

1 Freedays(5일중 4일 사용)

Mobile Processor Programming

assignment #2: Single-cycle MIPS

모바일시스템공학과 32217259 문서영

[목차]

1. 개요

2. 배경지식

3. 코드 설명

4. 출력 예시

5. 어려웠던 점 및 후기

1. 개요

이번 과제는 single-cycle architecture MIPS 를 구현하는 프로젝트로, fetch, decode, evaluate address, fetch operands, execute, store result 과정을 거칩니다.

2. 배경지식

1. ISA(Instruction Set Architecture)

ISA 는 명령어 집합 구조로, 마이크로프로세서가 인식해서 기능을 이해하고 실행할 수 있는 기계어 명령어를 말합니다. 즉 하드웨어와 소프트웨어 사이의 인터페이스 역할을 하며 컴퓨터 성능과 호환성에 영향을 줍니다. ISA 는 RISC(Reduced Instruction Set Computer)와 CISC(Complex Instruction Set Computer)로 나뉩니다.

RISC 는 간단하고 빠른 명령어들로 구성된 프로세서이기에 하드웨어 회로가 단순하고, 발열과 전력소모가 적습니다. 이에 명령어 실행 속도가 빠르고, pipelining 기법을 쉽게 적용할 수 있다는 장점이 있습니다. 하지만 같은 내용을 처리하는 데 더 많은 메모리 공간이 필요하며, 고급 언어와의 호환성이 떨어집니다.

CISC 는 복잡하고 다양한 명령어들로 구성된 프로세서로, 코드 밀도가 높아 메모리 공간을 절약할 수 있어 복잡한 연산을 한 번에 수행할 수 있으며, 고급 언어와 호환성이 좋다는 장점이 있습니다. 하지만 하드웨어 회로가 복잡하여 가격이 비싸고 발열이 많으며 전력소모가 큼니다. 또한 명령어 실행 속도가 느리며, pipelining 기법에 적용하기 어렵습니다.

2. MIPS

MIPS 는 Microprocessor without Interlocked Pipeline Stages 의 약자로, ARM 과 같이 RISC 기반의 ISA 를 가집니다. RISC 답게 단순하고 구조화된 명령어 구조를 가지며, 컴파일러 성능에 의존하는 경향이 있습니다.

[MIPS 명령어 체계]

MIPS 명령어 체계는 총 3 가지 종류로 이루어져 있습니다.

BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct	
	31	26 25	21 20	16 15	11 10	6 5	0
I	opcode	rs	rt	immediate			
	31	26 25	21 20	16 15	0		
J	opcode	address					
	31	26 25	0				

<그림 1>

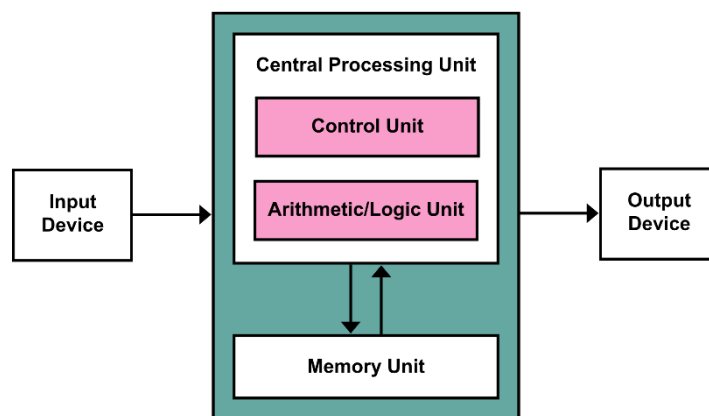
Opcode: 해당 명령어가 실행할 연산의 종류를 정의합니다.

R-type: 연산을 할 때, 2 개의 레지스터 값을 이용하여 연산을 한 다음, 다른 레지스터 하나에 연산한 값을 기록합니다. 이 때 연산에 사용되는 레지스터는 각각 rs(register source), rt(register target)이며, 연산을 한 후 rd(register direction)에 저장됩니다.

I-type: R-type 과 비슷하게 2 개의 값을 이용해 연산을 한 다음 다른 하나의 레지스터에 저장하지만, 연산할 값을 하나는 레지스터(rs)에서 가져오고, 다른 하나는 임의의 값(imm_immediate)이 되며, 레지스터 rt 에 저장합니다.

J-type: 무조건 분기 명령어로, 특정 메모리 주소로 바로 이동이 필요한 경우에 사용되는 명령어이며 이동할 메모리 주소를 연산에 사용합니다.

3. Von Neumann Architecture



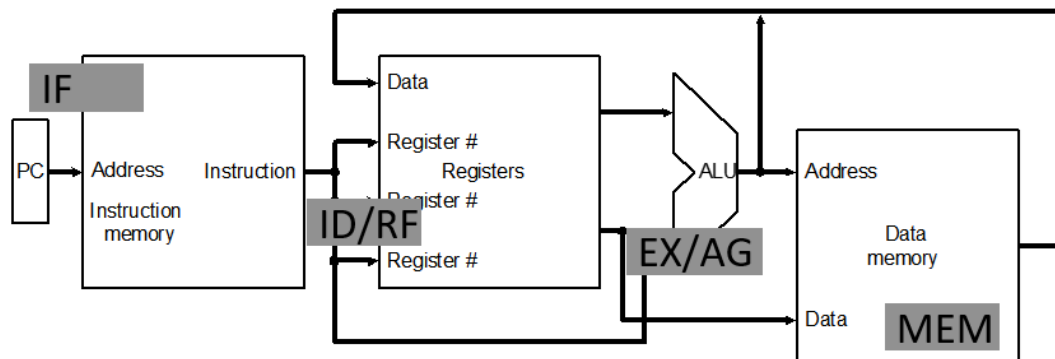
<그림 2>

폰 노이만 구조는 현재 컴퓨터에서 가장 많이 사용되는 기본적인 아키텍처입니다. 메모리, 입출력장치가 CPU 를 중심으로 연결되어 있고, 프로그램과 데이터가 같은 메모리 공간에 저장되어 처리되는 구조입니다.

CPU 는 메모리에 저장된 프로그램을 실행하며, 프로그램에서 필요한 데이터 또한 메모리에서 가져옵니다. 이런 구조에서 CPU 가 메모리와 입출력 장치에 직접 접근이 가능하기 때문에 처리 속도가 매우 빠릅니다. 폰 노이만 구조는 명령어와 데이터가 동일한 메모리에 저장되어 있기 때문에, 명령어와 데이터를 구분하기 위한 별도의 제어 신호가 필요하지 않다는 점입니다. CPU 는 메모리에서 순차적으로 명령어를 읽어와 실행하는 방식으로 동작합니다.

4. Instruction Cycle(명령 주기)_ instruction processing

명령 주기는 CPU 가 메모리로부터 1 개의 명령어를 가져와 이를 수행하는 과정을 말하며 5 단계로 이루어져 순차적으로 이루어집니다.



<그림 3>

1) Fetch(Instruction fetch _IF): CPU 가 메모리에서 다음으로 실행할 명령어를 가져옵니다. 이 때 CPU 는 메모리 주소 버스를 이용해 메모리 주소를 지정하고, 데이터 버스를 통해 명령어를 가져옵니다. 이 과정에서 순차적으로 한 번에 처리되는 데이터 단위는 1word 이며, 현재 CPU 가 처리하는 최소 정보처리 단위로서 대부분 64bits 를 사용하고 있습니다.

2) Decode(Instruction decode and register operand fetch _ID/RF): CPU 가 가져온 명령어를 해독하여 어떤 작업을 수행해야 하는지 결정합니다. 이때 MIPS 에서는 opcode 를 사용하여 연산자(operator)를 구분하고 해당 명령어에 필요한 레지스터나 메모리 주소를 결정합니다.

3) Execute(Execute/Evaluate memory address _EX/AG): CPU 가 결정한 명령어를 해석하고 실행합니다. 이 때 CPU 는 연산을 수행하기 위해 필요한 데이터를 레지스터에서 저장하거나 가져옵니다.

4) Memory Access(Memory operand fetch _MEM): CPU 가 메모리에 접근하여 데이터를 읽거나 쓸 수 있습니다. CPU 는 메모리 주소를 결정하고, 데이터 버스를 통해 메모리와 데이터를 주고받습니다.

5) Write Back(Store/writeback result _WB): 계산된 결과를 레지스터에 저장합니다. CPU 는 결과를 저장할 위치를 결정하고, 데이터 버스를 통해 결과를 저장합니다.

single-cycle processor 는 각 instruction 이 single clock cycle 이 걸리는 프로세서이며, 모든 state 가 instruction 의 실행이 끝난 후에 update 되는 구조입니다. combinational logic 과 sequential logic(state)로 구성되어 있습니다. 이와 비교할 때, multi-cycle processor 가 있는데, 이는 single-cycle processor 와 달리 instruction 의 실행 동안에 업데이트 될 수 있습니다. 이를 봤을 때 single-cycle processor 의 큰 단점은 가장 느린 instruction 이 사이클 타임을 결정한다는 점입니다.

5. datapath

Datapath 는 CPU 에서 데이터와 주소를 처리할 수 있는 요소들을 말하며, 레지스터, ALU, MUX, 메모리 등이 datapath 라 말할 수 있습니다.

1) Instruction fetch

위의 instruction processing 과정에 따라 fetch 를 먼저 살펴보도록 하겠습니다.

PC(Program Counter)는 실행할 명령어의 주소를 저장하고 있는 레지스터입니다.

Instruction memory 는 명령어 주소를 받아 그 주소에 맞는 32 bit 형식(R, I, J format)의 instruction 을 내보냅니다.

Add 는 덧셈을 위한 요소로, 다음 명령어의 주소를 가리키기 위해($pc + 4$) 존재합니다. 4 를 더하는 이유는 instruction 의 길이가 32bit 이고, 주소 하나가 1byte 이므로 4byte 뒤의 주소가 다음 명령어의 주소이기 때문입니다.

2 -1) R-format

r-format 은 위에서 살펴본 것처럼 레지스터 2 개의 값(rs, rt)을 연산하여 다른 레지스터(rd)에 저장하는 instruction 을 포함하고 있습니다. 과정을 설명해 보자면, Rs 와 rt 는 각각 read register1, read register2 에 저장되고, 각 레지스터에 저장된 값을 read data1, read data2 로 읽어 ALU 로 넘어가 연산 후 ALUresult 에 저장됩니다. 이 값을 레지스터의 write data 로 불러와 write register 인 rt 안에 저장됩니다.

2-2) I-format

i-format 은 일반적으로 rs 를 read register1 로 가져온 레지스터에 대한 값을 read data1 로 읽어 ALU data1 로 가져오고, 임의의 값 imm 을 sign extend 시켜 alu 로 보낸 후 ALU 연산을 합니다. 이 값을 레지스터의 write data 로 보내 write register 인 rt 값에 저장됩니다.

2-3) LW/SW instruction

Load/store 연산 또한 I-format 을 이루고 있기 때문에 위의 I-format 처럼 ALU 에 값을 저장하는 것까지는 같지만, alu result 를 address 로 저장하여 memory 를 사용한다는 점에서 큰 차이를 가집니다.

두 연산의 살펴보자면, load 는 memory 에서 읽은 데이터(read data)를 rt 레지스터로 가져오고, store 는 레지스터 rt 의 값을 memory 로 읽어(write data) alu 연산하여 가져온 메모리의 주소에 저장된다는 점에서 차이가 있습니다.

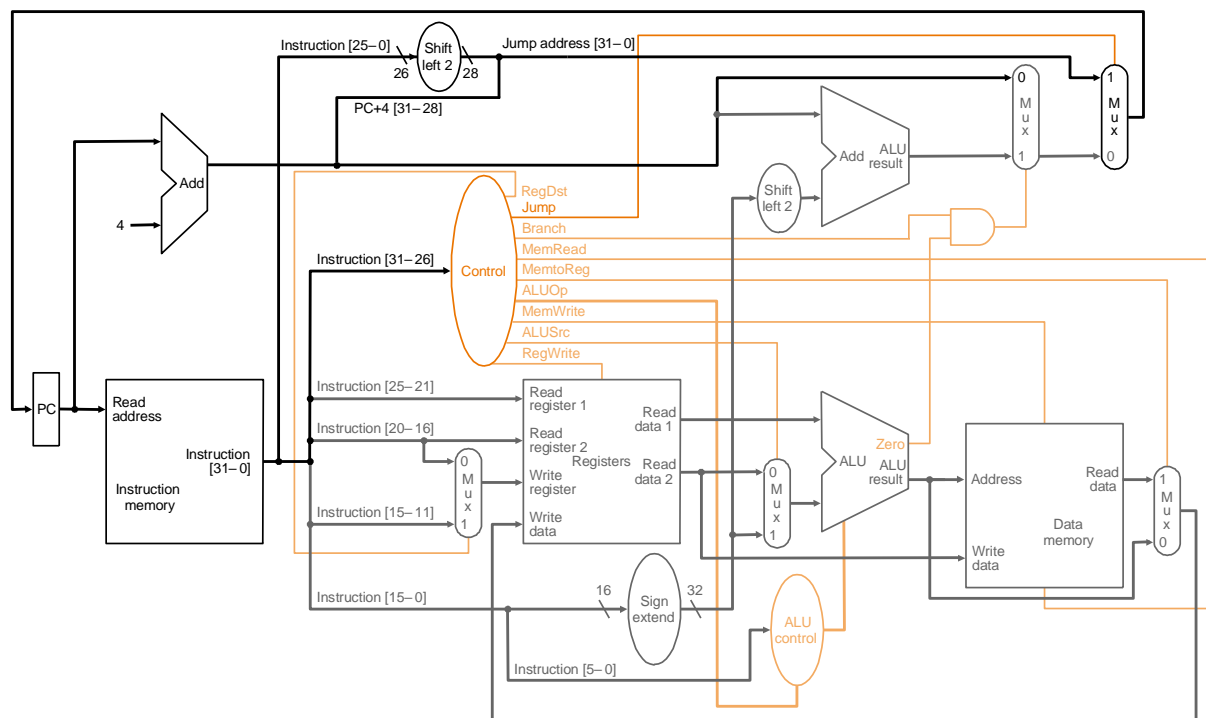
2-4) Branch Instruction

Branch(bne, beq) 또한 I-type 으로 I-type 의 연산 결과를 따르지만, add 연산을 하는 다른 ALU 연산과 달리 sub 연산으로 rs 와 rt 값을 비교하여 pc 값을 업데이트 한다는 점에서 차이가 있습니다.

2-5) Jump Instruction

Jump instruction 은 j-type 의 형식으로, register 를 사용하는 다른 instruction 과 달리 오직 pc 와 immediate value 를 sign extend 시킨 값을 사용합니다.

Instruction 은 control unit 에 따라 실행되며, fetch, decode, evaluate address, fetch operands, execute, store result 의 단계가 있습니다.



<그림 4>

<그림 4>는 모든 instruction 의 datapath 구조화한 사진입니다.

3. 코드 설명

- 실행

Visual studio code 를 사용하여 만들었으며, 언어는 C 언어를 사용했습니다.

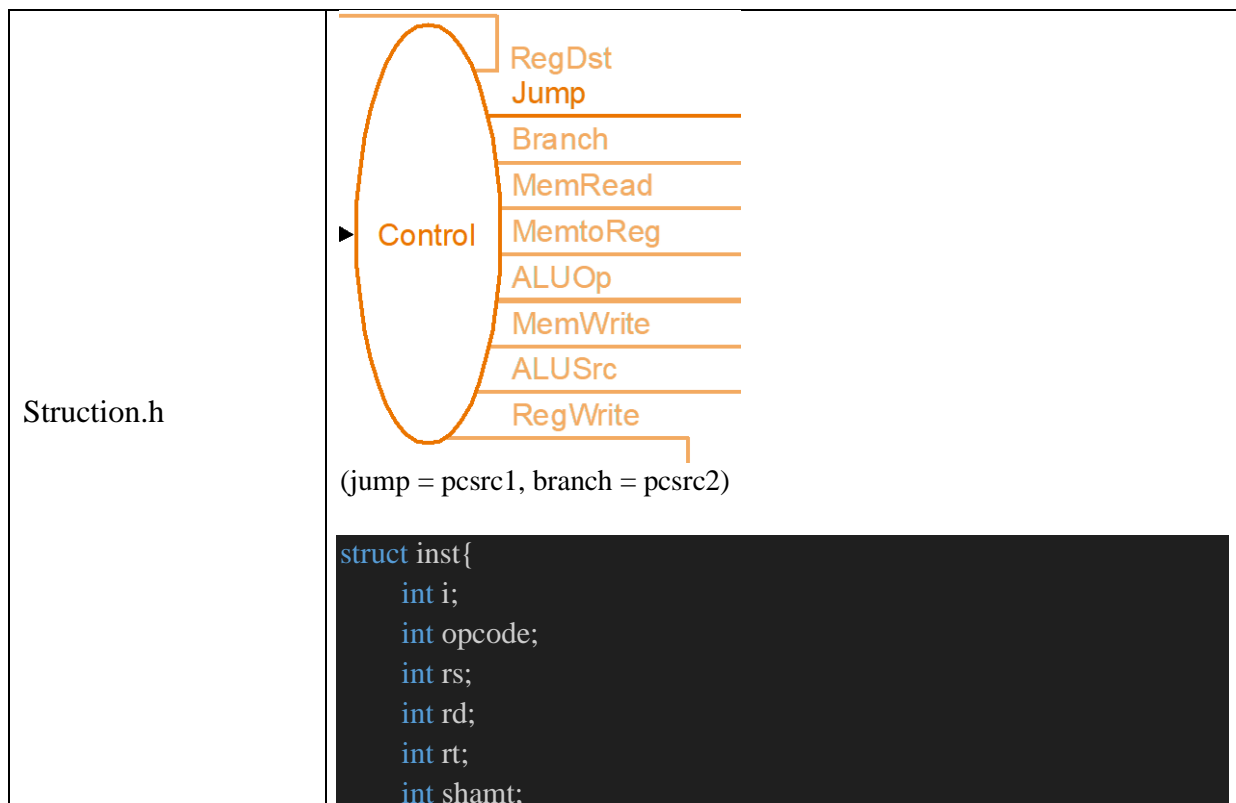
```
top\univ\4-1\camp_test> gcc ./newmips.c -o singlemips
top\univ\4-1\camp_test> ./singlemips .\test_prog\input4.bin
```

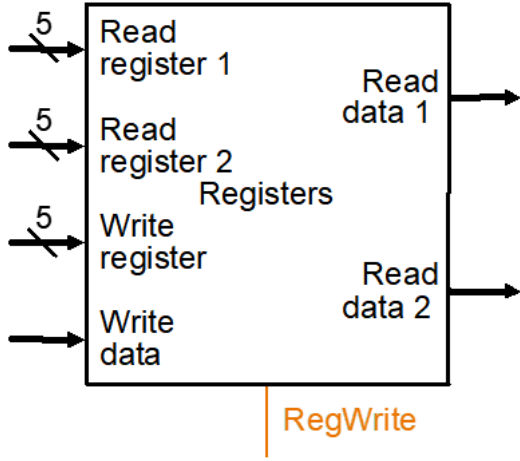
[출력 방법 예시]

원래는 오직 switch-case 문을 사용하여 opcode 에 따라 instruction 을 입력했는데, single-cycle 의 개념을 다시 보고, single-bit control signal 을 사용하여 각 register, alu, memory 값으로 나누어 주었습니다. 이에 따라 switch-case 문으로만 만든 mips 와 control signal 에 따른 single-cycle mips 를 같이 넣었습니다.

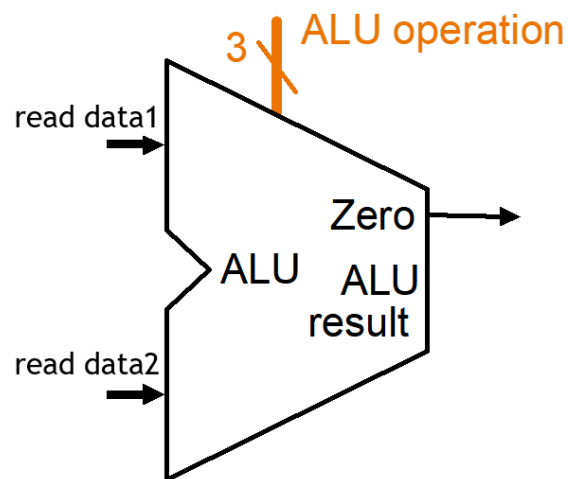
또한 각 결과를 출력할 때 값이 상당히 많기 때문에 DEBUG_PC(#ifdef~ #endif)를 사용하여 각 structure 의 출력을 분리하였습니다.

1. structure



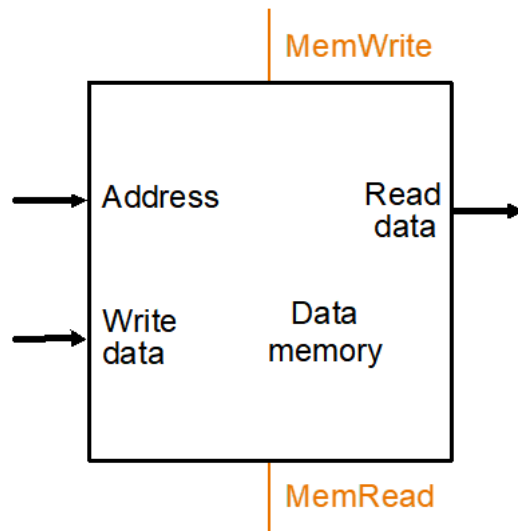
	<pre> int func; int imm; int imm_s; }; </pre>
Control.h	<pre> struct control{ int regdest; // op = 0 (r-type) int pcsrc1; // op = j, jal int pcsrc2; // op = bxx && b 만족 int memread; // op = lw int memtoreg; // op = lw int aluop; // op = 0 -> 1, op = j-type -> 0, // op = lw, sw -> 2, op = bxx -> 3, // op = i-type -> 4 int memwrite; // op = sw int alusrc; // op != 0, beq, bne int regwrite; // op != sw, bxx, j, jr }; </pre>
Register.h (register)	 <pre> struct regist{ int readreg1; int readreg2; int writereg; int writedata; int rddata1; int rddata2; }; </pre>

aluInst.h

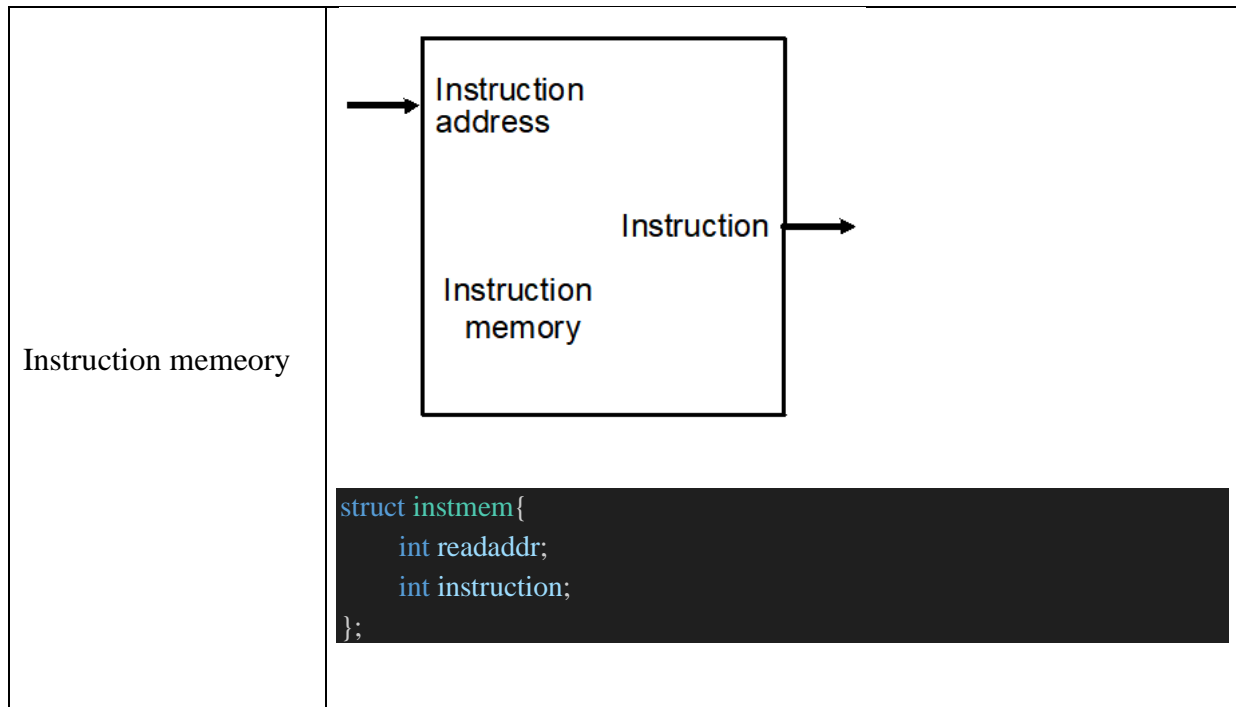


```
struct alu{  
    int rddata1;  
    int rddata2;  
    int alurestult;  
};
```

Memory.h
(data memory)



```
struct memor{  
    int address;  
    int writedataMem;  
    int readdataMem;  
};
```



2. global value

Int mem[0x400000];

Long int reg[32] = {0};

// reg[29] = 0x1000000, reg[31] = 0xffffffff (r29와 r31는 지역 변수로 넣어주었습니다.)

Int pc = 0;

2. function

void controlunit(int opcode, struct control *cont_, int funct, int rs, int rt);

: single-bit control signals 결정하는 함수로, opcode에 따라 값을 결정했습니다.

int alu(int opcode, struct regist *regist_, struct alu *alu_);

: bxx, lw, sw 이외의 나머지 I-type instruction 수행하는 함수입니다. opcode에 따라 switch-case 문을 사용하였고, register에 따라 alu 값을 변화시켜서 register의 write data 값을 return 하도록 했습니다.

- 밑의 opzero 함수와 같이 unsigned과 signed value를 구분하기 위해 변수를 따로 설정하였습니다.

int opzero(int funct, struct regist *regist_, int shamt);

: opcode = 0인 R-type instruction 수행하는 함수입니다. R-type의 instruction 중 jal은 다른 instruction과 달리 pc에 reg[rs]값을 저장하는 (only pc update)operation이기에 funct = 0x8(jal)일 때는 바로 break; 하도록 설정했습니다. 또한 나머지 R-type instruction은 funct에 따라 switch-case 문을 사용하여 변화시켰습니다.

Return 값은 register의 reg[regist_>wrtiereg] (= reg[rt])로 설정했습니다.

- addu와 and 같이 unsigned와 signed value를 구분하기 위해 변수를 따로 설정하였고 이 값을 reg[rt] (regist_>writedata) 값에 저장해주었습니다.

```
unsigned int Alures_;  
signed int Alures;
```

Main function

1. 먼저 file을 오픈하여 opcode, rs, rd, rt, shamt, funct, imm, imm_s, addr로 분리시켜 **메모리 배열 (mem[0x0x400000])에 저장**해줍니다.

2. Instruction fetch, Instruction decode

control bit에 따라 **register, alu, memory값을 설정**해줍니다. (single-cycle processor 참조)

3. Execute

Control bit에서 aluoperation에 따라 **연산을 수행하여 aluresult 값을 변환**시켜 줍니다.

- j-type은 pc를 변환시키는 operation이기에 수행하지 않고 지나갑니다.

- opcode = 0인 operation은 opzero function으로 function code에 따라 값을 설정해 줍니다.

- sw와 lw operation에서는 aluresult가 address(addr = r[rs] + imm_s)이기에 따로 분리했습니다.

- bne와 beq의 aluresult는 이 operation이 만족할 때 넣어줄 값을 설정하였습니다.

(BranchAddr = { 14{immediate[15], immediate, 2'b0 } = imm_s * 4}

- I-type일 때의 aluresult는 opzero function처럼 alu function으로 분리하여 opcode에 따라 값을 설정했고, r[rt]에 aluresult를 넣어주었습니다.

4. memory & write back

Memory는 **lw와 sw**만 사용합니다. 그렇기에 address에 aluresult로 받은 값을 바로 넣어주었고, 나머지는 control bit에 따라 sw(memwrite)와 lw(memread, memtoreg)를 구분하였습니다.

5. **pc update** (control bit: psrc1, psrc2)

Psrc1 == 1일 때는 j-type operation이고, psrc2 == 1일 때는 bxx operation입니다. 만약 둘 다 아니면 pc에 4를 증가시켜 넣어주었고 jal operation은 $pc = r[rs]$ 값이기 때문에 따로 빼주었습니다.

4. 출력 예시

1. simple.bin

```
PS C:\Users\82106\Desktop\univ\4-1\camp_test> .\singlemips .\test_prog\simple.bin
-----
return value(value in r2) : 0(0x0)
number of executed instruction : 8
number of (executed) R-type instruction : 4
number of I-type instruction : 4
number of J-type instruction : 0
number of memory access instruction : 2
number of taken branches : 0
-----
```

2. simple2.bin

```
PS C:\Users\82106\Desktop\univ\4-1\camp_test> .\singlemips .\test_prog\simple2.bin
-----
return value(value in r2) : 100(0x64)
number of executed instruction : 10
number of (executed) R-type instruction : 3
number of I-type instruction : 7
number of J-type instruction : 0
number of memory access instruction : 4
number of taken branches : 0
-----
```

3. simple3.bin

```
PS C:\Users\82106\Desktop\univ\4-1\camp_test> .\singlemips .\test_prog\simple3.bin
-----
return value(value in r2) : 5050(0x13ba)
number of executed instruction : 1330
number of (executed) R-type instruction : 409
number of I-type instruction : 818
number of J-type instruction : 1
number of memory access instruction : 613
number of taken branches : 102
-----
```

4. simple4.bin

```
PS C:\Users\82106\Desktop\univ\4-1\camp_test> .\singlemips .\test_prog\simple4.bin
-----
return value(value in r2) : 55(0x37)
number of executed instruction : 243
number of (executed) R-type instruction : 79
number of I-type instruction : 143
number of J-type instruction : 11
number of memory access instruction : 100
number of taken branches : 10
-----
```

5. fib.bin

```
PS C:\Users\82106\Desktop\univ\4-1\camp_test> .\singlemips .\test_prog\fib.bin
-----
return value(value in r2) : 55(0x37)
number of executed instruction : 2679
number of (executed) R-type instruction : 818
number of I-type instruction : 1588
number of J-type instruction : 164
number of memory access instruction : 1095
number of taken branches : 109
-----
```

5. gcd.bin

```
PS C:\Users\82106\Desktop\univ\4-1\camp_test> .\singlemips .\test_prog\gcd.bin
-----
return value(value in r2) : 1(0x1)
number of executed instruction : 1061
number of (executed) R-type instruction : 359
number of I-type instruction : 564
number of J-type instruction : 65
number of memory access instruction : 486
number of taken branches : 73
-----
```


6. input4.bin

```
PS C:\Users\82106\Desktop\univ\4-1\camp_test> .\singlemips .\test_prog\input4.bin
-----
return value(value in r2) : 85(0x55)
number of executed instruction : 23372706
number of (executed) R-type instruction : 10152862
number of I-type instruction : 11190042
number of J-type instruction : 103
number of memory access instruction : 7116606
number of taken branches : 2029699
-----
```

5. 어려웠던 점 및 후기

맨 처음에는 single-cycle 을 잘 이해하지 못하고 switch-case 문만을 이용해서 값을 출력했습니다. 거의 제출 3-4 일 전에 깨닫고 고쳤기 때문에 코드가 중구난방으로 분리되어 있을 수도 있습니다. 이를 통해서 코드를 작성하기 전에 손으로 개념을 익히는 게 중요하다는 점을 깨닫게 되었고, 중간고사에서 그렸던 cycle architecture 가 매우 도움이 되었습니다.

대부분의 operation 은 register 만을 사용하는 연산이기 때문에 고쳐야 할 점이 많이 없었는데, sw 와 lw, bne/beq, j-type operation 은 memory 와 pc 를 변화시키는 연산이기에 switch-case 문을 이용할 때보다 더 어렵게 느껴졌습니다. 특히 sw/lw operation 에서는 control bit 에 따라 모든 값을 분리하다 보니 register 의 값을 memory 에 잘못 넣어 분명히 같은 operation 인데, 어디 있는지에 따라 값이 달라져서 더욱 혼란스러웠습니다.

이렇게 코드를 길게 써본 경험이 없어서 처음에는 구현해야 할 것들이 많아 어렵게 느껴졌지만, instruction format(R,I,J instruction format)형식에 따라 반복되는 부분이 있어서 생각보다 어렵지 않았습니다. 하지만 코드를 잘못 작성한 탓에 제대로 수정할 시간이 부족해 추가 과제인 jalr 를 수행하지 못하게 되었습니다. 그렇기에 아쉬움이 많은데, 이 점은 다음 과제인 파이프라인을 사용한 mips 를 만드는 과제에서 보충하여 깔끔한 코드를 작성할 수 있도록 하겠습니다.

****ChatGPT 관련:** 각 operation 의 설명이 검색을 해도 부족하여 사용 예시를 통해 이를 제대로 확인하기 위해 사용했습니다.