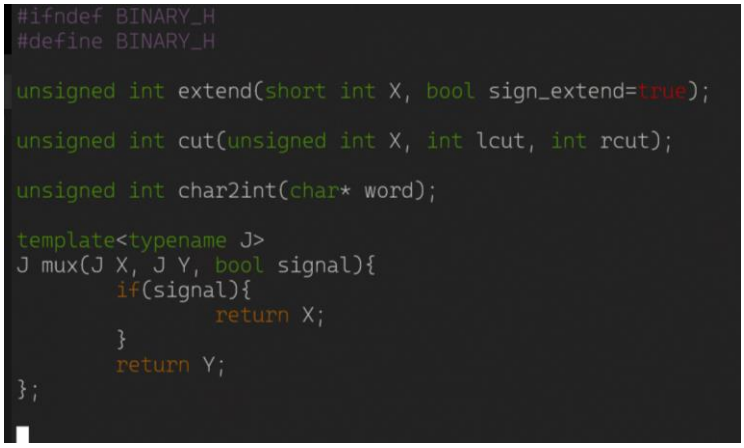
컴퓨터 구조 Homework #2

201911050 도담록, 2023-04-04

1. 코드 설명

i. Binary.h/Binary.cpp

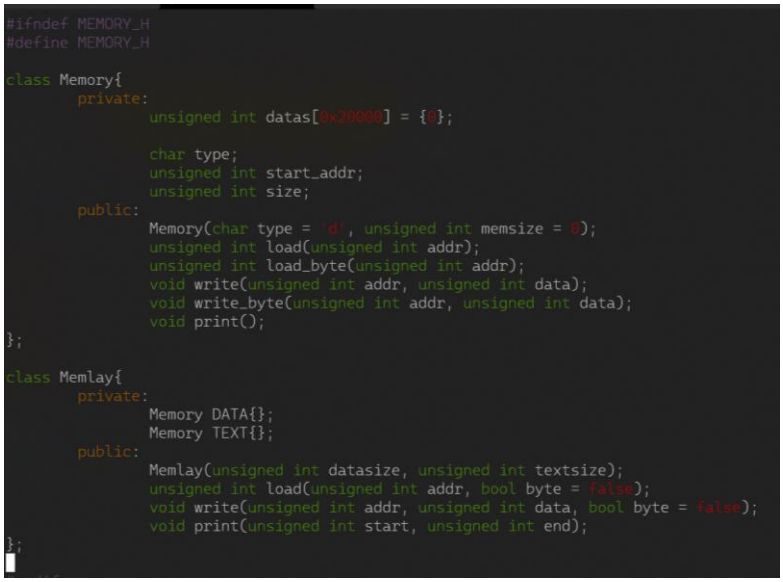


```
#ifndef BINARY_H
#define BINARY_H

unsigned int extend(short int X, bool sign_extend=true);

unsigned int cut(unsigned int X, int lcut, int rcut);

unsigned int char2int(char* word);

template<typename J>
J mux(J X, J Y, bool signal){
        if(signal){
                return X;
        }
        return Y;
};
```

Binary.h

Sign/zero extension을 위한 extend 함수, 왼쪽/오른쪽에서 bit를 잘라 주는 cut 함수, signal을 받아 둘 중 하나를 선택하는 mux 함수가 구현되어 있다.

ii. Memory.h/Memory.cpp



```
#ifndef MEMORY_H
#define MEMORY_H

class Memory{
        private:
                unsigned int datas[0x20000] = {0};

                char type;
                unsigned int start_addr;
                unsigned int size;
        public:
                Memory(char type = 'd', unsigned int memsize = 0);
                unsigned int load(unsigned int addr);
                unsigned int load_byte(unsigned int addr);
                void write(unsigned int addr, unsigned int data);
                void write_byte(unsigned int addr, unsigned int data);
                void print();
};

class Memlay{
        private:
                Memory DATA{};
                Memory TEXT{};
        public:
                Memlay(unsigned int datasize, unsigned int textsize);
                unsigned int load(unsigned int addr, bool byte = false);
                void write(unsigned int addr, unsigned int data, bool byte = false);
                void print(unsigned int start, unsigned int end);
};
```

Memory.h

Memory를 Array 형태로 구현하기 위해, 0x20000개의 word가 저장된 두 개(TEXT, DATA)의 메모리를 합쳐 Memlay로 구현하였다.

Write, load, print는 입력 Addr에 따라 Text section인지 Data section인지 확인해, 두 개의 메모리 중 하나에 접근한다. 없는 주소에 접근하면 0을 뱉는다.

Memory의 print는 test용이며, Memlay의 print는 load를 통해 구현하였다.

Write/lode byte의 경우에는 Little endian을 사용하는 윈도우의 data를 읽어 오기 때문에 Word의 주소에서 3-(modulo)만큼 포인터를 움직여 읽고 쓰도록 처리하였다. Write_byte는 Sb를 위한 것으로, 정해진 위치에 1byte를 저장한다. Write는 Wrte_byte를 4번 call하는 것으로, Load는 Word 단위로 data를 가져오는 것으로 구현하였다.

iii. Main 함수의 구조

Main(hw2_201911050.cpp)의 구조 입력 인자를 받아 파일을 읽어 메모리에 저장하고, sector별로 나누어서 동작을 진행한다.

실제 하드웨어의 경우에는 프로그램과 다르게 동작한다. 하드웨어의 논리 회로들은 “언제나” 작동하고 있지만, 프로그램의 경우에는 한 줄 한 줄 실행된다.

따라서, 실제 하드웨어처럼 작동하게 하려면 순서를 지켜서(아직 나오지 않은 값을 input으로 받지 않도록) flow를 설계해야 한다.

따라서, ALU나 MUX 등이 input으로 잘못된 값을 받지 않기 위해 sector를 나누어서 순서대로 값들을 뒤로 넘겨주도록 한다.

Sector은 7번까지 있는데, 앞 Sector의 input은 뒤 Sector의 결과값과 무관하기 때문에, sector 순서대로 진행하면 잘못된 input을 받는 문제가 발생하지 않는다.

기본 구조는 ppt의 구조를 참조했으며, 구현의 어려움이나 JR 등 특수한 경우를 위해 변형한 부분은 후에 자세히 설명하며 표기하도록 하겠다.

iv. Main 파일의 변수들

Textaddr. Dataaddr, PC, register는 main 함수 내에 정의되어 있다. (line 119-123) 앞의 둘은 여기서는 단순히 파일을 저장하는 주소를 의미한다.

맨 위에는 output 출력을 위한 변수들이 있다. bool들과 메모리의 시작/끝, cycle 등이 있다.

Opcode를 분석한 op, rs, rt, rd, shamt, target, imm_offset, funct가 정의되어 있으며, 이 값들은 유효한 타입의 instruction이 들어왔을 때만 변경된다. (line 21-29)

특수한 OP들을 나타내는 signal들이 존재한다. (line 33-35) BEQ와 BNE를 구분하기 위한 bool

BEQ, register에 있는 값으로 바로 분기하기 때문에 특수 처리가 필요했던 JR, J type임에도 retgister에(기존 구조에서 쓰지 않는, PC값) 값을 넣는 JAL 3개가 특수 처리를 위해 존재한다.

한 Sector를 실행할 때마다 결과값을 저장해 주어야 하기 때문에, Sector마다 결과값을 저장하는 변수들이 존재한다. (line 36-73)

v. 입력

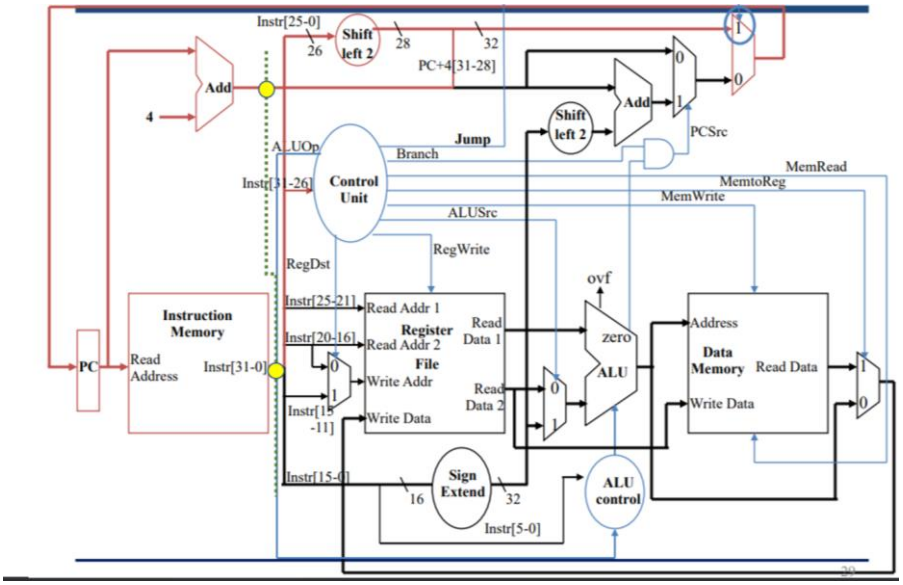
argc-1 (filename) 이전까지 “-d”, “-m”, “-n”이 나오는지 확인하고, 나온다면 (-d, -m의 경우) 그 뒤의 문자를 unsigned int로 변환하여 memstart, memend(메모리 출력 시작 위치, 끝 위치)에 저장한다.

Filename file을 열고, Memlay(전체 메모리) 클래스로 Mmemory를 선언해 그 안에 text section, data section을 저장하고 파일을 닫는다.

vi. Sector별 실행

① Sector 1: Load & PC

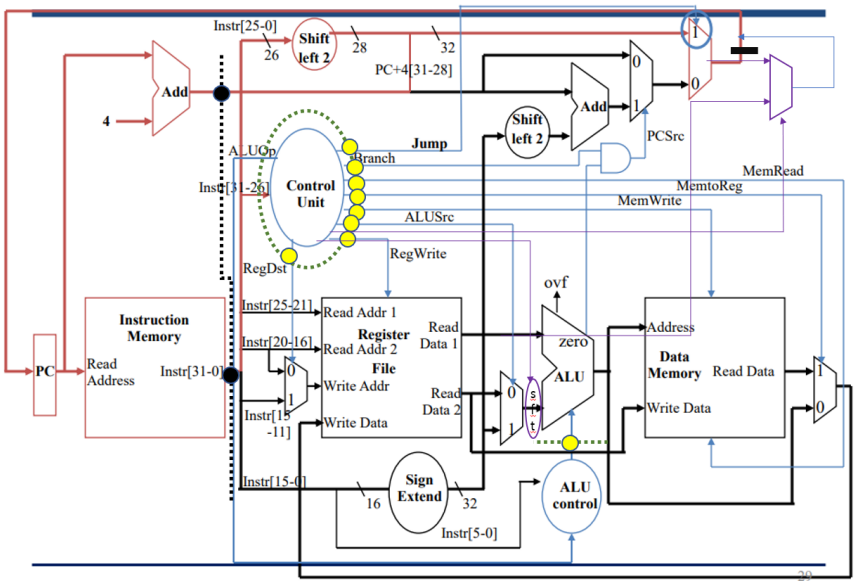
단순히 PC+4를 넘기고, Instruction을 불러온다.



② Sector 2: Control Unit, ALUctr

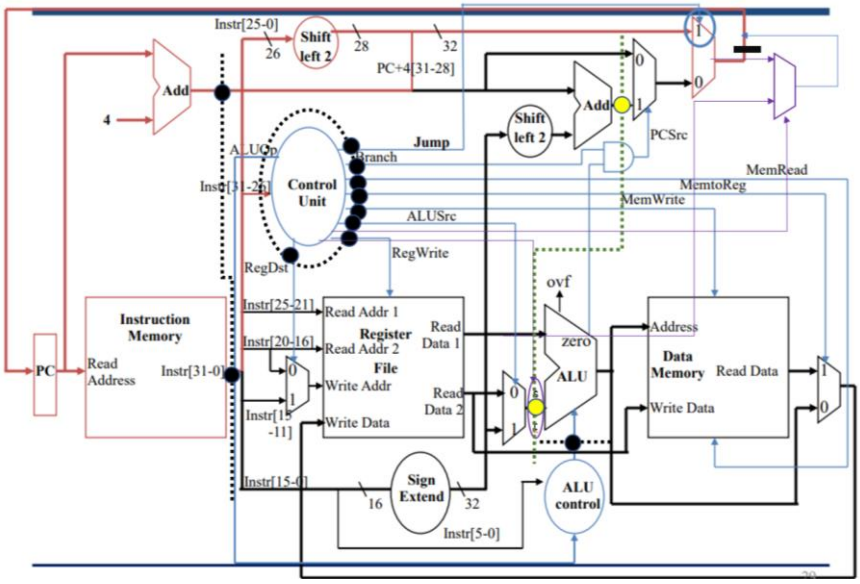
Instruction을 Control Unit에 보내, Signal 값들을 바꾼다. 이때, Byte(lb, sb를 위함), Sign(sign extend를 위함), Shift의 부호, JR, JAL, BEQ에 대한 신호가 추가로 들어갔고, ALUOp는 구현상의 편의를 위해 ALUctr로 변환되었다.

Instruction들의 op, rd, rs … 등으로의 분해도 진행되었기 때문에, ALU의 명령어까지 이 단계에서 계산할 수 있다.



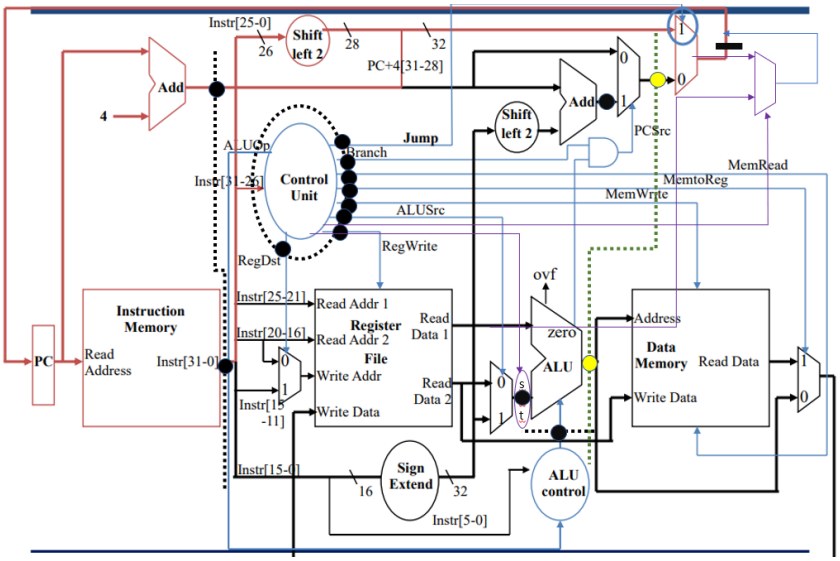
③ Sector 3: Register read/Branch addr

ALU의 input들을 준비한다. Register들을 읽고, imm이나 offset의 경우 extension을 해서 ALU의 두 번째 값에 넣어준다. Beq, Bne의 경우 branch할 addr도 계산해 준다.



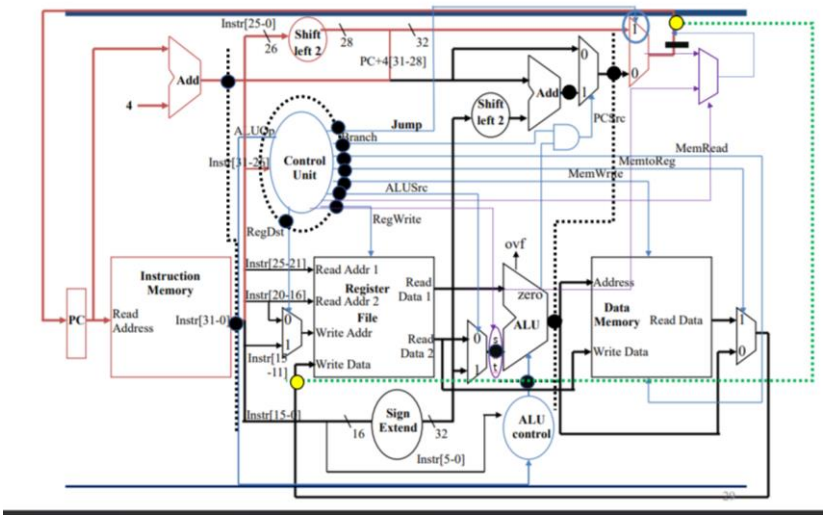
④ Sector 4: ALU

ALU 계산을 진행하고, beq/bne에 따라 PCSrc를 구해 mux에 넣어 값을 받아낸다.



⑤ Sector 5,6,7: PC calculation, Memory, Register Write

JR signal과 합쳐 최종 주소를 계산하고, (Sector 5) MemWrite Signal에 따라 Memory에 Write를 진행하고, MemtoReg/MemRead signal을 받아 mux에 넣는다(Sector 6). 그 뒤 RegWrite signal에 따라 Register에 데이터를 넣는다(Sector 7)



2. 과제의 컴파일 방법 및 컴파일 환경

Ubuntu 20.04 환경에서 g++ 9.4.0 버전을 이용하여 컴파일 하였다. Binary.h, binary.cpp, memory.h, memory.cpp, hw2_201911050.cpp, makefile을 같은 폴더에 넣고 커맨드에 make를 입력하면 output file 인 runfile 이 나온다. (실행 가능 파일)

Makefile의 구조는 다음과 같다.

Binary와 memory, main(hw2_201911050) file을 컴파일 한 뒤 링크해 준다.

```
CC = g++
OBJS = binary.o memory.o hw2_201911050.o
TARGET = runfile

runfile: $(OBJS)
	$(CC) $(OBJS) -o runfile

memory.o : memory.h memory.cpp
	$(CC) -c memory.cpp
binary.o : binary.h binary.cpp
	$(CC) -c binary.cpp
hw2_201911050.o: hw2_201911050.cpp binary.h
	$(CC) -c hw2_201911050.cpp

.PHONY: clean
clean:
	rm -f $(OBJS)
```

Makefile

3. 과제의 실행 방법 및 실행 환경

컴파일해서 나온 파일인 runfile을 실행한다. 실행 또한 Ubuntu 20.04 환경에서 진행하였다.

과제 메뉴얼에 적힌 대로,

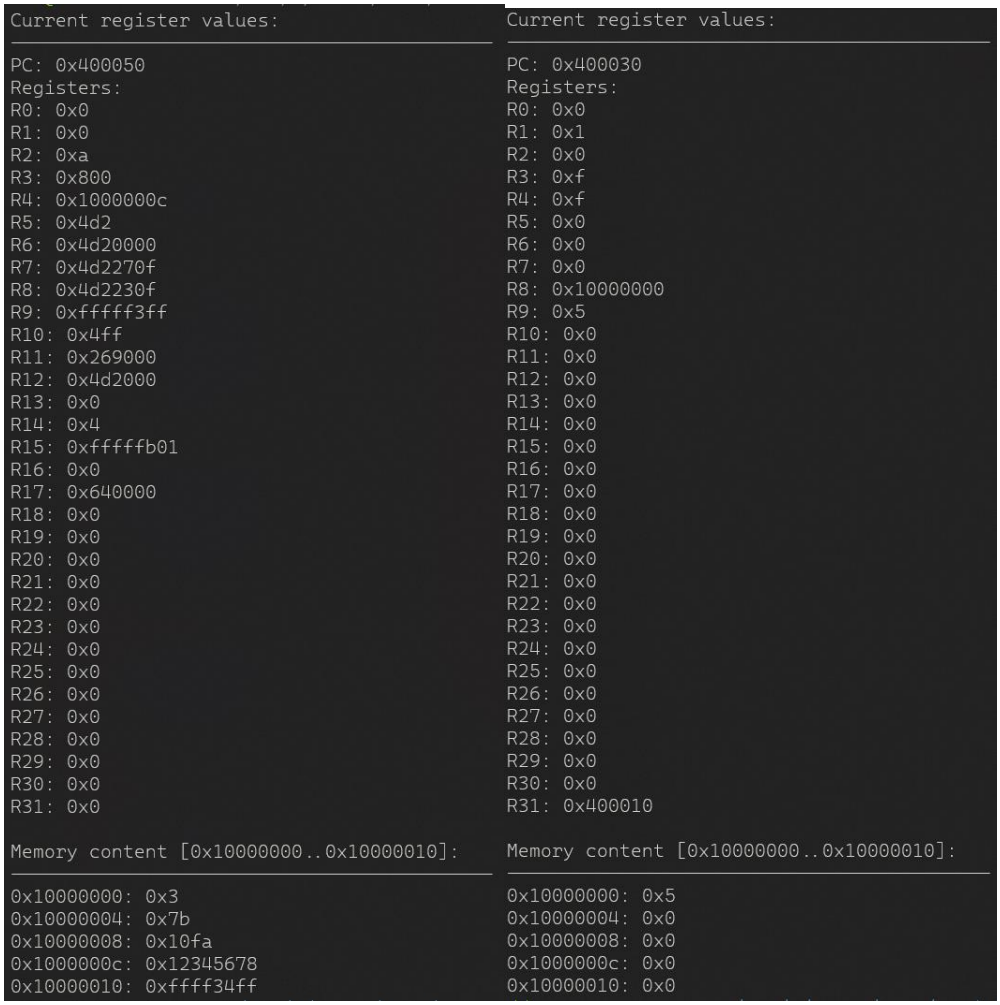
./runfile [-m addr1:addr2] [-d] [-n num_instruction] <input file> 으로 실행한다.

-m addr1:addr2를 입력하면 instruction들이 끝난 후 그 메모리를 콘솔에 출력한다.

-d를 입력하면 매 instruction마다 register 값을 형식에 맞춰 출력한다. 이때, -m이 있다면 메모리도 출력한다. -d가 없다면 모든 instruction이 끝나고 마지막 상태만 출력한다.

-n num_instruction이 있으면 **num_instruction**만큼만 실행한다.

Input file의 이름을 받아 그 파일을 실행한다**.**



Sample.o **Sample2.o**

실행 결과는 다음과 같다, -m option만 킨 상태이다.