**Report : Project #2 Single-Cycle MIPS**

**컴퓨터공학과 32191597 박민규**

use 2/5 free days



**과목 : 컴퓨터구조론**

**담당교수 : 유시환 교수님**

Contents

1. About Project -------------------------------- 2p

1-1. 프로젝트 설명 및 요구사항

1-2. 기본 개념

2. About Code --------------------------------- 9p

2-1. 환경 설명

2-2. 코드 설명

3. About Sample Program -------------------- 19p

4. About Conclusion -------------------------- 25p

**1. About Project**

**1-1 프로젝트 설명 및 요구사항**

본 프로젝트는 간단한 ISA를 가진 MIPS 시뮬레이터 프로그램을 구현하는 프로젝트입니다. 이 프로그램은 input으로 들어온 실행 가능한 binary 파일의 각 명령어들을 해석하여 해당 소스코드의 결과와 동일한 결과를 얻는 것이 목표입니다. 이 시뮬레이터에 포함된 ISA는 MIPS Green Sheet의 CORE INSTRUCTION SET의 31개의 명령어 중 6개의 명령어(lbu, lhu, ll, sb, sc, sh)를 제외한 25개의 명령어와 JALR(Jump And Link using Register)입니다. 각각의 명령어는 Single-Cycle MIPS를 따라 IF(Instruction Fetch), ID(Instruction Decode), EXE(EXEcution), MEM(MEMory access), WB(Write Back)의 다섯 단계를 거치는데, 이것은 실제 CPU와 동일하게 작동해야 합니다. 본 프로젝트를 통해 컴퓨터 내부 동작과 Single Cycle에 대해 이해하는 것이 목표입니다.

프로젝트의 요구사항은 다음과 같습니다.

**1) 기본 요구사항**

- MIPS binary program(.bin파일)을 input(실행 파일의 인자)으로 합니다.

\* visual studio code의 경우 hex editor extension을 설치해 확인할 수 있습니다.

\* 필요에 따라 .bin파일을 적절히 수정해야 합니다.

ex) jal 명령어의 경우 컴파일 시 JumpAddr이 0으로 되어있는데 원하는 값으로 적절히 수정할 필요가 있습니다.

- 실제 CPU와 동일하게 구현해야 합니다.

- 레지스터 : R[0] ~ R[31]까지 존재하며 LR(R[31])과 SP(R[29])를 제외한 모든 레지스터는 0으로 초기화되어야 합니다. LR과 SP의 초기값은 각각 0xFFFF:FFFF, 0x100:0000입니다.

- 메모리 : 메모리는 16MB입니다. 단, Instruction memory와 data memory를 하나의 메모리로 구현합니다.

- PC : PC는 현재 명령어의 주소를 가지고 있는 레지스터로, 초기값은 프로그램의 시작주소인 0x0 입니다. 각 명령어에 대응하여 PC는 0x0 부터 4씩 증가하며 PC가 0xFFFF:FFFF일 때 프로그램이 종료됩니다.

- 각 사이클마다 아키텍쳐 상태의 변화가 생긴다면 user visible architectural state인 PC, GPR, Memory에 대한 상태를 출력해야 합니다.

\* 추가적으로, 한 사이클 내의 각 단계에서의 결과를 출력합니다.

- 명령어가 전부 실행되면, 다음과 같은 값들을 출력해야합니다.

- 최종 결과값 (R[2])

- 명령어 실행 횟수

- R-type 명령어 실행 횟수

- I-type 명령어 실행 횟수

- J-type 명령어 실행 횟수

- 메모리 액세스 명령어 실행 횟수

- branch 명령어 실행한 횟수

**2) Single-Cycle**

- 명령어 하나 당 하나의 사이클이 돌아야 합니다.

- OPCODE에 따라 Control Signals를 구한 후 MUX의 개념을 이용하여 Signal에 따라 각 명령어가 필요로 하는 단계에만 접근 가능하게 합니다.

- JALR (Jump And Link using Register) 명령어를 구현해야 합니다.

- rd에 리턴할 주소값을 저장하고 rs로 점프하는 R-type 명령어입니다.

- rd = 31, rs = 점프 타겟 레지스터, funct = 0x9, 나머지는 0입니다.

- JALR 명령어를 16진수로 하면 0x0?!0f809 입니다. 이때 '?'와 '!'는 미지수로 범위는 다음과 같습니다.

\*) 0 <= ? <= 3

\*) 0 <= ! <= E

**1-2 기본 개념**

**1) Single-Cycle에 대한 이해**

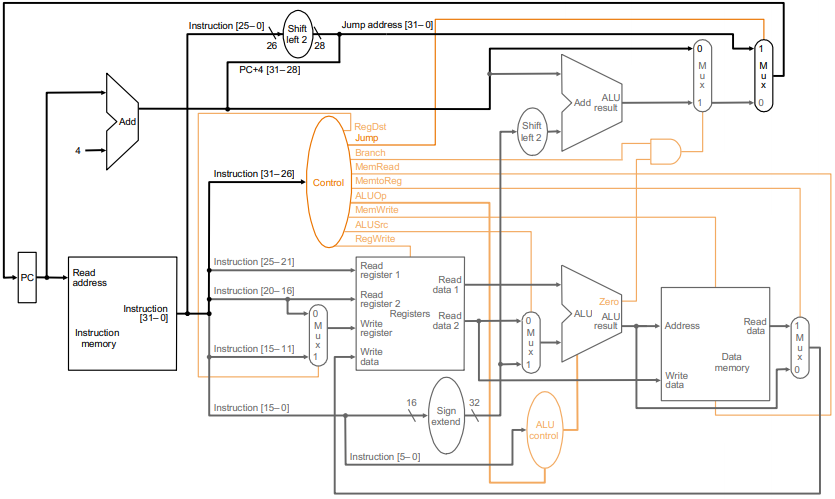
- 1 clock cycle에 하나의 명령어가 진행됩니다.

- 기본적으로 5단계로 진행됩니다. (IF, ID, EXE, MEM, WB)

- 각 명령어의 opcode에 따라 control signal의 값이 0 또는 1로 구성되며 control signal을 통해서 하드웨어를 통제하여 해당 명령어를 실행합니다.

- clock cycle time은 가장 시간이 오래 걸리는 명령어의 처리 시간보다 길어야 하므로 비효율적입니다.

- 하나의 장치(unit)이 연산을 수행하고 있을 때 다른 장치들은 사용하고 있지 않아 여러 개의 명령을 처리해야 할 경우 시간이 오래 걸립니다.



EXE

MEM

WB

ID

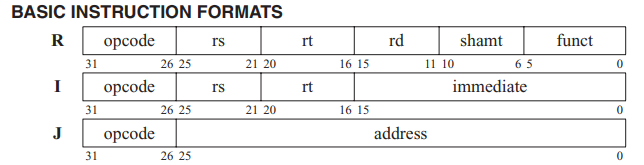
IF

<그림 1>

**1-1) Single-Cycle의 5단계**

- IF(Instruction Fetch) : PC를 이용하여 명령어가 들어있는 주소를 찾고 해당 명령어를 가져옵니다.

- ID(Instruction Decode) : 해당 명령어를 해석합니다. 각 타입에 대한 decode는 다음과 같습니다.



- EXE(EXEeuction) : 필요한 레지스터를 읽고 읽은 값으로 ALU 연산을 합니다.

- MEM(MEMory Acces) : 메모리에 접근하여 값을 저장하거나 가져옵니다. (LW, SW)

- WB(Write Back) : 연산 결과를 쓰기 레지스터에 저장합니다.

**1-2) Control Signal**

<그림 1>의 주황색 선 부분이 control signal입니다. 이 control signal을 통해서 해당하는 하드웨어를 통제합니다. 이때 control signal에는 Instruction[31-26]인 opcode에 의해 1 또는 0의 값이 저장됩니다. signal의 값에 따라 다음 <표 1>과 같습니다.

|  |  |  |
| --- | --- | --- |
| 0 or 1  Control Signal | De-asserted (=0) | Asserted (=1) |
| RegDst | Write Register  = rt | Write Register  = rd |
| MemRead | Memory  read disabled | Memory  read enabled |
| MemtoReg | Write data (WB)  = ALU result | Write data (WB)  = Read data in Memory |
| MemWrite | Memory  write disabled | Memory  write enabled |
| ALUSrc | ALU 2nd input  = Read data2 in Register | ALU 2nd input  = sign-extended imm |
| RegWrite | GPR  write disabled | GPR  write enabled |
| PCSrc1 (=Jump) | next PC  = PCSrc2 | next PC  = JumpAddr |
| PCSrc2 | PCSrc1 2nd input  = PC + 4 | PCSrc1 2nd input  = PC + 4 + BranchAddr |

<표 1>

**2) 코드 구현을 위한 개념**

**2-1) 메모리 사이즈**

1)의 기본 요구사항에 따르면, SP의 초기값은 0x100:0000입니다. 이 값은 16MB이므로 메모리 사이즈도 같게 해야 합니다. 하지만 코드 구현시 메모리를 int형 배열로 선언을 해야 합니다. int형은 4Byte이므로 int형 메모리 배열의 크기는 16 / 4 = 4MB입니다. 따라서 메모리 사이즈는 4MB로 설정해야 하며, MEM 단계와 같이 메모리에 접근을 할 때에도 4로 나누어 접근을 해야 합니다.

\* 1MB = 2^20B

16MB = 2^24B, 2^24 = 16^6 = 0x100:0000

4MB = 2^22B, 2^22 = 16^5 \* 4 = 0x40:0000

**2-2) 명령어 decode (bit 연산)**

32bit의 명령어를 decode하기 위해서는 opcode를 우선적으로 알아야 합니다. 그 opcode를 바탕으로 Format을 파악하여 원하는 값들을 얻어 각각 저장해야 합니다. 그러기 위해서 32bit 숫자에서 원하는 자리의 값들을 얻어올 수 있는 방법을 알아야 합니다. 바로 bit 연산을 이용할 것입니다.

1. left shift : 비트가 원하는 위치 수 만큼 왼쪽으로 이동합니다. 연산으로 비워진 비트 위치는 0으로 채워집니다.

2. right shift : 비트가 원하는 위치 수 만큼 오른쪽으로 이동합니다. 부호 없는 숫자의 경우 연산으로 비워진 비트 위치는 0으로 채워집니다. 부호 있는 숫자의 경우 부호 비트는 비워진 비트를 채우는 데 사용됩니다. 즉, 숫자가 양수이면 0, 음수이면 1이 채워집니다.

3. bit mask : AND 연산으로 필요한 부분만 뽑아낼 수 있습니다.

예시) I-type 명령어 decode

1. opcode : 명령어에서 필요한 부분인 [31:26]을 얻어오기 위해 0xfc000000과 AND 연산을 한 후 26bit만큼 right shift 해줍니다.

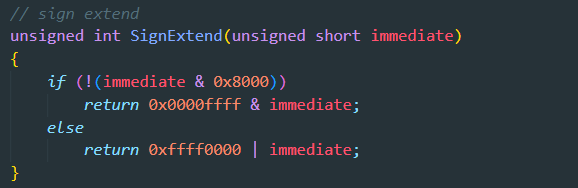
2. rs : 명령어에서 필요한 부분인 [25:21]를 얻어오기 위해 0x03e00000과 AND 연산을 한 후 21bit만큼 right shift 해줍니다.

3. rt : 명령어에서 필요한 부분인 [20:16]을 얻어오기 위해 0x001f0000과 AND 연산을 한 후 16bit만큼 right shift 해줍니다.

4. immediate : 명령어에서 필요한 부분인 [15:0]을 얻어오기 위해 0x0000ffff와 AND 연산을 해줍니다.

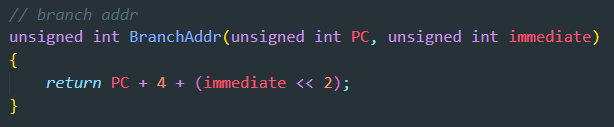
**2-3) SignExtImm, BranchAddr, JumpAddr 구현하기**

1. SignExtImm = { 16{immediate[15]}, immediate }

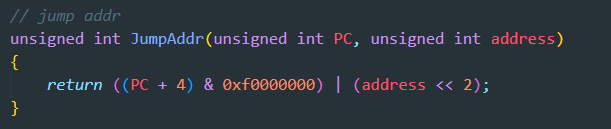
 명령어의 부호 비트가 0이라면 [31:16]을 0으로 채우고 1이라면 [31:16]을 1로 채워야 합니다.

2. BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }

명령어를 2만큼 left shift 해준 값에 PC + 4 를 더해줍니다.



3. JumpAddr = { PC+4[31:28], address, 2'b0 }

 주소를 2만큼 left shift 해준 후 PC+4의 마지막 4자리가 들어가므로 PC와 0xf000:0000을 AND 연산한 값에 OR연산을 해준 값을 return합니다..

**2. About Code**

**2-1 환경 설명**

사용 언어: C언어

개발 환경: OS : windows, Text editor : visual studio code

파일 정보: register.h, instruction.h, mips.h, ADDR.c, ALU.c CU.c, DM.c, IM.c, RF.c, main.c

컴파일 및 실행 방법:

1. 위의 파일들이 모두 있는 디렉토리에 위치합니다.

2. 컴파일을 다음과 같이 합니다.

-> gcc \*.c -o single\_cycle

3. 컴파일이 잘 되었다면 single\_cycle 실행 파일이 생성되었을 것입니다. ./single\_cycle에 원하는 binary 파일명을 입력하여 실행한 후, 결과를 확인합니다.

**2-2 코드 설명**

구현한 코드에서 코멘트가 필요하다고 생각되는 함수에 대해 설명합니다. 코드 설명 같은 경우 한글과 영어를 계속 왔다 갔다 하기에 어려움이 있기도 했고, 영어로 레포트를 쓰는 경험도 중요할 것 같아 영어로 작성 하였습니다.

**Control Unit (CU)**

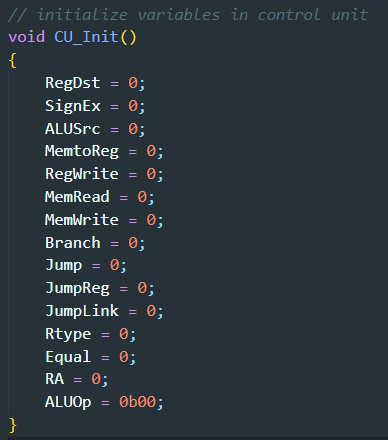


Figure 1 - CU\_Init (control unit init)

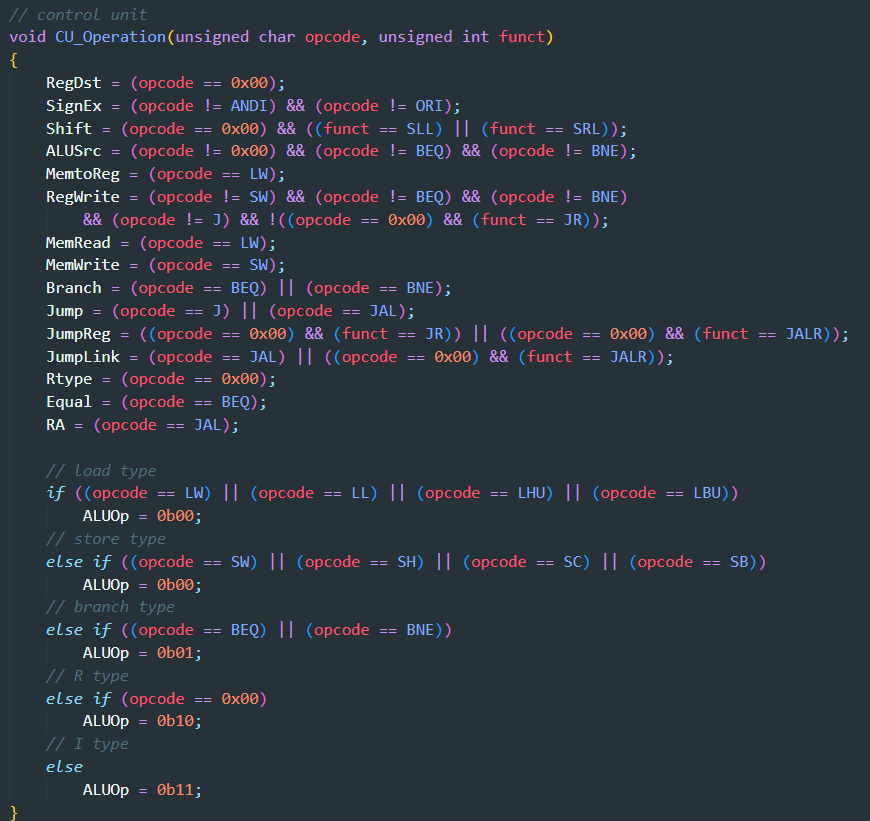


Figure 2 – CU Operation

Figure 1 shows the CU\_Init function, which initializes all the control bits to zero. Figure 2 shows the CUOperation function, which controls the control bits due to the given opcode or funct field. According to the control signal table in sections 4-2, we can implement the action of setting the bits like c codes in Figure 2. In the case of ALUOP, which is the ALU operation bit, because they need two bits to determine the ALU control bits, we used conditional statements to distinguish the needed ALU operation bit. By implementing the control unit components in this way, we can generate the control bits and set them in a proper situation.

**Instruction Memory (IM)**



Figure 3 - IM\_Init

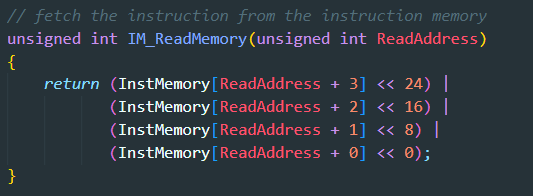


Figure 4 - IM\_ReadMemory

Figure 3 shows IM\_Init. This function reads the following binary program for every 4 bytes until it reaches the end of the file. Since the data in the binary file is stored in Little-Endian, we converted the byte order into Big-Endian form and stored them in the instruction memory per 1 byte. Figure 4 shows the IM\_ReadMemory function that returns the instruction stored in the following Read Address.

**Registers (RF)**

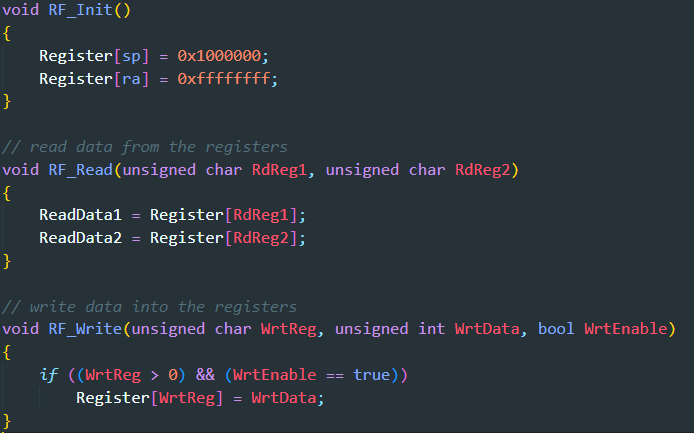


Figure 5 - RF Functions

Figure 5 shows the functions implemented in the RF module. RF\_Init function initializes the register r29 and r31 in the data represented in the requirement specification, which is 0x0100:0000 and 0xffff:ffff. RF\_Read returns the read data from the given registers and returns them through the first and second GPR read data. In RF\_Write, if the RegWrite control bit is 1, it writes the given data into the following register.

**Data Memory (DM)**

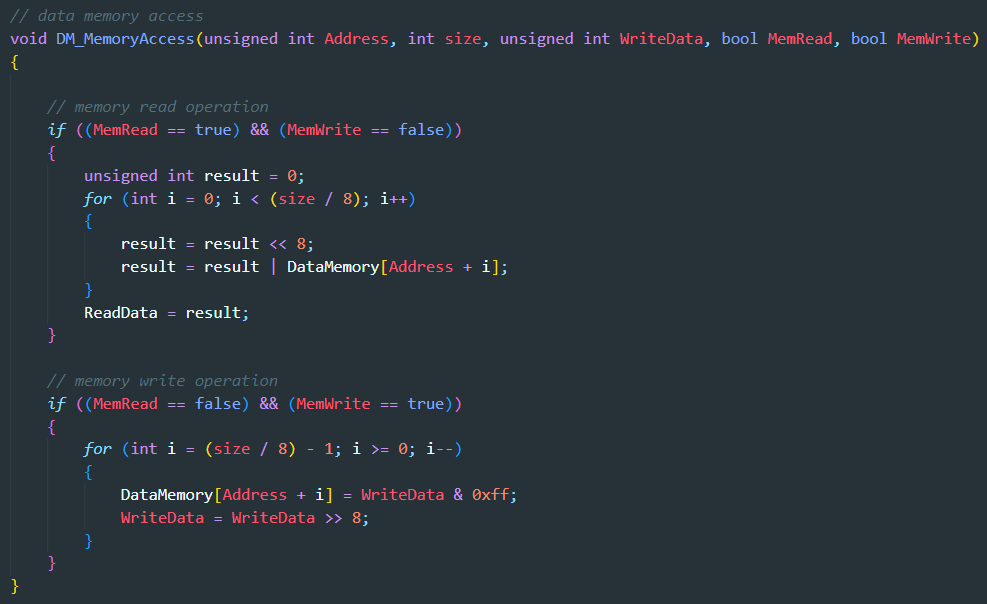


Figure 6 - Data Memory

Figure 6 shows the DM\_MemoryAccess function. If the MemRead bit is 1, then it returns the read data from the given address to the GPR read data. If MemWrite bit is 1, then it writes the data into the following address in the data memory per byte.

**MUX**

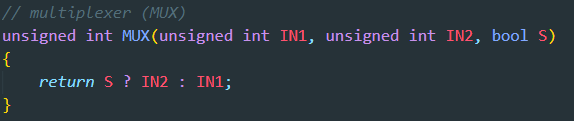


Figure 7 - Multiplexer

Figure 7 shows a multiplexer (MUX). We could use 3-input MUX rather than 2-input MUX, however, because we have to add an additional signal for increasing the input of MUX, we decided to implement only 2-input MUX.

**Main**

The main module follows the sequence of instruction execution in Von Neumann architecture and single-cycle MIPS architecture: IF - ID/RF - EX/AG – MEM - WEB. Figure 18 shows this sequence of execution, and from now on, we will check each phase one by one.

**Load Program and Initialize**

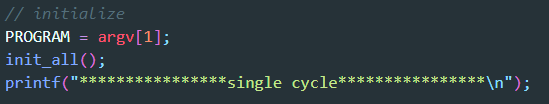


Figure 8 - Load and Init

Figure 8 shows how we loaded the program to the instruction memory. In the initial phase, we read all of the programs as we represented in 5-1, IM\_Init. IM\_Init function is in the init\_all function. Init\_all function is explained in the next section.

**Initialize**

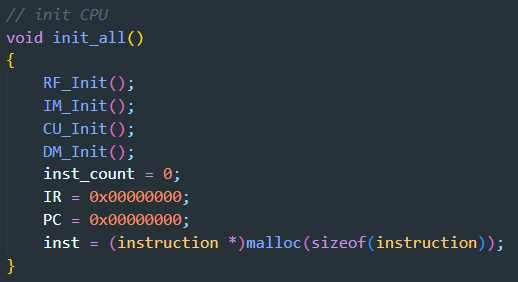


Figure 9 - Init

Figure 9 show the init\_all function. It initializes all of the components and variables.

**Instruction Fetch (IF)**

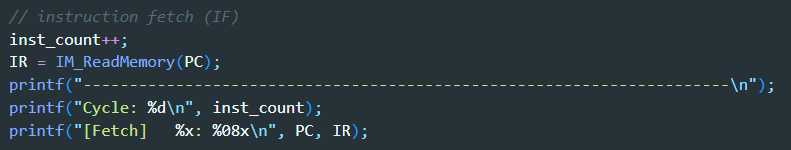


Figure 10 - Instruction Fetch

Figure 10 shows the instruction fetch phase. By using the IM\_ReadMemory function, we can get the instructions in the address of the PC.

**Instruction Decode / Register Operand Fetch (ID / RF)**

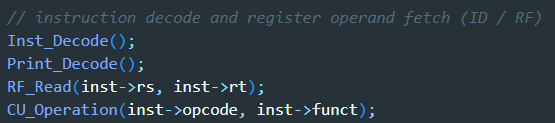


Figure 11 – Instruction Decode and Register Operand Fetch

Figure 11 shows the ID / RF phase of the program. We decode the instruction by using the Inst\_Decode function and print the result of the decode by using Print\_Decode. Then, we fetch the operand from the first and second GPR read register according to the decoded instruction and return the value of the first and second GPR read data. Then, we set the control unit bits by using the CU\_Operation function.

**Execute / Evaluate Memory Address (EX / AG)**

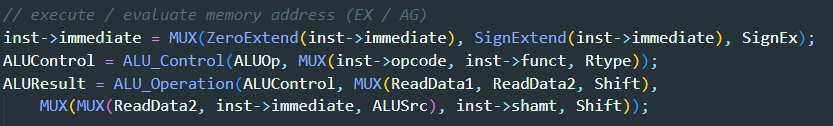


Figure 12 - EX / AG

Figure 12 shows the data paths and their implementation in a C code of EX / AG phases. From top to bottom, each red square shows each implementation of the parts. First, we control whether we will use sign-extended or zero-extended immediate. Second, we decide whether opcode or funct filed flows into the ALU Control unit. Third, we decide the input variables that will flow into the ALU unit and be operated. By implementing the simulator in this way, we can execute the instructions such as R-type and I-type arithmetic operations, and shift operations.

**Memory Operand Fetch (MEM)**

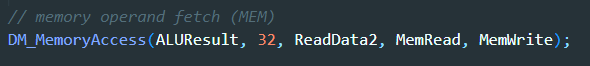


Figure 13 - MEM

Figure 13 shows the implementation of the memory access function of data memory. The interesting thing is that the number ‘32’ is in the function variable. This refers to the size of the data or memory that we are trying to access. In this project, because the given programs only use load word (LW) and store word (SW) instructions, we only use the size of 32 bits of the data. We just fixed them in the number of 32 bits. However, if there comes a new program that uses instructions that access less than 32 bits, such as store byte (SB) or load half-word unsigned (LHU), then we will implement those in the later project by using this size field.

**Store / Writeback Result (WB)**

Figure 14 - WB

Figure 14 shows the data paths and their implementation in the c code of the WB phase. From left to right, each red square shows the implementations of each part. First, it determines which register should go into the write register: rt, rd, and r31. Next, it chooses which data should flow into the write data port of the register’s component. By implementing this method, we can write the following data into the given register if the RegWrite bit is 1.

**PC Update**

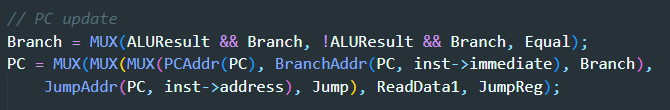


Figure 15 - PC Update

Figure 15 shows the data flow and implementation in the c code of the PC update phase. From left to right, each red square shows the implementations of each part. First, it distinguishes whether the given signal is branch equal or branch not equal. If it is branch equal, it chooses the NAND gate. Else, it chooses AND gate. Second, it decides the PC update value in PC adder, branch adder, jump adder, and First GPR read data, which is data from rs register in instructions such as JR or JALR.

**3. About Sample Programs**

다음은 예제 소스코드에 대한 설명입니다.

1. simple.c : 오직 리턴만하는 함수

2. simple2.c : 100을 리턴하는 함수

3. simple3.c : 1부터 100까지의 합을 리턴하는 함수 (반복문)

4. simple4.c : 1부터 10까지의 합을 리턴하는 함수 (재귀)

5. gcd.c : 0x1298과 0x09387의 최대공약수를 리턴하는 함수 (재귀)

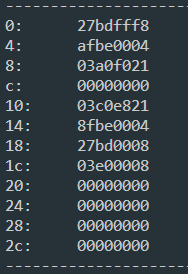
6. fib.c : 피보나치 수열의 10번째 숫자를 리턴하는 함수 (재귀)

7. input4.c : input\_data.h에 저장되어 있는 값들 중 102번째 작은 값을 리턴하는 함수

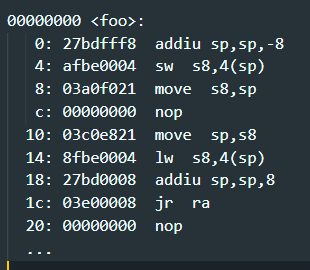
다음은 출력에 대한 설명입니다. 각 예제 소스코드에 대한 출력 결과를 보여주고 각 타입과 주요 명령어에 대한 설명을 할 것입니다.

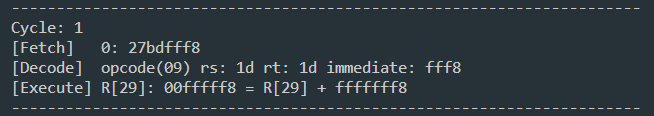
**3-1 simple.c**

우선 simple.bin에 있는 명령어를 순서대로 Instruction Memory에 저장합니다. 그것을 출력하여 원하는 16진수의 명령어들이 메모리에 잘 저장되어있는지 확인합니다.

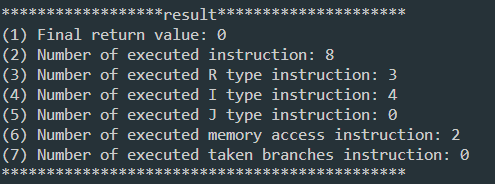


simple.mips.asm과 비교하여 명령어들이 잘 저장되었는지 확인합니다.

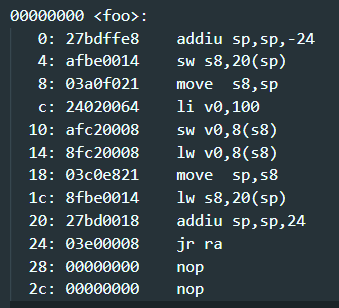


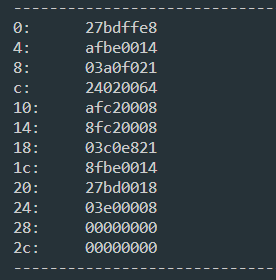
사이클 한번이 돌 때마다 다음 그림과 같이 표시가 되게 됩니다.

몇 번째 사이클인지 나타내 주는 첫 줄, 해당 사이클에서 fetch 해 온 명령어, decode한 값, execute한 결과가 출력됩니다.

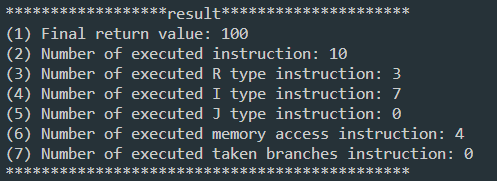
프로그램이 종료되면 result 값이 나오는데, 여기서 final return value, number of executed instruction, number of executed R type instruction, etc … 등이 출력됩니다.

**3-2. simple2.c**

위에서 한 과정을 동일하게 반복합니다.



다음과 같은 결과가 나옵니다.

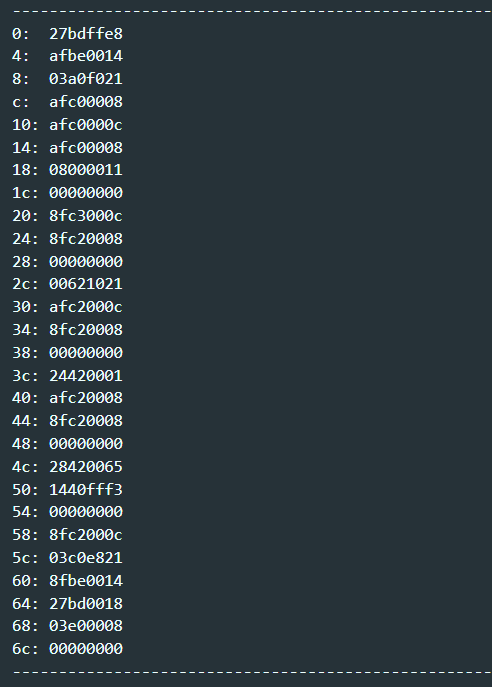


**3-3. simple3.c**

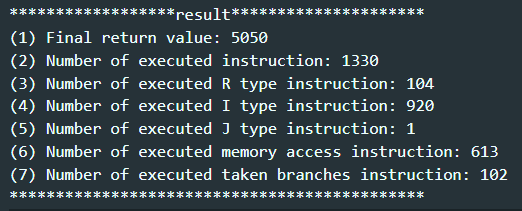
위에서 한 과정을 반복합니다. simple3 cycle의 횟수가 많아져서 .txt파일에 저장한 뒤 결과를 확인합니다.

****

저장된 instruction은 다음과 같습니다.

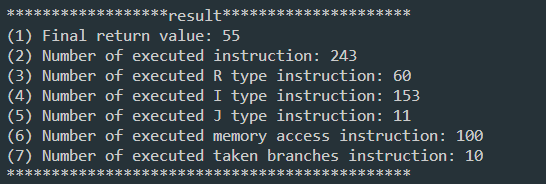


실행 결과는 다음과 같습니다.



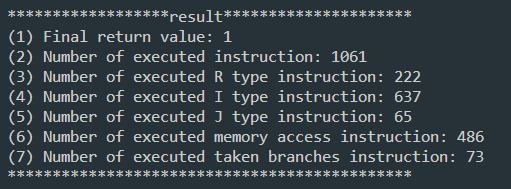
**3-4. simple4.c**

메모리에 instruction이 정상적으로 저장된 것을 확인했고, 결과는 다음과 같습니다.

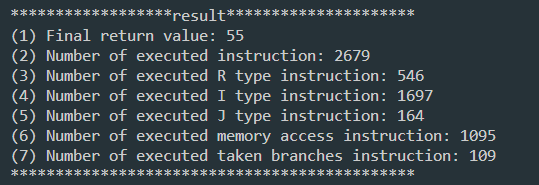


**3-5. gcd.c**

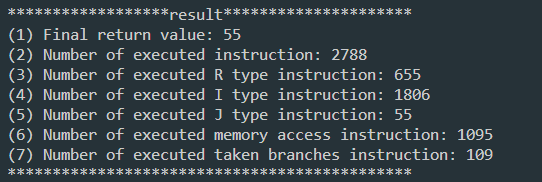
메모리에 instruction이 정상적으로 저장된 것을 확인했고, 결과는 다음과 같습니다.

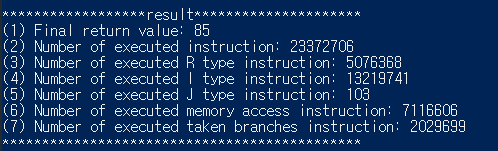


**3-6. fib.c**

메모리에 instruction이 정상적으로 저장된 것을 확인했고, 결과는 다음과 같습니다.

**3-7. fib2.c**

Linux 환경이 아닌 Windows에서 binary file을 어떻게 수정해야 할지 도저히 방법이 떠오르지 않아 아는 지인에게 부탁해 수정을 한 후, fib.bin에 있는 JAL 명령어를 모두 JALR 명령어로 바꿔 준 후 출력을 하였습니다.

**3-8. input4.c**

나머지 파일들은 대략 몇 초 만에 결과가 나오는 반면, input4.bin 파일은 실행을 시킨 후 1시간 이상이 걸린 후 에야 결과를 볼 수 있었습니다. (2) Number of executed instruction 의 수를 보면 2300만번 이상의 cycle이 돈 것을 볼 수 있습니다. 처음엔 몇 분이 지나도 결과가 나오지 않아 무한루프를 돌고 있는 것 아닌가 라는 생각 때문에 제대로 코드를 작성 하였음에도 불구하고 시간을 더 쓰게 되었습니다. 결과가 딱 나왔을 때의 쾌감은 말로 설명하기 힘들 정도였습니다.

**4. About Conclusion**

**배운 점과 어려웠던 점**

본 프로젝트 진행을 위해 요구되는 기본 지식이 많았습니다. 지난 컴퓨터구조론 수업에서 배운MIPS 명령어에 대한 이해와 그것의 응용인 MIPS Single-Cycle에 대한 이해가 완벽해야 코드를 짤 수 있었습니다. 그것에 대해 이해가 부족한 상황에서 코드를 짜기 시작하기가 굉장히 어려웠습니다. 하지만 수업을 따라가고 스스로도 계속 찾아보며 공부를 하면서 개념이 명확히 잡혔습니다. 사실 이번 파트에서 배운 점을 쓰지만, 지금까지 레포트에 작성한 내용 대부분을 배웠습니다. 그래서 다음에 혹은 모르는 사람이 레포트만 보아도 개념이 쉽게 이해가 될 수 있게 작성을 하였습니다.

**C언어**

C언어로 구현하면서 스스로 아직 많이 부족하다고 느꼈던 부분이 많았습니다.

- 이번에 **구조체**를 처음 사용해보면서 공부를 해서 코드 구현에 적용하는데 까지 조금 시간이 걸렸습니다. 특히 구조체를 인자로 받고 사용할 때 방법을 몰라서 계속 에러가 나는 어려움을 겪었습니다. 하지만 이번 프로젝트를 통해 구조체에 대해 많이 배웠고 특히 구조체 변수를 다룰 때와 구조체 포인터를 다룰 때의 경우는 명확히 구분할 수 있게 되었습니다..

- **헤더 파일**을 만들어 각 파일들을 링크하는데에 완벽하게 이해가 되었습니다. 초반에 헤더 파일에 함수들을 넣어 컴파일을 했지만 링크가 제대로 되지 않아서 어려움을 겪었습니다. 하지만 이내 잘못을 깨닫고 링크되는 함수들만을 넣어 잘 링크를 시켜서 헤더 파일 사용에 대해서 자신감이 생겼습니다.

**코드 구현 - 잘한 점과 아쉬웠던 점**

**- 깔끔하게**

저번 프로젝트에서 아쉬웠던 부분 중 하나가 코드를 난잡하게 짰던 부분이었습니다. 그래서 이번에는 최대한 깔끔하고 가독성이 좋게 짜려고 노력을 했습니다. 최대한 메인함수의 내용을 줄이고 반복문만 읽어봐도 어떤식으로 동작하는지 파악할 수 있게 함수들을 네이밍 했고 함수를 짤 때에도 최대한 가독성을 주었습니다.

제가 지금까지 해보았던 프로젝트 중에 가장 오랜 시간이 걸린 프로젝트였습니다. 만약에 어떠한 정보가 주어진 상태에서 그것에 대해 구현하라고 했으면 오래 걸리지 않았을텐데 MIPS Single-Cycle이라는 개념을 이해하는데 어려움을 겪어서 코드 구현을 시작하는게 어려웠습니다. 또한 개념을 다 이해했더라도 적용시켜야할 개념들이 여러개로 연결 되어있어서 이것을 한 번에 생각하여 구현하기 어려웠습니다. 하지만 공들인만큼 성공적으로 구현할 수 있게 되어 많은 성취감을 느꼈으며 자신감 또한 생겼습니다. 요약하자면 이번 프로젝트를 통해 C언어의 구조체, 헤더 파일에 대한 이해와 MIPS-Single-Cycle의 개념을 완벽히 숙지하게 되었습니다. 실행 결과가 이번 프로젝트처럼 자신의 한계를 뛰어넘게 해주는 것을 성공 했을 때의 카타르시스를 다시 한번 느끼게 되었습니다.