Project2

-Single-cycle MIPS-

모바일시스템공학과

32212061

변윤성

Left days 5

1. Intro

컴퓨터구조론의 2번째 과제인 single-cycle MIPS emulator를 만들고 구현되는 과정을 코드로 보면서 명령어들이 어떤 식으로 작동하고 순서가 어떻게 되는지 알 수 있는 과제이다. Microarchitecture는 프로세스 내부에서 명령어가 가져와지고(fetch) 해독하고(decode) 실행하고(execute) 저장하는(write-back) 과정을 거치면서 어떻게 처리되는지를 이해한다. 초반에는 개념에 대한 정의와 설명을 하며 나중에 나올 코드에 대해서 설명한다.

2. Concept

2-1. Von Neumann Computer

폰 노이만 구조는 1945년 폰 노이만이 제안한 이래 현대 컴퓨터 설계의 기본이 된 구조로 크게 4가지 구성으로 되어 있다.

1. 제어 유닛
   - Program Counter(PC) 로 Personal Computer가 아닌 다음에 실행할 명령어의 메모리 주소를 가리키는 곳이다.

2. 연산 유닛
   - 산술논리장치(ALU)와 레지스터(Register)를 포함한다.

   - ALU는 실제 연산과 논리 연산 등을 수행해서 레지스터나 메모리에 저장한다.

3. 메모리
   - 명령어와 데이터를 주소 공간에 저장한다.

4. 입/출력 장치
   - 사용자와 컴퓨터 간에 데이터를 주고받는 장치로 마우스, 키보드 등이 해당된다.

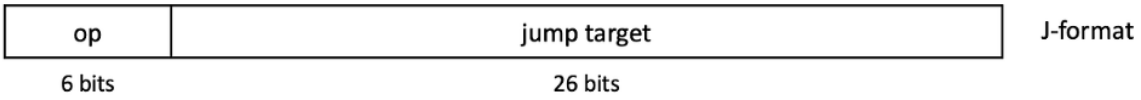
폰 노이만 구조에서는 아래에서 설명할 fetch->decode->execute->write-back 순서로 실행된다.

2-2. MIPS ISA

MIPS(Microprocessor without Interlocked Pipeline Stages)는 32비트 고정 길이 명령어 집합 구조이다. MIPS는 명령어를 크게 3가지로 구분한다.

- R-Type: 명령어는 레지스터 간의 연산을 수행하고 명령어의 상위 6비트(opcode)가 0으로 고정되고 마지막 6비트(funct)가 실제 연산을 결정한다.

- I-Type: 명령어는 immediate값을 사용하거나 메모리 접근 (LW, SW)를 수행한다. 16비트 immediate값은 Sign-extend(확장) 32비트로 확장해서 ALU, 주소 계산에 사용된다.



- J-Type: 명령어는 26비트 주소 필드를 통해서 대용량 코드 공간에서 점프를 지원하고 이 주소는 PC 상위 4비트와 결합해서 최종 32비트 점프 주소를 생성한다.

2-3. Single-Cycle

단일 사이클은 모든 명령어를 한번의 사이클안에 처리하도록 설계하는 구조이다. 즉, 명령어마다 다음 다섯 단계를 순차적으로 수행하고 다음 명령어로 넘어가지만 모든 명령어가 아래의 단계가 필요한 것은 아니다.

1. Fetch: PC가 가리키는 메모리 주소에서 32비트 명령어를 읽고 다음 명령어를 실행하기 위해서 PC에 4를 더한다.
2. Decode: fetch 단계에서 가져온 명령어를 해석해서 opcode, rs, rt, rd, immediate를 분리하고 어떤 레지스터 주소가 들어갈지 결정한다.
3. Execute: ALU를 통해 실제 연산이나 산술 연산을 수행하거나 메모리 주소를 계산해서 다음 단계에 보내준다.
4. MEM: 메모리에 접근하는 단계로 LW/SW 로 메모리에 있는 데이터를 읽거나 데이터를 메모리에 저장하는 단계이다.
5. Write-Back: ALU의 계산 결과나 메모리에서 읽은 데이터를 최종 레지스터에 기록하는 단계이다.

이러한 설계는 제어 로직이 단순해서 구현과 디버깅이 강하다는 장점이 있다. 이러한 설계 속에서 파이프 라인 구조를 도입해서 여러 개의 단계를 병렬로 작동하게 해서 CPU 처리 속도를 높이는 방식이다.

2-4. Control Signal Table

아래의 표로 구현했다.

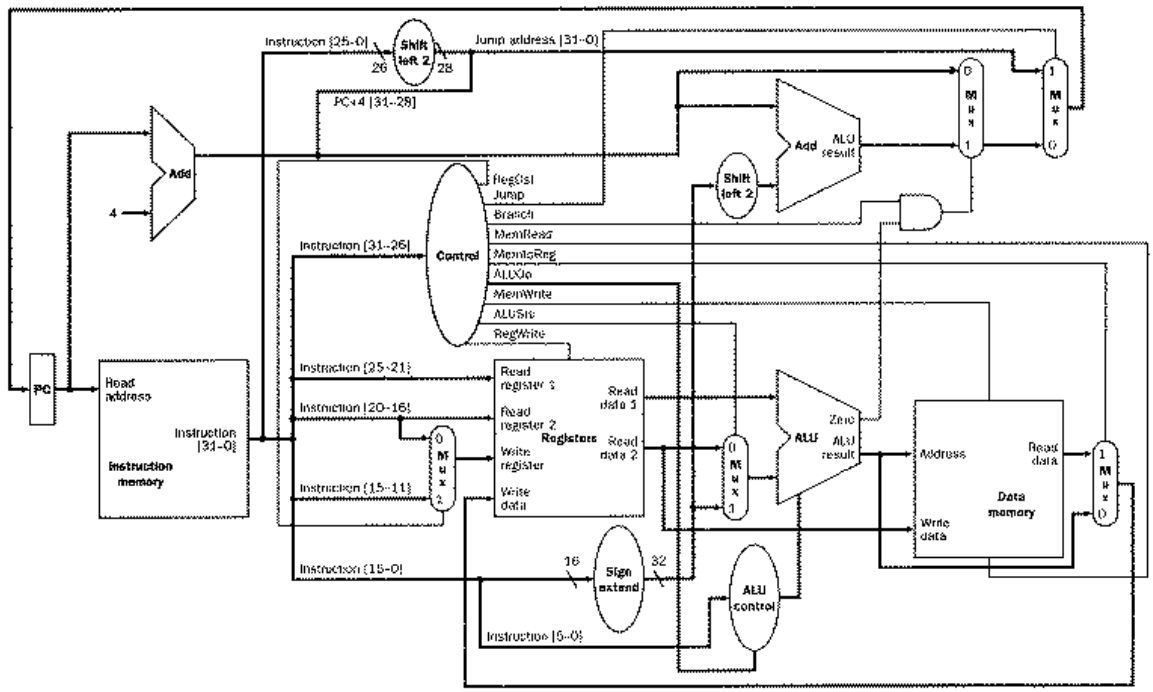
MIPS Control Signals

신호 이름	기능	Datapath 제어 대상
RegDst	레지스터 파일에 쓰기 대상으로 rt(I-타입) 또는 rd(R-타입), 또는 링크용 r31을 선택	쓰기 레지스터 MUX

ALUSrc	ALU의 두 번째 입력으로 레지스터 값(0) 또는 부호 확장된 즉시값(1)을 선택	ALU 입력 MUX
MemtoReg	레지스터에 쓰기할 데이터로 ALU 결과(0) 또는 메모리에서 읽어온 값(1)을 선택	쓰기 데이터 MUX
RegWrite	레지스터 파일 쓰기 허용 여부 (1이면 쓰기, 0이면 쓰기 억제)	레지스터 파일 쓰기 enable
MemRead	데이터 메모리 읽기 수행 여부 (1이면 메모리 → 버스)	데이터 메모리 읽기 enable
MemWrite	데이터 메모리 쓰기 수행 여부 (1이면 레지스터 → 메모리)	데이터 메모리 쓰기 enable
Branch	분기 명령어(beq/bne)일 때, 조건이 참이면 분기	분기 판단 논리 (AND/NOT gate)
Jump	무조건 점프(j, jal) 시 분기	PC 업데이트 MUX
JumpReg	레지스터 점프(jr, jalr) 시 rs 레지스터 값을 다음 PC로 사용	PC 업데이트 MUX
JumpLink	jal, jalr 시 리턴 주소 (PC+4 또는 PC+8)를 r31에 저장	쓰기 레지스터 (r31) 및 쓰기 데이터 MUX
ALUOp	ALU 제어 신호(2비트): 00=덧셈, 01=뺄셈, 10=R-타입 funct 해석 등	ALU Control 유닛

Single-Cycle MIPS에서 Control Signal은 어떤 동작을 할지 결정하는 역할을 한다. CPU 내부의 제어장치가 opcode, funct 필드를 해석해서 신호를 결정한다

2.5 DataPath



위의 그림에 따라 ADD, LW, SW 등 명령어들이 흐름의 다 다르며 명령어마다 필요한 구조도 다르다. 대표적인 명령어들을 소개했다.

1. ADD 명령어는 두 레지스터 값을 더해서 결과를 다른 레지스터에 저장한다. 요약하면 Register File에서 두 값(rs,rt)를 읽어 ALU에서 연산을 수행하고 그 결과를 다시 Register File에 저장하는 과정을 거친다.

2. LW(Load Word) 명령어는 메모리로부터 데이터를 읽어서 레지스터에 저장한다. Register File로부터 주소를 읽어오고 ALU를 통해 실제 메모리 주소를 계산한 뒤 메모리로부터 데이터를 읽어서 다시 레지스터에 저장하는 흐름이다.

3. SW(Store Word) 명령어는 레지스터에 있는 데이터를 메모리에 저장하는 역할을 한다. SW 명령어는 결과를 다시 레지스터 파일로 저장하지 않고 단순히 메모리에 데이터를 쓰는 거에 집중한다. Register File, ALU, Data Memory를 순서대로 거치면서 동작한다.

4. BEQ(Branch on Equal) 명령어는 두 레지스터 값이 같을 경우 분기해서 새로운 주소로 이동하는 역할이다. 비교와 분기 결정을 위해 Register File과 ALU를 많이 활용하면 경우에 따라 PC의 값을 변경시킨다.

5. J(Jump) 명령어는 주어진 주소로 점프 시키는 명령어이다. J 명령어는 ALU 연산 없이 단순히 Jump 주소를 계산해서 PC를 덮어쓰는 방식으로 동작한다.

3. Implementation

3.1 초기화 initialize()

```
24    void initialize() {
25        memset(Reg, 0, sizeof(Reg));
26        memset(memory, 0, sizeof(memory));
27        Reg[29] = MEM_SIZE;      // 메모리의 맨 위 초기화
28        Reg[31] = 0xFFFFFFFF;    // 잘못된 복귀 막기
29        pc = 0;
30    }
```

실행하기 전에 모든 아키텍쳐를 깨끗한 초기 상태로 만든다. 스택 포인터 $sp(레지스터 29)를 메모리의 최상단 주소 (MEM_SIZE)로 설정하고 리턴 주소 레지스터 $ra(레지스터 31)에만 0xFFFFFFFF를 할당한다. PC가 이 값을 만나면 프로그램 종료를 하고 PC를 0으로 설정함으로써 fetch를 시작하도록 준비한다.

3.1.1 Endian System

```
31    uint32_t to_big_endian(uint32_t v) {
32        return ((v >> 24) & 0xFF) | ((v >> 8) & 0xFF00) | ((v << 8) & 0xFF0000) | ((v << 24) & 0xFF000000);
33    }
```

PC는 Little Endian 시스템(메모리에 가장 작은 바이트(최하위 바이트)를 먼저 저장)인데 MIPS는 보통 Big Endian(메모리에 가장 큰 바이트(최상위 바이트)를 먼저 저장)이다. 그래서 fread()로 읽으면 바이트 순서가 어긋나기 때문에 to_big_endian()으로 바이트를 바로잡고 읽어야 된다. 그렇지 않으면 명령어가 뒤틀리기 때문에 CPU가 이상한 명령어를 실행하게 된다.

3.2 명령어 로드 load_program()

```
36    void load_program(const char* fn) {
37        FILE* fp = fopen(fn, "rb"); if (!fp) { fprintf(stderr, "Cannot open %s₩n", fn); exit(1); }
38        uint32_t addr = 0, tmp;
39        while (fread(&tmp, sizeof(tmp), 1, fp) == 1) {
40            tmp = to_big_endian(tmp);
41            if (addr + 4 > MEM_SIZE) break;
42            memcpy(&memory[addr], &tmp, 4);
43            addr += 4;
44        }
45        fclose(fp);
46    }
```

load_program() 함수는 바이너리 파일을 4byte씩 읽어서 to_big_endian()을 통해 변환한 뒤 내부 메모리 배열(memory[])에 순차적으로 복사한다. 메모리 상의 주소 증가는 항상 4byte 단위이므로 다음 instruction은 addr+=4에 저장된다. 이로써 "instruction memory"가 완성된다.

3.2.1 instruction memory

Instruction memory는 프로그램이 실행할 명령어들을 저장해놓은 메모리 영역을 의미한다. CPU가 수행할 명령어들은 모두 이 instruction memory에 순서대로 저장되어 있으며 프로세서는 프로그램 카운터(PC) 값에 따라 instruction memory를 참조해 해당 주소에 저장된 명령어를 가져온다.

3.3 패치 단계 fetch()

```
48  uint32_t fetch() {
49      if (pc + 4 > MEM_SIZE) { fprintf(stderr, "Fetch invalid PC 0x%X₩n", pc); exit(1); }
50      return *(uint32_t*)&memory[pc];
51  }
```

Fetch()는 현재 PC가 가리키는 메모리 위치에서 32비트 명령어를 읽어 반환한다. 내부적으로 memory[pc] 주소에서 4byte를 그대로 로드하므로 실제 instruction memory를 통한 읽기(read)와 동일한 동작을 수행한다.

3.4 디코드 decode()

```
53  void decode(uint32_t inst, int* op, int* rs, int* rt, int* rd, int* sh, int* fn, int* im, int* addr) {
54      *op = (inst >> 26) & 0x3F; *rs = (inst >> 21) & 0x1F; *rt = (inst >> 16) & 0x1F;
55      *rd = (inst >> 11) & 0x1F; *sh = (inst >> 6) & 0x1F; *fn = inst & 0x3F;
56      *im = inst & 0xFFFF; *addr = inst & 0x3FFFFFF;
57  }
58
```

decode() 함수는 32비트 워드를 MIPS 명령어 (opcode, rs, rt, rd, shamt, funct, immediate)에 맞춰 필드를 분리한다. 이러한 과정을 통해 control unit이 어떤 제어 신호(control signal)를 설정할지 어떤 레지스터를 읽어올지 결정할 수 있다. 이 부분은 MIPS sing-cycle의 ID/RF(Instruction Decode & Register Fetch)를 소프트웨어 적으로 구현한 부분이다.

3.4.1 ID/RF 단계(Instruction Decode & Register Fetch)

fetch 단계에서 읽어온 32비트 명령어를 해석하고 명령어 실행에 필요한 레지스터 값을 읽어오는 과정을 말한다. ID/RF 단계는 명령어를 해석하고 필요한 소스 데이터를 준비하며, 이후 명령어 실행을 위해 제어 신호를 설정하는 핵심적이고 필수적인 과정이라고 할 수 있다.

3.5 실행 단계 execute()

```
void execute(int op, int fn, int rs, int rt, int rd, int sh, int im, int addrField) {
    uint32_t se = (im & 0x8000) ? (im | 0xFFFF0000) : im;
    uint32_t brt = pc + 4 + (se << 2);
    uint32_t jmp = (pc & 0xF0000000) | (addrField << 2);
    uint32_t res, md;
    switch (op) {
    case 0x00: // R-type
        cnt_r_type++;
        switch (fn) {
        case 0x20: res = Reg[rs] + Reg[rt]; Reg[rd] = res; printf("[Execute] ADD R[%d]=0x%08X\n", rd, res); break;
        case 0x21: res = Reg[rs] + Reg[rt]; Reg[rd] = res; printf("[Execute] ADDU R[%d]=0x%08X\n", rd, res); break;
        case 0x22: res = Reg[rs] - Reg[rt]; Reg[rd] = res; printf("[Execute] SUB R[%d]=0x%08X\n", rd, res); break;
        case 0x23: res = Reg[rs] - Reg[rt]; Reg[rd] = res; printf("[Execute] SUBU R[%d]=0x%08X\n", rd, res); break;
        case 0x24: res = Reg[rs] & Reg[rt]; Reg[rd] = res; printf("[Execute] AND R[%d]=0x%08X\n", rd, res); break;
        case 0x25: res = Reg[rs] | Reg[rt]; Reg[rd] = res; printf("[Execute] OR R[%d]=0x%08X\n", rd, res); break;
        case 0x2A: res = ((int32_t)Reg[rs] < (int32_t)Reg[rt]) ? 1 : 0; Reg[rd] = res; printf("[Execute] SLT R[%d]=%d\n", rd, res); break;
        case 0x00: res = Reg[rt] << sh; Reg[rd] = res; printf("[Execute] SLL R[%d]=0x%08X\n", rd, res); break;
        case 0x02: res = Reg[rt] >> sh; Reg[rd] = res; printf("[Execute] SRL R[%d]=0x%08X\n", rd, res); break;
        case 0x08: pc = Reg[rs]; printf("[Execute] JR to 0x%08X\n", pc); return;
        default: printf("[Execute] Unknown R-funct 0x%02X\n", fn);
        }
        pc += 4;
        break;
```

```
    case 0x08: // ADDI
        cnt_i_type++; Reg[rt] = Reg[rs] + se; printf("[Execute] ADDI R[%d]=0x%08X\n", rt, Reg[rt]); pc += 4; break;
    case 0x09: // ADDIU
        cnt_i_type++; Reg[rt] = Reg[rs] + se; printf("[Execute] ADDIU R[%d]=0x%08X\n", rt, Reg[rt]); pc += 4; break;
    case 0x0A: // SLTI
        cnt_i_type++; Reg[rt] = ((int32_t)Reg[rs] < (int32_t)se) ? 1 : 0; printf("[Execute] SLTI R[%d]=%d\n", rt, Reg[rt]); pc += 4; break;
    case 0x0C: // ANDI
        cnt_i_type++; Reg[rt] = Reg[rs] & (im & 0xFFFF); printf("[Execute] ANDI R[%d]=0x%08X\n", rt, Reg[rt]); pc += 4; break;
    case 0x0D: // ORI
        cnt_i_type++; Reg[rt] = Reg[rs] | (im & 0xFFFF); printf("[Execute] ORI R[%d]=0x%08X\n", rt, Reg[rt]); pc += 4; break;
    case 0x0F: // LUI
        cnt_i_type++; Reg[rt] = im << 16; printf("[Execute] LUI R[%d]=0x%08X\n", rt, Reg[rt]); pc += 4; break;
    case 0x23: // LW
        cnt_mem_read++; md = *(uint32_t*)&memory[Reg[rs] + se]; Reg[rt] = md;
        printf("[Execute] LW R[%d]=M[R[%d]+0x%X]=0x%08X\n", rt, rs, se, md); pc += 4; break;
    case 0x2B: // SW
        cnt_mem_write++; if (Reg[rs] + se + 4 > MEM_SIZE) { fprintf(stderr, "Store OOB 0x%X\n", Reg[rs] + se); exit(1); }
        *(uint32_t*)&memory[Reg[rs] + se] = Reg[rt];
        printf("[Execute] SW M[R[%d]+0x%X]=0x%08X\n", rs, se, Reg[rt]); pc += 4; break;
    case 0x04: // BEQ
        cnt_branches++; printf("[Execute] %s\n", (Reg[rs] == Reg[rt]) ? "Branch taken" : "Branch not taken");
        if (Reg[rs] == Reg[rt]) { cnt_taken_branches++; pc = brt; }
        else pc += 4; break;
    case 0x05: // BNE
        cnt_branches++; printf("[Execute] %s\n", (Reg[rs] != Reg[rt]) ? "Branch taken" : "Branch not taken");
        if (Reg[rs] != Reg[rt]) { cnt_taken_branches++; pc = brt; }
        else pc += 4; break;
    case 0x02: // J
        cnt_j_type++; printf("[Execute] J to 0x%08X\n", jmp); pc = jmp; break;
    case 0x03: // JAL
        cnt_j_type++; Reg[31] = pc + 8; printf("[Execute] JAL, R[31]=0x%08X\n", Reg[31]); pc = jmp; break;
    default:
        printf("[Execute] Unknown opcode 0x%02X\n", op); pc += 4;
    }
}
```

execute()는 opcode와 funct 필드에 따라서 명령어를 구분하고 실제 데이터 연산이나 논리 연산을 수행한다.

R-타입 연산: opcode 0x00에 들어온 뒤 funct 값을 보고 ADD, SUB, AND, OR, SLT, SLL, SRL 같은 레지스터 간 연산을 수행하고, 결과를 Reg[rd]에 저장한다.

I-타입 연산: ADDI, ADDIU, SLTI, ANDI, ORI, LUI 등의 immediate값 연산은 분기 대상(branch target) 및 immediate값의 특성에 따라 sign-extend 또

는 zero-extend 과정을 거쳐 ALU에 전달한다.

메모리 접근: LW는 Reg[rs] + signExtImm 주소의 메모리에서 32비트 워드를 불러와 Reg[rt]에 쓰고, SW는 Reg[rs] + signExtImm 주소에 Reg[rt] 값을 저장한다.

분기 및 점프: BEQ/BNE는 pc+4 + (signExtImm<<2)를 브랜치 타겟으로 사용하며, 조건이 만족되면 pc를 그 주소로 갱신한다. J/JAL은 상위 PC 비트를 결합해 26비트 즉시 점프 주소를 생성하고, JAL의 경우 Reg[31]에 pc+8(delay slot을 고려한 리턴 주소)를 저장한다.

모든 연산 직후에는 항상 PC를 다음 명령어로 이동시키거나, 분기/점프 된 새 PC로 설정하도록 설정했다.

3.5.1 immediate 값

메모리에 접근하지 않고 명령어를 읽는 순간 바로 순간 바로 쓸 수 있는 값이라서 immdiate(즉시)값이라고 한다. immediate값은 16비트인데 ALU는 32비트 연산을 하니까 sign-extend, zero-extend를 해야 한다. 부호를 유지해야 할 때는 sign-extend 또는 앞에 0을 채워서 확장하는 경우는 zero-extend를 써서 확장시킨다.

3.6 메인 함수 main()

```
int main(int argc, char** argv) {
    const char* f = (argc > 1 ? argv[1] : "simple.bin"); //파일 이름 변경 부분
    initialize(); load_program(f);
    while (pc != 0xFFFFFFFF) {
        printf("----------------------------------------₩n");
        printf("Cycle: %d₩n", cycle + 1);
        uint32_t inst = fetch(); printf("[Fetch] 0x%08X:0x%08X₩n", pc, inst);
        int op, rs, rt, rd, sh, fn, im, ad;
        decode(inst, &op, &rs, &rt, &rd, &sh, &fn, &im, &ad);
        printf("[Decode] opcode(0x%02X) rs:%d rt:%d rd:%d sh:%d fn(0x%02X) imm:%d adr:0x%X₩n", op, rs, rt, rd, sh, fn, im, ad);
        execute(op, fn, rs, rt, rd, sh, im, ad);
        cycle++;
    }
    printf("****************result****************₩n");
    printf("(1) Final return value: %d₩n", Reg[2]);
    printf("(2) Total instructions: %d₩n", cycle);
    printf("(3) R-type: %d₩n", cnt_r_type);
    printf("(4) I-type: %d₩n", cnt_i_type);
    printf("(5) J-type: %d₩n", cnt_j_type);
    printf("(6) Mem access: %d₩n", cnt_mem_read + cnt_mem_write);
    printf("(7) Taken branches: %d₩n", cnt_taken_branches);
    printf("****************************************₩n");
    return 0;
}
```

main() 함수는 프로그램을 메모리에 올리고 한 사이클마다 명령을 처리한 뒤 마지막에 통계를 출력하는 역할이다.

전체 코드의 설명도 같이 하자면 argc, argv를 통해 사용자로부터 실행할 파일을 받아온다. 만약 명령 인수가 주어지지 않는다면 simple.bin을 사용하도록 처리했다. 그 다음 initialize() 호출을 통해 모든 레지스터와 메모리를 0으로 초기화하고 스택 포인터($sp)와 리턴 주소($ra) 프로그램 카운터(pc)를 ISA 명세에 맞는 초기값으로 설정합니다. 곧바로 load_program(f)가 실행되어 지정된 파일을 읽어드린다.

이제 while (pc != 0xFFFFFFFF) 루프에 진입하는데 여기서 0xFFFFFFFF는 리턴 주소($ra)로 쓰여 프로그램 종료 조건이 된다. 매 반복마다 우선 구분선(-)을 출력하고 사이클 번호(cycle+1)를 찍어 현재 사이클이 몇 번 째인지 표시한다. 그 후 fetch()를 호출하여 메모리에서 현재 pc가 가리키는 32비트 명령어 워드를 읽어오고 이를 16진수 형태로 출력한다.

그 다음의 명령어를 decode()에 넘기면 내부에서 상위 6비트의 opcode와 rs, rt, rd, shamt, funct, immediate, 주소 필드 등 MIPS ISA가 정의한 대로 각 부분을 비트 연산으로 분리한다. 분리된 필드들은 다시 한번 모두 출력되어 "ID/RF 단계"에서 올바른 값이 나오는지 눈으로 확인할 수 있게 한다.

execute() 함수가 single-cycle microarchitecture의 EX, MEM, WB 단계를 몽땅 처리한다. R-타입 산술·논리 연산, I-타입 immediate값 연산, LW/SW 메모리 접근, BEQ/BNE 분기, J/JAL/JR 점프 같은 모든 명령을 switch문과 내부 비트 연산으로

분기 처리해 주며 수행 결과를 레지스터나 메모리에 기록하고 pc를 적절히 갱신한다.

마지막으로 사이클 카운터를 1 증가시키고 루프를 반복하며 pc가 끝나는 주소(0xFFFFFFFF)에 도달하면 빠져나온다. 루프를 벗어나면 총 수행된 사이클(=명령어) 수, R/I/J 타입별 집계, 메모리 접근 횟수, 분기 성공 횟수 등 미리 선언된 전역 통계 변수를 모조리 출력한 뒤 종료된다.

이 전 과정은 실제 하드웨어의 Fetch→Decode→Execute→Memory→Write-Back을 구현한 것이다.

4. Result

4.1 simple.bin

```
----------------------------------------
Cycle: 6
[Fetch] 0x00000014:0x8FBE0004
[Decode] opcode(0x23) rs:29 rt:30 rd:0 sh:0 fn(0x04) imm:4 adr:0x3BE0004
[Execute] LW R[30]=M[R[29]+0x4]=0x00000000
----------------------------------------
Cycle: 7
[Fetch] 0x00000018:0x27BD0008
[Decode] opcode(0x09) rs:29 rt:29 rd:0 sh:0 fn(0x08) imm:8 adr:0x3BD0008
[Execute] ADDIU R[29]=0x00100000
----------------------------------------
Cycle: 8
[Fetch] 0x0000001C:0x03E00008
[Decode] opcode(0x00) rs:31 rt:0 rd:0 sh:0 fn(0x08) imm:8 adr:0x3E00008
[Execute] JR to 0xFFFFFFFF
**********result**********
(1) Final return value: 0
(2) Total instructions: 8
(3) R-type: 4
(4) I-type: 2
(5) J-type: 0
(6) Mem access: 2
(7) Taken branches: 0
****************************************
```

4.2 simple2.bin

```
Cycle: 8
[Fetch] 0x0000001C:0x8FBE0014
[Decode] opcode(0x23) rs:29 rt:30 rd:0 sh:0 fn(0x14) imm:20 adr:0x3BE0014
[Execute] LW R[30]=M[R[29]+0x14]=0x00000000
----------------------------------------
Cycle: 9
[Fetch] 0x00000020:0x27BD0018
[Decode] opcode(0x09) rs:29 rt:29 rd:0 sh:0 fn(0x18) imm:24 adr:0x3BD0018
[Execute] ADDIU R[29]=0x00100000
----------------------------------------
Cycle: 10
[Fetch] 0x00000024:0x03E00008
[Decode] opcode(0x00) rs:31 rt:0 rd:0 sh:0 fn(0x08) imm:8 adr:0x3E00008
[Execute] JR to 0xFFFFFFFF
*****************result*****************
(1) Final return value: 100
(2) Total instructions: 10
(3) R-type: 3
(4) I-type: 3
(5) J-type: 0
(6) Mem access: 4
(7) Taken branches: 0
****************************************
```

4.3 simple3.bin

```
Cycle: 1328
[Fetch] 0x00000060:0x8FBE0014
[Decode] opcode(0x23) rs:29 rt:30 rd:0 sh:0 fn(0x14) imm:20 adr:0x3BE0014
[Execute] LW R[30]=M[R[29]+0x14]=0x00000000
----------------------------------------
Cycle: 1329
[Fetch] 0x00000064:0x27BD0018
[Decode] opcode(0x09) rs:29 rt:29 rd:0 sh:0 fn(0x18) imm:24 adr:0x3BD0018
[Execute] ADDIU R[29]=0x00100000
----------------------------------------
Cycle: 1330
[Fetch] 0x00000068:0x03E00008
[Decode] opcode(0x00) rs:31 rt:0 rd:0 sh:0 fn(0x08) imm:8 adr:0x3E00008
[Execute] JR to 0xFFFFFFFF
*****************result*****************
(1) Final return value: 5050
(2) Total instructions: 1330
(3) R-type: 409
(4) I-type: 205
(5) J-type: 1
(6) Mem access: 613
(7) Taken branches: 101
****************************************
```

4.4 simple4.bin

```
Cycle: 241
[Fetch] 0x00000024:0x8FBE0018
[Decode] opcode(0x23) rs:29 rt:30 rd:0 sh:0 fn(0x18) imm:24 adr:0x3BE0018
[Execute] LW R[30]=M[R[29]+0x18]=0x00000000
----------------------------------------
Cycle: 242
[Fetch] 0x00000028:0x27BD0020
[Decode] opcode(0x09) rs:29 rt:29 rd:0 sh:0 fn(0x20) imm:32 adr:0x3BD0020
[Execute] ADDIU R[29]=0x00100000
----------------------------------------
Cycle: 243
[Fetch] 0x0000002C:0x03E00008
[Decode] opcode(0x00) rs:31 rt:0 rd:0 sh:0 fn(0x08) imm:8 adr:0x3E00008
[Execute] JR to 0xFFFFFFFF
*****************result*****************
(1) Final return value: 55
(2) Total instructions: 243
(3) R-type: 79
(4) I-type: 43
(5) J-type: 11
(6) Mem access: 100
(7) Taken branches: 9
****************************************
```

4.5 gcd.bin

```
Cycle: 1059
[Fetch] 0x0000003C:0x8FBE0028
[Decode] opcode(0x23) rs:29 rt:30 rd:0 sh:0 fn(0x28) imm:40 adr:0x3BE0028
[Execute] LW R[30]=M[R[29]+0x28]=0x00000000
----------------------------------------
Cycle: 1060
[Fetch] 0x00000040:0x27BD0030
[Decode] opcode(0x09) rs:29 rt:29 rd:0 sh:0 fn(0x30) imm:48 adr:0x3BD0030
[Execute] ADDIU R[29]=0x00100000
----------------------------------------
Cycle: 1061
[Fetch] 0x00000044:0x03E00008
[Decode] opcode(0x00) rs:31 rt:0 rd:0 sh:0 fn(0x08) imm:8 adr:0x3E00008
[Execute] JR to 0xFFFFFFFF
*****************result*****************
(1) Final return value: 1
(2) Total instructions: 1061
(3) R-type: 359
(4) I-type: 78
(5) J-type: 65
(6) Mem access: 486
(7) Taken branches: 45
****************************************
```

4.6 input4.bin

이 코드로 그대로 실행시켰더니 무한루프를 돌아서 이유를 찾아봤더니 JALR이 없어서 실행이 안되는 것이다. 그래서 아래의 JALR을 추가한 코드로 실행을 시켰다.

4.6.1 +JALR 추가

```
90          case 0x09: // JALR 추가
91              Reg[rd] = pc + 8;
92              pc = Reg[rs];
93              printf("[Execute] JALR: Save 0x%08X to R[%d], Jump to 0x%08X₩n", Reg[rd], rd, pc);
94              return;
```

JALR은 JAL과 JR을 합친 것으로 특정 레지스터에 저장된 주소로 점프하고 현재 PC+8을 다른 레지스터($rd)에 저장하는 명령어이다. 결과적으로 JALR 명령어는 Register File을 통해 JUMP 주소를 읽고 PC를 덮어쓰며 동시에 현재 위치에 대한 복귀 주소를 다른 레지스터에 저장하는 복합적인 동작을 단일 사이클 안에서 완성한다.

다른 명령어들은 실행이 끝나고 PC+=4를 해서 다음 명령어들로 넘어가지만 JALR은 명령어가 PC를 덮어써서 점프하니까 PC+=4를 안 한다.

```
----------------------------------------
Cycle: 23372703
[Fetch] 0x00018EE4:0x27DD1C78
[Decode] opcode(0x09) rs:30 rt:29 rd:3 sh:17 fn(0x38) imm:7288 adr:0x3DD1C78
[Execute] ADDIU R[29]=0x000F8010
----------------------------------------
Cycle: 23372704
[Fetch] 0x00018EE8:0x8FBE7FEC
[Decode] opcode(0x23) rs:29 rt:30 rd:15 sh:31 fn(0x2C) imm:32748 adr:0x3BE7FEC
[Execute] LW R[30]=M[R[29]+0x7FEC]=0x00000000
----------------------------------------
Cycle: 23372705
[Fetch] 0x00018EEC:0x27BD7FF0
[Decode] opcode(0x09) rs:29 rt:29 rd:15 sh:31 fn(0x30) imm:32752 adr:0x3BD7FF0
[Execute] ADDIU R[29]=0x00100000
----------------------------------------
Cycle: 23372706
[Fetch] 0x00018EF0:0x03E00008
[Decode] opcode(0x00) rs:31 rt:0 rd:0 sh:0 fn(0x08) imm:8 adr:0x3E00008
[Execute] JR to 0xFFFFFFFF
*****************result*****************
(1) Final return value: 85
(2) Total instructions: 23372706
(3) R-type: 10152862
(4) I-type: 4073436
(5) J-type: 103
(6) Mem access: 7116606
(7) Taken branches: 2028830
****************************************
```

4.7 fib.bin

```
Cycle: 2677
[Fetch] 0x00000030:0x8FBE0020
[Decode] opcode(0x23) rs:29 rt:30 rd:0 sh:0 fn(0x20) imm:32 adr:0x3BE0020
[Execute] LW R[30]=M[R[29]+0x20]=0x00000000
----------------------------------------
Cycle: 2678
[Fetch] 0x00000034:0x27BD0028
[Decode] opcode(0x09) rs:29 rt:29 rd:0 sh:0 fn(0x28) imm:40 adr:0x3BD0028
[Execute] ADDIU R[29]=0x00100000
----------------------------------------
Cycle: 2679
[Fetch] 0x00000038:0x03E00008
[Decode] opcode(0x00) rs:31 rt:0 rd:0 sh:0 fn(0x08) imm:8 adr:0x3E00008
[Execute] JR to 0xFFFFFFFF
*****************result*****************
(1) Final return value: 55
(2) Total instructions: 2679
(3) R-type: 818
(4) I-type: 493
(5) J-type: 164
(6) Mem access: 1095
(7) Taken branches: 54
****************************************
```

5. Feelings

이번 과제를 하며 명령어가 어떻게 실행되는지를 몸소 체감할 수 있었다. 처음에는 그냥 단순히 돌아가는 순서만 이론적으로 배웠다면 코드를 짜면서 각각의 단계가 어떤 역할을 하고 레지스터와 ALU 등 사이를 어떻게 오고 가는지를 과제를 하면서 배웠다. 시험과 같이 공부하니 훨씬 수월하다고 느꼈고 input4.bin을 실행했을 때 또 다시 무한 루프에 걸리는 걸 보고 다시 처음으로 돌아간 거 같은 기분이 들었지만 알고 보니 JALR을 구현하지 않아서 발생한 문제였다. 한 사이클마다 나오는 값이 달라서 CPU가 살아있는 듯한 느낌을 받았으며 과제를 하며 무작정 구현을 하기보다 시작하기전 구상을 하고 들어가는 것이 더 중요하다고 느꼈다.