목차

1. 과제 개요

1.1 과제 목표

본 과제는 16비트 단순 명령어 컴퓨터(CPU16)를 시뮬레이션하는 프로그램을 구현하는 것이다. CPU의 핵심 구성요소인 ALU, CU, 레지스터(PC, IR, AC)를 클래스로 설계하고, 4096바이트 크기의 메모리를 구현하여 콘솔에서 어셈블리 코드를 입력받아 실시간으로 실행하고 메모리 상태를 확인할 수 있는 시스템을 개발하는 것을 목표로 한다.

1.2 요구사항 분석

1) CPU 클래스 설계

ALU(산술논리장치), CU(제어장치), PC(프로그램 카운터), IR(명령어 레지스터), AC(누산기) 클래스를 구현하여 CPU를 구성한다.

2) Memory 클래스 설계

- 구조: 실제 데이터를 저장할 4096바이트 공간과 유효 범위(0~4095) 내에서 이동 가능한 메모리 포인터를 설계한다.

- 동작: 읽기, 쓰기, 초기화, 포인터 설정, 메모리 내용 보기 기능을 구현한다.

3) 16비트 단순 명령어 컴퓨터 구현

- 처리 대상 명령어: LDA, STA, ADD, MUL, DIV, MOD, SEA (총 7개)

- 필수 테스트: SEA 7EF → ADD 4 → MUL 40 → STA 8

- 선택 테스트: DIV와 MOD를 활용한 임의의 프로그램

위 테스트 프로그램들을 메모리에 로드한 후 명령어를 한 줄씩 읽고 해독하여 실행하며, 각 단계에서 메모리의 상태 변화를 분석한다.

4) CPU

CPU16은 16비트 단순 명령어 컴퓨터의 중앙처리장치를 처리하는 클래스이다. ALU(산술논리장치), CU(제어장치), 그리고 세 개의 레지스터(PC, IR, AC)로 구성되며, Fetch → Decode → Execute 사이클을 통해 명령어를 처리한다.

(1) ALU

ALU는 CPU에서 실제 산술 연산을 수행하는 장치이다. AC(누산기) 레지스터의 값과 피연산자를 받아 덧셈, 곱셈, 나눗셈, 나머지 연산을 처리하고 그 결과를 반환한다.

```
class ALU {
public:
    uint16_t add(uint16_t a, uint16_t b) {
        return a + b;
    }

    uint16_t multiply(uint16_t a, uint16_t b) {
        return a * b;
    }

    uint16_t divide(uint16_t a, uint16_t b) {
        if (b == 0) {
            cerr << "[Error] Division by zero" << endl;
            return 0;
        }
        return a / b;
    }
```

```
    uint16_t modulo(uint16_t a, uint16_t b) {
        if (b == 0) {
            cerr << "[Error] Modulo by zero" << endl;
            return 0;
        }
        return a % b;
    }
};
```

각 함수는 두 개의 16비트 값을 입력받아 연산 결과를 반환한다. 나눗셈과 나머지 연산의 경우 0으로 나누는 예외 상황을 처리하여 에러 메시지를 출력하고 0을 반환한다.

해당 함수들은 사용자가 직접 호출하지 않으며, ADD, MUL, DIV, MOD 명령어 실행 시 CPU 내부에서 자동으로 호출된다.

(2) CU

CU는 명령어를 해독(Decode)하고 어떤 동작을 수행해야 하는지 결정하는 장치이다. IR(명령어 레지스터)에 저장된 16비트 명령어에서 상위 4비트(opcode)를 추출하여 해당 명령어의 종류를 식별한다.

CU 클래스는 opcode 해독 및 변환 함수를 제공한다.

```
enum Opcode {
    OP_LDA = 0x0,    // 0000
    OP_STA = 0x1,    // 0001
    OP_ADD = 0x2,    // 0010
    OP_MUL = 0x3,    // 0011
    OP_DIV = 0x4,    // 0100
```

```
    OP_MOD = 0x5,     // 0101
    OP_SEA = 0xF      // 1111
};

class CU {
public:
    Opcode decode(uint16_t instruction) {
        return static_cast<Opcode>((instruction >> 12) & 0x0F);
    }

    string opcodeToString(Opcode op) {
        switch(op) {
            case OP_LDA: return "LDA";
            case OP_STA: return "STA";
            case OP_ADD: return "ADD";
            case OP_MUL: return "MUL";
            case OP_DIV: return "DIV";
            case OP_MOD: return "MOD";
            case OP_SEA: return "SEA";
            default: return "UNKNOWN";
        }
    }

    int stringToOpcode(const string& str) {
        if (str == "LDA") return OP_LDA;
```

```
        if (str == "STA") return OP_STA;
        // ... 생략
        return -1;
    }
};
```

- **decode():** 16비트 명령어에서 상위 4비트를 추출하여 opcode를 반환한다.

- **opcodeToString():** opcode를 문자열로 변환하여 실행 과정 출력 시 사용한다.

- **stringToOpcode():** 어셈블리 코드 입력 시 문자열을 opcode로 변환한다.

위의 함수들은 사용자가 직접 호출하지 않으며, 명령어 실행 시 Decode 단계에서 내부적으로 사용된다.

(3) PC

PC는 다음에 실행할 명령어가 저장된 메모리 주소를 가리키는 레지스터이다. 명령어를 Fetch한 후 자동으로 2씩 증가하여 다음 명령어 위치를 가리킨다. (16비트 명령어 = 2바이트)

```
//Register 클래스
class Registerable {
public:
    virtual void set(uint16_t value) = 0;
    virtual uint16_t get() const = 0;
    virtual void reset() = 0;
    virtual string getName() const = 0;
    virtual ~Registerable() {}
};
```

```
class Register : public Registerable {
protected:
    uint16_t value;
    string name;

public:
    Register(const string& regName) : value(0), name(regName) {}
    virtual ~Register() {}

    void set(uint16_t val) override { value = val; }
    uint16_t get() const override { return value; }
    void reset() override { value = 0; }
    string getName() const override { return name; }

    void print() const {
        cout << name << ": 0x" << hex << uppercase
            << setw(4) << setfill('0') << value << dec << endl;
    }
};
```

```
//PC 클래스
class PC : public Register {
public:
```

```
    PC() : Register("PC") {}


    void increment() { value += 2; }   // 16bit = 2bytes씩 증가
};
```

- **set(uint16_t val)`:** PC 값을 특정 주소로 설정한다. (프로그램 시작 시 사용)

- **get():** 현재 PC 값을 반환한다.

- **increment():** PC 값을 2 증가시킨다. (Fetch 후 자동 호출)

- **reset():** PC 값을 0으로 초기화한다.

쉘에서 사용할 때는 `reg` 명령어를 입력하여 PC를 포함한 모든 레지스터의 현재 상태를 확인할 수 있다.

```
CPU16> reg

----- Register Status -----
PC: 0x0000
IR: 0x0000
AC: 0x0000
---------------------------
```

(4) IR

IR은 현재 실행 중인 명령어를 저장하는 레지스터이다. Fetch 단계에서 메모리로부터 읽어온 16비트 명령어가 IR에 저장되며, Decode 단계에서 이 값을 분석하여 opcode(상위 4비트)와 operand(하위 12비트)를 추출한다.

IR 클래스는 Register 클래스를 상속받아 구현하였으며, opcode와 operand를 추출하는 함수를 추가로 제공한다.

```
class IR : public Register {
```

```
public:
    IR() : Register("IR") {}

    uint16_t getOpcode() const { return (value >> 12) & 0x0F; }
    uint16_t getOperand() const { return value & 0x0FFF; }
};
```

- **set(uint16_t val):** Fetch된 명령어를 IR에 저장한다.

- **get():** 현재 저장된 16비트 명령어 전체를 반환한다.

- **getOpcode():** 상위 4비트(opcode)를 추출하여 반환한다.

- **getOperand():** 하위 12비트(operand)를 추출하여 반환한다.

(5) AC

AC는 모든 산술 연산의 결과가 AC에 저장되며, 다음 연산의 입력값으로 사용된다. LDA 명령어로 메모리에서 값을 로드하거나, SEA 명령어로 즉시값을 저장하고, STA 명령어로 AC의 값을 메모리에 저장할 수 있다.

AC 클래스는 Register 클래스를 상속받아 구현하였다.

```
class AC : public Register {
public:
    AC() : Register("AC") {}
};
```

- **set(uint16_t val):** AC에 값을 저장한다.

- **get():** AC의 현재 값을 반환한다.

- **reset():** AC 값을 0으로 초기화한다.

명령어별 AC 동작을 정리하면 아래와 같다

명령어	AC 동작
LDA x	AC ← Memory[x] (메모리에서 값 로드)
STA X	Memory[x] ← AC (AC 값을 메모리에 저장)
ADD X	AC ← AC + x
MUL X	AC ← AC × x
DIV X	AC ← AC ÷ x (몫)
MOD X	AC ← AC % x (나머지)
SEA X	AC ← x (즉시값 로드) AC ← AC × x

```
CPU16> step

[Fetch] PC=0x0 -> IR=0x0008
[Decode] Opcode=LDA, Operand=0x8
[Execute] LDA: AC = Memory[0x8] = 0x0

----- Register Status -----
PC: 0x0002
IR: 0x0008
AC: 0x0000
---------------------------
```

`step` 명령어로 한 줄씩 실행하며 각 명령어 실행 후 AC 값의 변화를 추적할 수 있다.

3. Memory

3.1 구조

1) 저장 공간

메모리는 4096바이트(2^12) 크기의 저장 공간을 가지며, unsigned char 배열로 구현하였다. 각 바이트는 0~255 범위의 값을 저장할 수 있으며, 16비트 명령어나 데이터를 저장할 때는 연속된 2바이트를 사용한다.

```
static const unsigned int MEMORY_SIZE = 4096;

unsigned char data[MEMORY_SIZE];
```

이와 같이 메모리 구조를 설계했을 때 16비트 값 0xF7EF를 주소 0에 저장하면 다음과 같이 배치된다:

- 주소 0: 0xF7 (상위 바이트)

- 주소 1: 0xEF (하위 바이트)

2) 메모리 포인터

메모리 포인터(mpt)는 현재 작업 중인 메모리 위치를 가리키는 변수로, 0부터 4095까지의 유효 범위 내에서 이동할 수 있다. unsigned int 타입으로 선언하여 음수 값이 들어오는 것을 방지하였으며, 포인터를 활용하면 주소를 직접 지정하지 않고도 현재 위치에서 읽기/쓰기 작업을 수행할 수 있다.

```
unsigned int mpt = 0;   // 유효 범위: 0~4095
```

위와 같이 만든 메모리 포인터는 아래와 같은 일을 한다

메모리 포인터는 다음과 같은 기능을 제공한다:

- 포인터 초기화: 메모리의 시작 주소(0)로 설정

- 포인터 설정: 임의의 유효한 주소로 변경

- 포인터 유효성 확인: 입력된 주소가 유효 범위(0~4095) 내에 있는지 검사

2.2 동작 정의 및 구현

1) 읽기

메모리에서 16비트 데이터를 읽어오는 기능으로, 세 가지 방식을 제공한다.

(1) 특정 주소에서 읽기 (read(unsigned int address))

- Input: 읽고자 하는 메모리 주소 (unsigned int)

- Output: 해당 주소에서 읽은 16비트 값 (uint16_t)

- 동작: 입력받은 주소와 그 다음 주소(address+1)의 유효성을 검사한 후, 두 바이트를 조합하여 16비트 정수로 반환한다. 유효하지 않은 주소인 경우 에러 메시지를 출력하고 0을 반환한다.

- 쉘 명령어: read <주소> (예: read 8)

```
uint16_t Memory::read(unsigned int address) {
    if (!isValidAddress(address) || !isValidAddress(address + 1)) {
        cerr << "[Error] Invalid address for read: " << address << endl;
        return 0;
    }
    return (static_cast<uint16_t>(data[address]) << 8) |
           static_cast<uint16_t>(data[address + 1]);
}
```

(2) **현재 포인터 위치에서 읽기 (read())**

- Input: 없음 (현재 메모리 포인터 위치 사용)

- Output: 현재 포인터가 가리키는 주소에서 읽은 16비트 값 (uint16_t)

- 동작: 내부적으로 read(mpt)를 호출하여 현재 메모리 포인터가 가리키는 위치에서 16비트 값을

읽어 반환한다.

```
uint16_t Memory::read() {

    return read(mpt);

}
```

(3) **메모리 시작 주소에서 읽기 (readFromStart())**

- Input: 없음 (시작 주소 0 사용)

- Output: 주소 0에서 읽은 16비트 값 (uint16_t)

- 동작: 내부적으로 read(0)을 호출하여 메모리의 첫 번째 주소에서 16비트 값을 읽어 반환한다.

```
uint16_t Memory::readFromStart() {

    return read(0);

}
```

(4) 쉘에서의 읽기 명령어

1. read(단일 주소 읽기)

- 형식: read <주소>

- 동작: 지정한 주소에서 `read(address)` 함수를 호출하여 16비트 값 하나를 읽고, 그 결과를 화면에 출력한다.

- 출력 예시:

```
CPU16> read 2
Memory[0x2] = 0x2004
```

2. dump(범위 읽기)

- 형식: dump <시작주소> <끝주소>

- 동작: 시작 주소부터 끝 주소까지 2바이트 단위로 `read(address)` 함수를 반복 호출하여 해당 범위의 메모리 내용을 16진수로 출력한다.

- 출력 예시:

```
CPU16> dump 0 7

========== Memory Dump ==========
Address Range: 0x000 - 0x007
=================================
0x000: 0xF7EF
0x002: 0x2004
0x004: 0x3040
0x006: 0x1008
=================================
```

2) 쓰기

메모리에 16비트 데이터를 저장하는 기능으로, 세 가지 방식을 제공한다.

(1) 특정 주소에 쓰기 (write(unsigned int address, uint16_t value))

- Input: 저장할 메모리 주소 (unsigned int), 저장할 16비트 값 (uint16_t)

- Output: 없음 (void)

- 동작: 입력받은 주소와 그 다음 주소(address+1)의 유효성을 검사한 후, 16비트 값을 두 바이트로 분리하여 저장한다. Big Endian 방식에 따라 상위 바이트를 낮은 주소에, 하위 바이트를 높은 주소에 저장한다. 유효하지 않은 주소인 경우 에러 메시지를 출력하고 저장을 수행하지 않는다.

```
void Memory::write(unsigned int address, uint16_t value) {

    if (!isValidAddress(address) || !isValidAddress(address + 1)) {

        cerr << "[Error] Invalid address for write: " << address << endl;

        return;

    }
```

```
    data[address] = static_cast<unsigned char>((value >> 8) & 0xFF);

    data[address + 1] = static_cast<unsigned char>(value & 0xFF);

}
```

(2) 현재 포인터 위치에 쓰기 (write(uint16_t value))

- Input: 저장할 16비트 값 (uint16_t)

- Output: 없음

- 동작: 내부적으로 write(mpt, value)를 호출하여 현재 메모리 포인터가 가리키는 위치에 16비트 값을 저장한다.

- 사용: 사용자가 직접 호출하지 않으며, 어셈블리 코드를 메모리에 로드할 때 내부적으로 사용된다.

```
void Memory::write(uint16_t value) {

    write(mpt, value);

}
```

(3) 메모리 시작 주소에 쓰기 (writeToStart(uint16_t value))

- Input: 저장할 16비트 값 (uint16_t)

- Output: 없음 (void)

- 동작: 내부적으로 write(0, value)를 호출하여 메모리의 첫 번째 주소에 16비트 값을 저장한다.

- 사용: 사용자가 직접 호출하지 않으며, 프로그램을 시작 주소부터 로드할 때 내부적으로 사용된다.

```
void Memory::writeToStart(uint16_t value) {

    write(0, value);
```

```
}
```

3) 초기화

메모리의 내용을 초기화하는 기능으로, 세 가지 방식을 제공한다.

(1) **특정 주소 초기화 (clear(unsigned int address))**

- Input: 초기화할 메모리 주소 (unsigned int)

- Output: 없음 (void)

- 동작: 내부적으로 write(address, 0)을 호출하여 해당 주소부터 2바이트(16비트)를 0으로 초기화한다.

- 쉘 명령어: clear <주소>

```
void Memory::clear(unsigned int address) {

    write(address, 0);

}
```

(2) **메모리 전체를 0으로 초기화 (clearAll())**

- Input: 없음

- Output: 없음 (void)

- 동작: memset 함수를 사용하여 메모리 전체(4096바이트)를 0으로 초기화한다.

- 쉘 명령어: clear (주소 없이 입력)

```
void Memory::clearAll() {

    memset(data, 0, MEMORY_SIZE);
```

```
}
```

(3) **메모리 전체를 임의의 값으로 초기화 (fillAll(unsigned char value))**

- Input: 초기화할 값 (unsigned char)

- Output: 없음 (void)

- 동작: memset 함수를 사용하여 메모리 전체(4096바이트)를 입력받은 값으로 채운다.

- 사용: 사용자가 직접 호출하지 않으며, 메모리 테스트나 디버깅 시 내부적으로 사용된다.

```
void Memory::fillAll(unsigned char value) {
    memset(data, value, MEMORY_SIZE);
}
```

4) 포인터 설정

메모리 포인터를 관리하는 기능으로, 세 가지 함수를 제공한다.

(1) **포인터 초기화 (resetPointer())**

- Input: 없음

- Output: 없음 (void)

- 동작: 메모리 포인터를 시작 주소인 0으로 설정한다.

- 쉘 명령어: resetptr

```
void Memory::resetPointer() {
    mpt = 0;
```

```
}
```

(2) **포인터 설정 (setPointer(unsigned int address))**

- Input: 설정할 주소 값 (unsigned int)

- Output: 설정 성공 여부 (bool)

- 동작: 입력받은 주소의 유효성을 검사한 후, 유효한 경우 포인터를 해당 주소로 변경하고 true를 반환한다. 유효하지 않은 주소인 경우 에러 메시지를 출력하고 false를 반환한다.

- 쉘 명령어: ptr <주소>

```
bool Memory::setPointer(unsigned int address) {
    if (!isValidAddress(address)) {
        cerr << "[Error] Invalid pointer address: " << address << endl;
        return false;
    }
    mpt = address;
    return true;
}
```

(3) 현재 포인터 값 반환 (getPointer())

- Input : 확인할 주소 값 (unsigned int)

- Output : 유효 여부 (bool)

- 동작 : 입력받은 주소가 메모리의 유효 범위(0~4095) 내에 있는지 검사한다. 주소가 MEMORY_SIZE(4096)보다 작으면 true를, 그렇지 않으면 false를 반환한다. 이 함수는 포인터 설정뿐만 아니라 메모리 읽기, 쓰기 등 모든 메모리 접근 연산에서 주소 유효성을 검증하는 데 사용된다.

- 쉘 명령어: ptr (주소 없이 입력하면 현재 포인터 값 출력)

```
bool Memory::isValidAddress(unsigned int address) {
    return address < MEMORY_SIZE;
}
```

5) 메모리 내용 보기

메모리의 내용을 화면에 출력하는 기능으로, 세 가지 방식을 제공한다.

(1) **전체 메모리 내용 보기 (dump(bool binary))**

- Input: 2진수 출력 여부 (bool, 기본값 false)

- Output: 없음 (콘솔에 출력)

- 동작: 메모리 전체(0x000 ~ 0xFFF, 4096바이트)의 내용을 2바이트 단위로 출력한다. binary가 false이면 16진수로, true이면 2진수로 출력한다.

- 쉘 명령어:

  - 16진수 출력: dumpall
  - 2진수 출력: dumpall bin

- 구현

```
void Memory::dump(unsigned int startAddr, unsigned int endAddr, bool binary) {
    if (!isValidAddress(startAddr) || !isValidAddress(endAddr)) {
        cerr << "[Error] Invalid address range" << endl;
        return;
    }
```

```
    if (startAddr > endAddr) {
        cerr << "[Error] Start address must be <= end address" << endl;
        return;
    }
    // 2바이트 단위로 순회하며 출력
    for (unsigned int addr = startAddr; addr <= endAddr; addr += 2) {
        uint16_t value = read(addr);
        // binary 값에 따라 출력 형식 결정
    }
}
```

- 출력 예시:

```
CPU16> dumpall

========== Memory Dump ==========
Address Range: 0x000 - 0xFFF
=================================
0x000: 0xF7EF
0x002: 0x2004
0x004: 0x3040
0x006: 0x1008
0x008: 0x0000
0x00A: 0x0000
0x00C: 0x0000
0x00E: 0x0000
0x010: 0x0000
0x012: 0x0000
0x014: 0x0000
0x016: 0x0000
0x018: 0x0000
0x01A: 0x0000
0x01C: 0x0000
0x01E: 0x0000
0x020: 0x0000
0x022: 0x0000
0x024: 0x0000
0x026: 0x0000
0x028: 0x0000
0x02A: 0x0000
0x02C: 0x0000
0x02E: 0x0000
0x030: 0x0000
0x032: 0x0000
```

(2) 임의 범위 메모리 내용 보기 (dump(unsigned int startAddr, unsigned int endAddr, bool

binary))

- Input: 시작 주소 (unsigned int), 끝 주소 (unsigned int), 2진수 출력 여부 (bool)

- Output: 없음 (콘솔에 출력)

- 동작: 시작 주소와 끝 주소의 유효성을 검사하고, 시작 주소가 끝 주소보다 작거나 같은지 확인한다. 유효한 경우 지정된 범위의 메모리 내용을 2바이트(16비트) 단위로 출력한다. 유효하지 않은 범위인 경우 에러 메시지를 출력한다.

- 쉘 명령어: dump <시작주소> <끝주소>

- 출력 예시:

```
CPU16> dump 0 7

========== Memory Dump ==========
Address Range: 0x000 – 0x007
=================================
0x000: 0xF7EF
0x002: 0x2004
0x004: 0x3040
0x006: 0x1008
=================================
```

(3) 16진수 / 2진수 선택 출력

- 동작: 모든 메모리 출력 명령어에서 16진수 또는 2진수 형식을 선택할 수 있다. 기본값은 16진수이다.

- 쉘 명령어:

    - 16진수 출력: `dump <시작> <끝>` (예: `dump 0 f`)

    - 2진수 출력: `dumpbin <시작> <끝>` (예: `dumpbin 0 7`) - 출력 예시 비교:

- 출력 예시

16진수 출력	2진수 출력

```
CPU16> dump 0 7

========== Memory Dump ==========
Address Range: 0x000 - 0x007
==================================
0x000: 0xF7EF
0x002: 0x2004
0x004: 0x3040
0x006: 0x1008
==================================
```

```
CPU16> dumpbin 0 7

========== Memory Dump ==========
Address Range: 0x000 - 0x007
==================================
0x000: 1111 0111 1110 1111
0x002: 0010 0000 0000 0100
0x004: 0011 0000 0100 0000
0x006: 0001 0000 0000 1000
==================================
```

4. 단순 명령어 컴퓨터

4.1 명령어의 구조와 종류

1) 명령어의 구조

본 시스템에서 사용하는 명령어는 16비트로 구성되며, 상위 4비트는 연산 코드(opcode), 하위 12비트는 피연산자(operand)를 나타낸다.

```
| 15 14 13 12 | 11 10 9 8 7 6 5 4 3 2 1 0 |

|   opcode    | operand                  |

|    (4비트)    | (12비트)                  |
```

- **opcode (4비트)**: 수행할 연산의 종류를 지정한다. 0000부터 1111까지 총 16가지 명령어를 표현할 수 있으며, 본 과제에서는 7개의 명령어를 사용한다.

- **operand (12비트)**: 연산에 사용할 피연산자를 지정한다. 16진수 정수 상수(0x000 ~ 0xFFF) 또는 메모리 주소를 나타낸다.

2) 명령어의 종류

본 시스템은 총 7개의 명령어를 지원한다.

(1) LDA

- **opcode**: 0000 (0x0)

- **형식**: LDA x

- **동작**: 메모리 주소 x에서 16비트 값을 읽어 AC 레지스터에 저장한다.

- **연산**: AC ← Memory[x:x+1]

(2) STA

- **opcode**: 0001 (0x1)

- **형식**: STA x

- **동작**: AC 레지스터의 값을 메모리 주소 x에 저장한다.

- **연산**: Memory[x:x+1] ← AC

(3) ADD (Add)

- **opcode**: 0010 (0x2)

- **형식**: ADD x

- **동작**: AC 레지스터의 값과 피연산자 x를 더하여 AC에 저장한다.

- **연산**: AC ← AC + x

(4) MUL (Multiply)

- **opcode**: 0011 (0x3)

- **형식**: MUL x

- **동작**: AC 레지스터의 값과 피연산자 x를 곱하여 AC에 저장한다.

- **연산**: AC ← AC × x

(5) DIV (Divide)

- **opcode**: 0100 (0x4)

- **형식**: DIV x

- **동작**: AC 레지스터의 값을 피연산자 x로 나눈 몫을 AC에 저장한다.

- **연산**: AC ← AC ÷ x

(6) MOD (Modulo)

- **opcode**: 0101 (0x5)

- **형식**: MOD x

- **동작**: AC 레지스터의 값을 피연산자 x로 나눈 나머지를 AC에 저장한다.

- **연산**: AC ← AC % x

(7) SEA (Set Accumulator)

- **opcode**: 1111 (0xF)

- **형식**: SEA x

- **동작**: 피연산자 x의 값을 즉시값으로 AC 레지스터에 저장한다.

- **연산**: AC ← x

4.2 Test1

1) 실행

```
CPU16> load 0
Enter assembly code (type 'END' to finish):
  > SEA 7EF
  > ADD 4
  > MUL 40
  > STA 8
  > END

Loading program at address 0x0:
  0x000: SEA 7EF -> 0xf7ef
  0x002: ADD 4 -> 0x2004
  0x004: MUL 40 -> 0x3040
  0x006: STA 8 -> 0x1008
Loaded 4 instructions.
```

```
CPU16> prepare 0 7

========== Step Mode Ready ==========
Start Address: 0x0
End Address: 0x7
Type 'step' or 's' to execute next instruction
=====================================
```

2) 분석

(1) 초기상태

```
CPU16> dump 0 7

========== Memory Dump ==========
Address Range: 0x000 - 0x007
==================================
0x000: 0xF7EF
0x002: 0x2004
0x004: 0x3040
0x006: 0x1008
==================================
```

초기에는 입력한 4개의 명령어가 operator, operand 의 구조로 하여 메모리에 저장되어 있는 것을 확인할 수 있다.

(2) SEA 7EF

```
CPU16> step

[Fetch] PC=0x0 -> IR=0xF7EF
[Decode] Opcode=SEA, Operand=0x7EF
[Execute] SEA: AC = 0x7EF

----- Register Status -----
PC: 0x0002
IR: 0xF7EF
AC: 0x07EF
---------------------------
```

```
CPU16> dump 0 9

========== Memory Dump ==========
Address Range: 0x000 - 0x009
=================================
0x000: 0xF7EF
0x002: 0x2004
0x004: 0x3040
0x006: 0x1008
0x008: 0x0000
=================================
```

Fetch 단계에서 PC가 가리키는 주소 0에서 명령어 0xF7EF를 읽어 IR에 저장한다. Decode 단계에서 IR의 상위 4비트(1111)를 분석하여 SEA 명령어임을 확인하고, 하위 12비트(0x7EF)를 피연산자로 추출한다. Execute 단계에서 피연산자 0x7EF(10진수 2031)를 AC에 저장한다.

실행 후 PC는 다음 명령어 위치인 0x0002로 2 증가하고, IR에는 방금 실행한 명령어 0xF7EF가 저장되어 있으며, AC에는 새로 저장된 값 0x07EF(10진수 2031)가 들어있다. SEA 명령어는 AC에만 값을 저장하므로 메모리 변화는 없다.

(3) ADD 4

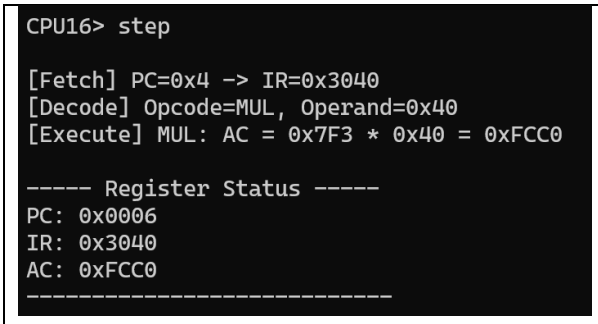
```
CPU16> step

[Fetch] PC=0x2 -> IR=0x2004
[Decode] Opcode=ADD, Operand=0x4
[Execute] ADD: AC = 0x7EF + 0x4 = 0x7F3

----- Register Status -----
PC: 0x0004
IR: 0x2004
AC: 0x07F3
---------------------------
```

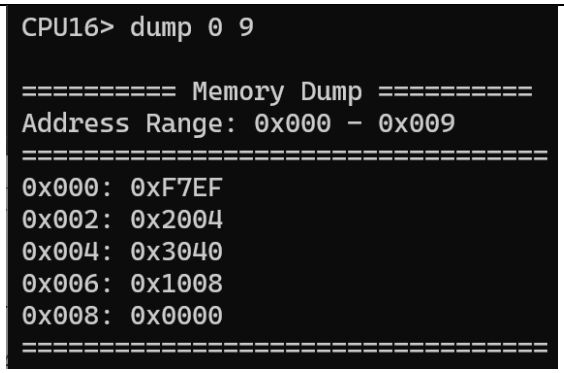
```
CPU16> dump 0 9

========== Memory Dump ==========
Address Range: 0x000 - 0x009
=================================
0x000: 0xF7EF
0x002: 0x2004
0x004: 0x3040
0x006: 0x1008
0x008: 0x0000
=================================
```

Fetch 단계에서 PC가 가리키는 주소 2에서 명령어 0x2004를 읽어 IR에 저장한다. Decode 단계에서 IR의 상위 4비트(0010)를 분석하여 ADD 명령어임을 확인하고, 하위 12비트(0x4)를 피연산자로 추출한다. Execute 단계에서 AC의 현재 값(0x7EF)에 피연산자(0x4)를 더하여 결과값 0x7F3(10진수 2035)을 AC에 저장한다.

실행 후 PC는 다음 명령어 위치인 0x0004로 2 증가하고, IR에는 방금 실행한 명령어 0x2004가 저장되어 있으며, AC에는 연산 결과 0x07F3(10진수 2035)이 들어있다. ADD 명령어는 AC 값만 변경하므로 메모리 변화는 없다.

(4) MUL 40 실행

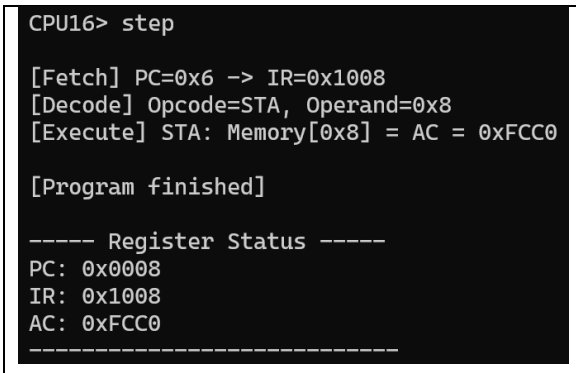


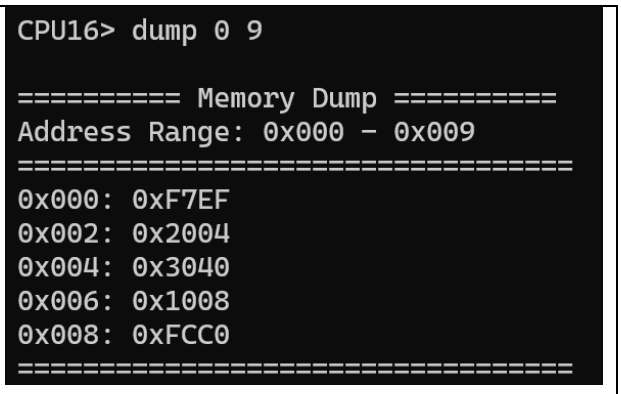


Fetch 단계에서 PC가 가리키는 주소 4에서 명령어 0x3040을 읽어 IR에 저장한다. Decode 단계에서 IR의 상위 4비트(0011)를 분석하여 MUL 명령어임을 확인하고, 하위 12비트(0x40)를 피연산자로 추출한다. Execute 단계에서 AC의 현재 값(0x7F3)에 피연산자(0x40, 10진수 64)를 곱하여 결과값 0xFCC0(10진수 64704)을 AC에 저장한다.

실행 후 PC는 다음 명령어 위치인 0x0006으로 2 증가하고, IR에는 방금 실행한 명령어 0x3040이 저장되어 있으며, AC에는 연산 결과 0xFCC0(10진수 64704)이 들어있다. MUL 명령어는 AC 값만 변경하므로 메모리 변화는 없다.

(5) STA 8 실행





Fetch 단계에서 PC가 가리키는 주소 6에서 명령어 0x1008을 읽어 IR에 저장한다. Decode 단계에서 IR의 상위 4비트(0001)를 분석하여 STA 명령어임을 확인하고, 하위 12비트(0x8)를 저장할 메모리 주소로 추출한다. Execute 단계에서 AC의 현재 값(0xFCC0)을 메모리 주소 8에 저장한다.

실행 후 PC는 0x0008로 2 증가하여 프로그램이 종료되고, IR에는 방금 실행한 명령어 0x1008이 저장되어 있으며, AC는 0xFCC0으로 값 변경 없이 유지된다. STA 명령어 실행으로 메모리 주소 0x008의 값이 0x0000에서 0xFCC0으로 변경된다.

4.3 Test2

1) 코드 설계

```
SEA 64

DIV A

STA 10

SEA 65

MOD A

STA 12
```

본 코드는 DIV(나눗셈)와 MOD(나머지) 명령어의 동작을 확인하기 위해 작성되었다. 100을 10으로 나눈 몫(10)과 101을 10으로 나눈 나머지(1)를 각각 계산하여 메모리에 저장한다.

2) 실행

```
CPU16> load 0
Enter assembly code (type 'END' to finish):
  > SEA 64
  > DIV A
  > STA 10
  > SEA 65
  > MOD A
  > STA 12
  > END

Loading program at address 0x0:
  0x000: SEA 64 -> 0xf064
  0x002: DIV A -> 0x400a
  0x004: STA 10 -> 0x1010
  0x006: SEA 65 -> 0xf065
  0x008: MOD A -> 0x500a
  0x00a: STA 12 -> 0x1012
Loaded 6 instructions.
```

3) 분석

(1) SEA 64

```
CPU16> s

[Fetch] PC=0x0 -> IR=0xf064
[Decode] Opcode=SEA, Operand=0x64
[Execute] SEA: AC = 0x64

----- Register Status -----
PC: 0x0002
IR: 0xF064
AC: 0x0064
---------------------------
```

```
CPU16> dump 0 15

========== Memory Dump ==========
Address Range: 0x000 - 0x015
==================================
0x000: 0xF064
0x002: 0x400A
0x004: 0x1010
0x006: 0xF065
0x008: 0x500A
0x00A: 0x1012
0x00C: 0x0000
0x00E: 0x0000
0x010: 0x0000
0x012: 0x0000
0x014: 0x0000
==================================
```

Fetch 단계에서 PC가 가리키는 주소 0에서 명령어 0xF064를 읽어 IR에 저장한다. Decode 단계에서 IR의 상위 4비트(1111)를 분석하여 SEA 명령어임을 확인하고, 하위 12비트(0x64)를 피연산자로 추출한다. Execute 단계에서 피연산자 0x64(10진수 100)를 AC에 즉시 저장한다.

실행 후 PC는 다음 명령어 위치인 0x0002로 2 증가하고, IR에는 방금 실행한 명령어 0xF064가 저장되어 있으며, AC에는 새로 저장된 값 0x0064(10진수 100)가 들어있다. SEA 명령어는 AC에만 값을 저장하므로 메모리 변화는 없다.

(2) DIV A

```
CPU16> s

[Fetch] PC=0x2 -> IR=0x400A
[Decode] Opcode=DIV, Operand=0xA
[Execute] DIV: AC = 0x64 / 0xA = 0xA

----- Register Status -----
PC: 0x0004
IR: 0x400A
AC: 0x000A
---------------------------
```

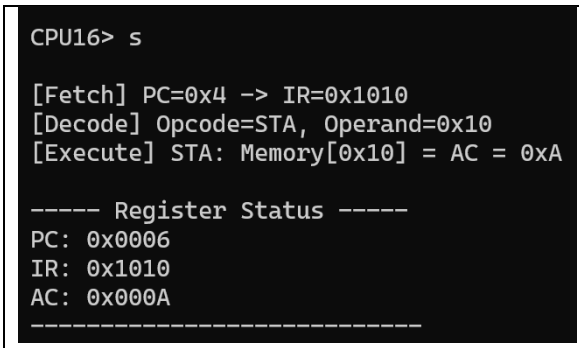
```
CPU16> dump 0 15

========== Memory Dump ==========
Address Range: 0x000 - 0x015
==================================
0x000: 0xF064
0x002: 0x400A
0x004: 0x1010
0x006: 0xF065
0x008: 0x500A
0x00A: 0x1012
0x00C: 0x0000
0x00E: 0x0000
0x010: 0x0000
0x012: 0x0000
0x014: 0x0000
==================================
```

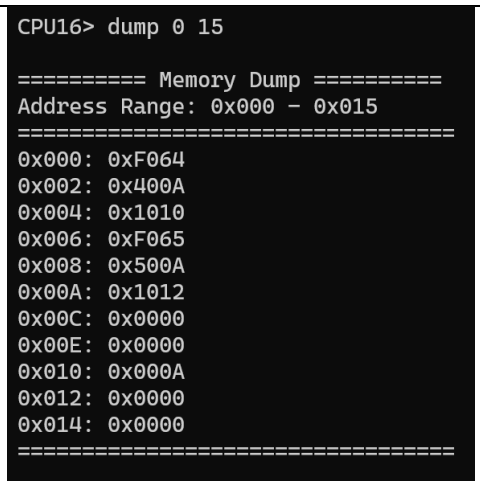
Fetch 단계에서 PC가 가리키는 주소 2에서 명령어 0x400A를 읽어 IR에 저장한다. Decode 단계에서 IR의 상위 4비트(0100)를 분석하여 DIV 명령어임을 확인하고, 하위 12비트(0xA)를 피연산자로

추출한다. Execute 단계에서 AC의 현재 값(0x64, 10진수 100)을 피연산자(0xA, 10진수 10)로 나눈 몫 0xA(10진수 10)를 AC에 저장한다.

실행 후 PC는 다음 명령어 위치인 0x0004로 2 증가하고, IR에는 방금 실행한 명령어 0x400A가 저장되어 있으며, AC에는 연산 결과 0x000A(10진수 10)가 들어있다. DIV 명령어는 AC 값만 변경하므로 메모리 변화는 없다.

(3) STA 10





Fetch 단계에서 PC가 가리키는 주소 4에서 명령어 0x1010을 읽어 IR에 저장한다. Decode 단계에서 IR의 상위 4비트(0001)를 분석하여 STA 명령어임을 확인하고, 하위 12비트(0x10)를 저장할 메모리 주소로 추출한다. Execute 단계에서 AC의 현재 값(0xA)을 메모리 주소 0x10에 저장한다.

실행 후 PC는 다음 명령어 위치인 0x0006으로 2 증가하고, IR에는 방금 실행한 명령어 0x1010이 저장되어 있으며, AC는 0x000A로 값 변경 없이 유지된다. STA 명령어 실행으로 메모리 주소 0x010의 값이 0x0000에서 0x000A로 변경된다.

(4) SEA 65

```
CPU16> s

[Fetch] PC=0x6 -> IR=0xF065
[Decode] Opcode=SEA, Operand=0x65
[Execute] SEA: AC = 0x65

----- Register Status -----
PC: 0x0008
IR: 0xF065
AC: 0x0065
---------------------------
```

```
CPU16> dump 0 15

========== Memory Dump ==========
Address Range: 0x000 - 0x015
==================================
0x000: 0xF064
0x002: 0x400A
0x004: 0x1010
0x006: 0xF065
0x008: 0x500A
0x00A: 0x1012
0x00C: 0x0000
0x00E: 0x0000
0x010: 0x000A
0x012: 0x0000
0x014: 0x0000
==================================
```

Fetch 단계에서 PC가 가리키는 주소 6에서 명령어 0xF065를 읽어 IR에 저장한다. Decode 단계에서 IR의 상위 4비트(1111)를 분석하여 SEA 명령어임을 확인하고, 하위 12비트(0x65)를 피연산자로 추출한다. Execute 단계에서 피연산자 0x65(10진수 101)를 AC에 즉시 저장한다.

실행 후 PC는 다음 명령어 위치인 0x0008로 2 증가하고, IR에는 방금 실행한 명령어 0xF065가 저장되어 있으며, AC에는 새로 저장된 값 0x0065(10진수 101)가 들어있다. SEA 명령어는 AC에만 값을 저장하므로 메모리 변화는 없다.

(5) MOD A

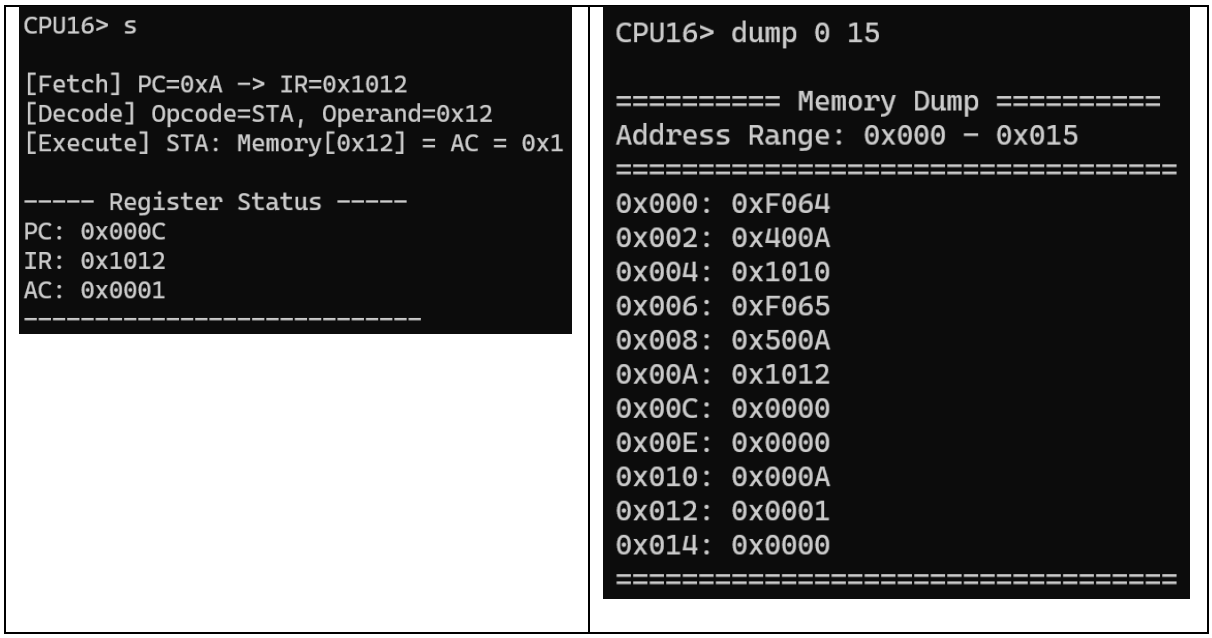
```
CPU16> s

[Fetch] PC=0x8 -> IR=0x500A
[Decode] Opcode=MOD, Operand=0xA
[Execute] MOD: AC = 0x65 % 0xA = 0x1

----- Register Status -----
PC: 0x000A
IR: 0x500A
AC: 0x0001
---------------------------
```

```
CPU16> dump 0 15

========== Memory Dump ==========
Address Range: 0x000 - 0x015
==================================
0x000: 0xF064
0x002: 0x400A
0x004: 0x1010
0x006: 0xF065
0x008: 0x500A
0x00A: 0x1012
0x00C: 0x0000
0x00E: 0x0000
0x010: 0x000A
0x012: 0x0000
0x014: 0x0000
==================================
```

Fetch 단계에서 PC가 가리키는 주소 8에서 명령어 0x500A를 읽어 IR에 저장한다. Decode 단계에서 IR의 상위 4비트(0101)를 분석하여 MOD 명령어임을 확인하고, 하위 12비트(0xA)를 피연산자로 추출한다. Execute 단계에서 AC의 현재 값(0x65, 10진수 101)을 피연산자(0xA, 10진수 10)로 나눈 나머지 0x1(10진수 1)을 AC에 저장한다.

실행 후 PC는 다음 명령어 위치인 0x000A로 2 증가하고, IR에는 방금 실행한 명령어 0x500A가 저장되어 있으며, AC에는 연산 결과 0x0001(10진수 1)이 들어있다. MOD 명령어는 AC 값만 변경하므로 메모리 변화는 없다.

(6) STA 12



Fetch 단계에서 PC가 가리키는 주소 0xA에서 명령어 0x1012를 읽어 IR에 저장한다. Decode 단계에서 IR의 상위 4비트(0001)를 분석하여 STA 명령어임을 확인하고, 하위 12비트(0x12)를 저장할 메모리 주소로 추출한다. Execute 단계에서 AC의 현재 값(0x1)을 메모리 주소 0x12에 저장한다.

실행 후 PC는 0x000C로 2 증가하여 프로그램이 종료되고, IR에는 방금 실행한 명령어 0x1012가 저장되어 있으며, AC는 0x0001로 값 변경 없이 유지된다. STA 명령어 실행으로 메모리 주소 0x012의 값이 0x0000에서 0x0001로 변경된다.