 바로

pc 로 이동하기 때문에 사실 don't care 와 다름없다.) 기타 모든 instruction 의 경우에는 + 연산을 수행하면 되며, alu 연산고 ㅏ무관하다. (lwd, swd 등)

Datapath

Datapath 는 IF, ID, EX, MEM, WB 단계를 따라 데이터가 이동하는 흐름을 나타내고 있다. Datapath 모듈은 총 4 개의 Pipeline Register 을 포함하며, 각 pipeline register 은 다음 값을 갖고 있다.

IF/ID	ID/EX	EX/MEM	MEM/WB
PC	PC	PC	PC
INSTRUCTION	INSTRUCTION	INSTRUCTION	INSTRUCTION
	RDEST	RDEST	RDEST
	REG2	REG2	
	REG1		
	IMMEDIATE	ALUOUT	MEMOUT
	FLUSH	ALUIN2	OUT
	RS		
	RT		
	ALUSRC		
	REGWRITE	REGWRITE	REGWRITE
	MEMWRITE	MEMWRITE	
	MEMREAD	MEMREAD	
	MEMTOREG	MEMTOREG	MEMTOREG
	PCTOREG	PCTOREG	PCTOREG
	WWD	WWD	
	NEWINST	NEWINST	NEWINST
	ALU		WWD
	HALTED	HALTED	HALTED
	AVAILABLE	AVAILABLE	AVAILABLE
	USERS		
	USERT		
	REGDST		
	ALUOP		

각 pipeline register 은 posedge clk 시점에 이전 pipeline register 의 해당하는 값으로 latch 된다. 단, data hazard 가 일어나면 IFID register 와 IDEX register 는 0 이 되거나 데이트하지

는다. IDEX register 을 0 으로 만드는 것은 이후의 모든 control bit 을 0 으로 만들어 아무 일도 하지 않는 operator 로 만들어 버리는 것과 같다.

INSTRUCTION FETCH

Instruction Fetch 단계의 대표적인 작업은 Memory 로부터 PC 에 해당하는 Instruction 을 인출하는 작업이다. 본 lab 에서는 address_1 을 PC Register 값, read_m 을 항상 1 로 놓아 항상 instruction 을 인출하게 했고, posedge clk 일 때 IFID_INSTR 레지스터에 메모리에서 읽어들인 데이터(flush 일 경우 0x0000)를 latch 한다.

다음으로, Instruction Fetch 단계에서 하는 중요한 일은 분기 예측이다. PC 를 Branch Predictor 에 연결해 BTB 의 target 혹은 PC + 1 을 저장한다. 이를 위해 IF 단계의 PC 레지스터와 Branch Predictor 을 연결한다.

INSTRUCTION DECODE

본 설계의 Instruction Decode 단계에서는 많은 양의 작업을 수행한다. 먼저 Instruction 을 rs index, rt index, rd index, immediate, jump target, opcode, func_code 등으로 분리한다.

```
assign id_instruction = IFID_INSTR;
assign id_instr_opcode = id_instruction[15:12];
assign id_instr_rs = id_instruction[11:10];
assign id_instr_rt = id_instruction[9:8];
assign id_instr_rd = id_instruction[7:6];
assign id_instr_func = id_instruction[5:0];
assign id_instr_jmp = id_instruction[11:0];
assign id_immediate[7:0] = id_instruction[7:0];
assign id_immediate[15:8] = id_instruction[7] == 1 ? 8'hff : 8'h00;
```

다음으로, Register File 에 접근하여 rs, rt, rd 레지스터의 값을 구한다. 이를 위해 앞서 만들어 놓은 Register File 모듈과 연결한다. Instruction 의 opcode 와 func_code 로부터 control bit 들을 계산한다. 이를 위해 앞서 만들어 놓은 Control Unit 과 Register File 과 연결한다.

끝으로 Branch instruction 의 경우 분기가 taken 되었는지를 검사하고, Branch, jmp, jpr 을 포함하여 모든 instruction 에 대해 실제로 다음 이동할 PC 값을 구한다. Branch Taken 여부는 Branch ALU 모듈을 이용해 확인한다. 실제 PC 값은 다음과 같은 방법으로 구현하였다.

```
assign id_next_pc_branch = id_bcond ? IFID_PC + id_immediate + 1 : IFID_PC + 1;
assign id_next_pc = IFID_PC + 1;
assign id_next_pc_jmp[11:0] = id_instr_jmp;
assign id_next_pc_jmp[15:12] = 4'h0;
```

```
assign id_next_pc_jalr = id_bj_input1;

assign actual_next_pc = (PRE_PC == 0 || IDEX_FLUSH) ? PRE_PC :
                        c_pcsrc == 1 ? id_next_pc_branch :
                        c_pcsrc == 2 ? id_next_pc_jmp :
                        c_pcsrc == 3 ? id_next_pc_jalr : id_next_pc;
assign is_flush = PRE_PC != actual_next_pc;
```

flush 는 현재 명령어를 인출하고 있는 IF 단계의 PC register 가 이전 instruction 의 '정답' instruction 과 다를 때 일어난다. 예를 들어 branch 에서 틀렸다고 분기 예측을 하여 PC + 1 로 이동을 했지만 실제로 branch 는 taken 되어 branch target 으로 이동해야 할 때 flush 가 일어나 명령어를 다 지우게 된다. 그 경우에는 actual next pc 를 PC 레지스터와 같은 값을 유지하여 flush 가 끝나고 안전하게 다음 instruction 부터 받아올 수 있도록 한다.

Branch 및 JALR 에 사용되는 값은 data hazard 를 고려하여 forwarding 받아온 값을 이용한다. (Forwarding 참조)

EXECUTION

메인 연산 모듈인 ALU 모듈을 사용하여 연산한다. 본 Lab 에서 EX 단계의 작업은 비교적 간단하게 이루어진다. 역시 data hazard 를 처리하기 위해 forwarding 받아온 값을 이용한다. (Forwarding 참조)

MEMORY

MEM 단계는 LWD, SWD instruction 을 처리하기 위해 메인 메모리에 접근하는 stage 이다. read_m2, write_m2 는 EX/MEM pipeline register 의 control bit 값을 그대로 가져온다. LWD 의 경우 가져온 값 data2 를 memory_output 에 연결하고, SWD 의 경우 data2 에 rt 값을 저장할 수 있도록 한다.

```
assign data2 = write_m2 ? EXMEM_REG2 : 16'bz;
...
MEMWB_MEMOUT <= data2;
```

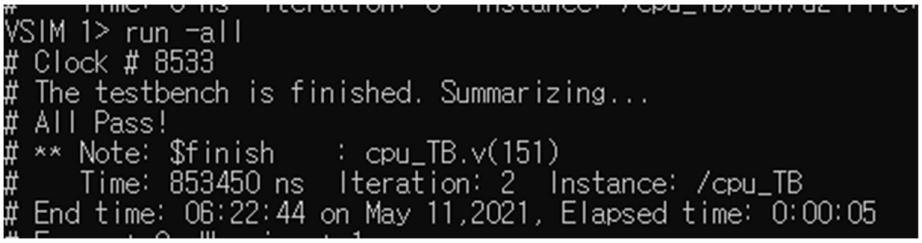
WRITEBACK

Writeback 단계는 값을 Register 에 다시 쓰는 단계이다. Pipeline Register 을 따라 진행된 reg_write, pc_to_reg, rdest(dest register)에 의존하여 RF 에서 쓰기를 진행한다. jal, jrl 의 경우 예외적으로 $2 레지스터에 ALU 나 MEM 에서 계산된 값 대신 pc + 1 을 저장한다. RF 의 쓰기 과정은 negedge clk 에서 진행된다.

DISCUSSION

Pipelined vs Multi-Cycle CPU 성능 비교

앞서 작성한 Multi-Cycle CPU 의 경우 총 6533 cycle 을 소요하였다.



반면, 이번에 작성한 Pipelined CPU 는 총 1279 cycle 을 소요한다.

```
# Clock # 1279
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : cpu_TB.v(153)
#    Time: 128050 ns  Iteration: 2  Instance: /cpu_TB
# End time: 05:47:07 on May 11,2021, Elapsed time: 0:00:02
# Errors: 0, Warnings: 0
```

Multi Cycle 에 비해 약 5 배 정도 향상된 성능을 보인다는 것을 알 수 있다.

Multi Cycle 은 여러 개의 cycle 에 걸쳐 하나의 instruction 을 처리하고 각 instruction 이 microinstruction 으로 쪼개지긴 하지만, 결국 하나의 instruction 만을 한 번에 처리하므로 throughput 이 낮다. 반면 pipelined cpu 는 한 instruction 을 5 단계로 나누기 throughput 도 일렬로 수행할 때보다 최대 5 배 증가함을 알 수 있다.

Branch Predictor 별 성능 비교

본 CPU 는 Branch Predictor 을 사용한다. Branch Predict 를 어떻게 하느냐에 따라 성능이 소폭 달라지는 것을 알 수 있다.

방법	Cycle
Always Not Taken	1296
Always Taken	1200
2-bit Global Saturation	1279

결과를 보면 Always Taken 이 가장 적은 cycle 만에 끝나는 것을 확인할 수 있었다. 주어진 testbench 의 경우 taken 되는 branch 가 더 많은 것을 알 수 있었다. global saturation 의 bit 수에 변화를 주었으나 1200 과 1296 사이의 cycle 을 얻었다.

개선점 및 주안점

우선, Branch 및 jrl, jpr instruction 을 ID 단계로 옮겼다. 이는 Branch, jpr, jrl 로 인해 발생하는 stall 을 한 stage 줄이는 데에 효과적이다. 하지만 이 이전의 instruction 에 의해 data hazard 가 발생할 경우 EX 단계에 비해 한 단계 더 hazard 가 발생하기 때문에 이 경우에는 시간 지연이 똑같다. 또한 그 결과 EX 단계에 비해 ID 단계에서 일을 더 많이 수행하게 되었고, EX 단계에서 처리하는 것에 비해 hazard, forwarding 등의 설계가 다소 복잡해졌다.

처음에는 EX 단계의 ALU 연산에 대해서만 forwarding 을 구현하고 ID 단계 연산의 경우 forwarding 을 구현하지 않으려 했지만, 그 결과 총 cycle 이 1539 정도를 얻었다. 이는 forwarding 을 구현하기 전과 큰 차이다. 실제로 testbench 후반에 갈 수록 jmp 와 branch 의 비율이 많아지고 data hazard 도 잦아지기 때문에 stall 하나가 치명적이다.

CONCLUSION

본 lab 에서는 TSC 명령어를 IF, ID, EX, MEM, WB 다섯 단계로 나누어 Pipeline 화하는 CPU 를 구현하였다. branch 와 jump register 목적지는 ID 단계에서 연산하였고 Data Hazard 를 해결하기 위한 Data Forwarding 및 분기로 인한 시간 지연을 막기 위한 Branch Predictor 을 도입하였다.

Pipelined CPU Lab 을 통해 현대 CPU 구조의 핵심이라 할 수 있는 Pipelined CPU 에 대해 심도 깊게 이해할 수 있었다. 구현 및 문제 해결뿐만 아니라 더 좋은 디자인을 고민하는 과정에 많은 시간과 노력을 소모하였다. 힘든 과제였지만 소중한 경험으로 남을 것이라 확신한다.