lab 5. pipelined cpu

20190316 유병호

20190065 강두경

# Introduction

이번 랩의 목표는 Verilog를 이용해 pipelined-cpu를 구현하는 것이다. 이번에 구현해야 할 pipeline-cpu는 총 5개의 isntruction이 pipelined되어있는 cpu이다.

파이프라인을 구현하기 위해, cpu의 동작은 IF, ID, EX, MEM, WB 5개의 stage로 나뉘어진다. 그리고 IF 단계에서 instruction을 가져오기 위한 주소를 저장하는 PC 레지스터와 IF/ID, ID/EX, EX/MEM, MEM/WB 4개의 Pipeline Register가 추가되어 CPU 내의 각각 PIPELINED되어 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. 뿐만 아니라 Branch 또한 branch condition을 ID 단계에서 계산하도록 설계하였다. 그렇기 때문에 jump/branch instuction일 때는 모두 한번씩 stall 된 뒤에, jump instuction의 ID 단계에서 계산된 다음 instuction에 대한 pc로 cpu의 IF 단계를 수행한다. EX 단계에서 발생하는 data hazard와 같이 ID 단계에서도 data hazard가 발생할 수 있는데, (Branch, Jump Register 명령어의 경우) 이 경우에도 별도로 data forwarding을 시키는 구조를 구현하였다.

Branch Predictor의 경우 Global 2-bit saturation predictor를 사용하였다. 총 16비트의 PC 중 하위 8비트를 index로 갖고, 상위 8비트를 tag로 갖는 size = 256인 BTB를 사용하였다.

# DESIGN

## CONTROL UNIT

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

module control\_unit (opcode, func\_code, is\_available, clk, reset\_n, branch, reg\_dst, alu\_op, alu\_src, mem\_write, mem\_read, mem\_to\_reg, pc\_src, pc\_to\_reg, halt, 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 Unit을 가져오는 것은 ID 단계에서 진행된다. 따라서 IFID 레지스터에 있는 instruction으로부터 opcode와 func\_code를 받아와 다양한 control bit들을 발생시킨다. clk와 reset\_n는 input으로 받지만, register을 사용하지 않으므로 사용하지 않는다.

Instruction = 0x0000은 TSC 상에서 BNE $0, $0, 0을 의미하는데 $0과 $0은 항상 같으므로 분기가 성립할 수 없으므로 자명하게 어떤 behavior도 가지지 않는 no-op로 볼 수 있다. 본 Pipelined CPU 구현에서는 bubblify를 할 때마다 instruction을 0x0000으로 강제했으며, 이 때는 is\_available을 0으로 두어 일부 control bit을 0으로 강제하도록 한다.

다음은 각종 발생시키는 control bit들에 대한 설명이다.

|  |  |  |
| --- | --- | --- |
| Control Bit | 설명 | 대상 Instruction |
| branch | branch 명령어이다. | BNE, BEQ, BGZ, BLZ |
| reg\_dst | 값을 쓸 레지스터를 결정한다.  10 : $2에 작성  01 : rt에 작성  00 : rd에 작성 | 10 : JRL, JAL  01 : LWD, ADI, ORI, LHI  00 : ALU\* |
| alu\_op | ALU의 연산 방식을 결정한다. |  |
| alu\_src | ALU의 2번째 input에 register가 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 register에 저장할 지를 결정한다. | JAL, JRL |
| halt | 프로그램을 멈춘다. | HLT |
| reg\_write | Register에 값을 Write-back한다. | ALU\*, ADI, ORI, LHI, LWD, JAL, JRL |
| alu | opcode == 15이고, func\_code == 0 ~ 8인 RS, RT 값을 이용해 RD 값을 저장하는 “평범한” instruction을 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 단게에서 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 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.

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/MEM이나 MEM/WB pipeline register에서 작성하려고 하는 레지스터의 위치가 ID 혹은 EX에서 Read하려고 하는 레지스터의 위치와 같을 때 Forwarding을 시켜줄 수 있다. 또한 EX/MEM이나 MEM/WB에서는 레지스터를 작성해야 하고, (regwrite control bit가 1이어야 한다) 레지스터를 사용하려는 ID/EX나 EX/MEM에서는 그 레지스터를 읽어서 아용해야 한다. (use\_rs, use\_rt 비트를 사용한다.)

alu1\_sel, alu2\_sel, id1\_sel, id2\_sel 이렇게 4개의 2bit wire output으로 연결되어 있다. 이 output에 따라 ID와 ALU에서 forwarding을 할지를 결정한다.

**<input>**  
IDEX\_rs, IDEX\_rt : IDEX register에 저장되어 있는 instruction의 rs 와 rt 의 index.

id\_rs, id\_rt : ID 단계에서 수행하고 있는 instruction의 rs와 rt의 index  
EXMEM\_rdest : EXMEM register에 저장되어 있는 instruction의 rdest 의 index.  
MEMWB\_rdest : MEMWB register에 저장되어 있는 instruction의 rdest 의 index.

id\_use\_rs, id\_use\_rt : ID 단계에서 해당 instruction을 사용하는 지를 결정.  
EXMEMC\_regwrite : EXMEM register에 저장되어 있는 instructon의 regwrite control bit 값.  
MEMWBC\_regwrite : MEMWBC register에 저장되어 있는 instructon의 regwrite control bit 값.

**<output>**  
ALU1\_sel : alu의 input 1으로 들어오게 될 값을 결정하는 2비트 signal.  
ALU2\_sel : alu의 input 1으로 들어오게 될 값을 결정하는 2비트 signal.

id1\_sel : ID 단계에서 연산하는 instruction에 대해 input 1로 들어오게 될 값을 결정하는 2bit signal

id2\_sel : ID 단계에서 연산하는 instruction에 대해 input 2로 들어오게 될 값을 결정하는 2bit signal

ID 단계의 경우 branch와 jrl, jpr instruction 만이 값을 생성하므로 이 경우에만 forwarding을 고려해주면 된다. 따라서 id\_use\_rs, id\_use\_rt 비트를 별도로 마련하여 이 경우에만 forwarding이 진행되도록 한다. 반면 EX 단계의 경우 대부분의 instruction이 rs를 사용하고, rt의 경우 forwarding unit 외부에서 rt 레지스터 사용 여부를 처리한다. 또한 ALU에서 출력된 값은 instruction에 따라 reg나 memory에 write할 지가 결정되기 때문에, 꼭 use\_rs, use\_rt 비트를 사용하지 않아도 원치 않는 data forwarding이 일어나지 않는다.

## HAZARD DETECT

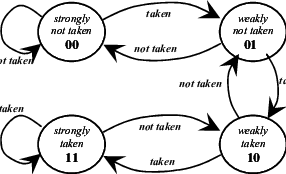
위 forwarding을 통해서 RAW dependency에 의한 data hazard는 대부분 해결되었지만, 아직 load instruction의 경우에는 data를 MEM stage에서 생성할 수 있기 때문에 이부분에 대해서는 해결이 되지 않았다. 따라서 load instruction과 다음 pipeline의 instruction간의 RAW dependency가 있을 경우헤는 한번 stall을 해주어야 한다.

또한, jpr, jrl, branch와 같이 ID 단계에서 연산을 수행하는 경우 distance가 1일 때 반드시 hazard가 발생하며, distance가 2이면 forwarding을 통해 hazard를 해결할 수 있으나 EX/MEM register의 memread가 1이면 이 레지스터로부터 바로 값을 얻어올 수 없으므로 hazard가 일어난다.

ex 단계의 data hazard와 id 단계의 data hazard를 나누어 bit를 반환한다.

## Branch Prediction

Branch와 Jump의 경우 분기 예측을 통해 원치 않는 flush를 줄일 수 있다. IF 단계에서 PC를 갖고, PC의 하위 비트로 이루어진 캐시와 유사한 ‘BTB’ 버퍼를 이용해 다음 PC를 불러온다. 본 Lab에서는 BTB로 총 16비트 PC의 하위 8비트를 index로 사용하였으며 상위 8비트를 tag로 사용하였다. 또한 Global 2-bit saturation predictor을 적용하였다.



그런데 만약 분기 예측이 틀리게 되면 PC는 ID 단계에서 계산된 실제 목적지 PC로 이동해야 한다. 따라서 이 때는 BTB에서 가져온 값이 아닌 input으로 받은 실제 목적지 PC를 반환해야 한다. 또한 PC의 tag가 BTB의 PC index에 해당하는 tag와 일치하면, (cache hit과 유사) jump instruction이거나 Saturation이 taken을 가리킬 때 B2B의 목적지 값을 반환한다.

Branch Predictor에서 초기에 instruction의 branch / jump 목적지를 저장하는 것은 소프트웨어 방법으로 해결하거나, reset\_n 상태일 때 직접 메모리 주소의 target을 가져와 직접 다 대입하는 방식으로 해결할 수 있으나 이 부분은 구현하지 않았다. 따라서 처음 시작할 때에는 모든 index의 tag는 0000000이며 next PC는 PC + 1을 가리킨다. 하지만 각 instruction을 만나면서 instruction이 Branch or Jump type이면 BTB에 해당 instruction과 실제 target 값이 기록된다. 따라서 다음에 같은 instruction을 만났을 때 BTB tag와 일치하게 된다면 분기 예측을 성공적으로 진행할 수 있게 된다.

## Datapath

본 pipelined cpu의 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 4개의 파이프라인 레지스터도 포함하고 있다. Pipelined CPU이기 때문에 모든 instruction은 IF, ID, EX, MEM, WB의 다섯 단계로 이루어지며 각 단계에 따라 분리된 행동을 취한다.

* IF (Instruction Fetch)
  + PC 값을 index로 하여 memory에 접근하여 instruction을 받아 온다.
  + PC 값이 Branch Predictor로 입력되어 다음 PC를 전달받아 저장한다.
* ID (Instruction Decode)
  + Instruction을 rs index, rt index, rd index, immediate, jump target, opcode, func\_code 등으로 분리한다.
  + Register File에 접근하여 rs, rt, rd 레지스터의 값을 구한다.
  + Instruction의 opcode와 func\_code로부터 control bit들을 계산한다.
  + Branch instruction의 경우 분기가 taken되었는지를 검사한다.
  + Branch, jmp, jpr을 포함하여 모든 instruction에 대해 실제로 다음 이동할 PC 값을 구한다.
* EX (Execution)
  + RS, RT, Immediate 값을 토대로 ALU에서 메인 연산을 수행한다.
* MEM (Memory)
  + LWD의 경우 메모리에서 값을 읽어 온다.
  + SWD의 경우 register의 값을 메모리에 저장한다.
* WB (Write-Back)
  + 레지스터에 EX 단계에서 구한 값을 저장한다.
  + JAL, JPR의 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;

ALU는 연산을 처리하는 EX 단계의 모듈이다. 이 모듈은 control unit으로부터 alu\_op(func\_code) 값을 받아 그 값에 따라 서로 다른 연산을 취해준다. TSC에 존재하는 모든 연산을 10가지로 압축하였다. alu의 output은 combinational logic에 의해 계산되어 연결된다.

Branch ALU는 ID 단계에서 Branch의 진위 여부를 판별하기 위한 소형 ALU이다. opcode에 따라 A, B가 같은지 혹은 다른지, A가 0보다 큰지 작은지를 판별하여 bcond를 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 opcode와 func\_code를 이용해 instruction들을 type에 따라서 분리해주었다. 비슷한 것끼리 묶는 작업을 통해 control\_bit들을 계산할 때 코드의 직관성과 가독성을 높이고 자주 사용되는 bit을 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, beq

    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 instuction의 type에 따라서 control bit 들을 계산해준다. Design에서 설명한 각 control unit의 해당 instruction과 동일한 instruction에 대해서만 각 control bit가 1이 되도록 한다. is\_available의 경우 instruction이 0x000일 때만 0을 반환하는데, 이 경우 완전히 ‘빈 명령어’로 약속하기에 원래였으면 1이 되었을 c\_branch나 id\_use\_rs, id\_use\_rt도 모두 0이 된다.

## 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. Forwarding은 다시 EX 단계의 Forwarding과 ID 단계의 Forwarding으로 나뉜다.

EX 단계의 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] = 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\_sel은 1) 2) case와 4) 5) case의 datapath내에서 dataforwarding을 수행해줄 때 어떤 data 값을 이용해서 data forwarding을 해줄지 결정해는 signal이다.

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;

datapath에서는 이렇게 alu\_sel 값을 통해 ALU의 input에 RF에서 받아온 값을 넣을지, EXMEM에서 write할 값을 넣을 지, MEMWB에서 write할 값을 넣을 지를 결정한다. 끝으로 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.

ID 단계의 forwarding도 위와 매우 유사하다. 레지스터에 작성될 결과 값은 EX 단계에서 생성되기 때문에 마찬가지로 EX/MEM 레지스터에서 값을 forwarding하거나, MEM/WB 레지스터에서 값을 forwarding하거나, Register File에서 값을 Internal Forwarding애 주어야 한다. 따라서 EX 단계의 forwarding과 똑같은 6개의 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 forwarding은 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\_writedata와 wb\_writedata는 각각 다음의 값으로 assign된다.

assign ex\_writedata = EXMEMC\_PCTOREG ? EXMEM\_PC + 1 : EXMEM\_ALUOUT;

assign wb\_writedata = MEMWBC\_PCTOREG ? MEMWB\_PC + 1 : MEMWB\_OUT;

## HAZARD DETECT

많은 data hazard는 forwarding으로 처리할 수 있지만, 그럼에도 불구하고 hazard가 일어난다.

EX 단계에서의 hazard는 바로 앞선 명령어가 RAW dependency를 일으키는 LWD instruction일 때 발생한다. 이 경우 결과값은 MEM 단계에서 produce되기 때문에 EXMEM register에서는 forwarding할 수 없다. 이 경우에는 어쩔 수 없이 한 단계를 stall하게 된다.

ID 단계에서의 hazard는 앞선 명령어와 RAW dependency를 일으킬 때나, 두 번 앞선 명령어가 RAW dependency를 일으키는 LWD instruction일 때 발생한다. EX 단계보다 한 단계 더 간다고 간주할 수 있다. 마찬가지로 이 경우 한 단계를 stall하게 된다.

output으로는 EX 단계의 hazard와 ID 단계의 hazard를 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

PC의 하위 8비트를 Index를, 상위 8비트를 Tag로 사용하는 BTB를 구현하였다. reset\_n이 0일 때 이 BTB를 초기화한다. 처음에는 일단 모든 entry의 target을 PC + 1 로 결정한다. 당연하겠지만, 이 경우 처음에 tag가 일치하는 branch나 jump를 만나게 되면 taken으로 예측을 해도 분기에 실패하게 된다. 하지만 분기 실패 여부는 PC 값의 차이로 검사하기 때문에 전체 behavior에는 아무 지장이 없으며, 이후에는 branch나 jump에 대해 tag와 target 값이 latch되기 때문에 해당 index에 대해서는 제대로 BTB를 사용할 수 있다. 예외적으로 PC = 0은 35번째 instruction으로 이동하는 jump 문이기 때문에 TARGET[0] = 35로 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 Predictor을 사용하는데, 이는 branch 문일 때만 값이 변화한다. TAG와 TARGET의 경우 branch, jump문일 때 모두 값이 변화한다. 단 data hazard 상태일 경우에는 업데이트하지 않는다. branch가 data hazard 문에 걸려 한 stage 이상 stall 하는 경우 bcond 값이 제대로 계산되지 않았음에도 불구하고 saturation predictor가 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

ALU는 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\_OP는 control\_unit에서 결정한다.

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

                (opcode == 5) ? 4'd3 :

                (opcode == 6) ? 4'd8 :

                (wwd || jr) ? 4'd9 : 4'd0;

우선, opcode가 15이고 func\_code가 0~7인 명령어의 경우 각 명령어에 맞는 연산 방법을 사용한다. opcode = 5는 ORI 명령어로 OR 연산을 사용하며, 이는 opcode == 15이고 func\_code == 3인 ORR 연산과 행동이 동일하다. opcode = 6은 LHI 명령어로 imm << 8 연산을 수행한다. wwd와 jr의 경우 rs의 값이 그대로 들어간다. (jr의 경우 id 단계에서 바로 pc로 이동하기 때문에 사실 don’t care와 다름없다.) 기타 모든 instruction의 경우에는 + 연산을 수행하면 되며, alu 연산고 ㅏ무관하다. (lwd, swd 등)

## Datapath

Datapath는 IF, ID, EX, MEM, WB 단계를 따라 데이터가 이동하는 흐름을 나타내고 있다. Datapath 모듈은 총 4개의 Pipeline Register을 포함하며, 각 pipeline register은 다음 값을 갖고 있다.

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

각 pipeline register은 posedge clk 시점에 이전 pipeline register의 해당하는 값으로 latch된다. 단, data hazard가 일어나면 IFID register와 IDEX register는 0이 되거나 데이트하지 는다. IDEX register을 0으로 만드는 것은 이후의 모든 control bit을 0으로 만들어 아무 일도 하지 않는 operator로 만들어 버리는 것과 같다.

### Instruction Fetch

Instruction Fetch 단계의 대표적인 작업은 Memory로부터 PC에 해당하는 Instruction을 인출하는 작업이다. 본 lab에서는 address\_1을 PC Register 값, read\_m을 항상 1로 놓아 항상 instruction을 인출하게 했고, posedge clk일 때 IFID\_INSTR 레지스터에 메모리에서 읽어들인 데이터(flush일 경우 0x0000)를 latch한다.

다음으로, Instruction Fetch 단계에서 하는 중요한 일은 분기 예측이다. PC를 Branch Predictor에 연결해 BTB의 target 혹은 PC + 1을 저장한다. 이를 위해 IF 단계의 PC 레지스터와 Branch Predictor을 연결한다.

### INSTRUCTION DECODE

본 설계의 Instruction Decode 단계에서는 많은 양의 작업을 수행한다. 먼저 Instruction을 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 File에 접근하여 rs, rt, rd 레지스터의 값을 구한다. 이를 위해 앞서 만들어 놓은 Register File 모듈과 연결한다. Instruction의 opcode와 func\_code로부터 control bit들을 계산한다. 이를 위해 앞서 만들어 놓은 Control Unit과 Register File과 연결한다.

끝으로 Branch instruction의 경우 분기가 taken되었는지를 검사하고, Branch, jmp, jpr을 포함하여 모든 instruction에 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;

flush는 현재 명령어를 인출하고 있는 IF 단계의 PC register가 이전 instruction의 ‘정답’ instruction과 다를 때 일어난다. 예를 들어 branch에서 틀렸다고 분기 예측을 하여 PC + 1로 이동을 했지만 실제로 branch는 taken되어 branch target으로 이동해야 할 때 flush가 일어나 명령어를 다 지우게 된다. 그 경우에는 actual next pc를 PC 레지스터와 같은 값을 유지하여 flush가 끝나고 안전하게 다음 instruction부터 받아올 수 있도록 한다.

Branch 및 JALR에 사용되는 값은 data hazard를 고려하여 forwarding 받아온 값을 이용한다. (Forwarding 참조)

### EXECUTION

메인 연산 모듈인 ALU모듈을 사용하여 연산한다. 본 Lab에서 EX 단계의 작업은 비교적 간단하게 이루어진다. 역시 data hazard를 처리하기 위해 forwarding 받아온 값을 이용한다. (Forwarding 참조)

### MEMORY

MEM 단계는 LWD, SWD instruction을 처리하기 위해 메인 메모리에 접근하는 stage이다. read\_m2, write\_m2는 EX/MEM pipeline register의 control bit 값을 그대로 가져온다. LWD의 경우 가져온 값 data2를 memory\_output에 연결하고, SWD의 경우 data2에 rt 값을 저장할 수 있도록 한다.

assign data2 = write\_m2 ? EXMEM\_REG2 : 16'bz;

...

MEMWB\_MEMOUT <= data2;

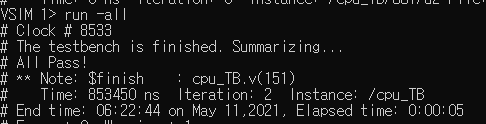
### WRITEBACK

Writeback 단계는 값을 Register에 다시 쓰는 단계이다. Pipeline Register을 따라 진행된 reg\_write, pc\_to\_reg, rdest(dest register)에 의존하여 RF에서 쓰기를 진행한다. jal, jrl의 경우 예외적으로 $2 레지스터에 ALU나 MEM에서 계산된 값 대신 pc + 1을 저장한다. RF의 쓰기 과정은 negedge clk에서 진행된다.

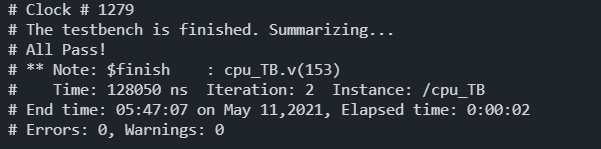
# Discussion

## Pipelined vs Multi-Cycle CPU 성능 비교

앞서 작성한 Multi-Cycle CPU의 경우 총 6533 cycle을 소요하였다.



반면, 이번에 작성한 Pipelined CPU는 총 1279 cycle을 소요한다.



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

Multi Cycle은 여러 개의 cycle에 걸쳐 하나의 instruction을 처리하고 각 instruction이 microinstruction으로 쪼개지긴 하지만, 결국 하나의 instruction만을 한 번에 처리하므로 throughput이 낮다. 반면 pipelined cpu는 한 instruction을 5단계로 나누기 throughput도 일렬로 수행할 때보다 최대 5배 증가함을 알 수 있다.

## Branch Predictor 별 성능 비교

본 CPU는 Branch Predictor을 사용한다. Branch Predict를 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 Taken이 가장 적은 cycle만에 끝나는 것을 확인할 수 있었다. 주어진 testbench의 경우 taken되는 branch가 더 많은 것을 알 수 있었다. global saturation의 bit 수에 변화를 주었으나 1200과 1296 사이의 cycle을 얻었다.

## 개선점 및 주안점

우선, Branch 및 jrl, jpr instruction을 ID 단계로 옮겼다. 이는 Branch, jpr, jrl로 인해 발생하는 stall을 한 stage 줄이는 데에 효과적이다. 하지만 이 이전의 instruction에 의해 data hazard가 발생할 경우 EX 단계에 비해 한 단계 더 hazard가 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 forwarding을 구현하고 ID 단계 연산의 경우 forwarding을 구현하지 않으려 했지만, 그 결과 총 cycle이 1539 정도를 얻었다. 이는 forwarding을 구현하기 전과 큰 차이다. 실제로 testbench 후반에 갈 수록 jmp와 branch의 비율이 많아지고 data hazard도 잦아지기 때문에 stall 하나가 치명적이다.

# COnCLUSION

본 lab에서는 TSC 명령어를 IF, ID, EX, MEM, WB 다섯 단계로 나누어 Pipeline화하는 CPU를 구현하였다. branch와 jump register 목적지는 ID 단계에서 연산하였고 Data Hazard를 해결하기 위한 Data Forwarding 및 분기로 인한 시간 지연을 막기 위한 Branch Predictor을 도입하였다.

Pipelined CPU Lab을 통해 현대 CPU 구조의 핵심이라 할 수 있는 Pipelined CPU에 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.