COMPUTER ARCHITECTURE
HW2 :: SINGLE-CYCLE MIPS

By using C programming

모바일시스템공학과 손보경 (32212190)
ssg020910@naver,com

2023.06.01.

목 차

1. Introduction

중간고사 이전 수업에서 컴퓨터 성능은 세 가지 주요 요인, 즉 명령어 개수, 클럭 사이클 시간, 명령어당 클럭 사이클 수(CPI)에 의해 결정된다는 것을 알았다. 앞선 수업에서 살펴본 바와 같이 컴파일러와 명령어 집합 구조가 프로그램에 필요한 명령어 개수를 결정한다. 그러나 클럭 사이클 시간과 명령어당 클럭 사이클 수는 프로세서의 구현 방법에 따라 결정된다.

이 보고서에서는 싱글 사이클 아키텍처의 작동 방식, 구형 방법, 그리고 프로젝트의 평가에 관해 설명한다. 이 프로젝트의 첫 번째 단계는 싱글 사이클 아키텍처에 대한 요구 사항을 명시하는 것이다. 두 번째로는 아키텍처 구현과 밀접한 관련이 있는 개념을 설명한다. 세 번째로는 직접 구현한 프로그램의 구조에 대해 설명한다. 그런 다음, 구현한 시뮬레이터를 사용하여 이진 프로그램을 실행한 결과 몇 가지를 보여준다. 마지막으로 요구사항을 기반으로 싱글 사이클 시뮬레이터를 평가한다.

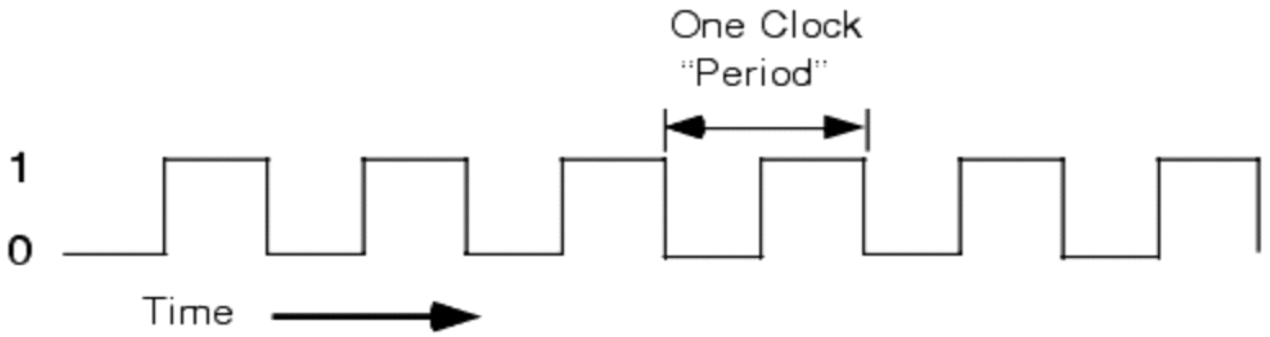
2. Requirements

Index	Requirement
1	Make a MIPS CPU emulator
2	Before the execution, the binary file is loaded into the memory
3	Assume that all register values are all zero, except for RA(r31) and SP(r29), of which value is 0xFFFFFFFF and 0x1000000
4	To begin the machine, it moves PC value to the very beginning point of the program, which is the address zero
5	MIPS executes instructions in the following stages: • Instruction fetch • Instruction decode • Execution • Load/store result to memory • Write back to reg. file
6	When your PC becomes 0xFFFFFFFF, your machine completes execution Your application is loaded to 0x0, and stack pointer is 0x1000000
7	Need to fix the jump address of each function call in manual
8	At the end of each cycle, the emulator prints out all of the changed states from the previous ones • A set of general registers, PC, and memory • Print out only changed states

3. Concepts

싱글 사이클 MIPS 시뮬레이터를 들어가기 전에, 주로 구현에 사용되는 개념인 클럭 및 사이클, 폰노이만 구조, ISA, MIPS, Datapath 및 control signal에 대해 간략히 설명하겠다.

3-1. Clock, Cycle

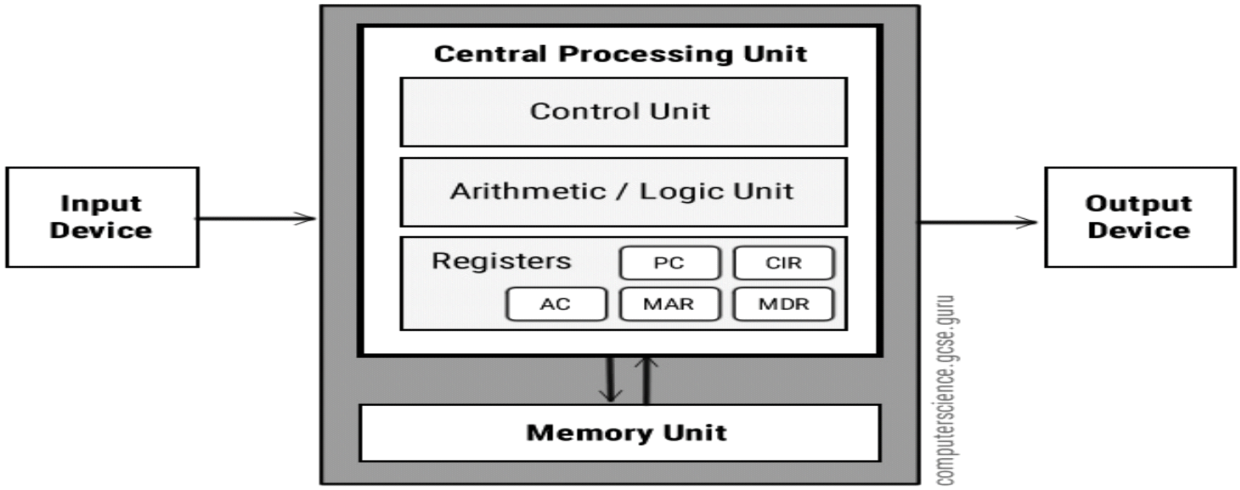


Clock Signal은 논리 상태 H(1)와 L(0)이 주기적으로 나타나는 방형파 신호를 말한다. 이는 CPU의 속도를 나타내는 단위로 쓰이는데, Clock은 1초 동안 파장이 한 번 움직이는 시간을 의미한다. 이 시간 동안 처리하는 데이터양에 따라 CPU 속도가 달라진다. Single cycle은 한 클럭당 하나의 명령어를 실행시키는 구조로 되어 있다.

Instruction Cycle은 CPU가 메모리로부터 프로그램된 한 개의 기계어 명령어를 가져와 어떠한 동작을 요구하는지 결정하고 명령어가 요구하는 동작들을 단계별로 수행하는 과정이다. 따라서 CPU clock cycle은 CPU가 명령어를 실행하기 위해 데이터를 가져오고, 명령어를 해석하고, 실행하는 단계를 말하며 clock cycle time은 한 clock 당 수행 시간을 의미한다.

Single cycle이란 한 clock에 하나의 명령어가 수행되는 구조이다. 한 명령어가 완료되어야 다음 명령어가 실행되는 구조이기 때문에 clock cycle time은 비교적 긴 편에 속한다. Single cycle의 가장 큰 특징은 모든 명령이 한 사이클에 수행된다는 것이다. 즉 한 번의 전기 신호가 컴퓨터에 가해지면, 모든 명령이 실행된다는 것이다. 이를 위해 모든 명령어는 같은 CPI(Clock per instruction)와 Cycle Time을 가지게 된다. 이는 가장 오랜 시간이 걸리는 명령에 clock이 맞춰져 있다는 것을 뜻한다. 이러한 구조의 단점은 모든 명령어가 모든 단계를 실행해야 한다는 것이다. Clock cycle time은 가장 긴 시간을 가지는 명령어인 LW, SW 시간에 맞춰져 있다. 하지만 그 밖의 명령어들은 이 모든 단계를 거치지 않아도 실행을 완료할 수 있다. 예를 들어, JUMP의 경우 굳이 Memory access, Write back 단계가 필요하지 않다. 하지만 Single Cycle의 경우 JUMP 명령어 또한 이 모든 단계를 거쳐야 실행이 완료되기 때문에 이 부분에 있어서 비효율적이다. 따라서 한 Cycle 내에서도 단계를 나누어 처리하는 Multi Cycle 기법이 도입됐다.

3-2. Von Neumann Computer



폰 노이만 구조는 1945년 존 폰 노이만이 제시한 컴퓨터 아키텍처이다. 위 그림은 폰 노이만 아키텍처의 모델을 보여준다. 폰 노이만의 컴퓨터는 산술 논리 장치(ALU)와 프로세서 레지스터를 포함하는 처리 유닛, 명령어 레지스터(IR)와 명령어 포인터(IP)를 포함하는 제어 유닛(CU), 데이터와 명령어를 저장하는 메모리, 그리고 입력과 출력 메커니즘으로 구성된다. 폰 노이만 아키텍처의 의미는 공통 BUS의 공유로 인해 명령어 검색과 데이터 연산이 동시에 발생할 수 없는 저장 프로그램 컴퓨터를 의미한다.

대부분의 컴퓨터와 마찬가지로, 싱글 사이클 MIPS 아키텍처도 폰 노이만 아키텍처의 개념을 따른다. 이는 명령어 검색(fetch) - 해석(decode) - 실행(execute) - 저장(store)의 실행 순서를 따른다. 먼저, 제어 유닛은 메모리로부터 다음 명령어를 가져온다. 제어 유닛 내의 명령어 포인터(IP) 또는 프로그램 카운터(PC)에는 가져올 다음 명령어의 주소가 포함되어 있다. 메모리 유닛은 지정된 주소에 저장된 바이트를 읽어와 데이터 버스를 통해 제어 유닛으로 전송한다. 명령어 레지스터(IR)는 메모리 유닛에서 받은 명령어의 바이트를 저장한다. 제어 유닛은 또한 IP 또는 PC 값을 증가시켜 새로운 다음 명령어의 주소를 저장한다. 그다음, 제어 유닛은 IR에 저장된 명령어를 해석한다. 세 번째로, 처리 유닛은 명령어를 실행한다. 마지막으로, 제어 유닛은 결과를 메모리에 저장한다.

3-3. ISA (Instruction Set Architecture)

명령어 세트 아키텍처(ISA)는 컴퓨터의 추상적인 모델 중 하나로, 소프트웨어에 의해 CPU가 제어되는 방식을 정의한다. ISA는 하드웨어와 소프트웨어 간의 인터페이스 역할을 하며, 프로세서가 수행할 수 있는 작업과 그 수행 방법을 지정한다. ISA는 사용자가 하드웨어와 상호작용할 수 있는 유일한 방법으로 볼 수 있다. 프로그래머의 매뉴얼로 생각할 수 있는데, 어셈블리 언어 프로그래머, 컴파일러 작성자, 응용 프로그래머가 볼 수 있는 컴퓨터의 부분이다. ISA는 지원하는 데이터 유형, 레지스터, 하드웨어가 주 메모리를 관리하는 방식, 가상 메모리와 같은 주요 기

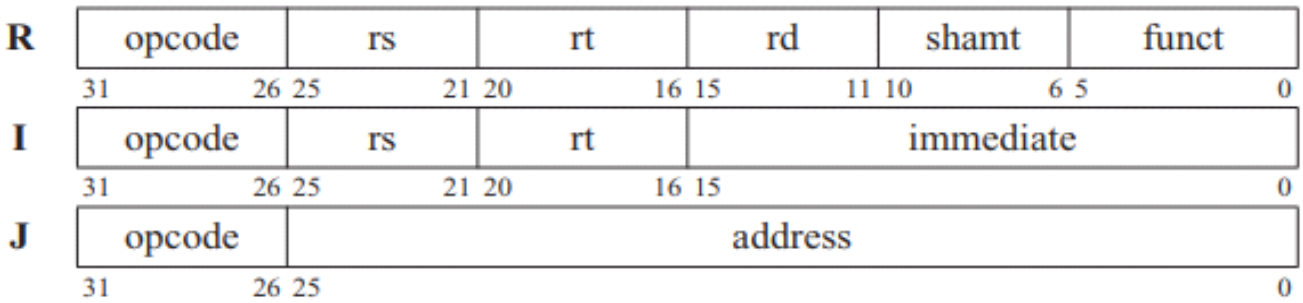
능, 마이크로프로세서가 실행할 명령어, 다중 ISA 구현의 입출력 모델 등을 정의한다. ISA는 명령어나 다른 기능을 추가함으로써 확장할 수 있으며, 더 큰 주소와 데이터 값의 지원을 위해 추가 지원도 가능하다. 각 CPU는 자체적인 ISA를 가지고 있다. 예를 들어, AMD는 AMD64와 ARM을 사용하고, Intel은 x86을 사용한다. 이 프로젝트에서는 MIPS ISA를 사용하며, 이는 MIPS Technologies에 의해 개발된 것이다.

3-4. MIPS (Microprocessor without Interlocked Pipeline Stages)

MIPS (Microprocessor without Interlocked Pipeline Stages)는 MIPS Technologies에서 개발된 RISC (Reduced Instruction Set Computer) ISA이다. MIPS ISA는 RISC 유형의 ISA이며, 모든 명령어의 길이가 32비트로 설정되어 있다. CPU가 다른 작업을 처리하기 위해 필요한 다른 정보에 따라 작업 유형을 구별하기 위해 여러 형식이 필요하다. 따라서 MIPS-32 ISA에서는 명령어를 R-Type, I-Type 및 J-Type 세 가지 유형으로 구별한다.

R-Type 명령어는 레지스터 간의 산술 및 논리 연산을 수행하는 데 사용된다. 이 유형의 명령어는 연산에 필요한 레지스터들의 번호와 명령어 코드를 포함한다. I-Type 명령어는 레지스터와 즉시 값(Immediate value)을 사용하여 연산을 수행하는 데 사용된다. 주로 데이터 전송, 분기 및 상수 연산과 관련된 작업에 사용된다. J-Type 명령어는 점프 및 분기 명령어로 사용된다. 이 유형의 명령어는 목적지 주소를 나타내는 점프 대상 필드를 포함한다.

MIPS ISA의 이러한 형식을 사용하여 다양한 유형의 명령어를 구별하고, 해당 명령어를 처리하는 데 필요한 정보를 명확하게 지정할 수 있다.

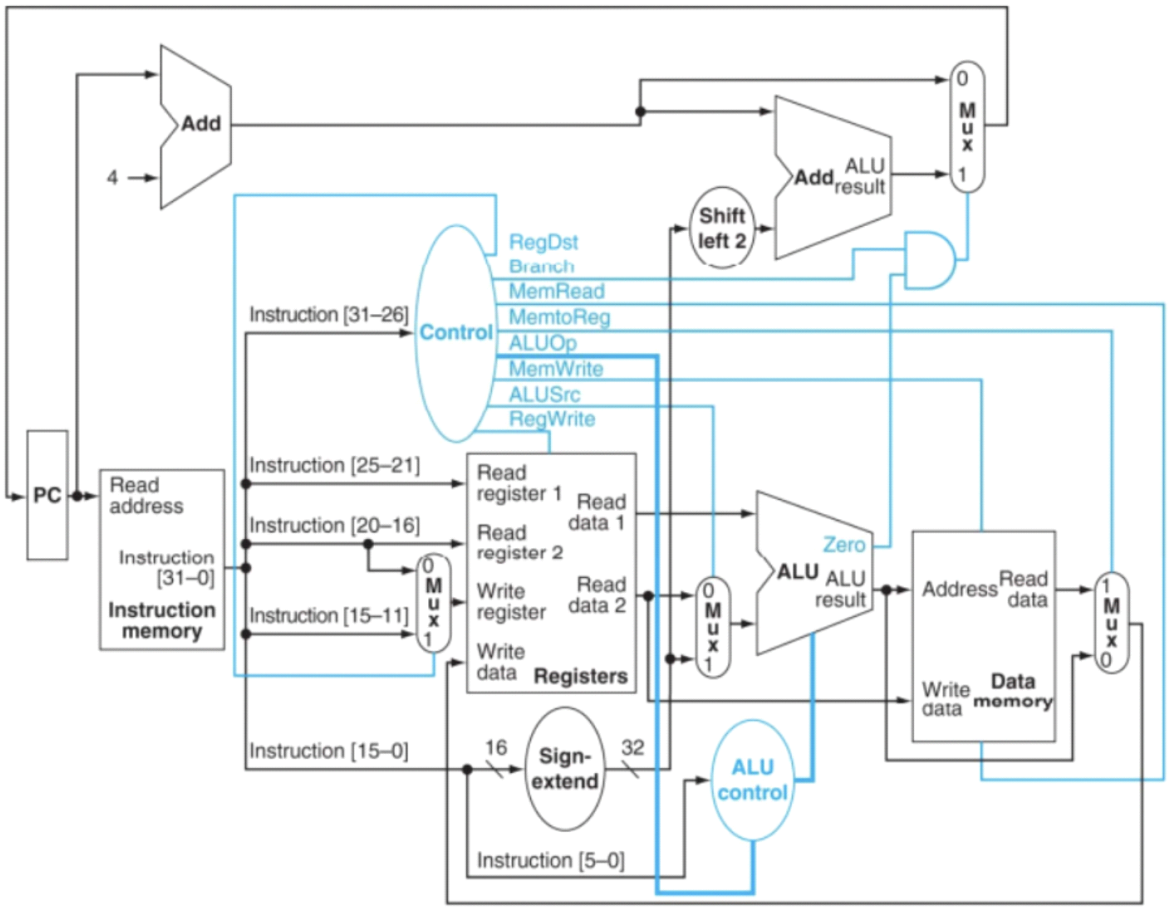


3-5. Datapath, control signal

데이터 패스(Data Path)는 컴퓨터 시스템에서 데이터 처리와 관련된 부분을 의미한다. 일반적으로 중앙 처리 장치(CPU) 내부의 하드웨어 구성 요소들을 가리키는 용어로 사용된다. 데이터 패스는 연산 유닛, 레지스터, 데이터 버스, 제어 신호 등으로 구성된다. 데이터 패스는 프로세서의 명령어 실행 단계에서 데이터를 처리하고 조작하는 역할을 담당한다. 데이터 패스는 주로 중

앙 처리 장치의 핵심 부분이며, 명령어를 해독하고 실행하는 역할을 수행한다. 데이터 패스의 구성은 프로세서의 설계에 따라 달라질 수 있으며, 다양한 프로세서 아키텍처에서 사용된다.

Control Logic이란 Control signal을 결정하는 하드웨어 구성 요소를 말한다. Control signal은 control unit을 관리해서 instruction에 따라서 적절하게 돌아가게끔 해주는 신호를 말하는데, Control signal은 하드웨어의 특정 신호로 들어가 해당 작업을 제어하는 역할을 담당한다. 이를 바탕으로 전체적인 Basic Datapath를 그리자면 다음과 같다.



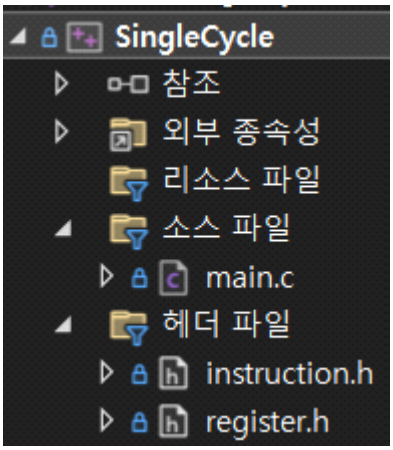
기본적인 구조는 Fetch, Decode, Execute, Memory, Write 실행 과정과 같은 구조로 되어 있다. Instruction fetch는 메모리에서 PC에 해당하는 명령어를 가져오는 단계이며, Instruction decode는 명령어를 ISA에 맞게 해석하고, 레지스터에서 적절한 값을 읽는다. Execute는 실행 단계로 이전 단계의 해석 값을 인자로 연산한다. Memory는 메모리를 읽거나 값을 쓰는 단계이며, Write back은 연산 결과를 레지스터에 저장하는 단계이다. 해당 작업에서 필요한 하드웨어를 가지고 동작한다. Fetch 단계 이후 Decode 단계에서 나온 Opcode로 Control Signal이 결정되고, 이 신호를 통해서 하드웨어의 작업을 제어한다. 예를 들어, MUX에 붙어 여러 개의 Input 값에 따른 Output 값 하나만 출력하게 하거나, Execution 단계의 ALU와 연결돼 해당 명령어에 맞는 연산을 수행하게 한다. (ALU Control) 또한, Memory Access 단계의 메모리에 연결돼 메모리값을 읽거나, 메모리값에 쓰는 연산을 할 수 있게끔 제어한다. (MemRead, MemWrite) 이렇게 제어가 필요한 이유는 CPU 안의 하드웨어 개수는 제한되어있고 Single

Cycle의 경우 한 clock에 모든 명령어가 다 실행될 수 있어야 하는 구조로 되어 있기 때문이다.

4. Implementation

코드 설명에 들어가기에 앞서, 본 프로그램은 C언어로 코드를 짰다. C언어를 사용한 이유는 물론 주언어이기 때문에 사용하였지만, 효율성 때문에 C언어를 선택하기도 했다. C언어는 저수준 언어로, 하드웨어와 밀접한 관련이 있다. 따라서 C언어로 작성된 코드는 일반적으로 효율적인 실행을 할 수 있다. 이는 싱글 사이클을 효율적으로 구현하고 빠른 실행 속도를 달성할 수 있는 장점이 될 수 있다. 지금부터 전체적인 구조와 함수 각각의 기능을 알아보도록 하겠다.

4-1. Program structure, global variables, functions



좌측 그림은 작성한 프로그램의 구조이다. instruction.h 파일은 R타입의 함수 코드, I 타입과 J 타입은 Opcode 코드를 정의했다. register.h는 레지스터 이름에 따른 레지스터 번호를 정의한 헤더 파일이다. 메인 파일은 Single Cycle을 구현하기 위한 전역 변수와 함수들을 선언하고 프로그램을 실행하는 파일이다. 메인 파일은 명령어 페치를 위한 전역 변수 및 컨트롤 유닛에 사용할 전역 변수, 레지스터와 메모리 엑세스를 위한 전역 변수 등 다양한 전역 변수와 명령어 수를 카운트하는 전역 변수, 싱글 사이클에서 각 단계를 수행할 수 있는 함수들이 선언되었다.

```
11  #define U_SIZE sizeof(unsigned int)
12
13  // instruction
14  unsigned int opcode, rs, rt, rd, shamt, funct, immediate, address;
15  unsigned int inst; // 페치 단계에서 명령어를 읽어올 때 쓰일 변수
16
17  // control unit variables
18  unsigned int RegDst, RegWrite, ALUSrc, PCSrc, MemRead, MemWrite, MemtoReg, Jump, Branch;
19
20  // register & memory global variables
21  unsigned int Memory[0x4000000] = { 0, };
22  unsigned int Register[32]      = { 0, };
23  unsigned int PC                = 0;
24  int change_pc_val              = 0;
25  unsigned int temp1, temp2      = 0;
26
27  // instruction count variables
28  int cycle_count  = 0;
29  int R_count      = 0;
30  int I_count      = 0;
31  int J_count      = 0;
32  int MemAcc_count = 0;
33  int branch_count = 0;
```

11 : 메모리를 4바이트씩 끊을 때 사용할 메크로이다.

15 : 페치 단계에서 명령어 한 줄을 읽어올 때 쓰이는 변수이다.

18 : 컨트롤 유닛에 사용될 변수다.

24 : 명령어 분기를 나타내기 위한 명령어다. Jump 한다면 1, 아니면 0을 저장한다.

25 : 메모리 액세스 연산이나 계산 결과를 임시로 저장하는 용도로 사용된다.

28 - 33 : 명령어들이 얼마나 사용됐는지 카운트에 사용되는 전역 변수다.

```
36      void init();
37      void read_mem(FILE* file);
38      int fetch(FILE* file);
39      int decode();
40      int Control_Signal();
41      int jump_Addr();
42      int Sign_Extend(int immediate);
43      int branch_Addr(int immediate);
44      int R_Type(int funct);
45      int I_Type(int opcode);
46      int J_Type(int opcode);
47      int Exe_and_WB();
```

36 : RA와 SP를 초기화할 때 사용된다.

37 : 파일에서 명령어를 읽어올 때 사용된다.

38 : 읽어온 명령어를 페치할 때 사용된다.

39 : 디코드할 때 사용된다.

40 - 46 : 명령어를 실행할 때 사용된다.

47 : 메모리 저장 및 레지스터에 저장할 때 사용된다.

4-2. Main

다음 코드는 주어진 파일(“input01.bin”)에서 명령어를 읽어 실행하는 메인 함수이다. 명령어가 끝날 때까지 무한 루프로 설정되어 있다. 무한 루프 안에는 각각의 함수를 호출하여 명령어 단계를 실행한다. 프로그램이 종료되면 반환 값, 사이클 수를 표시한다.

```
49  int main() {
50      FILE* file;
51      int ret = 0;
52
53      char* filename;
54      filename = "input01.bin";
55
56      file = fopen(filename, "r");
57      if (file == NULL) {
58          printf("Error: There is no file");
59          return 0;
60      }
61
62      read_mem(file);
63      fclose(file);
64
65      // initiallize
66      init();
67
68      // Run program
69      while (1) {
70          ret = fetch(file);
71          if (ret <= 0) break;
72
73          ret = decode();
74          if (ret <= 0) break;
75
76          ret = Control_Signal();
77          if (ret <= 0) break;
78
79          ret = Exe_and_WB();
80          if (ret <= 0) break;
81
82          cycle_count++;
83          printf("\n\n");
84      }
85
86      printf("================================================================\n");
87      printf("Return value (r2)                  : %d\n", Register[v0]);
88      printf("Total cycle                        : %d\n", cycle_count);
89      printf("Executed 'R' instruction           : %d\n", R_count);
90      printf("Executed 'I' instruction           : %d\n", I_count);
91      printf("Executed 'J' instruction           : %d\n", J_count);
92      printf("Number of branch taken             : %d\n", branch_count);
93      printf("Number of memory access instructions : %d\n", MemAcc_count);
94      printf("================================================================\n");
95
96      return 0;
97  }
```

53 - 63 : 파일 오픈 및 오류 처리, 메모리 읽기

66 : 초기화

69 - 84 : 프로그램 실행(F-D-E-M-W)

86 - 94 : 결과 출력

4-3. Init, Read_mem

초기화 작업을 수행하는 init 함수와 파일에서 데이터를 읽어와 메모리에 저장하는 read_mem 함수를 포함하고 있다. init 함수는 RA 레지스터와 스택 포인터를 초기화하고, read_mem 함수는 파일로부터 데이터를 읽어와서 메모리에 저장한다.

```
100  void init() {
101      Register[ra] = 0xFFFFFFFF;
102      Register[sp] = 0x10000000;
103  }
```

101 : Return Address를 0xFFFFFF으로 초기화

102 : Stack Pointer를 0x10000000으로 초기화

```
105  void read_mem(FILE* file) {
106      unsigned int ret = 0;
107      int i = 0;
108      int memVal;
109
110      while (true) {
111          int data    = 0;                                // 파일을 읽어오는 용도로 사용할 변수
112          ret         = fread(&data, 1, sizeof(int), file);   // data 변수에 파일로부터 4바이트 만큼 데이터를 읽어옴. ret 변수에는 fread 함수의 읽은 바이트 수가 저장됨
113          memVal      = ((data & 0xFF000000) >> 24) |
114                        ((data & 0x00FF0000) >> 8)  |
115                        ((data & 0x0000FF00) << 8)  |
116                        ((data & 0x000000FF) << 24);
117          Memory[i++] = memVal;                           // Memory에 값을 저장
118
119          if (ret != 4) break;                            // 4바이트를 읽지 못했다면 루프 탈출
120      }
121  }
```

112 : file에서 4바이트 데이터를 읽어온 후, ret에 fread()가 읽은 바이트 수 저장.

113 - 116 : data 값을 비트 연산을 사용하여 엔디안 변환을 수행한 후 memVal 변수에 저장.

117 : Memory 배열에 memVal 값을 저장.

119 : ret 변수가 4가 아니라면(즉, 4바이트를 읽지 못했다면) 루프를 탈출.

4-4. Fetch

명령어를 가져오는 작업을 수행한다. 주어진 파일로부터 명령어를 가져와서 inst 변수에 저장하고, 필요한 출력을 수행한 후, 반환 값을 결정하여 반환한다. 또한, nop 명령어인 경우에는 다음 명령어를 가져오기 위해 재귀적으로 fetch 함수를 호출한다.

```
123  int fetch(FILE* file) {
124      int ret = 1;
125      if (PC == 0xFFFFFFFF) return 0;                     // PC가 0xFFFFFFFF이면 종료
126      inst = Memory[PC / 4];
127
128      printf("Cycle[%03d](PC : 0x%08X)==========================================================\n\n", cycle_count + 1, PC);
129      if (inst == 0) {
130          PC += 4;
131          cycle_count++;
132          return fetch(file);  // nop 명령어일 경우 다시 fetch 호출
133      }
134      printf("[Fetch Inxtruction] %08X\n", inst);
135      return ret;
136  }
```

125 : PC가 0xFFFFFFFF인 경우 함수를 종료.

126 : Memory에서 명령어를 가져옴. Memory[PC / 4]는 PC를 4로 나눈 위치에 저장된 명령어를 가져오는 것을 의미한다.

129 - 133 : 가져온 명령어가 nop 이라면, PC를 4만큼, cycle_count를 1만큼 증가시키고 fetch 함수를 재귀로 호출하여 다음 명령어를 가져옴.

135 : 함수가 정상적으로 실행되었을 경우 ret 반환.

4-5. Decode

명령어를 해석하여 해당하는 유형을 판별하고 출력을 수행한다. 각각의 명령어 유형에 따라서 필요한 정보를 출력하고, 해당하는 카운트 변수를 증가시킨다. 마지막으로, 함수가 정상적으로 실행되었을 경우 ret 값을 반환한다.

```
138 int decode() {
139     int ret = 1;
140 
141     opcode    = (inst & 0xFC000000);                    // 비트 연산을 이용해 명령어 추출
142     opcode    = (opcode >> 26) & 0x3F;
143     rs        = (inst & 0x03E00000);
144     rs        = (rs >> 21) & 0x1F;
145     rt        = (inst & 0x001F0000);
146     rt        = (rt >> 16) & 0x1F;
147     rd        = (inst & 0x0000F800);
148     rd        = (rd >> 11) & 0x1F;
149     shamt     = (inst & 0x000007C0);
150     shamt     = shamt >> 6;
151     funct     = (inst & 0x0000003F);
152     immediate = (inst & 0x0000FFFF);
153     address   = (inst & 0x03FFFFFF);
154 
155     switch (opcode) {                           // opcode로 명령어 유형을 판별
156     case 0:                                     // R타입
157         printf("[Decode Instruction] type : R₩n");
158         printf("₩tOPCODE : 0x%X, RS : 0x%X (R[%d] = 0x%X), RT : 0x%X (R[%d] = 0x%X), RD : 0x%X (R[%d] = 0x%X), SHAMT : 0x%X, FUNCT : 0x%X₩n", opcode, rs, rs, Register[rs], rt, rt, Register[rt],
159         R_count++;
160         break;
161 
162     case J:
163         printf("[Decode Instruction] type : J₩n");
164         printf("₩tOPCODE : 0x%X, ADRESS : 0x%X₩n", opcode, address);
165         J_count++;
166         break;
167 
168     case JAL:
169         printf("[Decode Instruction] type : J₩n");
170         printf("₩tOPCODE : 0x%X, ADRESS : 0x%X₩n", opcode, address);
171         J_count++;
172         break;
173 
174     default:
175         printf("[Decode Instruction] type : I₩n");
176         printf("₩tOPCODE : 0x%X, RS : 0x%X (R[%d] = 0x%X), RT : 0x%X (R[%d] = 0x%X), IMM : 0x%X₩n", opcode, rs, rs, Register[rs], rt, rt, Register[rt], immediate);
177         I_count++;
178         break;
179     }
180 
181     return ret;
182 }
```

141 - 153 : 비트 연산을 이용하여 명령어에서 필요한 비트를 추출하여 변수에 저장.

155 : switch 문을 사용하여 opcode를 기반으로 명령어 유형을 판별.

156 - 160 : R 타입 명령어 수행

162 - 172 : J/JAL 타입 명령어 수행

175 - 178 : I 타입 명령어 수행

181 : 함수가 정상적으로 실행되었을 경우, ret 값을 반환한다.

4-6. Control_Signal

명령어의 opcode를 기반으로 제어 신호를 설정한다. 각 제어 신호는 해당하는 명령어 유형에 따라 설정되며, 설정된 값을 반환한다.

```
184 int Control_Signal() {
185     int ret = 1;
186 
187     // MUX
188     if (opcode == 0) {
189         RegDst = 1;
190     }
191     else {
192         RegDst = 0;
193     }
194 
195     if ((opcode == J) || (opcode == JAL)) {
196         Jump = 1;
197     }
198     else {
199         Jump = 0;
200     }
201 
```

```
202     if ((opcode == BEQ) || (opcode == BNE)) {
203         Branch = 1;
204     }
205     else {
206         Branch = 0;
207     }
208
209     if (opcode == LW) {
210         MemtoReg = 1;
211         MemRead = 1;
212     }
213     else {
214         MemtoReg = 0;
215         MemRead = 0;
216     }
217
218     if (opcode == SW) {
219         MemWrite = 1;
220     }
221     else {
222         MemWrite = 0;
223     }
224
225     if ((opcode != 0) && (opcode != BEQ) && (opcode != BNE)) {
226         ALUSrc = 1;
227     }
228     else {
229         ALUSrc = 0;
230     }
231
232
233     if ((opcode != SW) && (opcode != BEQ) && (opcode != BNE) && (opcode != J) && (opcode != JR)) {
234         RegWrite = 1;
235     }
236     else {
237         RegWrite = 0;
238     }
239
240     return ret;
241 }
```

188 - 193 : [RegDst] opcode가 0일 경우, RegDst를 1로 설정하고, 그 외의 경우에는 0으로 설정한다.

195 - 200 : [Jump] opcode가 J 또는 JAL인 경우, Jump를 1로 설정하고, 그 외의 경우에는 0으로 설정한다.

202 - 207 : [Branch] opcode가 BEQ 또는 BNE인 경우, Branch를 1로 설정하고, 그 외의 경우에는 0으로 설정한다.

209 - 216 : [MemtoReg와 MemRead] opcode가 LW인 경우, MemtoReg와 MemRead를 1로 설정하고, 그 외의 경우에는 0으로 설정한다.

218 - 223 : [MemWrite] opcode가 SW인 경우, MemWrite를 1로 설정하고, 그 외의 경우에는 0으로 설정한다.

225 - 230 : [ALUSrc] opcode가 SW, BEQ, BNE, J, JR이 아닌 경우, ALUSrc를 1로 설정하고, 그 외의 경우에는 0으로 설정한다.

233 - 238 : [RegWrite] opcode가 SW, BEQ, BNE, J, JR이 아닌 경우, RegWrite를 1로 설정하고, 그 외의 경우에는 0으로 설정한다.

240 : 함수가 정상적으로 실행되었을 경우, ret 값을 반환.

4-7. Sign_Extend

16비트 immediate 값을 부호 확장하여 32비트로 확장하는 작업을 수행하는 함수이다. 부호 확장은 immediate의 최상위 비트를 확인하여 1이면 상위 16비트를 1로 채우고, 0이면 하위 16비트를 그대로 유지한다. 최종적으로 부호 확장된 값을 반환한다.

```
243 int Sign_Extend(int immediate) {
244     int SignExtImm;
245     int SignBit = immediate >> 15;
246     if (SignBit == 1) {
247         SignExtImm = 0xFFFF0000 | immediate;
248     }
249     else {
250         SignExtImm = 0x0000FFFF & immediate;
251     }
252     return SignExtImm;
253 }
```

245 : SignBit 변수는 immediate의 최상위 비트인 15번째 비트를 추출한다.
246 - 248 : SignBit 값이 1이면, immediate는 음수이므로 0xFFFF0000와 비트 OR 연산을 수행하여 상위 16비트를 1로 채운다.
249 - 251 : SignBit 값이 0이면, immediate는 양수이므로 0x0000FFFF와 비트 AND 연산을 수행하여 하위 16비트를 그대로 유지한다.
252 : 함수가 정상적으로 실행되었을 경우, SignExtImm 값을 반환한다.

4-8. Jump_Addr, Branch_Addr

주어진 주소나 immediate 값을 사용하여 점프 주소나 분기 주소를 계산하는 함수이다. 각 함수는 주어진 주소나 immediate 값을 조작하여 최종 주소를 계산하고, 그 값을 반환한다.

```
255  int jump_Addr() {
256      int j = ((PC + 4) & 0xF0000000) | (address << 2);
257      return j;
258  }
259
260  int branch_Addr(int immediate) {
261      int b = Sign_Extend(immediate) << 2;
262      return b;
263  }
```

256 : (PC + 4) & 0xF0000000는 현재 주소의 상위 4비트를 유지하고, 하위 28비트를 0으로 만든다. address << 2는 점프 명령어의 하위 26비트를 왼쪽으로 2비트 시프트하여 계산된 주소를 가져온다. 두 값을 비트 OR 연산하여 최종 점프 주소를 계산한다.
257 : 함수가 정상적으로 실행되었을 경우, j 값을 반환한다.
261 : Sign_Extend(immediate)는 Sign_Extend 함수를 호출하여 immediate 값을 부호 확장한 값을 가져온다. Sign_Extend(immediate) << 2는 부호 확장된 immediate 값을 왼쪽으로 2비트 시프트하여 계산된 분기 주소를 가져온다.
262 : 함수가 정상적으로 실행되었을 경우, b 값을 반환한다.

4-9. R_Type

이 함수는 R 타입의 명령어를 실행하는 역할을 한다. funct 매개변수에 따라서 해당하는 R 타입의 연산을 수행하고, 레지스터에 결과를 저장하거나 PC 값을 변경한다.

```
266  int R_Type(int funct) {
267      switch (funct) {
268      case ADD:
269          Register[rd] = (int)((int)Register[rs] + (int)Register[rt]);
270          printf("[Write Back] R[%d] <- 0x%X\n", rd, Register[rd]);
271          break;
272
273      case ADDU:
274          Register[rd] = Register[rs] + Register[rt];
275          printf("[Write Back] R[%d] <- 0x%X\n", rd, Register[rd]);
276          break;
277
278      case AND:
279          Register[rd] = Register[rs] & Register[rt];
280          printf("[Write Back] R[%d] <- 0x%X\n", rd, Register[rd]);
281          break;
282
283      case JR:
284          PC = Register[rs];
285          change_pc_val = 1;
286          printf("[Write Back] PC <- 0x%X\n", PC);
287          break;
288
289      case JALR:
290          Register[rd] = PC + 4;
291          PC = Register[rs];
292          change_pc_val = 1;
293          printf("[Write Back] PC <- 0x%X\n", PC);
294          break;
```

```
        case NOR:
            Register[rd] = ~(Register[rs] | Register[rt]);
            printf("[Write Back] R[%d] <- 0x%X\n", rd, Register[rd]);
            break;

        case OR:
            Register[rd] = Register[rs] | Register[rt];
            printf("[Write Back] R[%d] <- 0x%X\n", rd, Register[rd]);
            break;

        case SLT:
            Register[rd] = ((int)Register[rs] < (int)Register[rt]) ? 1 : 0;
            printf("[Write Back] R[%d] <- 0x%X\n", rd, Register[rd]);
            break;

        case SLTU:
            Register[rd] = (Register[rs] < Register[rt]) ? 1 : 0;
            printf("[Write Back] R[%d] <- 0x%X\n", rd, Register[rd]);
            break;

        case SLL:
            Register[rd] = Register[rt] << shamt;
            printf("[Write Back] R[%d] <- 0x%X\n", rd, Register[rd]);
            break;

        case SRL:
            Register[rd] = Register[rt] >> shamt;
            printf("[Write Back] R[%d] <- 0x%X\n", rd, Register[rd]);
            break;

        case SUB:
            Register[rd] = (int)((int)Register[rs] - (int)Register[rt]);
            printf("[Write Back] R[%d] <- 0x%X\n", rd, Register[rd]);
            break;

        case SUBU:
            Register[rd] = Register[rs] - Register[rt];
            printf("[Write Back] R[%d] <- 0x%X\n", rd, Register[rd]);
            break;

        default:
            return -1;
        }

        return change_pc_val;
    }
```

위의 프로그램 코드를 switch 문, 명령어 유형별 동작, PC 값 변경, 반환 값으로 나눠 설명하도록 하겠다.

switch 문 : funct 값을 기반으로 다양한 경우의 수를 처리하기 위해 switch 문을 사용한다. funct 값을 확인하여 해당하는 명령어 유형을 식별한다.

명령어 유형별 동작: 각 case 문은 해당하는 명령어 유형에 따른 동작을 수행한다. 예를 들어, **case ADD** : 는 덧셈 명령어를 처리한다. 명령어 유형에 따라 레지스터 연산이 수행되고, 그 결과가 Register[rd]에 저장된다. 결과를 출력하고 break 문을 통해 switch 문을 종료한다.

PC 값 변경 : JR 명령어와 JALR 명령어는 PC 값을 변경한다. JR 명령어는 Register[rs] 값을 새로운 PC 값으로 설정한다. JALR 명령어는 Register[rs] 값을 새로운 PC 값으로 설정하고, Register[rd]에 현재 PC + 4 값을 저장한다.

반환 값 : change_pc_val 변수는 PC 값이 변경되었는지를 나타내는 플래그이다. change_pc_val 값이 1이면 PC가 변경되었음을 의미하고, 0이면 변경되지 않았음을 의미한다. 함수가 정상적으로 실행되었을 경우, change_pc_val 값을 반환한다.

4-10. J_Type

이 함수는 J 타입의 명령어를 실행하는 역할을 한다. opcode 매개변수에 따라 해당하는 J 타입의 명령어를 처리하고, PC 값을 변경한다.

```
int J_Type(int opcode) {
    unsigned int JumpAddr = jump_Addr();
    switch (opcode) {
    case J:
        PC = JumpAddr;
        change_pc_val = 1;
        printf("[Jump] PC = 0x%X", PC);
        break;
```

```
350         }
351     case JAL:
352         Register[ra] = (unsigned int)PC + 8;
353         PC = JumpAddr;
354         change_pc_val = 1;
355         printf("[Jump] PC = 0x%X", PC);
356         break;
357 
358     default:
359         return -1;
360     }
361 
362     return change_pc_val;
363 }
```

switch 문 : opcode 값을 기반으로 다양한 경우의 수를 처리하기 위해 switch 문을 사용한다. opcode 값을 확인하여 해당하는 명령어 유형을 식별한다.

명령어 유형별 동작 : 각 case 문은 해당하는 명령어 유형에 따른 동작을 수행한다. 예를 들어, case J:는 점프 명령어를 처리한다. 명령어 유형에 따라 PC 값을 변경한다. PC 변수에 JumpAddr 값을 저장하고, change_pc_val 변수를 1로 설정하여 PC가 변경되었음을 나타낸다.

JAL 명령어 : JAL 명령어의 경우 Register[ra]에 현재 PC + 8 값을 저장한다. 그 후 PC 변수에 JumpAddr 값을 저장하고, change_pc_val 변수를 1로 설정하여 PC가 변경되었음을 나타낸다.

반환 값 : change_pc_val 변수는 PC 값이 변경되었는지를 나타내는 플래그이다. change_pc_val 값이 1이면 PC가 변경되었음을 의미하고, 0이면 변경되지 않았음을 의미한다. 함수가 정상적으로 실행되었을 경우, change_pc_val 값을 반환한다.

4-11. I_Type

I 타입의 명령어를 처리하여 레지스터나 메모리에 대한 동작을 수행한다. 해당하는 명령어에 따라 레지스터나 메모리 값을 변경하거나 로드하고, 결과를 출력한다.

```
365 int I_Type(int opcode) {
366     int ZeroExtImm = immediate;
367     int SignExtImm = Sign_Extend(immediate);
368     int BranchAddr = branch_Addr(immediate);
369     switch (opcode) {
370     case ADDI:
371         Register[rt] = (int)((int)Register[rs] + SignExtImm);
372         printf("[Write Back] R[%d] <- 0x%X", rt, Register[rt]);
373         break;
374 
375     case ADDIU:
376         Register[rt] = (unsigned int)Register[rs] + SignExtImm;
377         printf("[Write Back] R[%d] <- 0x%X", rt, Register[rt]);
378         break;
379 
380     case ANDI:
381         Register[rt] = Register[rs] & ZeroExtImm;
382         printf("[Write Back] R[%d] <- 0x%X", rt, Register[rt]);
383         break;
384 
385     case BEQ:
386         if (Register[rs] == Register[rt]) {
387             PC = PC + 4 + BranchAddr;
388             change_pc_val = 1;
389             branch_count++;
390         }
391 
392         printf("[Branch] PC : 0x%X", PC);
393         break;
394 
395     case BNE:
396         if (Register[rs] != Register[rt]) {
397             PC = PC + 4 + BranchAddr;
398             change_pc_val = 1;
399             branch_count++;
400         }
401 
402         printf("[Branch] PC : 0x%X", PC);
403         break;
404 
405     case LBU:
406         temp1 = Register[rs] + (unsigned int)SignExtImm & 0x000000FF;
407         Register[rt] = (Memory[temp1 / U_SIZE]) & 0x000000FF;
408         printf("[Load] R[%d] <- Mem[0x%X] = 0x%X", rt, temp1 / U_SIZE, Register[rt]);
409         break;
410 
411     case LHU:
412         temp1 = Register[rs] + (unsigned int)SignExtImm & 0x0000FFFF;
413         Register[rt] = (Memory[temp1 / U_SIZE]) & 0x0000FFFF;
414         printf("[Load] R[%d] <- Mem[0x%X] = 0x%X", rt, temp1 / U_SIZE, Register[rt]);
415         break;
```

```
    case LL:
        Register[rt] = Memory[(Register[rs] + (unsigned int)SignExtImm) / U_SIZE];
        printf("[Load] R[%d] <- Mem[0x%X] = 0x%X", rt, (Register[rs] + (unsigned int)SignExtImm) / U_SIZE, Register[rt]);
        break;

    case LUI:
        Register[rt] = immediate << 16;
        printf("[Load] R[%d] <- 0x%X", rt, Register[rt]);
        break;

    case LW:
        Register[rt] = Memory[(Register[rs] + (unsigned int)SignExtImm) / U_SIZE];
        MemAcc_count++;
        printf("[Load] R[%d] <- Mem[0x%X] = 0x%X", rt, (Register[rs] + (unsigned int)SignExtImm) / U_SIZE, Register[rt]);
        break;

    case ORI:
        Register[rt] = Register[rs] | ZeroExtImm;
        printf("[Write Back] R[%d] <- 0x%X", rt, Register[rt]);
        break;

    case SLTI:
        Register[rt] = (Register[rs] < SignExtImm) ? 1 : 0;
        printf("[Write Back] R[%d] <- 0x%X", rt, Register[rt]);
        break;

    case SLTIU:
        Register[rt] = (Register[rs] < (unsigned int)SignExtImm) ? 1 : 0;
        printf("[Write Back] R[%d] <- 0x%X", rt, Register[rt]);
        break;

    case SB:
        temp1 = Register[rs] + (unsigned int)SignExtImm;
        temp2 = Memory[temp1 / U_SIZE];
        temp2 = temp2 & 0xFFFFFF00;
        temp2 = (Register[rt] & 0x000000FF) | temp2;
        Memory[temp1 / U_SIZE] = temp2;
        printf("[Store] Mem[0x%X] <- 0x%X", temp1 / U_SIZE, temp2);
        break;

    case SH:
        temp1 = Register[rs] + (unsigned int)SignExtImm;
        temp2 = Memory[temp1 / U_SIZE];
        temp2 = temp2 & 0xFFFF0000;
        temp2 = (Register[rt] & 0x0000FFFF) | temp2;
        Memory[temp1 / U_SIZE] = temp2;
        printf("[Store] Mem[0x%X] <- 0x%X", temp1 / U_SIZE, temp2);
        break;

    case SW:
        temp1 = Register[rs] + (unsigned int)SignExtImm;
        Memory[temp1 / U_SIZE] = Register[rt];
        MemAcc_count++;
        printf("[Store] Mem[0x%X] <- R[%d] = 0x%X", temp1 / U_SIZE, rt, Register[rt]);
        break;

    default:
        return -1;
    }
    return change_pc_val;
}
```

switch 문 : opcode 값을 기반으로 다양한 경우의 수를 처리하기 위해 switch 문을 사용한다. opcode 값을 확인하여 해당하는 명령어 유형을 식별한다.

명령어 유형별 동작 : 각 case 문은 해당하는 명령어 유형에 따른 동작을 수행한다. 예를 들어, case ADDI:는 레지스터에 피연산자와 즉시값을 더하는 명령어를 처리한다. 명령어 유형에 따라 해당하는 동작을 수행한다.

결과 출력 : 각 명령어 유형에 따라 결과를 출력한다. 예를 들어, ADDI 명령어의 경우 결과값을 Register[rt]에 저장하고, 해당 결과를 출력한다.

반환 값 : change_pc_val 변수는 PC 값이 변경되었는지를 나타내는 플래그이다. change_pc_val 값이 1이면 PC가 변경되었음을 의미하고, 0이면 변경되지 않았음을 의미한다. 함수가 정상적으로 실행되었을 경우, change_pc_val 값을 반환한다.

4-12. Exe_and_WB

이 함수는 명령어 실행 및 Write Back 단계를 수행하는 역할을 한다. 함수 내부에서는 주어진 opcode 값에 따라 R 타입, J 타입 또는 I 타입의 명령어를 처리한다.

```c
int Exe_and_WB() {
    int ret = 1;
    change_pc_val = 0;

    if (opcode == 0x00) {
        change_pc_val = R_Type(funct);
    }
    else if (opcode == 0x02 || opcode == 0x03) {
        change_pc_val = J_Type(opcode);
    }
    else {
        change_pc_val = I_Type(opcode);
    }

    // PC Update
    if (change_pc_val == 0) {
        printf("[PC Update] PC <- 0x%08X = 0x%08X + 4\n", PC + 4, PC);
        PC = PC + 4;
    }
    else if (change_pc_val == -1) {
        ret = change_pc_val;
    }
    else if (change_pc_val == 1) {
        printf("[PC Update] (JUMP) PC <- 0x%08X\n", PC);
    }

    return ret;
}
```

변수 초기화 : change_pc_val 값을 0으로 초기화한다. 이 값은 PC 변경 여부를 나타내는 플래그이다. ret 변수를 1로 초기화한다. 이 변수는 함수의 반환 값을 나타낸다.

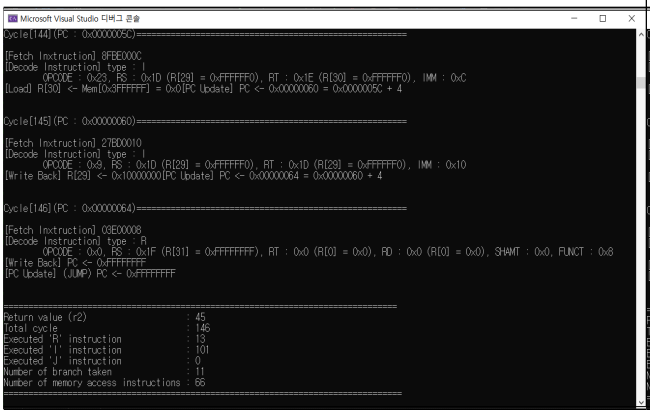
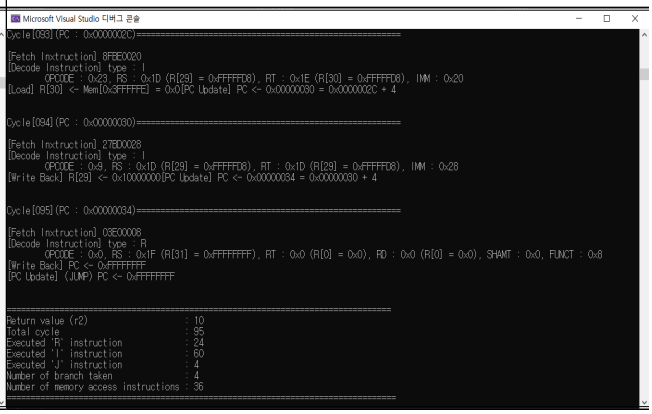
명령어 유형에 따른 처리 : opcode 값이 0x00인 경우, R 타입의 명령어로 간주하고 R_Type 함수를 호출한다. opcode 값이 0x02 또는 0x03인 경우, J 타입의 명령어로 간주하고 J_Type 함수를 호출한다. 그 외의 경우, I 타입의 명령어로 간주하고 I_Type 함수를 호출한다. 각각의 함수 호출은 해당하는 명령어 유형에 대한 처리를 수행하고, change_pc_val 값을 반환한다.

PC 업데이트 : change_pc_val 값에 따라 PC 값을 업데이트한다. change_pc_val 값이 0인 경우, PC 값을 현재 PC 값에 4를 더한 값으로 업데이트한다. 이는 다음 명령어를 가리키게 된다. change_pc_val 값이 -1인 경우, ret 값을 change_pc_val로 설정한다. 이는 잘못된 명령어나 오류가 발생한 경우를 나타낸다. change_pc_val 값이 1인 경우, PC 값이 이미 J 타입의 명령어에 의해 변경되었음을 의미하므로 PC 값을 변경하지 않고 출력한다.

반환 값 : ret 값을 반환한다. 이 값은 함수의 실행 결과를 나타낸다. 정상적으로 실행되었을 경우 1을 반환하며, 오류가 발생한 경우 -1을 반환한다.

5. Result

위에서 본 코드를 돌리면 어떻게 출력이 되는지 알아보도록 하겠다.

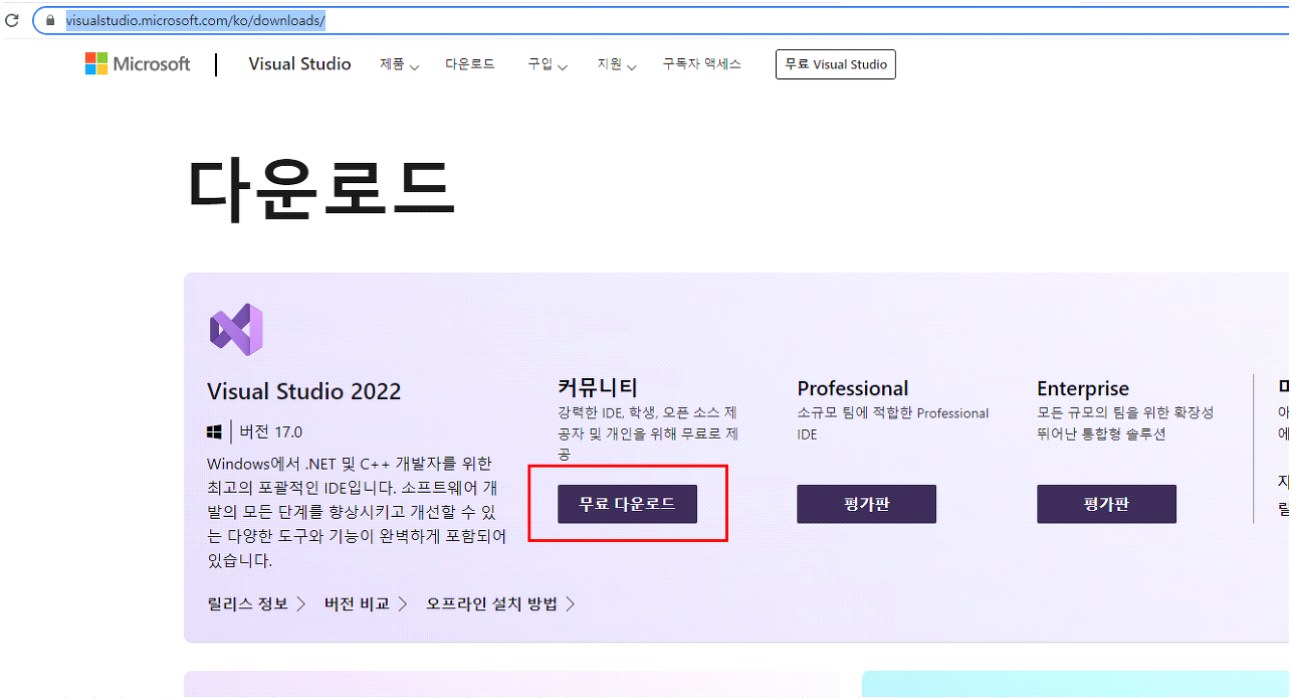
input01.bin	input02.bin
	

위의 그림과 같이 각각의 Cycle에서 수행하는 명령어를 모두 잘 출력하고 있으며 마지막 결과까지 정확히 출력하는 것을 볼 수 있다.

6. Build Environment

본 노트북의 운영체제는 윈도우 10이고, 시스템은 64비트 운영체제이다. 프로그램은 Visual Studio 2022를 사용하였으며, 설치방법은 다음과 같다.

1. 다운로드 사이트 접속 후 다운로드



2. 설치할 항목 선택 후 설치 진행 : 내려받은 파일을 실행하면 설치할 항목들이 나타난다. 그 중, 맞는 항목을 선택 후 설치한다.
3. 이후 설치가 완료되면, 로그인하면 된다. 로그인 계정을 바로 사용하지 않을 때는 나중에 로그인 항목을 선택한다.

다음은 Visual Studio에서 프로그램을 실행하는 방법에 관해서 설명하겠다.

1. 해당 압축 폴더를 압축을 풀고, Visual Studio를 열어서 [새프로젝트 만들기] - [빈프로젝트]를 클릭한다.
2. Windows에서 프로젝트 폴더를 열어 c 파일과 바이너리 파일을 넣어준다. 바이너리 파일과 c 파일이 같은 위치에 있어야 함을 유의한다.
3. 다시 생성한 프로젝트에 들어가 상단에 있는 [로컬 Windows 디버거]를 클릭하여 프로그램을 실행한다. 위와 같은 방법으로 파일을 실행하면, 디버그 콘솔에서 출력물을 볼 수 있다.

6. Lesson

이렇게 MIPS 아키텍처를 기반으로한 싱글 사이클 시뮬레이터를 구현하였다. 주어진 입력 파일을 읽고, 명령어를 실행하여 레지스터와 메모리의 값을 조작하며 프로그램을 실행한다. 이 프로그램은 R 타입, I 타입, J 타입의 명령어를 처리할 수 있다. 명령어 실행 도중에는 다양한 통제 신호와 MUX를 사용하여 제어 유닛을 구성한다. 프로그램의 실행 결과로는 레지스터의 최종 값을 출력하고, 실행한 명령어의 종류와 수, 분기 횟수, 메모리 접근 횟수 등의 정보를 표시한다.

과제를 할 때, simple calculator보다 훨씬 어려워서 코딩하는 시간보다 고민하고 알고리즘을 짜는 시간이 더 많았던 것 같다. 데이터 패스를 보고 직접 짜는 것이 너무 어려웠고, 특히 nop 처리나, branch cycle, jump를 처리할 때 많이 헤맸던 것 같다. 하지만 과제를 위해 공부하면서 수업시간에 이해하지 못했던 개념을 이해할 수 있게 되어서 좋았던 것 같다.