Project 3

Pipeline

Prepared by ChoJungHee

32194394@dankook.ac.kr

개요

이 프로젝트의 목표는 MIPS ISA(Instruction Set Architecture)를 사용하는 Pipeline을 수행하는 C코드를 구현하는 것이 목표이다.

이 프로젝트에서 Single Cycle은 폰 노이만의 구조를 따라간다.

이 Single Cycle을 pipeline화 하여 구현한다.

간략한 프로그램의 전체적인 흐름은 다음과 같다.

1. 장치 초기화 및 기본값 설정
2. 프로그램 로드
3. Pipeline
    1. Memory Operation
    2. Writeback
    3. Fetch
    4. Decode
    5. Execute
    6. Hazard Detect (and fowarding)
    7. Latch update

이 보고서는 5가지 내용으로 구성되어 있다.

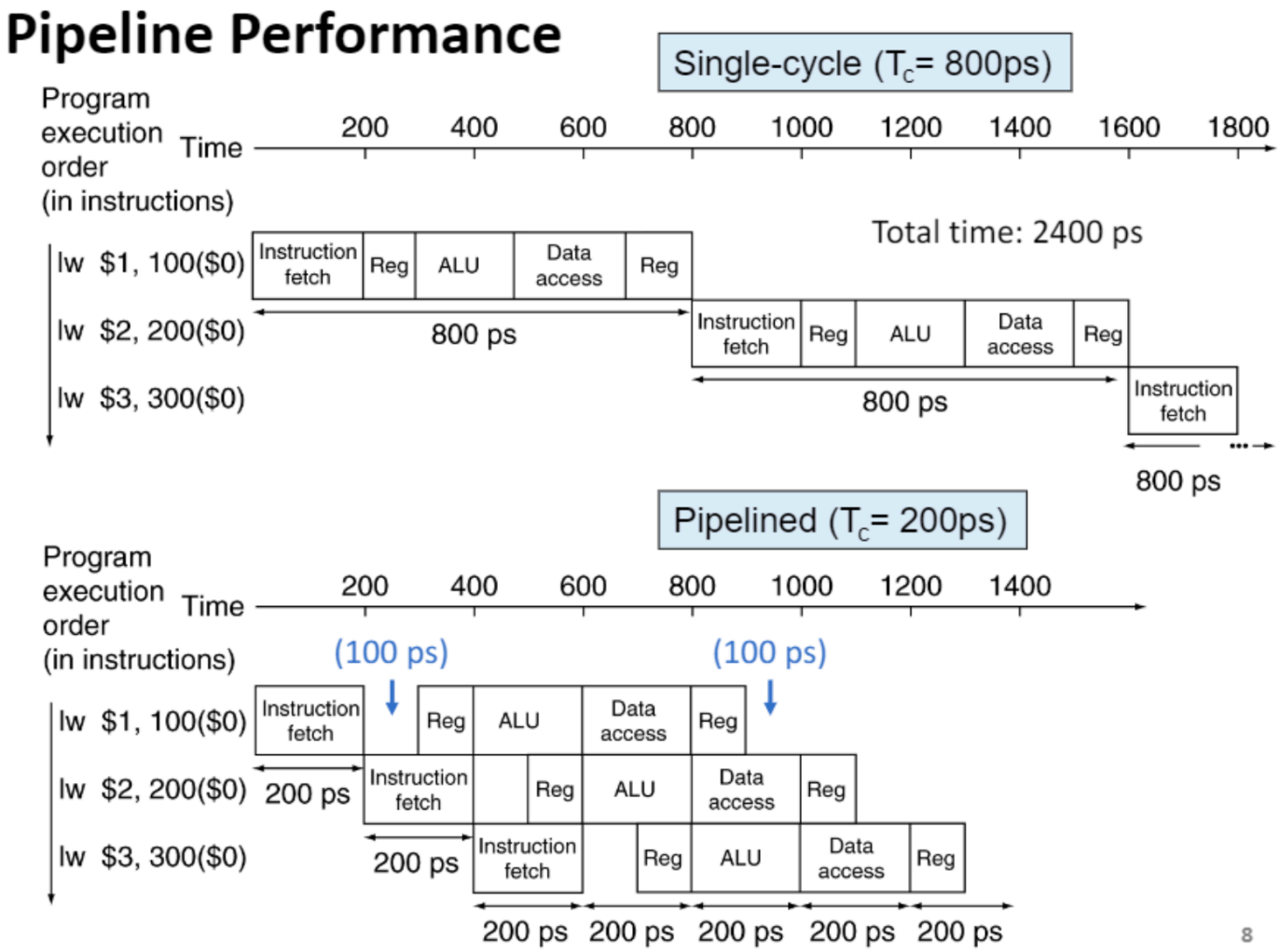
1. 프로젝트에서 사용되는 개념과 배경지식.
2. 프로젝트가 어떻게 디자인 되어있는지 서술.
3. 프로젝트를 어떤환경에서 어떠한 방식으로 빌드 했는지에 대한 정보를 제공.
4. 프로젝트의 실행 결과
5. 프로젝트를 하며 느꼈던 점과 힘들었던 점.

1. 배경

1.1 주요 개념

> single cycle에 관한 내용은 이전 프로젝트에서 서술하였으므로 생략

PIPELINE

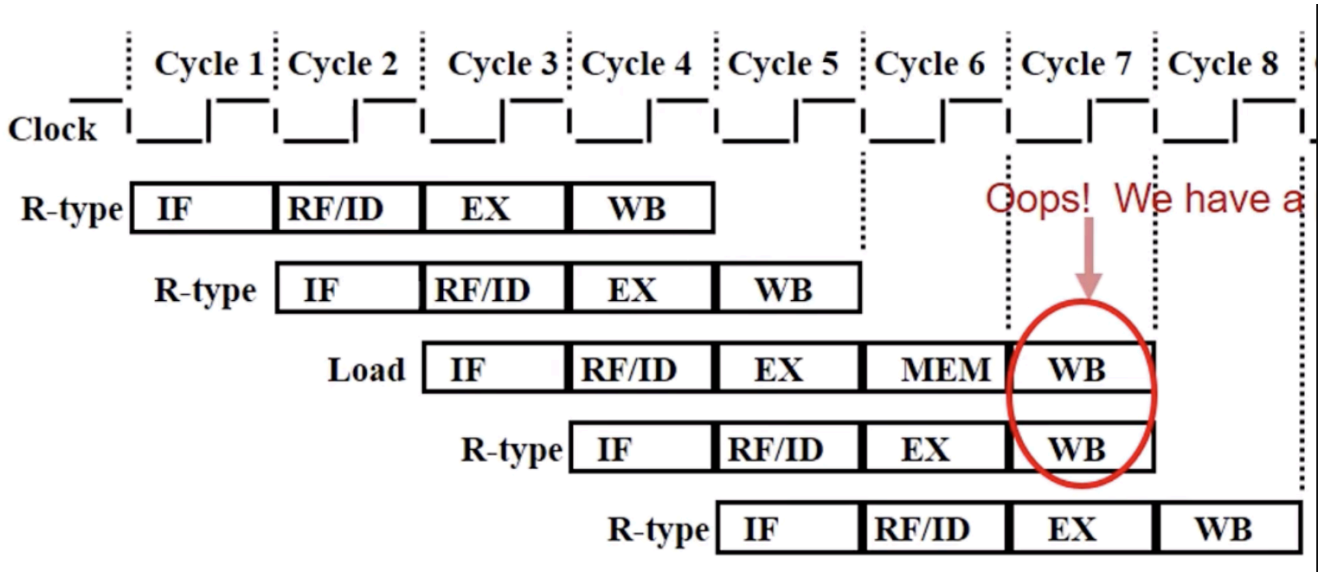


- 한 사이클에 서로 다른 명령어들을 병렬 실행하여 CPU 처리량을 최적화
- 단 병렬 실행시 여러가 문제점들이 존재하는데 그것을 Hazard라 함

Pipeline Hazard

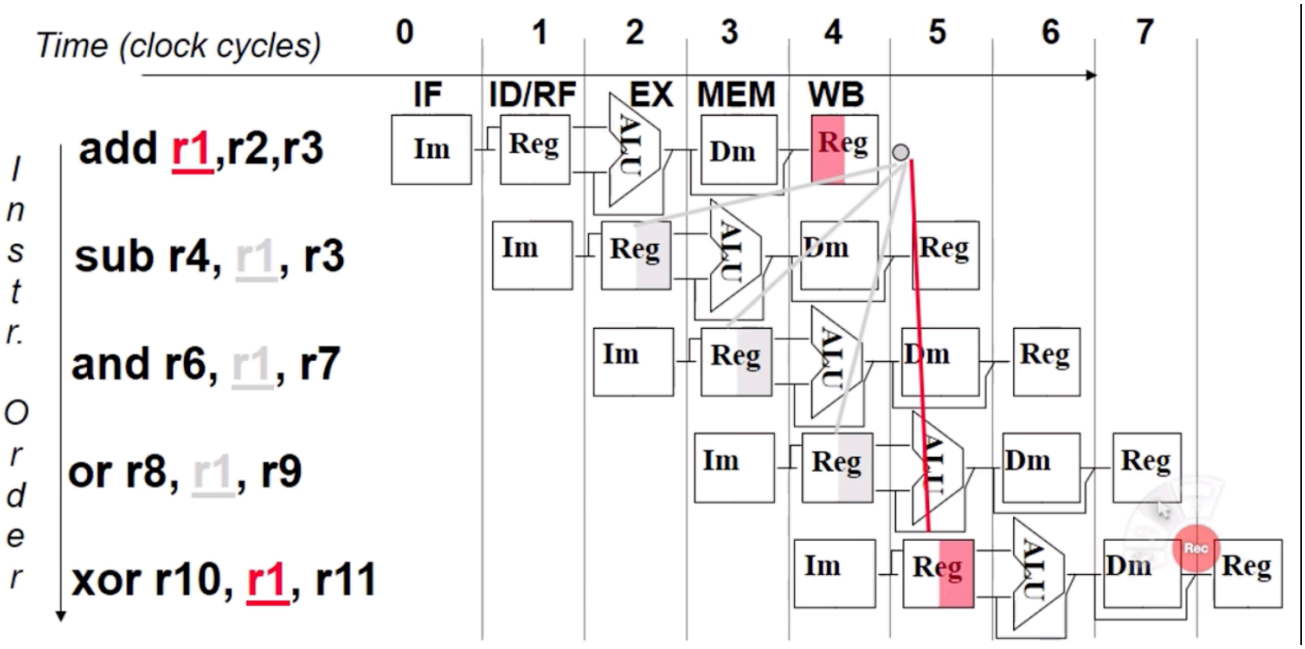
1. Structure Hazard

   여러 단계에서 같은 자원(메모리, 레지스터 등)를 동시에 참조하는 상황



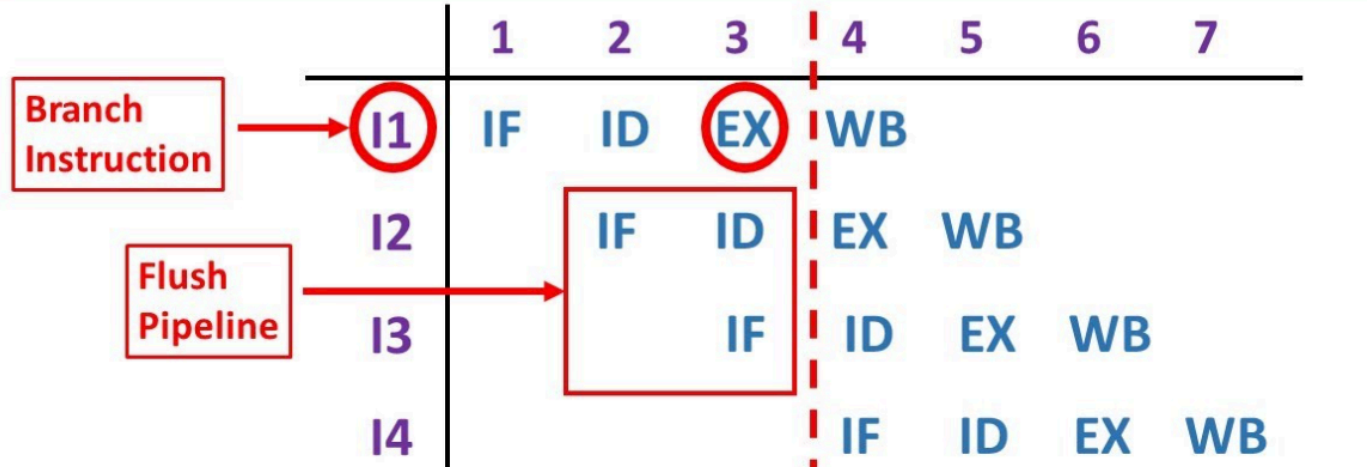
2. Data Hazard

   Data의 저장이 완료되기 전 읽어야 하는 상황 (load-use등 여러 케이스 존재)



3. Control Hazard

   Branch 혹은 jump명령어가 진행되면 점프 전까지 미리 만들어두었던 파이프라인이 쓸모없어져 flush해야하는 상황



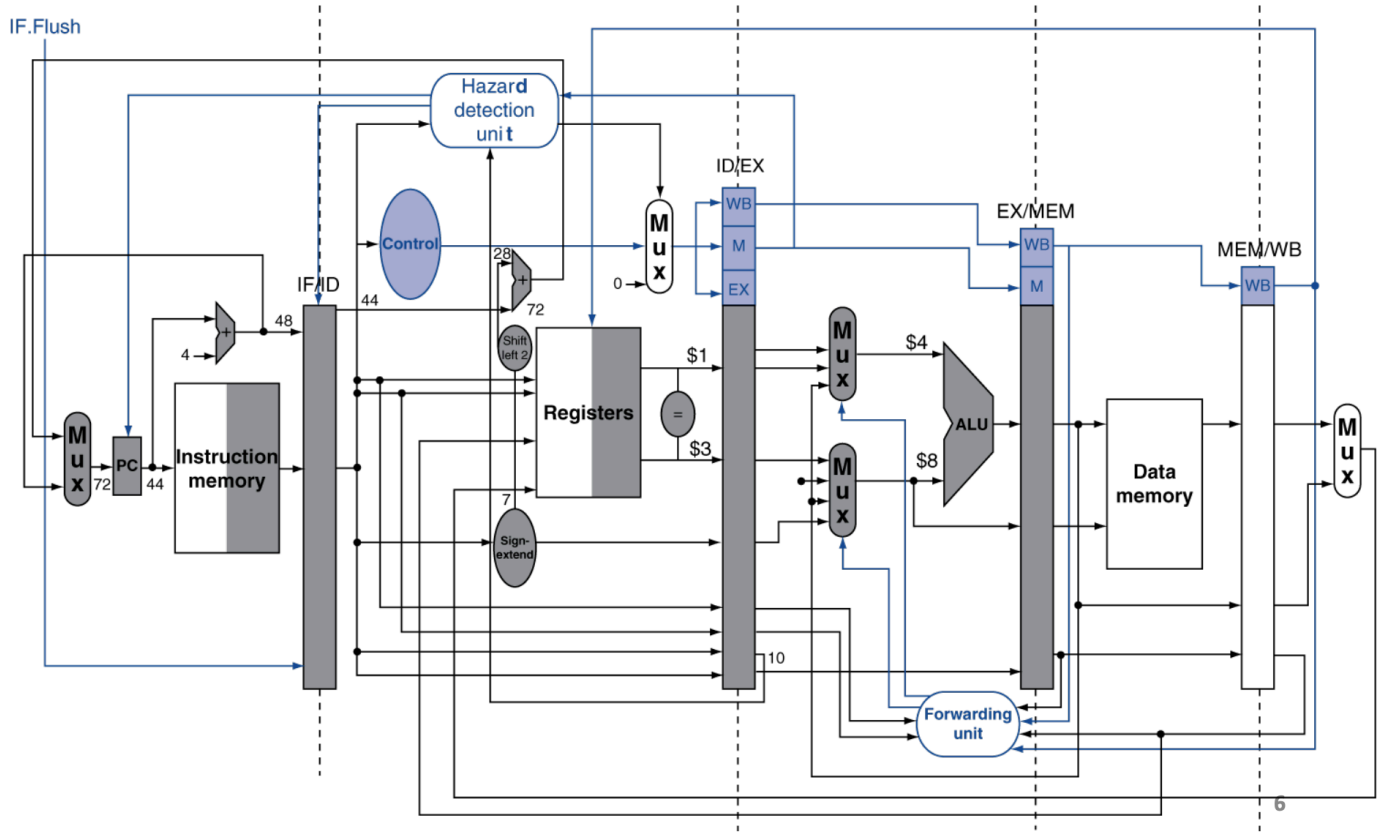
Hazard Solution

1. Structure Hazard
    - 모든 명령어의 단계 개수를 일치시켜 같은 자원의 소모를 예방한다.
    - 메모리의 분리 (IF, MEM)
2. Data Hazard
    - Detect and wait : 데이터가 저장될때까지 모든 명령어들을 stall
    - Forwarding : 데이터가 저장되기전 다음 명령어로 미리 보냄
3. Control Hazard
    - Detect and wait : 분기가 끝날때까지 모든 명령어들을 stall
    - Static branch prediction : 분기 명령어의 결과를 실제로 실행하지 않고 예측하는 방법 (컴파일 타임 분기 예측) , 항상 분기를 가져가거나 항상 가져가지 않거 둘중 하나.
    - Dynamic branch prediction : 분기 명령어의 결과를 저장하여 그 기록을 바탕으로 예측하는 방법 (런 타임 분기 예측), Last Time Branch Prediction, Level Local Branch Prediction등이 존재.

2. 구현

2.1 설계

2.1-1 데이터 패스



이번 과제에서의 데이터패스 과제 pdf에 나와있는 데이터패스에 저번 Single Cycle과제를 합쳐 만들었다. 한가지 데이터패스와 다른점이 있다면 hazard 감지 및 처리를 EXE단계 다음에 진행한다.

2.1-2 Latch

각 단계들의 값을 저장하는 래치.

한 단계당 in, out을 배열로 구분하여 Latch[0] == in , Latch[1] == out 으로 구분하였다.

모든 단계는 valid 값을 통하여 stall이 진행된다.

각 단계는 이전 단계의 out의 valid가 1일 경우에 단계를 진행한다.

명령어의 한 단계의 in부분의 valid를 0으로 바꾼다면 latch update를 통해 in->out이 복사되는데 valid비트가 복사되며 이후의 단계들이 stall 되는 원리이다(단계를 진행을 안하게 된다면 latch의 모든값이 0으로 초기화됨).

```
typedef struct IFID_latch {
    unsigned int valid;
    unsigned int next_PC;
    unsigned int data;

    unsigned int BH_found;
    unsigned int BH_index;
    unsigned int pretaken;
} IFID_latch;
```

BH_found , BH_index, pretaken은 동적 분기예측에 쓰이는 변수이며 data는 명령어를 저장한다.

```
typedef struct _IDEX_latch {
    CU control;
    unsigned int valid;
    unsigned int next_PC;
    unsigned int read_data1;
    unsigned int read_data2;

    unsigned int opcode;
    unsigned int rs;
    unsigned int rt;
    unsigned int rd;
    unsigned int addr;

    unsigned int imm;
    unsigned int sign_extened_data;
    unsigned int zero_extened_data;
    unsigned int lui_shifted_data;

    unsigned int shamt;
    unsigned int funct;

    unsigned int BH_found;
    unsigned int BH_index;
    unsigned int pretaken;
} IDEX_latch;
```

```
typedef struct _EXMEM_latch {
    CU control;
    ALU_flag flag;
    unsigned int valid;
    unsigned int next_PC;
    unsigned int branch_target;
    unsigned int alu_result;
    unsigned int read_data1;
    unsigned int read_data2;
    unsigned int rd;
} EXMEM_latch;
```

```
typedef struct _MEMWB_latch {
    CU control;
    unsigned int valid;
    unsigned int next_PC;
    unsigned int read_data;
    unsigned int alu_result;
    unsigned int rd;
} MEMWB_latch;
```

MEMWB래치의 read_data는 메모리로부터 읽어온 값이다.

```
/* in cpu */
IFID_latch IFID[2];
IDEX_latch IDEX[2];
EXMEM_latch EXMEM[2];
MEMWB_latch MEMWB[2];
```

```
void latch_update(CPU* cpu){
  memcpy(&(cpu->IFID[1]), &(cpu->IFID[0]), sizeof(IFID_latch));
  memcpy(&(cpu->IDEX[1]), &(cpu->IDEX[0]), sizeof(IDEX_latch));
  memcpy(&(cpu->EXMEM[1]), &(cpu->EXMEM[0]), sizeof(EXMEM_latch));
  memcpy(&(cpu->MEMWB[1]), &(cpu->MEMWB[0]), sizeof(MEMWB_latch));

  memset(&(cpu->IFID[0]), 0, sizeof(IFID_latch));
  memset(&(cpu->IDEX[0]), 0, sizeof(IDEX_latch));
  memset(&(cpu->EXMEM[0]), 0, sizeof(EXMEM_latch));
  memset(&(cpu->MEMWB[0]), 0, sizeof(MEMWB_latch));
}
```

한 사이클이 끝나면 래치를 업데이트한다. In -> out으로 복사하고 in을 초기화한다.

2.1-3 Branch Prediction

1. detect and wait

```
void detect_wait_execution(CPU* cpu, unsigned int branch_taken, unsigned int addr){
    if(cpu->IDEX[1].control.branch){
        cpu->IFID[0] = cpu->IFID[1];
        cpu->IDEX[0].valid = 0;

        if(branch_taken){
            cpu->reg.PC = addr;
        }
        else{
            cpu->reg.PC = cpu->IFID[1].next_PC-4;
        }
    }
}
```

분기라면 일단 stall

2. ANT

Always not taken

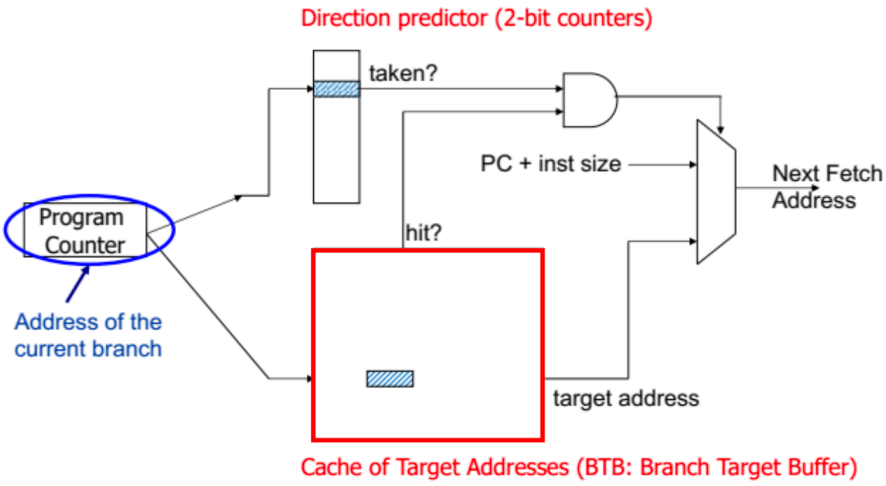
```
void ANT_prediction(CPU* cpu){
    cpu->reg.PC = cpu->reg.PC + 4;
    cpu->IFID[0].next_PC = cpu->reg.PC;
}

void ANT_execution(CPU* cpu, unsigned int branch_taken, unsigned int addr){
    if (branch_taken) {

    cpu->IFID[0].valid = 0;
    cpu->IFID[1].valid = 0;
    cpu->IDEX[0].valid = 0;
  }
  else {
    //pass
  }
    cpu->reg.PC = addr;
}
```

fetch 단계에서 prediction을 진행하고 실제 분기가 taken되면 예측이 틀렸기에 stall한다.

3. ALT

```
void ALT_prediction(CPU* cpu){
```

```
    int BH_index = (0x100 - 1) ;
    BH_index <<= 2;
    BH_index &= cpu->reg.PC;
    BH_index >>= 2;
  cpu->IFID[0].BH_index = BH_index;

  if (cpu->reg.PC == cpu->BH_unit[BH_index].branch_inst_addr) {
    cpu->IFID[0].BH_found = 1;
    cpu->IFID[0].next_PC = cpu->reg.PC + 4;

    cpu->reg.PC = cpu->BH_unit[BH_index].branch_target_addr;
  }
  else {
    cpu->IFID[0].BH_found = 0;
    cpu->reg.PC = cpu->reg.PC + 4;
    cpu->IFID[0].next_PC = cpu->reg.PC;
  }
}

void ALT_execution(CPU* cpu, unsigned int branch_taken, unsigned int addr){
  if (cpu->IDEX[1].BH_found==1) {
    if (branch_taken) {
      //pass
    }
    else {
      cpu->reg.PC = cpu->IDEX[1].next_PC;

      cpu->IFID[0].valid = 0;
      cpu->IFID[1].valid = 0;
      cpu->IDEX[0].valid = 0;
    }
  }
  else {
    if (branch_taken) {
      cpu->BH_unit[cpu->IDEX[1].BH_index].branch_inst_addr = cpu->IDEX[1].next_PC -
4;
      cpu->reg.PC = addr;
      cpu->BH_unit[cpu->IDEX[1].BH_index].branch_target_addr = cpu->reg.PC;

      cpu->IFID[0].valid = 0;
      cpu->IFID[1].valid = 0;
      cpu->IDEX[0].valid = 0;
    }
  }
}
```

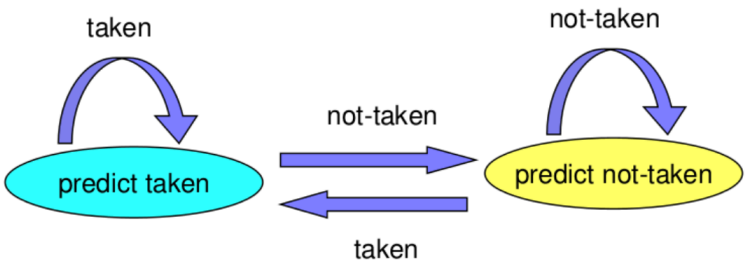


fetch단계에서 예측할때 BTB의 인덱싱을 위해 PC값을 사용한다. 해당 인덱스의 BTB값에 대한 유효한 정보가 존재한다면 분기한다. 분기경로는 분기의 예측 정보를 사용한다.

4. LTP

```
void LTP_prediction(CPU* cpu){
  int BH_index = (0x100 - 1) ;
    BH_index <<= 2;
    BH_index &= cpu->reg.PC;
    BH_index >>= 2;
  cpu->IFID[0].BH_index = BH_index;

  //0: not taken
  //1: taken
  if (cpu->reg.PC == cpu->BH_unit[BH_index].branch_inst_addr) {
    cpu->IFID[0].BH_found = 1;
    switch (cpu->BH_unit[BH_index].prediction_bit){
    case 0:     cpu->IFID[0].pretaken = 0;
              cpu->reg.PC = cpu->reg.PC + 4;
              cpu->IFID[0].next_PC = cpu->reg.PC;                    break;

        case 1:     cpu->IFID[0].pretaken = 1;
              cpu->IFID[0].next_PC = cpu->reg.PC + 4;
              cpu->reg.PC = cpu->BH_unit[BH_index].branch_target_addr;  break;
    }
  }
  else {
        cpu->reg.PC = cpu->reg.PC + 4;
    cpu->IFID[0].BH_found = 0;
    cpu->IFID[0].pretaken = 0;
    cpu->IFID[0].next_PC = cpu->reg.PC;
  }
}

void LTP_execution(CPU* cpu, unsigned int branch_taken, unsigned int addr){
    //0: not taken
    //1: taken
```

```
  if (cpu->IDEX[1].BH_found) { //found BH_unit in IF stage
    if (cpu->IDEX[1].pretaken) { //predicted taken
      if (branch_taken) {
        cpu->BH_unit[cpu->IDEX[1].BH_index].prediction_bit = 1;
      }
      else {
        cpu->BH_unit[cpu->IDEX[1].BH_index].prediction_bit = 0;
        cpu->reg.PC = cpu->IDEX[1].next_PC;

        cpu->IFID[0].valid = 0;
        cpu->IFID[1].valid = 0;
        cpu->IDEX[0].valid = 0;
      }
    }
    else { //predicted not taken
      if (branch_taken) {
        cpu->BH_unit[cpu->IDEX[1].BH_index].prediction_bit = 1;
        cpu->reg.PC = cpu->BH_unit[cpu->IDEX[1].BH_index].branch_target_addr;

        cpu->IFID[0].valid = 0;
        cpu->IFID[1].valid = 0;
        cpu->IDEX[0].valid = 0;
      }
      else {
        cpu->BH_unit[cpu->IDEX[1].BH_index].prediction_bit = 0;
      }
    }
  }
  else { // not found BH_unit in IF stage
    if (branch_taken) {
      cpu->BH_unit[cpu->IDEX[1].BH_index].branch_inst_addr = cpu->IDEX[1].next_PC -
4;
      cpu->reg.PC = addr;
      cpu->BH_unit[cpu->IDEX[1].BH_index].branch_target_addr = cpu->reg.PC;
      cpu->BH_unit[cpu->IDEX[1].BH_index].prediction_bit = 1;

      cpu->IFID[0].valid = 0;
      cpu->IFID[1].valid = 0;
      cpu->IDEX[0].valid = 0;
    }
  }
}
```



가장 최근의 결과로 분기예측을 하는 방법.

각BTB값에는 다음 분기에 대한 하나의 예측 비트가 포함되어 있으며, 예측이 실행되지 않으면 0으로 표시되고 예측이 실행되면 하나의 예측 비트가 1로 표시된다. 예측 비트는 분기 결과를 예측하는 데 사용한다.

2.1-4 BTB

Branch Target Buffer

```
typedef struct _branch_history {
    unsigned int branch_inst_addr;
    unsigned int branch_target_addr;
    unsigned int prediction_bit;
} BHU;
```

```
/* in cpu */
BHU BH_unit[0x100];
```

2.1-5 Fowarding

```
void H_detect(CPU* cpu){
  cpu->foward_unit[0] = 0b00;
  cpu->foward_unit[1] = 0b00;

  // EXMEM hazard
  if (cpu->EXMEM[0].rd != 0 && cpu->EXMEM[0].control.reg_write) {
        if(cpu->data_hazard_option == 0){ //wait
            if((cpu->EXMEM[0].rd == cpu->IDEX[0].rs) || (cpu->EXMEM[0].rd == cpu-
>IDEX[0].rt)){
                cpu->IFID[0].valid = 0;
                cpu->IFID[1].valid = 0;
                cpu->IDEX[0].valid = 0;
                cpu->reg.PC = cpu->EXMEM[0].next_PC;
            }
        }
        else if(cpu->data_hazard_option == 1){ //foward
```

```
            if (cpu->EXMEM[0].rd == cpu->IDEX[0].rs) {
                cpu->foward_unit[0] = 0b10;
            }
            if (cpu->EXMEM[0].rd == cpu->IDEX[0].rt) {
                cpu->foward_unit[1] = 0b10;
            }
        }
    }

    // MEMWB hazard
    if (cpu->MEMWB[0].rd != 0 && cpu->MEMWB[0].control.reg_write) {
        if(cpu->data_hazard_option == 0){ //wait
            if ((cpu->EXMEM[0].rd != cpu->IDEX[0].rt && cpu->MEMWB[0].rd == cpu-
>IDEX[0].rt) || (cpu->EXMEM[0].rd != cpu->IDEX[0].rs && cpu->MEMWB[0].rd == cpu-
>IDEX[0].rs)){
                //stall 1 but fail
                cpu->IFID[0].valid = 0;
                cpu->IFID[1].valid = 0;
                cpu->IDEX[0].valid = 0;
                cpu->reg.PC = cpu->EXMEM[0].next_PC;
            }
        }
        else if(cpu->data_hazard_option == 1){ //foward
            if (cpu->EXMEM[0].rd != cpu->IDEX[0].rs && cpu->MEMWB[0].rd == cpu-
>IDEX[0].rs) {
                cpu->foward_unit[0] = 0b01;
            }
            if (cpu->EXMEM[0].rd != cpu->IDEX[0].rt && cpu->MEMWB[0].rd == cpu-
>IDEX[0].rt) {
                cpu->foward_unit[1] = 0b01;
            }
        }
    }
}
```

옵션값을 통하여 data hazard를 wait할건지 fowarding 할건지 선택이 가능하다.

Forward_unit은 rs rt에 대한 Fowarding 정보가 담겨있다.

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The 1st ALU operand comes from the register file
ForwardA = 10	EX/MEM	The 1st ALU operand is forwarded from the prior ALU result
ForwardA = 01	MEM/WB	The 1st ALU operand is forwarded from data memory or from an earlier ALU result
ForwardB = 00	ID/EX	The 2nd ALU operand comes from the register file
ForwardB = 10	EX/MEM	The 2nd ALU operand is forwarded from the prior ALU result
ForwardB = 01	MEM/WB	The 2nd ALU operand is forwarded from data memory or from an earlier ALU result

빌드 환경

Ubuntu

> Ubuntu 22.04.2 LTS (GNU/Linux 5.15.0-69-generic aarch64)
>
> gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0

M2 MacBook Air

> Apple clang version 14.0.0 (clang-1400.0.29.202)
>
> Target: arm64-apple-darwin22.3.0
>
> Thread model: posix

Make command

> make main -> build the execution program
>
> make clean -> clean the object files

make main

```
jojeonghui@MacBook-AirM2 CA3 % make main
gcc -Wall -c src/alu.c -o src/alu.o
gcc -Wall -c src/acu.c -o src/acu.o
gcc -Wall -c src/cu.c -o src/cu.o
gcc -Wall -c src/memory.c -o src/memory.o
gcc -Wall -c src/register.c -o src/register.o
gcc -Wall -c src/cpu.c -o src/cpu.o
ar rv ./src/cycle.a ./src/alu.o ./src/acu.o ./src/cu.o ./src/memory.o ./src/register.o ./src/cpu.o
ar: creating archive ./src/cycle.a
a - ./src/alu.o
a - ./src/acu.o
a - ./src/cu.o
a - ./src/memory.o
a - ./src/register.o
a - ./src/cpu.o
gcc -Wall -c ./src/predictor.c -o ./src/predictor.o
gcc -Wall -o main main.c ./src/predictor.o -L. ./src/cycle.a
rm -f ./src/alu.o ./src/acu.o ./src/cu.o ./src/memory.o ./src/register.o ./src/cpu.o ./src/cycle.a ./src/predictor.o
jojeonghui@MacBook-AirM2 CA3 %
```

make clean

```
jojeonghui@MacBook-AirM2 CA3 % make clean
rm -f ./src/alu.o ./src/acu.o ./src/cu.o ./src/memory.o ./src/register.o ./src/cpu.o ./src/cycle.a main ./src/predictor.o
jojeonghui@MacBook-AirM2 CA3 %
```

결과

```
jojeonghui@MacBook-AirM2 CA3 % ./main
Error : Input file is not exist
 ./main (input file) (data hazard option) (control hazard option)
data hazard option
        detect and wait : 0
        fowarding : 1
control hazard option
        detect and wait : 0
        ANT : 1
        ALT : 2
        LTP : 3
```

실행 파일에 인자가 충분하지 않다면 안내 설명후 종료.

```
Error : Input file is not exist
 ./main (input file) (data hazard option) (control hazard option)
data hazard option
        detect and wait : 0
        fowarding : 1
control hazard option
        detect and wait : 0
        ANT : 1
        ALT : 2
        LTP : 3
jojeonghui@MacBook-AirM2 CA3 % ./main input/input1/input.bin 1 1
```

정상적인 실행파일 실행

결과

```
        rt<-[MEMWB Fowarding]

Cycle[169] (PC: 0x68)
 [IF]
        0x00000000       pc: 0x68
        [PC update] pc <- 0x6c
 [ID]
        type: R, opcode: 0x0, rs: 0x1f (R[31]=0xffffffff), rt: 0x0 (R[0]=0x0), rd: 0x0 (0), shamt: 0x0, funct: 0x8
 [EXE]
        rs: 1d(r[29]=fffff0), imm: fffff0, alu result : 1000000
 [MEM]
        [MEMORY ACCESS] read_data <- M[0x00fffffc] = 0
 [WB]
        r[29] <- fffff0
 [Fowarding]

Cycle[170] (PC: 0x6c)
 [IF]
        0x00000000       pc: 0x6c
        [PC update] pc <- 0x70
 [ID]
        Nop
 [EXE]
        rs: 1f(r[31]=ffffffff), rt: 0(r[0]=0), alu result : ffffffff
        [PC update (jump register)] pc <- 0xffffffff
 [MEM]
 [WB]
        r[30] <- 0
 [Fowarding]

=========================PROGRAM RESULT============================
Return value (R[2]) :                   45
Total Cycle :                           170
Executed 'R' instruction :              13
Executed 'I' instruction :              101
Executed 'J' instruction :              0
Number of Branch Taken :                11
Number of Memory Access Instruction :   66
Number of STALL :                       87
===================================================================
```

느낀점

기말고사 기간이 길어지며 과제할 시간이 상당히 부족했고, 구조를 짜도 적용하는중에 오류가 나면 컴파일 오류라면 다행이지만 아닌 경우 디버그를 하면서 오류를 수정하기가 너무 어려웠다. 또한 fowarding이나 dectect and wait에서 stall을 진행할때 mips-gcc가 자동으로 넣어주는 nop코드(jal 다음, lw 다음)를 고려해야할지 안해야할지 (컴파일러가 stall을 피하기 위한 행위들?)가 고민이 많이 되었다. 이번 학기 이런 재미있는 과제를 통하여 생각의 범위와 코드의 구현등에 대하여 깊은 생각을 가질 수 있어 매우 만족했다. 시간이 부족하여 마지막 과제에 모든 힘을 다해내지 못한게 아쉽다.