HW3_32194074_정민호_document

- 소개
  - 간단한 설명
    - 이전에 만들었던 Mips Single Cycle Simulator의 함수와 구조체를 적극 재사용하였으며, 가독성을 끌어올리기 위해 직관적인 알고리즘과 코드 작성을 중점으로 하였습니다.
    - 모든 Requirements를 충족합니다
    - Detect and forward/bypass 및 ID Stage 단계에서의 Branch Taken을 추가로 구현하였습니다.

- 배경
  - 중요한 개념
    - Pipeline
      - 명령어를 병렬로 실행하여 전체적인 Throughput을 줄이는 기술입니다. Single Cycle과 다른 점은 다음 유닛이 추가된 것 입니다.
        - Pipeline Register
          - 파이프라인 레지스터는 파이프라인의 각 단계 사이에 위치한 레지스터입니다. 파이프라인은 명령어를 여러 단계로 분할하여 동시에 실행하는데, 이 때 파이프라인 레지스터는 단계 사이에서 데이터를 전달하고 제어 신호를 저장하는 역할을 합니다. 각 레지스터는 다음과 같습니다
            - IF/ID Register
            - ID/EX Register
            - EX/MEM Reigster
            - MEM/WB Register
        - Hazard Detection Unit
          - 파이프라인에서는 명령어들이 동시에 실행되기 때문에 하자드(hazard)가 발생할 수 있습니다. 하자드는 명령어 실행 순서나 데이터 종속성으로 인해 잘못된 결과를 초래하는 상황을 의미합니다. 하자드 감지 유닛은 이러한 하자드를 감지하고 처리하는 역할을 합니다. 종류로는 Data hazard(데이터 종속성)과 Control hazard(제어 종속성)이 있습니다. 데이터 종속성은 한 명령어가 이전 명령어의 결과에 의존하는 경우 발생하며, 제어 종속성은 조건 분기 명령어와 같이 명령어 실행 흐름이 바뀌는 경우 발생합니다. 하자드 감지 유닛은 이러한 종속성을 검출하여 필요한 제어 신호를 생성하고, 명령어의 실행을 조절하여 하자드를 해결합니다.
        - Forwarding Unit
          - 포워딩은 데이터 종속성으로 인한 하자드를 해결하기 위한 기술입니다. 데이터 종속성이 있는 경우 이전 명령어의 결과를 사용하는 다음 명령어가 대기해야 하는 상황이 일어나는데, 대기 없이 바로 데이터를 전달하는 통로를 배치해둠으로써 대기 시간을 없애고 명령어 실행을 가속화합니다.
      - Pipleline의 순서
        - Instruction Fetch (IF)
          - 명령어 메모리에서 명령어를 가져오는 단계입니다. PC를 증가시켜 다음 명령어의 주소를 계산하고 해당 주소에서 명령어를 가져옵니다.
        - Instruction Decode (ID)
          - 가져온 명령어를 해석(Decode)하고 필요한 레지스터의 값들을 읽어옵니다.
        - Execute (EX)
          - ALU에서산술 및 논리 연산을 수행하는 단계입니다.
        - Memory Access (MEM)
          - 데이터 메모리에 접근하여 메모리에서 데이터를 읽거나(LW) 쓸 수 있습니다(SW)
        - Write Back (WB)

        - 최종 결과(ALUResult or memData)를 레지스터 파일에 쓰는 단계입니다.

    - 구현에 대한 특별한 고려사항
        - bin 파일에서 명령어 가지고 오기
            - scanf를 통해서 사용자가 이진 파일 둘 중 하나를 선택합니다.
            - fread 명령어를 이용하여 bin 파일에서 32bit씩 1줄씩 iMem이라는 u_int32_t 배열에 저장합니다.

        - 예외를 처리하는 방법
            - 파일 처리 과정에서 오류가 발생하면 즉시 프로그램을 종료합니다.
            - 명령어 처리 과정에서는 고정 Input에 의하여 오류가 발생하지 않기 때문에 구현하지 않았습니다.

        - Hazard 처리 구현
            - Data Forwarding/bypass
                - EX Stage 구현 단계에서 알고리즘을 그대로 삽입하여 간단하게 구현했습니다.
            - Load-Use Hazard
                - ID Stage 구현 단계에서 알고리즘을 그대로 삽입하여 간단하게 구현했습니다.
                - 컴파일러를 통해 추출한 bin 파일에서는 이미 Load-Use Hazard가 해결되어 nop이 추가 돼 있기 때문에, 본 프로그램에서는 Load-Use Hazard의 알고리즘 부분은 주석처리 했습니다.
            - Control Hazard
                - 별도의 ALU Unit을 삽입하여 구현했습니다.
                    - Adder
                    - Comparator

        - Memory 크기
            - Data Memory : Memory[0xFFFFFFF]
                - 처음에 Data Memory의 크기를 0xFFFFFFFF (32비트)로 만드려고 했으나, 크기 오류 때문에 0xFFFFFFF(28비트)로 구현하였습니다.
            - Instruction Memory : iMem[1000]
                - 예시 input.bin 파일보다 넉넉한 공간을 설정했습니다.

- 구현
    - 프로그램 설계 방식(디자인)
        - 모든 기능적 설명은 소스코드에 100줄 가량의 주석으로 작성해뒀습니다.
        - 이 프로그램은 함수화와 가독성에 초점을 둔 프로그램으로, 전체적인 흐름은 main()에서 진행하며, “00. HW3.pdf p.6” 이미지에 있는 각 유닛(AndGate 포함)들을 모두 함수나 알고리즘으로 구현했습니다.
        - 따라서 각 입출력 라인들은 함수의 매개변수(입력), 함수의 return(출력)으로 구현하였습니다.
        - 출력은 HW3 pdf에 있는 예시를 그대로 구현하였습니다.
        - 헤더파일과 소스 파일을 따로 만들어뒀습니다.
            - 헤더 파일 (struct.h)
                - 처음엔 main.c에 모두 구현해놨으나, 디버깅 과정에서 가독성 부분에서 어려움을 느끼게 되어 아예 struct.h를 만들어서 구조체, 열거형, 전역변수, 함수 선언부를 모아뒀습니다.
                - 구조체 및 열거형을 적극적으로 활용하여 소스코드의 가독성을 높였습니다

```
#define iMemSize 1000

// R
typedef enum {
    ADD = 0x20,
    ADDU = 0x21,
    AND = 0x24,
    JR = 0x08,
    JALR = 0x09,
    OR = 0x25,
    SLT = 0x2a,
    SLL = 0x00,
    SUB = 0x22
}enum_R;

// I
typedef enum {
    ADDI = 0x08,
    ADDIU = 0x09,
    ANDI = 0x0c,
    BEQ = 0x04,
    BNE = 0x05,
    LW = 0x23,
    SLTI = 0x0a,
    SLTIU = 0x0b,
    SW = 0x2b
} enum_I;

// J
typedef enum {
  J = 0x02,
  JAL = 0x03
} enum_J;

typedef struct instruction {
    char format;                // R, I, O
    u_int32_t instruction;      // Inst 31-00 (출력용)
    u_int8_t opcode : 6;        // Inst 31-26
    u_int8_t rs : 5;            // Inst 25-21
    u_int8_t rt : 5;            // Inst 20-16
    u_int8_t rd : 5;            // Inst 15-11
    u_int8_t shamt : 5;         // Inst 10-06
    u_int8_t func : 6;          // Inst 05-00
    u_int32_t addr : 26;        // Inst 25-00
    u_int32_t imm;          // Inst 15-00
} Instruction;

typedef struct {
    u_int8_t RegDst : 1; // regMux의 input으로, inst.rd(true) 또는 inst.rt(false)를 반환
    u_int8_t ALUSrc : 1; // ReadData2(false) 또는 Immediate(true) 둘 중 하나로 ALU에 반환
    u_int8_t MemtoReg : 1; // memMux의 input으로, readData(true) 또는 aluResult(false)로 반환
    u_int8_t RegWrite : 1; // processRegister의 input으로, 레지스터를 모드를 결정함
    u_int8_t MemRead : 1; // processData의 input으로, 메모리에서 값을 읽을 것인지(true) 안 읽을 것인지(false) 결정함
    u_int8_t MemWrite : 1; // processData의 input으로, 메모리로 값을 쓸 것인지(true) 안 쓸 것인지(false) 결정함
    u_int8_t Branch : 1; // pcAndgate의 input으로, (pc+4)+(imm<<2) 값을 pc로 할 것인지(true), (pc+4)로 할 것인지(fals
e) 결정함
    u_int8_t Jump : 1; // processJAddress의 input으로, pc를 Jump Address 값으로 할 것인지(true), pcAndgate의 값으로
 할 것인지(false) 결정함
    u_int8_t JR : 1; // processJAddress의 input으로, PC를 reg[$ra]값으로 할 것인지(true), 다른 값으로 할 것인지(false)
 결정함
    u_int8_t ALUOp; // ALUOp의 input으로, opcode를 읽고 ALUControl의 값을 정함
    u_int8_t IFFlush : 1; // IF 레지스터를 0으로 초기화 하는 Flush
    u_int8_t IDFlush : 1; // ID 레지스터를 0으로 초기화 하는 Flush
    u_int8_t EXFlush : 1; // EX 레지스터를 0으로 초기화 하는 Flush
} Control;

typedef struct ifid{
  Instruction inst;
  u_int32_t PC;
}IFID;

typedef struct idex{
  Instruction inst;
    u_int32_t PC;
    u_int32_t immediate;
  Control ctr;
  u_int32_t data1,data2;
  u_int32_t temp;
    u_int32_t ALUResult;
}IDEX;

typedef struct exmem{
  Instruction inst;
    u_int32_t PC;
    u_int32_t  immediate;
```

```
  Control ctr;
  u_int32_t data1, data2;
    u_int32_t temp;
  u_int32_t WriteReg, ALUResult, ReadData;
}EXMEM;

typedef struct memwb{
  Instruction inst;
    u_int32_t PC;
    u_int32_t  immediate;
  Control ctr;
  u_int32_t data1, data2;
    u_int32_t temp;
  u_int32_t WriteReg, ALUResult, ReadData;
}MEMWB;

// 레지스터 이름 열거형
typedef enum {
    $zero = 0, $at, $v0, $v1, $a0, $a1, $a2, $a3, $t0, $t1, $t2, $t3, $t4, $t5, $t6, $t7, $s0, $s1, $s2, $s3,
 $s4, $s5, $s6, $s7, $t8, $t9, $k0, $k1, $gp, $sp, $fp, $ra
} RegName_enum;

// 레지스터 이름 문자형
const char* RegName_str[] = {
    "$zero", "$at", "$v0", "$v1", "$a0", "$a1", "$a2", "$a3", "$t0", "$t1", "$t2", "$t3", "$t4", "$t5", "$t6",
"$t7", "$s0", "$s1", "$s2", "$s3", "$s4", "$s5", "$s6", "$s7", "$t8", "$t9", "$k0", "$k1", "$gp", "$sp", "$f
p", "$ra"
};
```

- 각 함수는 재사용성에 초점을 두고 작성했습니다. 자세한 설명은 마찬가지로 프로그램 코드 내의 주석에 기입했기 때문에 포괄적인 설명만 기입했습니다.

```
// 이진 파일에서 데이터를 한 줄씩(32비트) 읽은 다음 메모리 배열에 저장하는 함수
void readMipsBinary(FILE *file);

// R, I 타입의 명령어를 분석하여 연산하는 함수
void processALU();

// BEQ나 BNE와 같은 분기 명령어를 처리하는 함수
void BranchPrediction(u_int32_t opcode, u_int32_t data1, u_int32_t data2);
void BranchTaken();

// Binary를 명령어로 분할하여 저장하는 함수
void parseInstruction();

// OPCODE에 따라 Control값을 구분하는 함수
void processControl(u_int32_t opcode);

// 16비트 immdidate를 32비트 Unsigned int로 형변환하는 함수
u_int32_t signExtend(u_int16_t immediate);

// 각 Stage별 함수
void IF();
void ID();
void EX();
void MEM();
void WB();
```

- 소스 파일 main.c

```
#include <stdio.h>
#include <string.h>
#include "struct.h"

int main(){…

void readMipsBinary(FILE *file){…

void parseInstruction() {…

void processControl(u_int32_t opcode){…

u_int32_t signExtend(u_int16_t immediate) {…

void processALU() {…

void BranchPrediction(u_int32_t opcode, u_int32_t data1, u_int32_t data2){…

void BranchTaken() {…

void IF(){…

void ID(){…

void EX(){…

void MEM(){…

void WB(){…
```

- 보시는 바와 같이 main을 제외하면 함수의 정의부만 존재합니다. 중요한 구현 부분만 기술하겠습니다.
- int main()

```
int main(){
    FILE *file;

  printf("\n !! 1을 입력하시면 input1의 이진 파일을, 그 외 나머지를 입력하시면 input2의 이진 파일을 수행합니다 : ");
  int input;
  scanf("%d", &input);
  if (input == 1) file = fopen("input_1.bin", "r");
  else file = fopen("input_2.bin", "r");

    if (file == NULL){
        printf("파일을 열 수 없습니다.\n");
        return 0;
    }

  // 파일로부터 binary를 읽어서 Memory에 저장하고 파일을 닫음
  readMipsBinary(file);
  fclose(file);

    u_int32_t ret = 0;

  // 레지스터 초기화
  R[$sp] = 0x10000000;
  R[$ra] = 0xFFFFFFFF;

  while(1){
    printf("cycle: %d\n",cycle);

    IF();
    ID();
    EX();
    MEM();
```

```
    WB();

    // PC가 0xFFFFFFFF 라면 바로 종료
    if(PC == 0xFFFFFFFF) break;

    // 각 레지스터의 출력값을 입력값으로 초기화 (다음 명령어 실행을 위함)
    ifid[1] = ifid[0];
    idex[1] = idex[0];
    exmem[1] = exmem[0];
    memwb[1] = memwb[0];

    cycle++;
  }

    printf("\n=========================================\n");
    printf("| Return value (%s): 0x%x(%d)\n", RegName_str[$v0], R[$v0], R[$v0]);
    printf("| Total Cycle: %u\n", cycle);
    printf("| Executed 'R' instruction: %u\n", R_count);
    printf("| Executed 'I' instruction: %u\n", I_count);
    printf("| Executed 'J' instruction: %u\n", J_count);
    printf("| Number of Branch Taken: %u\n", branch_count);
  printf("| Number of Branch Not Taken Count: %d\n",not_branch_count);
    printf("| Number of Memory Access Instructions: %u\n", mem_count);
  printf("| Jump Count: %d\n",jump_count);
    printf("=========================================\n");

  return 0;
}
```

- 처음에 사용자로부터 1 또는 나머지 입력을 받아 input_1.bin 또는 input_2.bin을 읽어옵니다.
  - file이 없는 경우 예외처리합니다
- 이진파일에서 데이터를 읽고 iMem 배열에 저장하고 파일을 닫습니다.
- 레지스터를 초기값으로 초기화 합니다
- PC가 0xFFFFFFFF가 되기 전까지 5개의 Stage를 순차적으로 실행합니다.
- PC가 0xFFFFFFFF가 되면 결과값을 출력하고 종료합니다.
- ifid[1] = ifid[0] …
  - 0번 index를 값을 입력한 배열, 1번 index를 값을 출력할 배열로 정의했습니다.
  - 매 반복문마다 5개의 Stage 실행이 끝나면 마지막에 위와 같이 업데이트 합니다

- void processControl(u_int32_t opcode)

```
void processControl(u_int32_t opcode){
  // opcode가 Jump 계열인 경우
  if(opcode == J || opcode == JAL){
    // JAL이면 PC값으로 $ra 업데이트
    if (opcode == JAL) R[$ra] = idex[0].PC+8;

    // PC의 상위 4비트(256MB 범위)와 address의 4배수를 더한 값(offset)을 PC로 함
    PC = (ifid[1].PC & 0xF0000000) | (idex[0].inst.addr << 2);
    exe_count++;
    jump_count++;
  }

  if(opcode == 0x0){
        idex[0].ctr.RegDst = 1;
    } else {
        idex[0].ctr.RegDst = 0;
    }

    if((opcode == J) || (opcode == JAL)) {
        idex[0].ctr.Jump = 1;
    } else {
        idex[0].ctr.Jump = 0;
    }

  // Branch
    if((opcode == BEQ) || (opcode == BNE)) {
        idex[0].ctr.Branch = 1;
    } else {
        idex[0].ctr.Branch = 0;
    }

    if(opcode == LW) {
```

```
        idex[0].ctr.MemtoReg = 1;
        idex[0].ctr.MemRead = 1;
    } else {
        idex[0].ctr.MemtoReg = 0;
        idex[0].ctr.MemRead = 0;
    }

    if(opcode == SW) {
        idex[0].ctr.MemWrite = 1;
    } else {
        idex[0].ctr.MemWrite = 0;
    }

  if((opcode != 0) && (opcode != BEQ) && (opcode != BNE)){
        idex[0].ctr.ALUSrc=1;
    } else {
        idex[0].ctr.ALUSrc=0;
    }

  if((opcode != SW) && (opcode != BEQ)&& (opcode != BNE) && (opcode != JR) && (opcode != J) && (opcode != JAL))
        idex[0].ctr.RegWrite = 1;
    } else {
        idex[0].ctr.RegWrite = 0;
    }
}
```

- 각 Control 값을 기준으로 활성화 할 때 사용되는 명령어들을 조건문으로 나열해놓았습니다.

- void processALU()

```
void processALU() {
  u_int8_t opcode = idex[1].inst.opcode;
  u_int8_t func = idex[1].inst.func;
  u_int32_t data1 = idex[1].data1;
  u_int32_t data2 = idex[1].data2;

  // R 타입 명령어
  if (idex[1].ctr.RegDst == 1) {
    switch(func){
      case ADD:
        exmem[0].ALUResult = data1 + data2;
        exe_count++;
        break;

      case ADDU:
        exmem[0].ALUResult = data1 + data2;
        exe_count++;
        break;

      case AND:
        exmem[0].ALUResult = data1 & data2;
        exe_count++;
        break;

      case JR:
        PC=data1;
        memset(&ifid[0], 0, sizeof(IFID));
        memset(&idex[0], 0, sizeof(IDEX));
        break;

      case JALR:
        PC=data1;
        memset(&ifid[0], 0, sizeof(IFID));
        memset(&idex[0], 0, sizeof(IDEX));
        break;

      case OR:
        exmem[0].ALUResult = (data1|data2);
        exe_count++;
        break;

      case SLT:
        exmem[0].ALUResult = ((data1<data2) ? 1:0);
        exe_count++;
        break;

      case SLL:
        exmem[0].ALUResult = data2 << idex[1].inst.shamt;
        exe_count++;
        break;

      case SUB:
```

```
        exmem[0].ALUResult = data1 - data2;
        exe_count++;
        break;

      default:
        break;
      }

  }
    // I 타입 명령어
  else if (idex[1].ctr.ALUSrc==1) {
    u_int32_t temp;

    switch (opcode) {
      case ADDI:
      case ADDIU:
        exmem[0].ALUResult = data1 + idex[1].immediate;
        exe_count++;
        break;

      case ANDI:
        exmem[0].ALUResult = data1 & idex[1].immediate;
        exe_count++;
        break;

      case LW:
        exmem[0].temp = data1 + idex[1].immediate;
        exe_count++;
        break;

      case SLTI:
        exmem[0].ALUResult = (data1 < idex[1].immediate) ? 1 : 0;
        exe_count++;
        break;

      case SLTIU:
        exmem[0].ALUResult = (data1 < (u_int32_t)idex[1].immediate) ? 1 : 0;
        exe_count++;
        break;

      case SW:
        exmem[0].ALUResult = data2;
        exmem[0].temp = data1 + idex[1].immediate;
        exe_count++;
        break;

      default:
        break;
    }
  }
  // J 타입 명령어
  else if (idex[1].ctr.Branch==1) {
    BranchPrediction(opcode, data1, data2);
  }
}
```

- R, J, I 타입의 명령어를 opcode와 func 값으로 구분하여 ALUResult를 도출합니다.

- void ID()

```
void ID(){
  printf("\t[ID]\n");
  // Instruction을 유기적으로 구분하는 함수
  parseInstruction();

  // 16비트짜리 offset으로 변환
  idex[0].immediate = signExtend(ifid[1].inst.instruction);

  // Control Hazard를 위한 변수, Adder, Comparator
  u_int32_t opcode = idex[0].inst.opcode;
  u_int32_t offset = (idex[0].immediate << 2) + (idex[0].PC + 4);
  u_int8_t branch = (R[idex[0].inst.rs] == R[idex[0].inst.rt]);
  u_int8_t flag = 0;

  /*
    NOP 명령어가 이미 있기 때문에 에러 발생, 따라서 Load-Use Hazard는 비활성화

  // Load-Use Hazard
  if (idex[0].ctr.MemRead == 1) {
    // 바로 앞 명령어가 load이면서 앞 명령어의 Rd(I type은 Rd가 Rt)가 현재 명령어의 source와 똑같다면 Hazard 발생
    if ((idex[0].inst.rt == ifid[0].inst.rs) || (idex[0].inst.rt == ifid[0].inst.rt)) {
      int a;
      scanf("%d", &a);
      // Control값을 0으로 만들어 버린다. (다음 명령어 nop)
```

```
      memset(&idex[1].ctr, 0, sizeof(idex[1].ctr));

      // PC와 IFID 레지스터의 값을 유보한다.
      ifid[1] = ifid[0];
      PC -= 4;
    }
  }
  */

  // IFID 레지스터의 출력값을 IDEX 레지스터의 입력으로 초기화
  idex[0].inst.instruction = ifid[1].inst.instruction;
  idex[0].PC = ifid[1].PC;

  // OPCODE에 따라 Control값 반환
  processControl(idex[0].inst.opcode);

  // OPCODE가 BEQ 또는 BNE이면서 Branch가 Taken일 때
  if (opcode == BEQ) {
    if (branch) {
      flag = 1;
    }
  } else if (opcode == BNE) {
    if (!branch) {
      flag = 1;
    }
  }

  // PC를 branch offset으로 업데이트 해준다.
  if (flag == 1) {
    printf("\tBEQBNE! : 0x%x(%d)\n", offset, offset);
    ifid[1].PC = offset;
  }
}
```

- Control Hazard와 Load-Use Hazard 구현 알고리즘이 적용된 ID Stage 단계입니다.
- Load-Use Hazard는 해당 Input들에서 오히려 오류를 뿜어내기 때문에 주석처리 하였습니다.
- Control Hazard의 조건이 발동되면 PC를 업데이트 합니다.

- void EX()

```
void EX(){
  printf("\t[EX]\n");

  // rs, rt 레지스터 번호로 각각 레지스터 데이터 추출
  idex[0].data1 = R[idex[0].inst.rs];
    idex[0].data2 = R[idex[0].inst.rt];

  // Data forwarding/bypass
  if((idex[1].inst.rs != 0) && (idex[1].inst.rs == exmem[0].WriteReg) && (exmem[0].ctr.RegWrite)) {
    printf("\t\tFORWARDING! Rs Changed to EXMEM.ALUResult\n");
    idex[1].data1  = exmem[0].ALUResult;
  } else if((idex[1].inst.rs != 0) && (idex[1].inst.rs == memwb[0].WriteReg) && (memwb[0].ctr.RegWrite)){
    if(memwb[0].ctr.MemtoReg==1){
      printf("\t\tFORWARDING! Rs Changed to MEMWB.ReadData\n");
      idex[1].data1 = memwb[0].ReadData;
    } else {
      printf("\t\tFORWARDING! Rs Changed to MEMWB.ALUResult\n");
      idex[1].data1 = memwb[0].ALUResult;
    }
  }
  if((idex[1].inst.rt != 0) && (idex[1].inst.rt == exmem[0].WriteReg) && (exmem[0].ctr.RegWrite)){
    printf("\t\tFORWARDING! Rt Changed to EXMEM.ALUResult\n");
    idex[1].data2 = exmem[0].ALUResult;
  } else if((idex[1].inst.rt!=0) && (idex[1].inst.rt == memwb[0].WriteReg) && (memwb[0].ctr.RegWrite)){
    if(memwb[1].ctr.MemtoReg == 1){
      printf("\t\tFORWARDING! Rt Changed to MEMWB.ReadData\n");
      idex[1].data2 = memwb[0].ReadData;
    } else {
      printf("\t\tFORWARDING! Rt Changed to MEMWB.ALUResult\n");
      idex[1].data2 = memwb[0].ALUResult;
    }
  }

  // RegDst가 true일 경우 R 명령어, ALUSrc가 True일 경우 I 명령어, Branch가 True일 경우 BEQ 또는 BNE로 규정함
  processALU();

  // RegDst에 따라 WriteRegister를 다르게 설정함
  exmem[0].WriteReg = (idex[1].ctr.RegDst == 1) ? idex[1].inst.rd : idex[1].inst.rt;

  printf("\t\t");
  // 각 명령어에 따라 출력값을 다르게 설정
  if(idex[1].inst.format == 'R') {
```

```
    printf("Rs: 0x%x (R[%d](%s)=0x%x), Rt: 0x%x (R[%d](%s)=0x%x), Rd: 0x%x (R[%d](%s)=0x%x), Shamt: 0x%x, Func: (
    printf(", ALUResult : 0x%x\n", exmem[0].ALUResult);
  } else if (idex[1].inst.format == 'I') {
    printf("rs : 0x%x (R[%d](%s)=0x%x), rt : 0x%x (R[%d](%s)=0x%x), immediate : 0x%08x\n", idex[1].inst.rs, idex
  } else {
    printf("(Bubble or Stalled)\n");
  }

  // IDEX 레지스터의 출력을 EXMEM 레지스터의 입력으로 초기화
  exmem[0].inst = idex[1].inst;
    exmem[0].PC = idex[1].PC;
  exmem[0].ctr = idex[1].ctr;
  exmem[0].data1 = idex[1].data1;
  exmem[0].data2 = idex[1].data2;
}
```

- Data Hazard를 구현한 EX Stage 입니다.
- 별도로 함수로 구현하지 않고 바로 레지스터의 값을 변경하도록 구현했습니다.

- 이유를 포함하여 구현한 부분과 구현하지 않은 부분 설명(추가 점수를 받을 수 있는 부분에 대한 구현도 포함)
  - 구현
    - Input1, Input2에 사용된 모든 Mips 명령어 + 자주 쓰이는 명령어
      - 지원해야 하는 MIPS 명령어의 종류가 제시돼 있지 않기에, 제공된 2개의 input.bin 파일에서 사용된 명령어들과 주로 사용되는 명령어들을 구현했습니다.
        - R-Type
          - ADD
          - ADDU
          - AND
          - JR
          - JALR
          - OR
          - SLT
          - SLL
          - SUB
        - I-Type
          - ADDI
          - ADDIU
          - ANDI
          - BEQ
          - BNE
          - LW
          - SLTI
          - SLTIU
          - SW
        - J-Type
          - J
          - JAL
  - 미구현
    - Instruction Memory 크기 제한(1000)
      - 본 프로그램에서는 명령어의 갯수를 매크로를 통해 1000개로 제한했습니다. 동적 할당으로 늘리는 방안은 고정 Input이기에 넉넉하다고 판단하여 고려하지 않았습니다.

- Dynamic branch prediction
    - 처음에 BTB(Branch Target Buffer)를 이용하여 Dynamic Branch Prediction을 구현하려고 했으나, 생각보다 난이도가 높아서 구현하지 않았습니다.

- 환경
    - 테스트를 위한 개발 환경 구축 방법 (+스크린샷)
        - Mac OS
            1. Homebrew 설치
                a. Mac OS는 홈페이지에서 직접 프로그램을 다운로드 받고 설치하는 것 보단, Homebrew라는 프로그램을 통해 설치하는 것이 여러 방면에서 효율적 입니다.

                ```
                /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
                ```

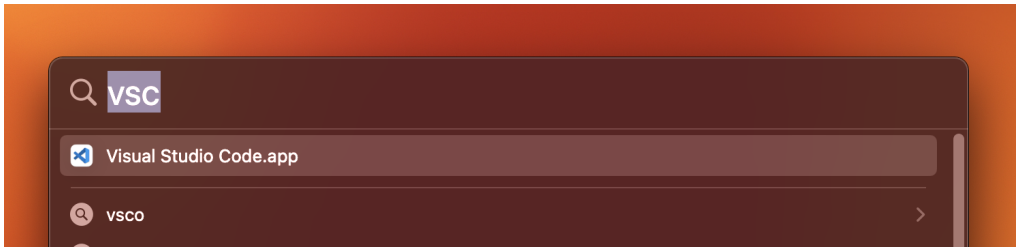
                b. 설치가 완료되면 자신의 환경에 맞게 다음 구문을 수정하여 터미널에 입력합니다.

                ```
                echo 'eval "$(/opt/homebrew/bin/brew shellenv)"' >> /Users/<유저명>/.zprofile
                eval "$(/opt/homebrew/bin/brew shellenv)"
                ```

            2. Visual Studio Code 설치
                a. Homebrew 설치가 완료되면 터미널에 다음과 같이 입력하여 VSC를 설치합니다.

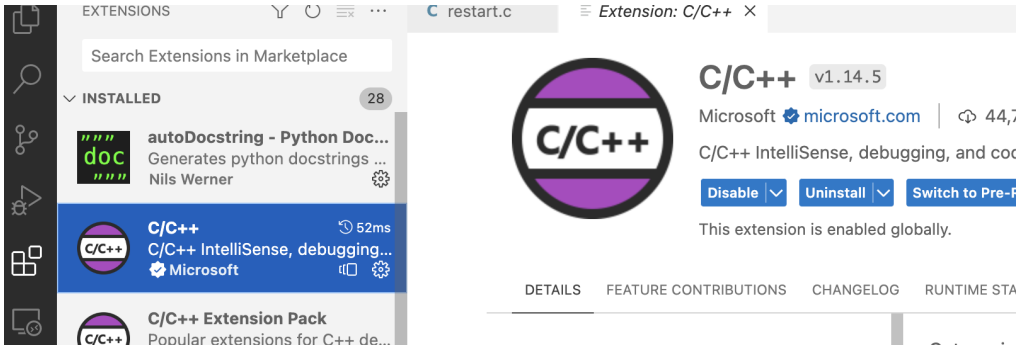
                ```
                brew install visual-studio-code
                ```

                b. Command 키와 Space Bar 를 동시에 눌러서 검색창을 띄운 뒤, VSC라고 입력하면 바로 Visual Studio Code를 찾을 수 있습니다.

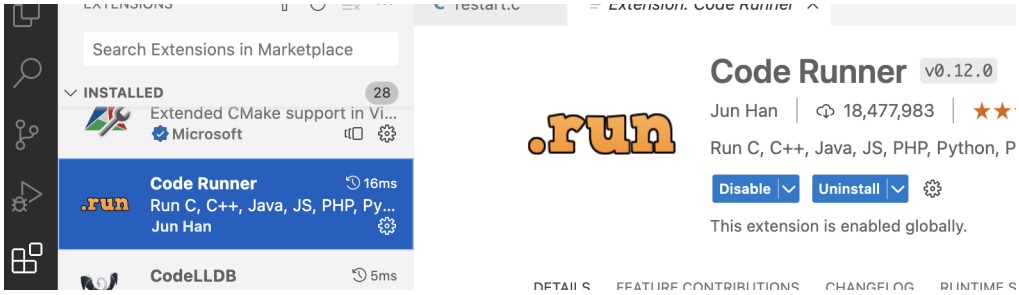
                

                또는 Launchpad에서 찾는 방법도 있음.

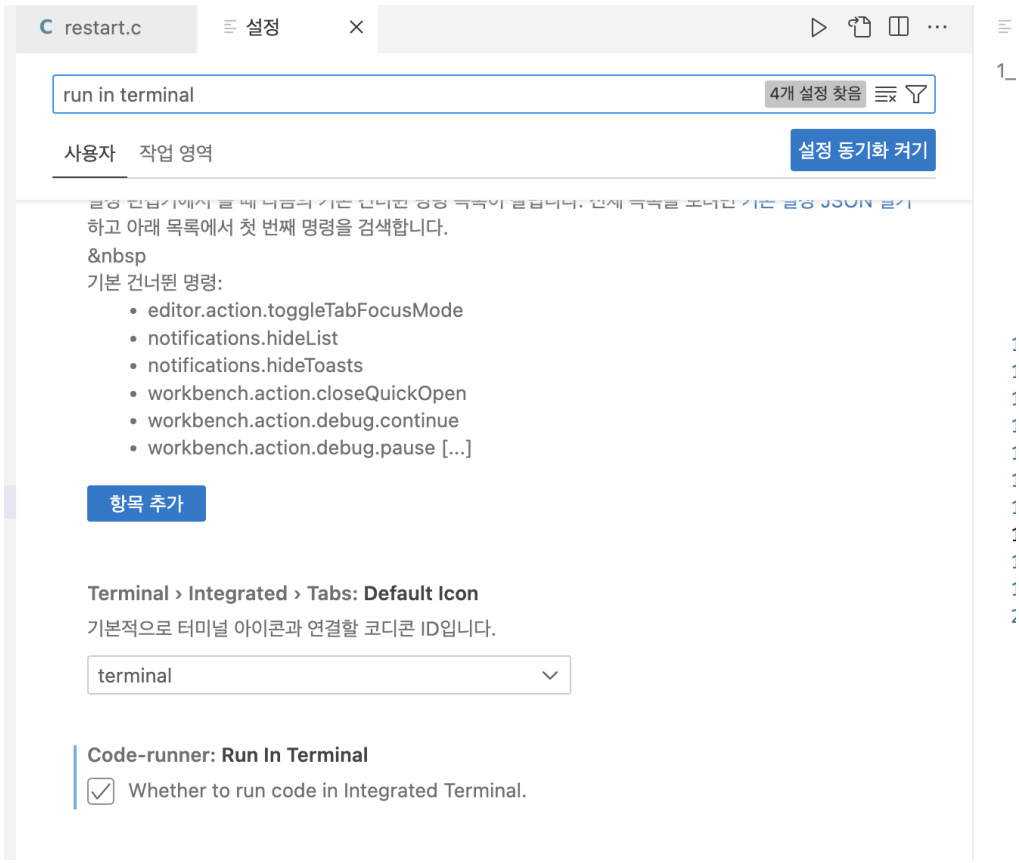
                c. 프로그램 좌측의 사각형 모양 아이콘 (Extension)을 누르고, “C/C++” 라는 확장자를 선택하여 설치합니다.

                d. 또한 Code Runner 라는 확장자도 함께 설치해줍니다.



e. Command 키와 , 키를 함께 눌러 설정창을 띄운 다음, run in terminal 이라고 검색하면 Code-runner: Run In Terminal 이라는 항목이 보이는데, 체크를 해주면 끝납니다.

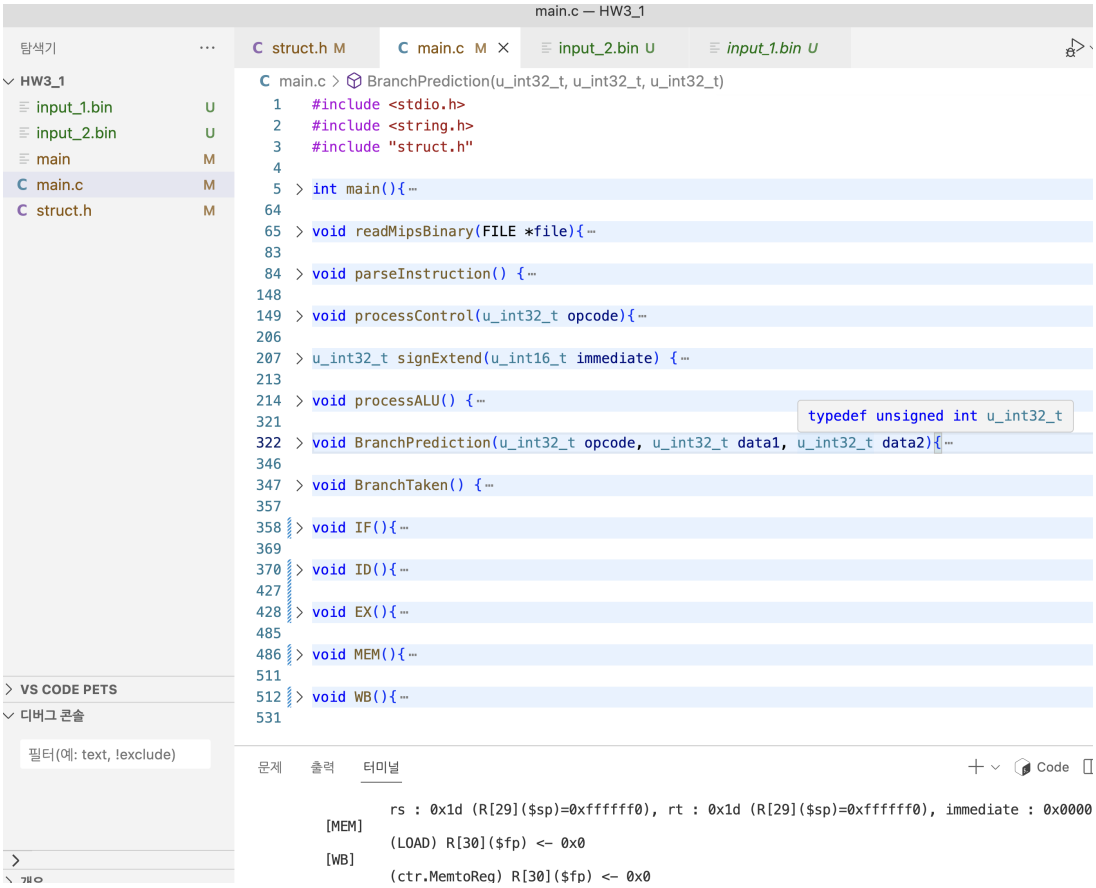


- 프로그램을 컴파일하고 실행하는 방법
  - 하단의 터미널에서 명령어를 차례대로 입력한다

```
gcc main.c

./a.out
```

  - Code Runner를 설치했다면 코드를 저장하고 우측 상단에 있는 재생 버튼의 토글 바를 눌러서 “Run Code”를 클릭하여 실행할 수 있다.



- 동작하는 증거와 설명이 담긴 스크린샷
  - 각 사이클마다 사용된 모든 값이 출력됩니다.
  - 마지막 결과는 교수님이 공개하신 출력 예시와 동일하게 작성했습니다.
  - input 파일 고르기

- 1 또는 아무값을 입력하면 즉시 시뮬레이션이 시작됩니다.
- input_1.bin

main.c — HW3_1

struct.h M | main.c M | input_2.bin U | input_1.bin U

main.c > ...

```
1   #include <stdio.h>
2   #include <string.h>
3   #include "struct.h"
4
5   int main(){…
64
65  void readMipsBinary(FILE *file){…
83
84  void parseInstruction() {…
148
149 void processControl(u_int32_t opcode){…
206
207 u_int32_t signExtend(u_int16_t immediate) {…
213
214 void processALU() {…
321
322 void BranchPrediction(u_int32_t opcode, u_int32_t data1, u_int32_t data2){…
```

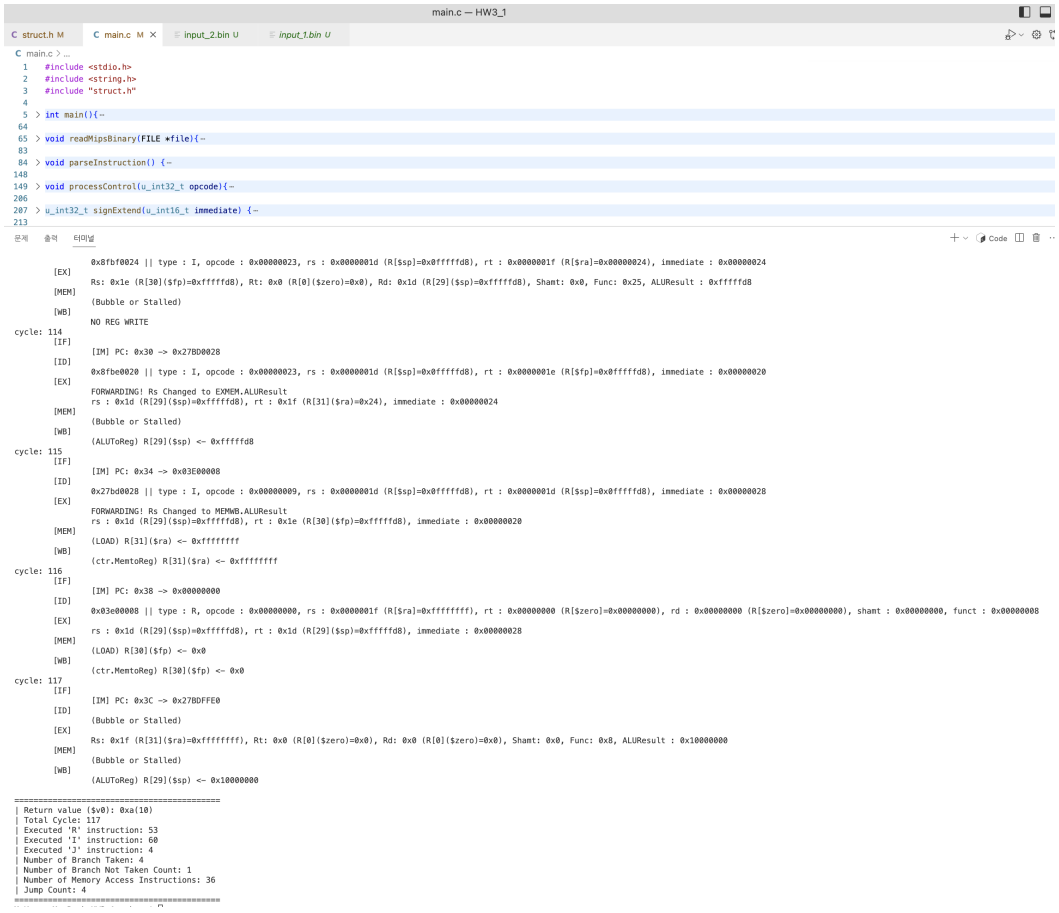
문제 출력 터미널

```
        [WB]
                NO REG WRITE
cycle: 167
        [IF]
                [IM] PC: 0x60 -> 0x27BD0010
        [ID]
                0x8fbe000c || type : I, opcode : 0x00000023, rs : 0x0000001d (R[$sp]=0x0ffffff0), rt : 0x0000001e (R[$fp]=0x0ffffff0), immediate : 0x0000000c
        [EX]
                Rs: 0x1e (R[30]($fp)=0xffffff0), Rt: 0x0 (R[0]($zero)=0x0), Rd: 0x1d (R[29]($sp)=0xffffff0), Shamt: 0x0, Func: 0x25, ALUResult : 0xffffff0
        [MEM]
                (LOAD) R[2]($v0) <- 0x2d
        [WB]
                (ctr.MemtoReg) R[2]($v0) <- 0x2d
cycle: 168
        [IF]
                [IM] PC: 0x64 -> 0x03E00008
        [ID]
                0x27bd0010 || type : I, opcode : 0x00000009, rs : 0x0000001d (R[$sp]=0x0ffffff0), rt : 0x0000001d (R[$sp]=0x0ffffff0), immediate : 0x00000010
        [EX]
                FORWARDING! Rs Changed to EXMEM.ALUResult
                rs : 0x1d (R[29]($sp)=0xffffff0), rt : 0x1e (R[30]($fp)=0xffffff0), immediate : 0x0000000c
        [MEM]
                (Bubble or Stalled)
        [WB]
                (ALUToReg) R[29]($sp) <- 0xffffff0
cycle: 169
        [IF]
                [IM] PC: 0x68 -> 0x00000000
        [ID]
                0x03e00008 || type : R, opcode : 0x00000000, rs : 0x0000001f (R[$ra]=0xffffffff), rt : 0x00000000 (R[$zero]=0x00000000), rd : 0x00000000 (R[$zero]=0x00000000), shamt
        [EX]
                FORWARDING! Rs Changed to MEMWB.ALUResult
                FORWARDING! Rt Changed to MEMWB.ALUResult
                rs : 0x1d (R[29]($sp)=0xffffff0), rt : 0x1d (R[29]($sp)=0xffffff0), immediate : 0x00000010
        [MEM]
                (LOAD) R[30]($fp) <- 0x0
        [WB]
                (ctr.MemtoReg) R[30]($fp) <- 0x0
cycle: 170
        [IF]
                [IM] PC: 0x6C -> 0x00000000
        [ID]
                (Bubble or Stalled)
        [EX]
                Rs: 0x1f (R[31]($ra)=0xffffffff), Rt: 0x0 (R[0]($zero)=0x0), Rd: 0x0 (R[0]($zero)=0x0), Shamt: 0x0, Func: 0x8, ALUResult : 0x10000000
        [MEM]
                (Bubble or Stalled)
        [WB]
                (ALUToReg) R[29]($sp) <- 0x10000000

=========================================
| Return value ($v0): 0x2d(45)
| Total Cycle: 170
| Executed 'R' instruction: 69
| Executed 'I' instruction: 101
| Executed 'J' instruction: 0
| Number of Branch Taken: 11
| Number of Branch Not Taken Count: 1
| Number of Memory Access Instructions: 66
| Jump Count: 0
=========================================
```

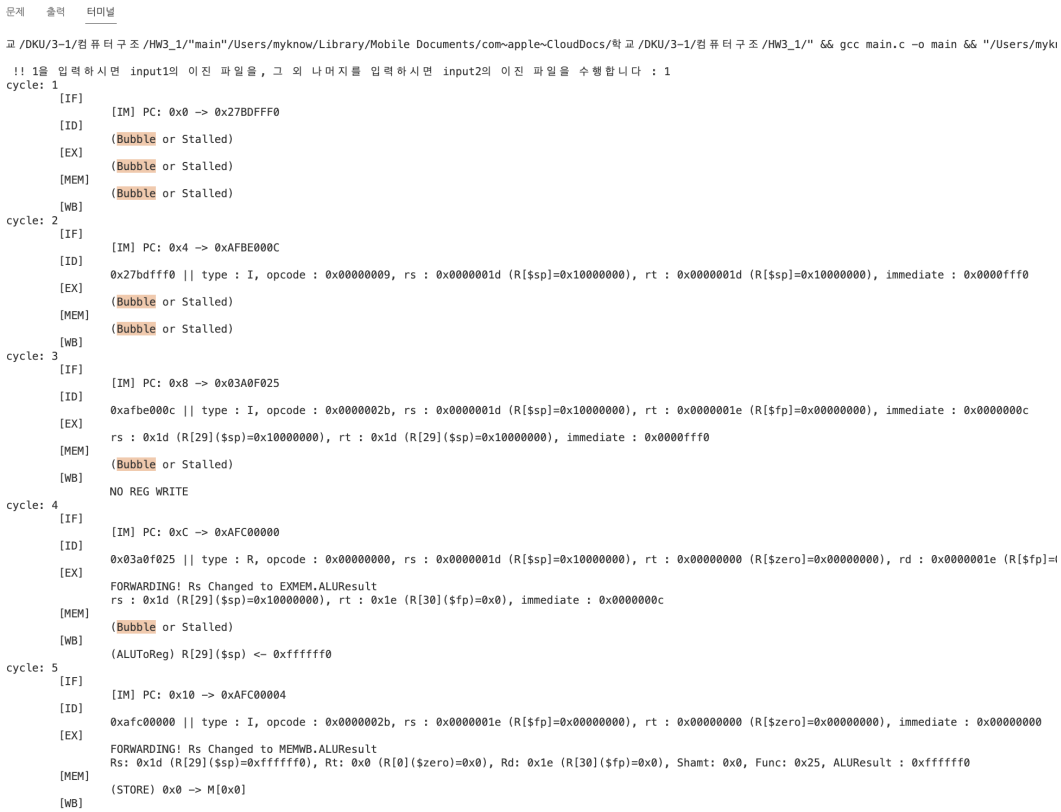
  - 정상적으로 0x2d(45)가 $v0 레지스터로 반환됐습니다.
  - 정상적으로 170회의 사이클을 수행했습니다.

- input2



- 정상적으로 0xa(10)이 $v0 레지스터로 반환됐습니다.
- 정상적으로 117회의 사이클을 수행했습니다.

- 5 Stage and Bubble(or Stall)



- Cycle 1번부터 정상적으로 Bubble이 사라지는 걸 관찰할 수 있습니다.
- EX 단계에서 FOWARDING! 이라는 문구를 통해 Data Fowarding이 정상적으로 이루어지는 걸 확인할 수 있습니다.

- Control Hazard

```
     [MEM]
               (Bubble or Stalled)
     [WB]
               (ALUToReg) R[2]($v0) <- 0x1
le: 68
     [IF]
               [IM] PC: 0x64 -> 0x1000000B
     [ID]
               0x24020001 || type : I, opcode : 0x00000009, rs : 0x00000000 (R[$zero]=0x00000000), rt :
     [EX]
               (Bubble or Stalled)
     [MEM]
               (Bubble or Stalled)
     [WB]
le: 69
     [IF]
               [IM] PC: 0x68 -> 0x00000000
     [ID]
               0x1000000b || type : I, opcode : 0x00000004, rs : 0x00000000 (R[$zero]=0x00000000), rt :
     BEQBNE! : 0x90(144)
     [EX]
               rs : 0x0 (R[0]($zero)=0x0), rt : 0x2 (R[2]($v0)=0x1), immediate : 0x00000001
     [MEM]
               (Bubble or Stalled)
     [WB]
               NO REG WRITE
le: 70
     [IF]
               [IM] PC: 0x6C -> 0x8FC20020
     [ID]
               (Bubble or Stalled)
     [EX]
               rs : 0x0 (R[0]($zero)=0x0), rt : 0x0 (R[0]($zero)=0x0), immediate : 0x0000000b
     [MEM]
               (Bubble or Stalled)
     [WB]
               (ALUToReg) R[2]($v0) <- 0x1
le: 71
     [IF]
               [IM] PC: 0x94 -> 0x03C0E825
     [ID]
               (Bubble or Stalled)
```

    - BEQBNE! : 0x90(144) 문구를 통해서, 정상적으로 Control Hazard를 대처하는 걸 볼 수 있습니다.

- 수업
    - 힘들었던 점
        - 처음에 5 스테이지를 나누는 과정을 거칠 때, 어떻게 하면 각 단계가 값을 한번에 공유하는 게 아니라 각 자 가지고 있을 수 있는지 엄청 고민을 많이 했습니다. 그러다가 차라리 WB부터 IF까지 역으로 계산하는 방법을 생각해봤는데 이 역시 버그가 우후죽순 생겨서 난항을 겪었습니다. 결국 배열로 입력과 출력을 따로 구현하여 처리하는 방법을 채택하게 되었습니다.
    - 과제를 하면서 생각한 점 등
        - 처음엔 논리적으로 매우 간단해보였는데, 직접 구현해보니 지식의 정수였다는 걸 깨닫게 됐습니다...