

# lab6\_report

## 0. 전체적인 구현 계획

### 1. Forwarding Unit 구현

- 파이프라인에서 발생하는 data hazard를 줄이기 위한 데이터 포워딩 경로 추가.
- 다음 세 가지 경우의 포워딩을 구현해야 함:
  - $\text{dist}(i,j)=1$  : EX  $\rightarrow$  EX
  - $\text{dist}(i,j)=2$  : MEM  $\rightarrow$  EX
  - $\text{dist}(i,j)>=3$  : 레지스터 파일에 있는 값 사용

```
if ( $\text{rs}_{\text{EX}} \neq 0$ ) && ( $\text{rs}_{\text{EX}}$  사용을 한다. ==  $\text{dest}_{\text{MEM}}$ ) &&  $\text{RegWrite}_{\text{MEM}}$  then
    forward operand from MEM stage // dist=1
else if ( $\text{rs}_{\text{EX}} \neq 0$ ) && ( $\text{rs}_{\text{EX}}$  사용을 한다. ==  $\text{dest}_{\text{WB}}$ ) &&  $\text{RegWrite}_{\text{WB}}$  then
    forward operand from WB stage // dist=2
else
    use the operand from register file // dist >= 3
    forwarding 안해줘도 됨.

Ordering matters!! Consider the following ...
```

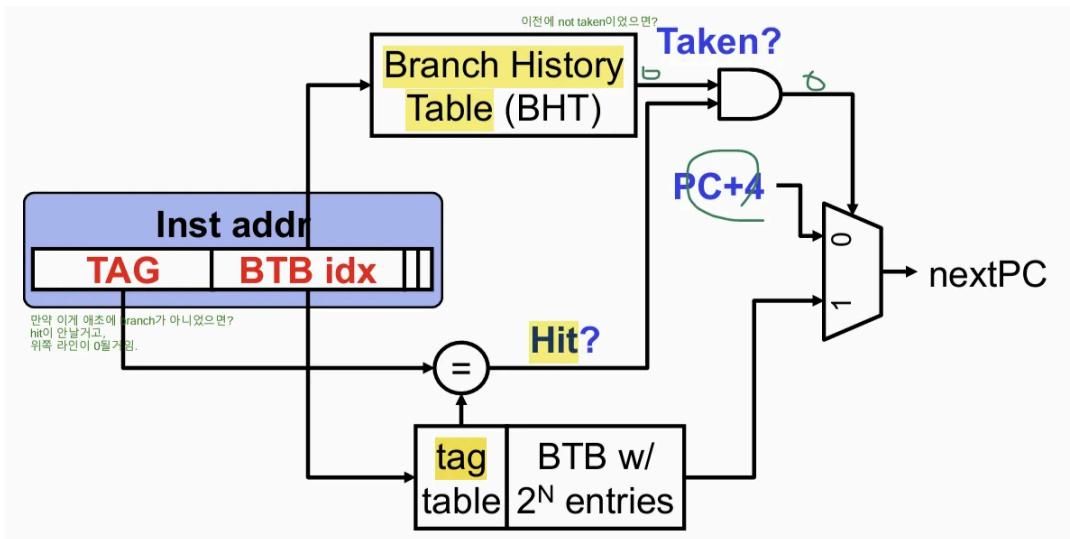


- Load-use hazard 발생 시 stall

$\text{ID/EX.MemRead} == 1$ 이고,  $\text{ID/EX.Rt} == \text{IF/ID.Rs or Rt}$  이면 stall  
EX에서 읽는 operand가 MEM에서 나오는 Load 결과의 경우

### 2. Branch Prediction 구현

- BTB (Branch Target Buffer)와 2-bit branch predictor 사용
- Branch target 및 taken 여부 예측
- JR 명령어는 예측 대상 제외



C++

## 1. FORWARDING.cpp

### 기능 구현

우선, internal forwarding을 제외하고,  
EX → EX forwarding과  
MEM → EX forwarding을 탐지하고, 신호를 보내내기 위한 모듈을 생성했다.

```
void computeForwarding(uint32_t id_ex_rs, uint32_t id_ex_rt,
                      uint32_t ex_mem_wr_addr, uint32_t ex_mem_RegWrite,
                      uint32_t mem_wb_wr_addr, uint32_t mem_wb_RegWrite,
                      ForwardingSignals *fw)
{
    // ForwardA
    if (ex_mem_RegWrite && ex_mem_wr_addr != 0 && ex_mem_wr_addr == id_ex_rs)
        fw->ForwardA = 2;
    else if (mem_wb_RegWrite && mem_wb_wr_addr != 0 && mem_wb_wr_addr == id_ex_rs)
        fw->ForwardA = 1;
    else
        fw->ForwardA = 0;

    // ForwardB
    if (ex_mem_RegWrite && ex_mem_wr_addr != 0 && ex_mem_wr_addr == id_ex_rt)
        fw->ForwardB = 2;
    else if (mem_wb_RegWrite && mem_wb_wr_addr != 0 && mem_wb_wr_addr == id_ex_rt)
        fw->ForwardB = 1;
    else
        fw->ForwardB = 0;
}
```

## 2. HAZARD.cpp

구현 계획

`ID/EX.MemRead == 1`이고, `ID/EX.Rt == IF/ID.Rs or Rt`이면 1 cycle stall할 수 있도록 기존 코드를 수정했다.

추가로, branch에 대해서는 `lw`의 뒤에 왔을 때 2 cycle stall을 할 수 있도록 코드를 수정했다.

## 기능 구현

`id_ex.rt` 는 `lw`의 목적지 register이자, 실제 load-use 의심 대상이다.

`id_ex.controls.MemRead` 가 1 이면 `lw` 이고,  
forwarding이 처리하는 나머지 hazard는 stall이 필요 없다.

```
    uint32_t use_rs = usesRS(opcode);
    uint32_t use_rt = usesRT(opcode, funct);

    // Load-use hazard only
```

다만, branch의 경우, 현재 ID단계에서 계산을 하고 있기 때문에, 2 cycle stall이 필요하다.

```
// Load-use hazard only
if (id_ex_MemRead &&
    ((id_ex_rt == rs && use_rs) || (id_ex_rt == rt && use_rt))) {
    // BEQ, BNE일 경우에는 2-cycle stall이 필요함
    if (opcode == OP_BEQ || opcode == OP_BNE)
        return 2;
    return 1;
}
```

## 3. BP.cpp

구현 계획

## 1. IF 단계 (Instruction Fetch)

- 현재 PC 값을 기반으로 BTB에 접근.
  - predict() 호출 → BTB hit + PHT 상태 기반으로 branch/jump 예측.
  - 예측이 taken이면 predicted\_target을 nextPC로 사용.
  - 예측이 not taken이면 그냥 PC + 4 를 nextPC로 사용

## 2. update()

- Branch, Jump에 대해  
    실제 분기 결과 (T/NT)와 실제 target address로 BTB/PHT 업데이트

## 기능 구현

우선, 명세에 맞춰서 BTB구조체를 생성했다.

PHT도 2bit counter로 이용할 수 있도록 만들어뒀다.

Index와 Tag부분을 slicing하는 함수들도 편의를 위해 추가했다.

### BP.h

```
// BTB Entry 구조체
enum BTBState {
    BTB_EMPTY = 0, // Not initialized
    BTB_JUMP = 1, // Jump (j, jal)
    BTB_BRANCH = 2 // Conditional Branch (beq, bne)
};

struct BTBEntry {
    BTBState state; // 0: empty, 1: jump, 2: branch
    uint32_t tag; // PC[31:8]
    uint32_t target; // predicted target
};
```

```
// 2-bit PHT (Pattern History Table) counter
enum PHTState {
    STRONGLY_NOT_TAKEN = 0,
    WEAKLY_NOT_TAKEN = 1,
    WEAKLY_TAKEN = 2,
    STRONGLY_TAKEN = 3
};
```

```
class BranchPredictor {
public:
    BranchPredictor();

    // 예측 관련 함수
    bool predict(uint32_t pc, uint32_t &predicted_target);
    void update(uint32_t pc, bool taken, uint32_t actual_target, bool isBranch); /

private:
    static const int BTB_SIZE = 64; // 6-bit index → 64-entry
    static const int PHT_SIZE = 256;
    BTBEntry btb[BTB_SIZE];
    PHTState pht[PHT_SIZE];

    uint32_t getBTBIndex(uint32_t pc);
    uint32_t getPHTIndex(uint32_t pc);
    uint32_t getTag(uint32_t pc);
    void updatePHT(uint32_t index, bool taken);
};
```

### BP.cpp

BTB entry가 valid한지 확인하고, BTB tag를 비교했다.

```
// 예측 함수
bool BranchPredictor::predict(uint32_t pc, uint32_t &nextPC) {
    uint32_t BTBIndex = getBTBIndex(pc);
    uint32_t PHTIndex = getPHTIndex(pc);
    uint32_t tag = getTag(pc);

    const BTBEntry &entry = btb[BTBIndex];

    if (entry.state == BTB_EMPTY || entry.tag != tag){
        nextPC = pc + 4;
        return false; // BTB Miss → predict not taken
    }
```

Entry가 Branch를 담고 있다면 pht를 확인하고,  
Jump를 담고 있다면 target으로 점프하도록 했다.

```
// BTB Hit
if (entry.state == BTB_BRANCH) {
    // Branch → PHT 확인
    if (pht[PHTIndex] >= WEAKLY_TAKEN) {
        nextPC = entry.target;
        return true; // predict taken
    } else {
        nextPC = pc + 4; // PC? PC+4?
        return false; // predict not taken
    }
} else if (entry.state == BTB_JUMP) {
    // Jump는 무조건 taken
    nextPC = entry.target;
    return true;
}
```

## 4. CPU.cpp

원활한 Forwarding을 위해 `wr_data` 계산이 우선될 수 있도록, 위치를 `tick()`의 최상단으로 조정했다.

```
//latch초기화는 마지막에 이루어지기 때문에 상관 없음.
wr_data = mem_wb.controls.MemtoReg ? mem_wb.mem_data : mem_wb.alu_result;
if(mem_wb.controls.SavePC){ //JAL일 경우
    mem_wb.wr_addr = 31;
    wr_data = mem_wb.PC + 4;
}
rf.write(mem_wb.wr_addr, wr_data, mem_wb.controls.RegWrite);
```

Internal Forwarding을 위해, 레지스터 파일에서 값을 읽은 이후, 조건을 확인하여, 값을 받을 수 있으면 Forwarding을 통해 값을 받도록 했다.

```
rf.read(parsed_inst.rs, parsed_inst.rt, &rs_data, &rt_data);

if (mem_wb.controls.RegWrite && mem_wb.wr_addr != 0) { // Internal Forwarding: WB
    if (mem_wb.wr_addr == parsed_inst.rs) {
        rs_data = wr_data;
    }
    if (mem_wb.wr_addr == parsed_inst.rt) {
        rt_data = wr_data;
    }
}
```

ALU연산 전에, forwarding을 받아야하는 상황인지 파악하기 위해 `computeForwarding()`을 호출하고, 그 결과로 얻은 두 신호, `fw.ForwardA`, `fw.ForwardB`에 따라 Forwarding을 진행했다.

```

computeForwarding(id_ex.rs, id_ex.rt,
                  ex_mem.wr_addr, ex_mem.controls.RegWrite,
                  mem_wb.wr_addr, mem_wb.controls.RegWrite,
                  &fw);
switch (fw.ForwardA) {
    case 0: operand1 = id_ex.rs_data; break;
    case 1: operand1 = wr_data; break;
    case 2: operand1 = ex_mem.alu_result; break;
}
if (id_ex.controls.ALUSrc)
    operand2 = id_ex.ext_imm;
else {
    switch (fw.ForwardB) {
        case 0: operand2 = id_ex.rt_data; break;
        case 1: operand2 = wr_data; break;
        case 2: operand2 = ex_mem.alu_result; break;
    }
}
alu.compute(operand1, operand2, id_ex.shamt, id_ex.controls.ALUOp, &alu_result);

```

Branch resolution의 경우 ID단계에서 이루어지고 있기 때문에, 해당 값도 forwarding을 받기 위해서는 WB→ID로의 forwarding이 추가로 필요했다.

```

// Forward from ID/EX (최신 연산 결과가 바로 뒤 cycle일 때도 고려)
uint32_t id_ex_wr_addr = (id_ex.controls.RegDst) ? id_ex.rd : id_ex.rt;
if (id_ex.controls.RegWrite && id_ex_wr_addr != 0) {
    if (id_ex_wr_addr == parsed_inst.rs) branch_rs = alu_result; // 필요
    if (id_ex_wr_addr == parsed_inst.rt) branch_rt = alu_result;
}

```

Branch resolution을 위한 준비과정으로, `actually_taken`, `actual_target` 을 계산했다.

다만, JR에 대해서는 BP를 하지 않기 때문에,

`controls.JR` 이 활성화된 경우, 다음 `PC` 값으로 예측 결과인 `PC_next` 를 이용하는 것이 아니라, 레지스터에서 직접 읽었거나, Forwarding을 통해 받았을 `rs_data`를 가지도록 했다.

```

if(controls.JR){
    PC = rs_data; //애는 여기서 계산해줘야함.
    // actual_target = rs_data;
} else if (controls.Branch && controls.ALUOp == ALU_EQ && branch_rs == branch_rt){
    // PC = if_id.PC + 4 + (ext_imm << 2); // BEQ 처리
    actual_target = if_id.PC + 4 + (ext_imm << 2);
} else if (controls.Branch && controls.ALUOp == ALU_NEQ && branch_rs != branch_rt){
    // PC = if_id.PC + 4 + (ext_imm << 2); // BNE 처리
    actual_target= if_id.PC + 4 + (ext_imm << 2);
} else if(controls.Jump){
    // PC = (if_id.PC & 0xF0000000) | (parsed_inst.immj << 2); //상위 4비트와 immj<<2
    actual_target = (if_id.PC & 0x0000000) | (parsed_inst.immj << 2);
} else {
    actual_target = PC + 4;
    actually_taken = false;
}

```

Branch resolution에서, JR연산이 아닌 경우, BP가 잘 진행되었는지 비교 연산을 수행했다. 그리고 그 결과로, BTB, PHT를 update해줬다.

```

//branch resolution
bool mispredict = false;
if(!controls.JR){
    if (if_id.predicted_taken != actually_taken){ // PHT가 잘 동작했는지. T/NT
        mispredict = true;
    } else if (if_id.predicted_taken && if_id.PC != actual_target) { // T이라면
        mispredict = true;
    }
}

bool is_Branch = controls.Branch; //근데 일단 이러면, jump가 꼭 아니어도, 그러니까 Branch이면 Branch로 설정
if(controls.Branch) is_Branch = controls.Branch;
else if(controls.Jump) is_Branch = 0;

if(!controls.JR && (controls.Branch || controls.Jump)) //JR은 고려 안하기. 이렇게 predictor.update(if_id.PC, actually_taken, actual_target, is_Branch);

```

latch를 업데이트하는 부분에서도, Forwarding이 필요하다.

`ex_mem.rt_data`의 경우, Forwarding을 받을 수 있기 때문에,  
이전에 생성된

`fw.ForwardB`의 신호 상태에 따라, 해당 값을 업데이트 해줬다.

misprediction이 발생했다면, JAL 연산인 경우를 제외하고, id\_ex 단계를 flush시켜줬다.

```

//id_ex, if_id 업데이트 w.flush, Stall 처리
if (mispredict) {
    if(controls.SavePC){
        id_ex.controls = controls;
        id_ex.ext_imm = ext_imm;
        id_ex.rs_data = rs_data;
        id_ex.rt_data = rt_data;
        id_ex.shamt = parsed_inst.shamt;
        id_ex.rs = parsed_inst.rs;
        id_ex.rt = parsed_inst.rt; //FORWARDING을 위해 추가.
        id_ex.rd = parsed_inst.rd;
        id_ex.immj = parsed_inst.immj; //jump연산을 위해 남겨둠
        id_ex.PC = if_id.PC;
    } else {
        id_ex = {};
        id_ex.PC = 0;
    }
    if_id = {};
    if_id.PC = 0;
    PC = actual_target; // 을바른 PC로 되돌림
} else {

```

## 5. 테스트 - testcase 1~7 통과

다음은, testcase 1~7 결과이다.

Mars로 출력한 결과(정답)

## cpp코드 출력 결과

모두 통과!

## 6. Trouble Shooting

우선, Forwarding을 구현한 후, 해당 코드가 test를 모두 통과하면, Branch Prediction을 추가할 예정이다.

따라서, Forwarding을 하며 만난 문제들을 먼저 기록하겠다.

# Forwarding

6-1. Forwarding 구현 시, branch 연산이 잘못되는 경우 존재.

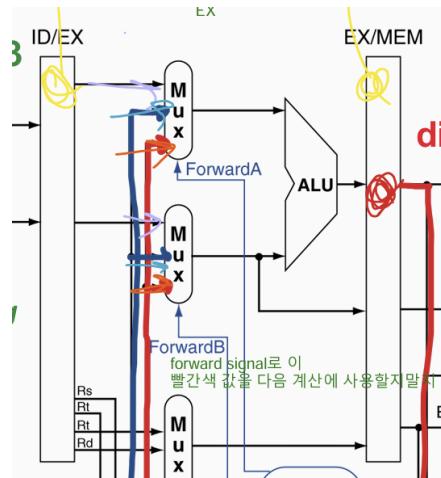
기존 코드에서는 ID단계에서 branch에 대한 연산을 진행했는데,

Forwarding으로 받아온 값을 기준으로 branch prediction이 진행된다고 생각해야하는데, 그러지

못했다.

## 6-2. SW가 값을 받지 못한다.

도표를 다시 살펴보니, SW에 대한 Forwarding도 이루어진다는 사실을 알 수 있었다. ForwardB 신호에 따라 얻어온 값을 EX/MEM latch로 넘겨주는 과정이 확인되었다.



```
//놀라운사실! MEM에 접근할 녀석도 Forwarding을 받는답니다!!
uint32_t forwarded_rt_data = id_ex.rt_data;
switch (fw.ForwardB) {
    case 0: forwarded_rt_data = id_ex.rt_data; break;
    case 1: forwarded_rt_data = wr_data; break;
    case 2: forwarded_rt_data = ex_mem.alu_result; break;
}
// 놀라운사실!
ex_mem.rt_data = forwarded_rt_data;
```

## 6-3. testcase 5 무한루프

```
PC: 000004EC, Instruction: j
branch test: branch
branch? 1
stall? 0
[DEBUG] Writing back PC
[DEBUG] rs_data (i.e., $ra) = 00000030
PC: 00000030, Instruction: j
```

`jr` (0x4e8)이 ID단계에서 해석할 때, `$ra` 레지스터에서 PC로 30을 읽은 모습이다. 즉, `jal` 연산이 PC+4를 저장하지 않고, PC를 저장해서 무한 루프가 발생했다.

`$ra`에 저장할 `wr_data`에 `+4`를 해서 해당 문제를 해결했다.

## 6-4. testcase 6 lw 다음 beq에서 forwarding 문제

현재 구현 방식에서는 `beq` 를 id단계에서 처리하고 있기 때문에,  
lw 이후 이를 사용하기 위해서는 2 cycle stall이 필요하다.  
하지만, 이 부분을 망각하고 있었다.

`HAZARD.cpp` 에 해당 부분을 수정하여 문제를 해결했다.

```
// Load-use hazard only
if (id_ex_MemRead &&
    ((id_ex_rt == rs && use_rs) || (id_ex_rt == rt && use_rt))) {
    // BEQ, BNE일 경우에는 2-cycle stall이 필요함
    if (opcode == OP_BEQ || opcode == OP_BNE)
        return 2;
    return 1;
}
```

## Branch Prediction

### 6-5. testcase 3, 4, 5에서 무한 루프 발생.

3,4,5의 전반적인 문제는 Branch Resolution을 잘못 처리하고 있었던 것이다.

ID단계에서 확인하는 것은, 바로 직전의 IF된 값이 정상적으로 Fetch된 값인지 이다.  
이를 고려하지 않고, 그보다 한 단계 더 다음 단계인 Fetch 예정인 값을 확인하고 있었다.

latch를 통해, 이전 BP 결과를 (T/NT, Target) 전달해주고, 비교함으로서 해당 문제를 해결할 수 있었다.

```
} else {
    if_id.instruction = instruction;
    if_id.PC = PC;
    if_id.predicted_taken = predicted_taken;
    PC = nextPC;
}
```

```
//branch resolution
bool mispredict = false;
if(!controls.JR){
    if (if_id.predicted_taken != actually_taken){ // PHT가 잘 동작했는지. T/NT
        mispredict = true;
    } else if (if_id.predicted_taken && if_id.PC != actual_target) { // T라면, BTB가 잘 동작했는지. (= 이전에 fetch)
        mispredict = true;
    }
    // printf("mispredict: %d, predicted_taken: %d, actually_taken: %d, predicted_target: %08x, actual_target: %08x\n",
    //        mispredict, predicted_taken, actually_taken, predicted_target, actual_target);
}

bool is_Branch = controls.Branch; //근데 일단 이러면, jump가 꼭 아니어도, 그러니까 다른 명령어여도 될 수 있는거 아닌가?
if(controls.Branch) is_Branch = controls.Branch;
else if(controls.Jump) is_Branch = 0;

if(!controls.JR && (controls.Branch || controls.Jump)) //JR은 고려 안하기. 이렇게 하면 항상 JR연산은 false뜨겠지!
    predictor.update(if_id.PC, actually_taken, actual_target, is_Branch);
```

또한, 5의 경우 무한루프가 발생하는 이유가, 늘 그렇지만 `jal` 연산에서, `$ra` 에 적절한 값을 써주지 못해서이다.

확인 결과 jal 연산 이후, 적히는 값이 0이었다. misprediction이 발생했을 때, SavePC 가 활성화되어 있는 경우, 값들을 온전하게 전달해줘야 하는데, if-else로 처리하지 않고 if로만 처리하여, 하단에서 넘겨준 값들이 덮어 씌워지고 있었다.

```
//id_ex, if_id 업데이트 w.flush, Stall 처리
if (mispredict) {
    if(controls.SavePC){
        id_ex.controls = controls;
        id_ex.ext_imm = ext_imm;
        id_ex.rs_data = rs_data;
        id_ex.rt_data = rt_data;
        id_ex.shamt = parsed_inst.shamt;
        id_ex.rs = parsed_inst.rs;
        id_ex.rt = parsed_inst.rt; //FORWARDING을 위해 추가.
        id_ex.rd = parsed_inst.rd;
        id_ex.immj = parsed_inst.immj; //jump연산을 위해 남겨둠
        id_ex.PC = if_id.PC;
    } else {
```

+ 생각해보니, mispredict를 탐지한 시점이라면, id\_ex latch에 들어가는 값들은 이미 온전한 값이다. 즉, flush될 이유가 없다. 따라서 코드를 다음과 같이 수정했다.  
SavePC의 여부와 관계없이 모든 값을 다음 latch로 넘겨주었다.  
SavePC는 여전히, WB단계에서 사용된다.

```
if (mispredict) {
    id_ex.controls = controls;
    id_ex.ext_imm = ext_imm;
    id_ex.rs_data = rs_data;
    id_ex.rt_data = rt_data;
    id_ex.shamt = parsed_inst.shamt;
    id_ex.rs = parsed_inst.rs;
    id_ex.rt = parsed_inst.rt; //FORWARDING을 위해 추가
    id_ex.rd = parsed_inst.rd;
    id_ex.immj = parsed_inst.immj; //jump연산을 위해 남겨둠
    id_ex.PC = if_id.PC;
    if_id = {};
    if_id.PC = 0;
    PC = actual_target; // 올바른 PC로 되돌림
} else {
```

## 6-6. testcase4를 수행할 때 결과가 계속 달라짐

```

havruta@BOOK-TCTPSJLV91:~/computerArchitectureHYU/HW6/cpp$ ./cpu testcase4.hex
Loading instruction memory...
Starting CPU simulation...
c
Simulation done successfully.
RF states after the program execution
$zero 00000000
$at c0bf0000
$v0 00000000
$v1 00000000
$a0 00000000
$a1 00000000
$a2 00000000
$a3 00000000
$t0 cceaa353c
$t1 5d5f78a6
$t2 b25816d6
$t3 996e093d
$t4 00000001
$t5 05cd1f4
$t6 00000000
$t7 00000000
$t8 00000000
$c0 c917ab77
$s1 00000000
$s2 00000000
$s3 00000000
$s4 00000000
$s5 00000000
$s6 00000000
$s7 00000000
$t8 00000000
$t9 00000000
$k0 00000000
$k1 00000000
$gp 00001800
$sp 00001800
$fp 00000000
$ra 00000000

```

성공한 결과

```

havruta@BOOK-TCTPSJLV91:~/computerArchitectureHYU/HW6/cpp$ ./cpu testcase4.hex
Loading instruction memory...
Starting CPU simulation...
c
Simulation done successfully.
RF states after the program execution
$zero 00000000
$at c0bf0000
$v0 00000000
$v1 00000000
$a0 00000000
$a1 00000000
$a2 00000000
$a3 00000000
$t0 f8712fed
$t1 a0352461
$t2 bbe2ac03
$t3 0a1d72c4
$t4 04e023ae
$t5 7a8812ba
$t6 00000000
$t7 00000000
$t8 00000000
$t9 00000000
$k0 00000000
$k1 00000000
$gp 00001800
$sp 00003ffc
$fp 00000000
$ra 00000000

```

일부 결과가 달라진 실패한 결과

```

// 초기화
BranchPredictor::BranchPredictor() {
    for (int i = 0; i < BTB_SIZE; ++i) {
        btb[i].state = BTB_EMPTY;
        btb[i].tag = 0;
        btb[i].target = 0;
    }
    for (int i = 0; i < PHT_SIZE; ++i){
        pht[i] = WEAKLY_NOT_TAKEN;
    }
}

```

pht에 대한 초기화를 진행하니 해당 문제가 해결되었다. (왜 인지는 모르겠다.)

추가적으로, Verilog를 구현하면서 누락한 부분인, instruction이 0인 경우 **mispredict** 계산하지 않도록 하는 부분을 추가했다.

아마 이 코드가 추가되지 않아서, 이전과 같은 상황이 발생했던 것으로 보인다.

```

bool mispredict = false;
if(instruction != 0 && !controls.JR){
    if(if_id.predicted_taken != actual)
        mispredict = true;
    else if(if_id.predicted_taken == 0)

```

## Verilog

# 1. FORWARDING.v

위에 설명했던 FORWARDING.cpp와 완전하게 동일한 구조로 설계했다.

```
always @(*) begin
    // ForwardA
    if (ex_mem_RegWrite && ex_mem_wr_addr != 0 && ex_mem_wr_addr == id_ex_rs)
        ForwardA = 2;
    else if (mem_wb_RegWrite && mem_wb_wr_addr != 0 && mem_wb_wr_addr == id_ex_rs)
        ForwardA = 1;
    else
        ForwardA = 0;

    // ForwardB
    if (ex_mem_RegWrite && ex_mem_wr_addr != 0 && ex_mem_wr_addr == id_ex_rt)
        ForwardB = 2;
    else if (mem_wb_RegWrite && mem_wb_wr_addr != 0 && mem_wb_wr_addr == id_ex_rt)
        ForwardB = 1;
    else
        ForwardB = 0;
```

# 2. HAZARD.v

위에 설명했던 HAZARD.cpp와 완전하게 동일한 구조로 설계했다.

다만, Verilog에서는 HAZARD 모듈을 매 사이클마다 접근해서 새로 계산을 진행하기 때문에, 2 cycle의 stall이 필요한 경우에도, 바로 다음 cycle에 0이 되는 경우가 있었다.

해당 문제는 `CPU.v`에서 `prev_hazard`라는 변수를 만들어, 2 cycle을 보장 받을 수 있도록 하여 해결했다.

```
always @(*) begin
    // 기본값: 스틀 없음
    hazard_stall = 0;

    // Load-Use hazard 검사
    if (id_ex_MemRead &&
        ((id_ex_rt == rs && use_rs) ||
         (id_ex_rt == rt && use_rt))) begin

        // BEQ, BNE는 2-cycle stall
        if (opcode == `OP_BEQ || opcode == `OP_BNE)
            hazard_stall = 2;
        else
            hazard_stall = 1;
    end
end
```

# 3. BP.v

위에 설명했던 BP.cpp와 완전하게 동일한 구조로 설계했다.

모듈 내부에 함수를 두어 연결할 수 있는 것이 아니기 때문에,  
update를 위한 변수들의 이름에 update\_를 붙여 predict를 위한 것과 이들을 구분할 수 있도록 했다.

```

// ----- BTB, PHT 메모리 -----
reg [1:0] btb_state [0:BTB_SIZE-1];
reg [23:0] btb_tag [0:BTB_SIZE-1];
reg [31:0] btb_target [0:BTB_SIZE-1];

reg [1:0] pht [0:PHT_SIZE-1];

// ----- Index & Tag 추출 -----
wire [5:0] btb_index = PC[7:2];
wire [7:0] pht_index = PC[9:2];
wire [23:0] pc_tag = PC[31:8];

wire [5:0] update_btb_index = update_pc[7:2];
wire [7:0] update_pht_index = update_pc[9:2];
wire [23:0] update_tag = update_pc[31:8];

```

예측 로직 자체는 cpp과 동일하다.

```

// ----- 예측 로직 -----
always @(*) begin
    if (btb_state[btb_index] == BTB_EMPTY || btb_tag[btb_index] != pc_tag) begin
        predict_taken = 0;
        nextPC = PC + 4;
    end else if (btb_state[btb_index] == BTB_JUMP) begin
        predict_taken = 1;
        nextPC = btb_target[btb_index];
    end else if (btb_state[btb_index] == BTB_BRANCH) begin
        if (pht[pht_index] >= WEAKLY_TAKEN) begin
            predict_taken = 1;
            nextPC = btb_target[btb_index];
        end else begin
            predict_taken = 0;
            nextPC = PC + 4;
        end
    end else begin
        predict_taken = 0;
        nextPC = PC + 4;
    end
end

```

## 4. CPU.v

우선, 구현의 편의성을 위해서 최대한 cpp과 변수명을 동일하게 맞추며, 새로 추가할 필요가 있는 변수들을 선언했다.

사진에 있는 값들 외에도

`ID_EX_rs` 도 추가로 필요했다. (상단에 선언됨)

```

//LAB06에서 추가됨 (FORWARDING, BP를 위한 값들)
wire [31:0]      nextPC;
wire           predict_taken;
wire [1:0]       ForwardA, ForwardB;
reg  [31:0]      Forwarded_rd_data2; //rt_data
reg           actually_taken;
reg  [31:0]      actual_target;
reg  [31:0]      branch_rs;
reg  [31:0]      branch_rt;
reg  [31:0]      ID_EX_wr_addr;
reg           IF_ID_predicted_taken; //이전에 예측했던 값 이용
reg           is_internal_forwarding_rs; //internal forwarding을 위해
reg           is_internal_forwarding_rt;
reg           prev_hazard;

```

우선, 대부분의 내용은 [CPU.cpp](#) 의 구현과 다르지 않다.

다만, 구현 중 유의해야 했던 점은

[PC](#) 에 값을 덮어 씌우면서 이용할 수 없기 때문에, 다음 cycle에 이용할 PC값을 저장하는 변수가 하나가 더 필요했다.

[nextPC](#) 는 cpp과 동일하게 BP에서 나온 결과를 담고 있게 했고,  
동시에

[PC\\_next\\_for\\_JR](#) 라는 변수를 두어, JR의 경우만 따로 처리할 수 있도록 했다.

```

if(JR) begin
    PC_next_for_JR = rd_data1; //PC_next_for_JR는 JR전용이야 이제..

```

Internal Forwarding의 경우에는 모듈들과 연결된 값을 그대로 덮어 씌울 수 없기 때문에, 1bit짜리  
변수를 두어, Internal Forwarding이 가능한 조건을 가질 경우,  
clock이 될 때 해당 값을 업데이트 해주었다.

```

is_internal_forwarding_rs = 0;
if(MEM_WB_RegWrite && MEM_WB_wr_addr != 0) begin
    if(MEM_WB_wr_addr == IF_ID_rs) is_internal_forwarding_rs = 1;
    if(MEM_WB_wr_addr == IF_ID_rt) is_internal_forwarding_rt = 1;
end

```

## 5. 테스트 - testcase 1~7 통과

Mars로 출력한 결과(정답)

## Verilog 코드 출력 결과

모두 통과!

## 6. Trouble Shooting

## 6-1. testcase 1 - Internal Forwarding 처리 문제

Internal forwarding을 어떻게 진행해야할지 모르겠는 상황에서, 우선 생략하고 testcase를 돌렸더니, 당연하게도 오류가 발생했다.

이전에, 다른 모듈과 연결된 와이어에 값을 덮어씌우는 것이 안된다는 것을 알게 되었기 때문에, Internal forwarding이 필요하다는 표시를 남겨두고, tick마다, 해당 값을 기반으로 다음 latch값을 업데이트 해주면 되겠다고 생각했다.

```

is_internal_forwarding_rs = 0;
is_internal_forwarding_rt = 0;
if(MEM_WB_RegWrite && MEM_WB_wr_addr != 0) begin
    if(MEM_WB_wr_addr == IF_ID_rs) is_internal_forwarding_rs = 1;
    if(MEM_WB_wr_addr == IF_ID_rt) is_internal_forwarding_rt = 1;
end

ID_EX_rd_data1 <= (is_internal_forwarding_rs) ? wr_data : rd_data1;
ID_EX_rd_data2 <= (is_internal_forwarding_rt) ? wr_data : rd_data2;
TD_EX_ext_imm <= ext_imm;

```

예상 그대로, 이 코드 수정을 통해 문제를 해결할 수 있었다.

Simulation success!!!

이 수정 이후, testcase 1, 2, 5, 7을 통과했다.

## 6-2. testcase 3, 4, 6에서 프로그램이 끝나지 않음.

디버깅을 통해 확인한 결과, `mispredict` 가 1로 올라갔다가 0으로 다시 내려오지 않는 듯한 모습을 보였다.

또한, ID단계의 명령어가

`00000000` 일 때에도 `mispredict` 가 감지되는 것을 확인했다.

`clock`이 발생할 때마다 0으로 초기화해주는 코드를 추가했고,  
명령어가

`00000000` 인 경우 `mispredict` 인지 확인하지 않도록 구현했다.

```

mispredict = 0;
if(IF_ID_instruction != 32'b0 && !JR) begin
    if (IF_ID_predicted_taken != actually_taken) begin // PHT가 잘 동작했는지.
        mispredict = 1;
    end else if (IF_ID_predicted_taken && IF_ID_PC != actual_target) begin /
        mispredict = 1;
    end
end

```

이처럼 수정한 후, testcase 3, 4가 해결되었따.

## 6-3. testcase 6 실패

디버깅 결과, `Hazard_stall`이 2라고 검출되었는데, 그 다음 cycle에는 0이 되어버렸다.  
즉, 충분히 유지되지 못했다.

```

PC: 0000003c, inst: 8f880000
IF_ID_PC: 00000038, IF_ID_instruction: 11090010
IF_ID_rs: 8, IF_ID_rt: 9
ID_EX_PC: 00000034, ID_EX_RegWrite: 1, ID_EX_rd: 0, ID_EX_rt: 9
ForwardA: 0, ForwardB: 0
operand1: 00001800, EX_MEM_alu_result: 00001800, wr_data: 0000180c, ID_EX_rd_data1: 00001800
operand2: 00000004, EX_MEM_alu_result: 00001800, wr_data: 0000180c, ID_EX_rd_data2: 00000000
EX_MEM_PC: 00000030, EX_MEM_RegWrite: 1, EX_MEM_wr_addr: 8
EX_MEM -> mem_write_data: 13b3bf44, EX_MEM_MemWrite: 0
MEM_WB_PC: 0000002c, MEM_WB_RegWrite: 0, MEM_WB_wr_addr: 8, MEM_WB_wr_data: 0000180c
Hazard_stall: 2
misprediction: 0
=====
PC: 0000003c, inst: 8f880000
IF_ID_PC: 00000038, IF_ID_instruction: 11090010
IF_ID_rs: 8, IF_ID_rt: 9
ID_EX_PC: 00000000, ID_EX_RegWrite: 0, ID_EX_rd: 0, ID_EX_rt: 0
ForwardA: 0, ForwardB: 0
operand1: 00000000, EX_MEM_alu_result: 00001804, wr_data: 44115b44, ID_EX_rd_data1: 00000000
operand2: 00000000, EX_MEM_alu_result: 00001804, wr_data: 44115b44, ID_EX_rd_data2: 00000000
EX_MEM_PC: 00000034, EX_MEM_RegWrite: 1, EX_MEM_wr_addr: 9
EX_MEM -> mem_write_data: 00000000, EX_MEM_MemWrite: 0
MEM_WB_PC: 00000030, MEM_WB_RegWrite: 1, MEM_WB_wr_addr: 8, MEM_WB_wr_data: 44115b44
Hazard_stall: 0
misprediction: 0
=====
```

단순하게 이전 cycle에 `hazard_stall` 이 2였는지 여부를 담고 있는 `prev_hazard` 를 생성하여 `hazard_stall`이 2가 감지된 경우 2 cycle을 멈출 수 있도록 했다.

```

if (hazard_stall > 0 || prev_hazard > 0) begin
    // ...
    if(hazard_stall == 2) prev_hazard <= 1;
    if(prev_hazard == 1) prev_hazard <= 0;
```

해당 코드를 추가한 이후 문제가 해결되었다.

Simulation success!!!