Project 2

Single cycle

Prepared by ChoJungHee

32194394@dankook.ac.kr

개요

이 프로젝트의 목표는 MIPS ISA(Instruction Set Architecture)을 사용하여 Single Cycle을 수행하는 C코드를 구현하는 것이 목표이다.

단 MIPS ISA의 모든 Instruction을 구현하는 것이 아니라 FPU와 high, low 레지스터를 이용하는 명령어는 제외한, 간단한 Single Cycle을 구현하는 것이다.

이 프로젝트에서 Single Cycle은 폰 노이만의 구조를 따라간다.

간략한 프로그램의 전체적인 흐름은 다음과 같다.

1. 장치 초기화 및 기본값 설정
2. 프로그램 로드
3. Single Cycle
    1. Fetch
    2. Decode
    3. Execute
    4. Memory Operation
    5. Writeback

이 보고서는 5가지 내용으로 구성되어 있다.

1. 프로젝트에서 사용되는 개념과 배경지식.
2. 프로젝트가 어떻게 디자인 되어있는지 서술.
3. 프로젝트를 어떤환경에서 어떠한 방식으로 빌드 했는지에 대한 정보를 제공.
4. 프로젝트의 실행 결과
5. 프로젝트를 하며 느꼈던 점과 힘들었던 점.

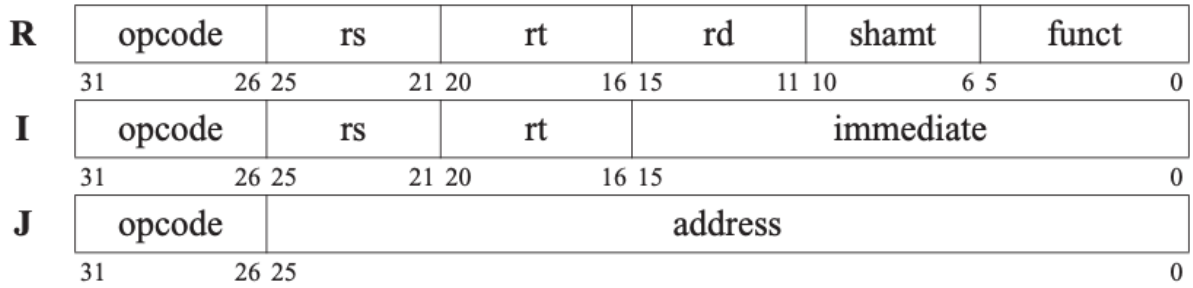
1. 배경

1.1 주요 개념

MIPS

- MIPS : Microprocessor without Interlocked Pipeline Stage
- RISC 구조 채용 : ReducedInstructionSetComputer.
    - 명령어의 개수가 적고 단순한 명령어로 이루어짐.
    - 명령어는 고정길이를 가짐.
    - 복잡한 연산도 기존의 적은 수의 명령어들의 조합으로 만들어 사용.
    - 명령어의 개수가 적고 길이가 고정되어 있기 때문에 해석시간이 상대적으로 짧음.
- MIPS-32 ISA : 32bit
- 명령어 체계 : 3가지의 타입을 가짐

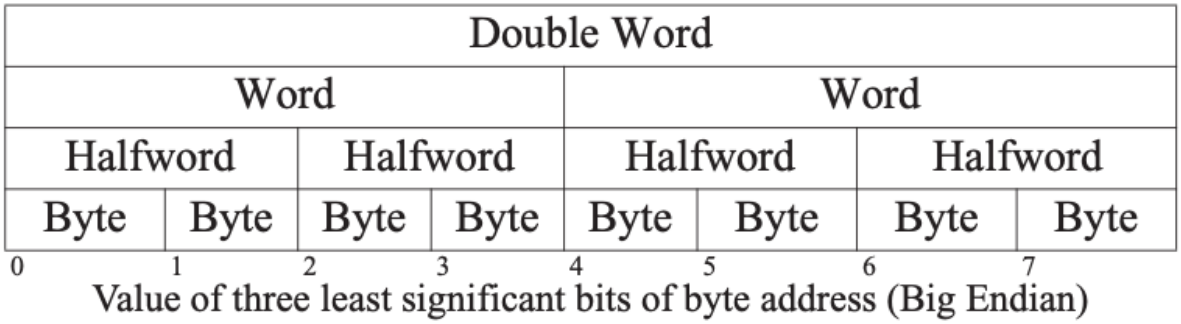
BASIC INSTRUCTION FORMATS



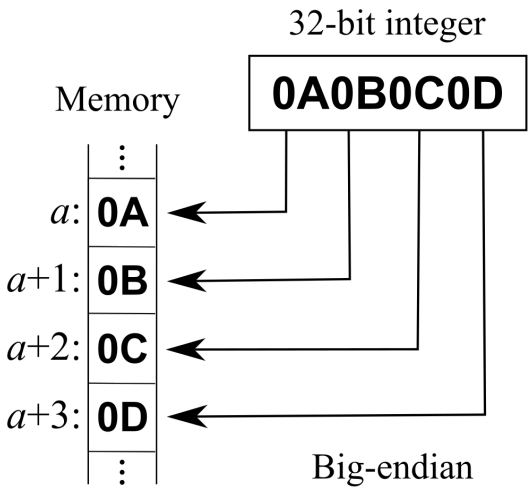
    - R type : **R**egister Type : 레지스터를 이용한 연산을 하는 명령어이다. opcode가 0이며 연산의 종류는 funct로 구분한다.
    - I type : **I**mmediate Type : 레지스터와 imm을 이용한 연산을 하는 명령어이다. imm은 명령어의 요구에 맞게 32비트로 확장되어 연산된다.
    - J type : **J**ump Type : pc레지스터의 값을 원하는 지점으로 점프시키기 위한 명령어이다. pc값은 4배수로 증가하기에 address값은 좌 쉬프트를 두번 진행한다.

- 데이터 저장 방식

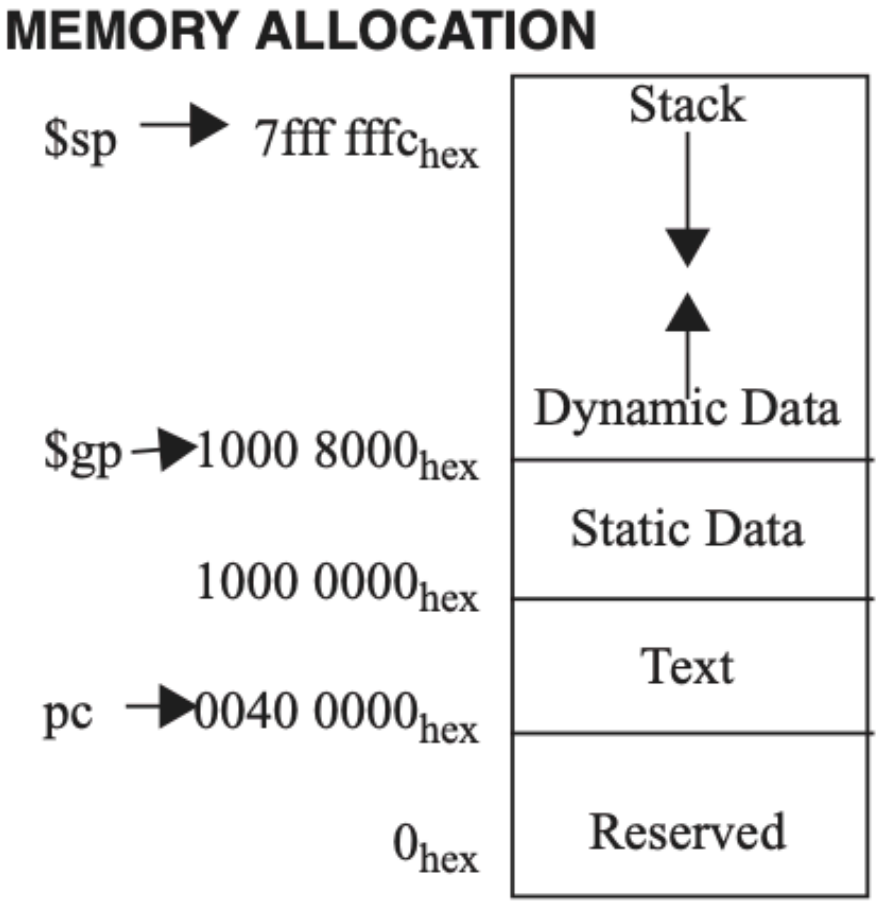
DATA ALIGNMENT



- Big endian 방식, Word = 4bytes

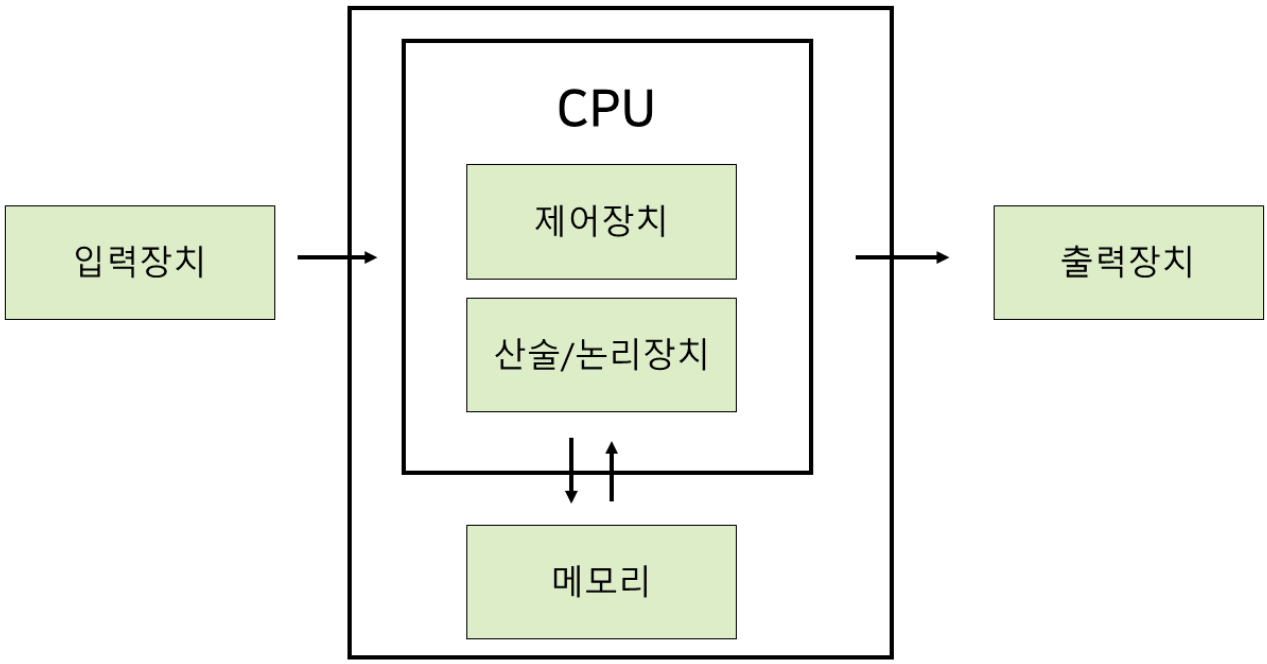


- 메모리 구조



- Text(instruction)와 stack(data)이 같은 메모리에 존재하기에 폰노이만 구조.

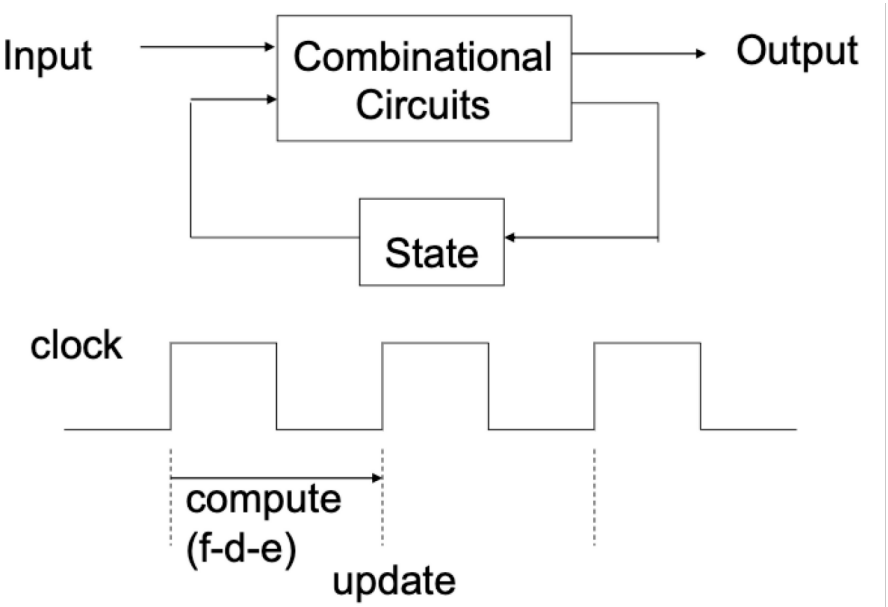
Von Neumann architecture



폰 노이만 구조 CPU, Memory, Program, I/O device 라는 큰 요소로 이루어져있다. 이 구조는 프로그램을 메모리에 내장(store)시켜 순차처리 하는 방식으로 데이터 메모리와 명령어 메모리가 따로 존재하지않고 하나의 버스만을 사용하는 메모리를 가진다.

Single Cycle

CPU는 내부의 존재하는 회로를 작동시키기 위해 일정한 주기를 가지는 신호를 발생시킨다. 이를 Clock이라고 한다.



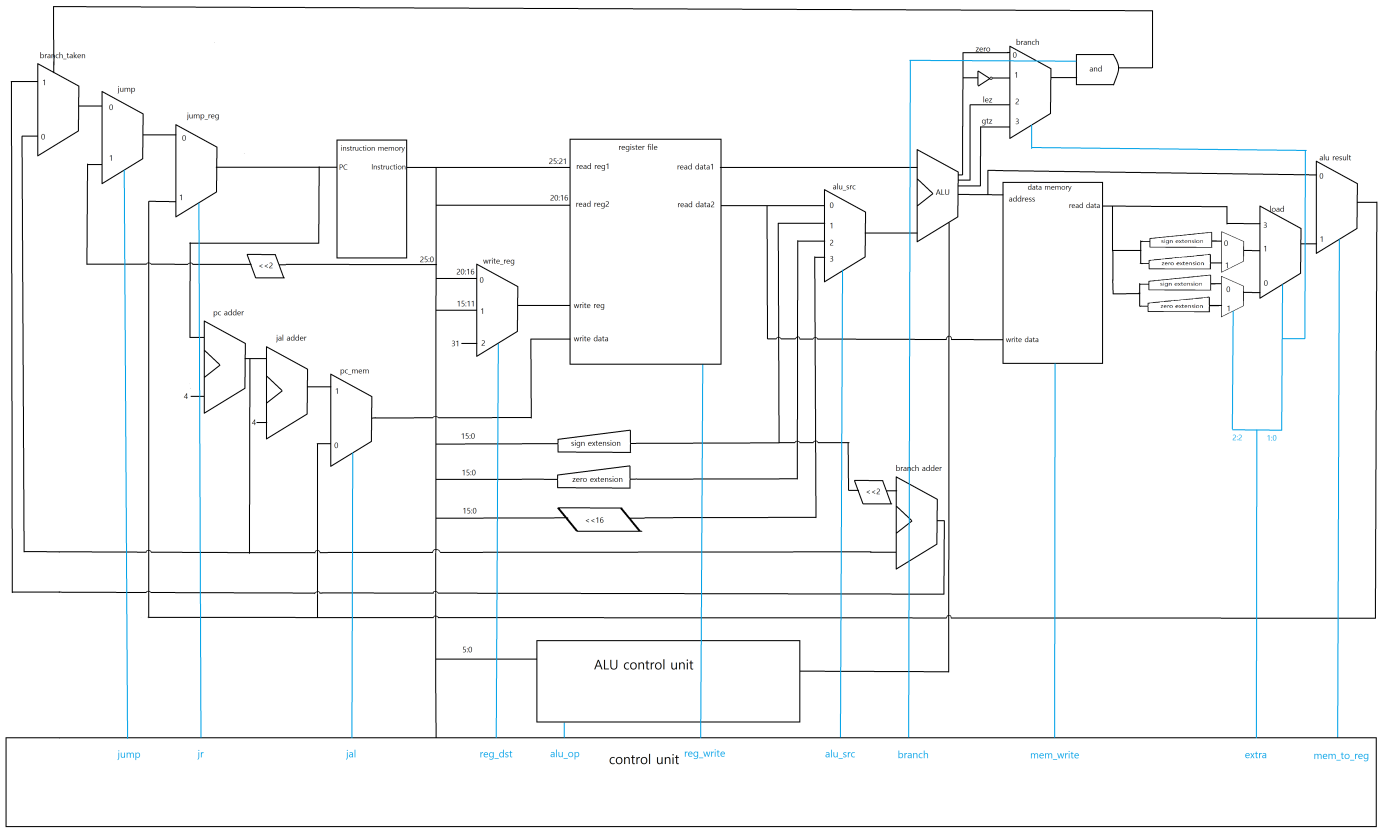
Clock신호가 상승한후 다음 Clock신호가 상승하기 전까지를 1 Clock Cylcle이라 한다.

Single Cycle이란 1 Clock Cycle안에 하나의 명령어를 처리 시키는 방식이다. 다음 Clock Cycle에는 새로운 명령어를 처리해야 한다.

2. 구현

2.1 설계

2.1-1 데이터 패스



이번 과제를 수행하기 위해 만든 Single Cycle의 datapath 이다. 이 datapath에서 장치들을 분리하여 구현했다. Alu control unit, Control unit, ALU, register, memroy로 파일을 분리하였으며 이 장치들을 모아 연결하여 기능을 가지는 CPU 파일이 존재한다. 이 CPU파일을 메인 파일에서 include하여 프로그램의 동작을 구현했다.

2.1-2 Control Unit signal

- reg_dst

  레지스터 파일의 write register는 이 신호를 통해 결정된다. R타입 명령어는 rd , 이외의 명령어들은 rt, jal(r)에 대해서만 31(ra)

- reg_write

  레지스터 파일의 write register에 write data를 저장 여부를 정하는 신호이다. Branch(beq, bne, ... etc) 명령어, Jump (j), store(sw)등의 레지스터에 결과를 저장할 필요가 없는경우에만 신호가 0이다.

- jump

  j , jal 와 같은 J타입 명령어 일 경우 신호가 활성화 된다. 활성화 된다면 명령어의 25:0 (address)<<2 로 pc값을 수정한다.

- jal

  jal, jalr 같은 경우 ra에 pc+8을 저장시켜야 하기에 그것을 결정하는 신호이다.

- jr

점프와 관련된 명령어중 jr일 경우 레지스터값으로 점프를 해야하기에 그것을 결정하는 신호이다.

- alu_src

  alu에서 연산을 하기위한 input2를 결정하기 위해 사용된다.

  - R타입, branch : 0 = 레지스터파일 read data2
  - 메모리 입출력(load, store), 산술연산 : 1 = 명령어 15:0 (imm)의 sign extension
  - 논리연산(and or nor xor) : 2 = 명령어 15:0 (imm)의 zero extension
  - Lui : 3 = 명령어 15:0 << 16

- alu_op

  alu에서 excute단게에서 사용 할 alu_control을 만들기 위해 1차적으로 opcode들이 수행하는 연산들에 대한 신호이다.

alu op	**Mean**
000	add (load, store, addi, addiu)
001	sub (branch)
010	and (andi)
011	or (ori)
100	xor (xori)
101	slt (slti, sltiu)
110	funct (R type)
111	no alu operation (j, jal)

- mem_to_reg

  결과로 alu result를 사용할지 load해온 값을 쓸지 정하는 신호이다.

- mem_write

  store연산에 사용되는 신호로 활성화 되어 있다면 메모리에 값을 저장한다.

- branch

  분기 명령어 (beq, bne, ... etc)일 경우 활성화 되는 신호이다. 분기 성공여부에 관여하는 신호이다.

- extra

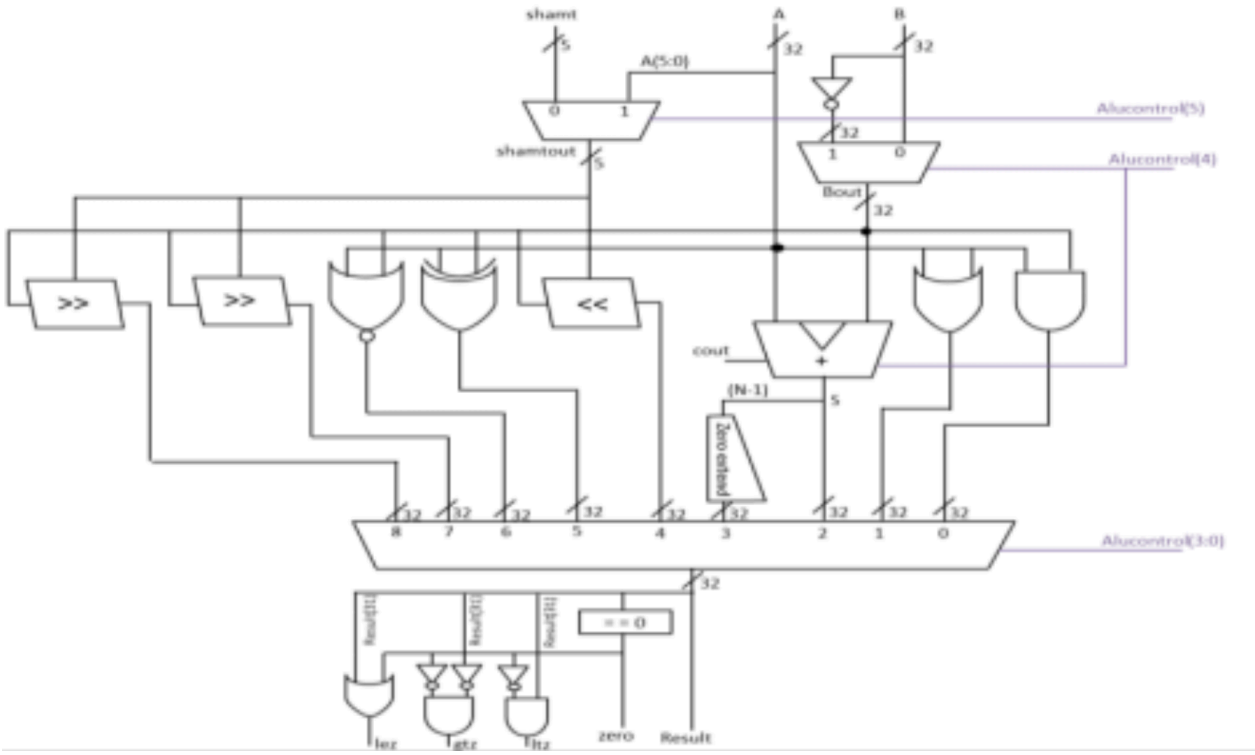
  특정 명령어들의 세부 분리(branch 명령어 종류 구분, store명령어 저장 바이트 수, load 명령어 읽을 바이트 수)를 위해 사용하는 3비트 신호이다.

2.1-3 ALU Control Unit signal

ALU Control Unit signal은 ALU에서 연산을 결정짓기 위해 사용하는 시그널로 ALU Control Unit에서 alu op와 명령어의 funct를 이용하여 생성한다.

alu control signal	mean
000000	And
000001	Or
000010	Add
000100	Sll
000101	Xor
000110	Nor
000111	Srl
001000	Sra
010010	Sub
010011	Slt
100100	Sllv
100111	Srlv
101000	Srav

2.1-4 ALU



2.2 코드 구현

구현 설명에 앞서 함수나 구조체를 보면 각 장치별 input과 output이 함수의 인자와 리턴값으로 있는게 아닌 구조체 내부의 멤버로서 존재하는데, 함수 인자와 리턴값을 안쓰고 저런 방식을 채택한 이유는 각 장치가 정말 서로 연결되어 있게 만들고 싶었기 때문이다. 물론 코드 구현 실력과 설계상의 이유(mux를 구조체를 사용하지 않고 배열로 사용했다가 out을 저장하는 곳이 없어 연결이 불가능 해짐)가 있어 모든 입출력을 연결시키지는 못했다.

---

acu (alu control unit)

```
typedef struct _ALU_CU{
    unsigned int* alu_op;
    unsigned int* funct;
    unsigned int* jal;
    unsigned int* jr;
    unsigned int alu_control;
}ALU_CU;

void ACU_operation(ALU_CU* acu);
```

alu_op와 jal, jr은 Control Unit 구조체에 존재하는 신호의 포인터값을 할당한다.

funct는 register 구조체 내부 IR 구조체의 funct 변수의 포인터값을 할당한다.

ACU_operation 함수 Control Unit 구조체의 alu_op 값을 이용하여 alu_control을 만든다.

---

alu

```
typedef struct _ALU_flag{
    int zero;
    int negative;
    int gtz;
    int lez;
    int carry;
    int overflow;
}ALU_flag;

typedef struct _ALU{
    unsigned int* alu_control;
    unsigned int* input1;
    unsigned int* input2;
    unsigned int* shamt;

    unsigned int input2_mux[2];
    unsigned int shamt_mux[2];
    unsigned int out_mux[9];
    ALU_flag alu_flag;
    unsigned int result;
}ALU;
```

```
void ALU_operation(ALU* alu, unsigned int* mux_out);
```

alu_control 은 alu control unit 의 alu control 변수의 포인터값을 할당한다.

input1은 레지스터 파일의 read data2 변수의 포인터 값을 할당하며 shamt값은 IR레지스터의 shamt변수의 포인터값을 할당한다.

input2는 ALU_operation함수의 인자인 mux_out값으로 할당한다.

mux_out 위치에 alu_src_mux의 결과가 들어간다.

ALU_operation은 alu control값에따라 input1과 input2를 연산한 결과를 result변수에 저장하는 함수이다.

세부 구현은 2.1-4의 흐름을 따라갔다.

---

cu

```
typedef struct _CU{
    unsigned int reg_dst;
    unsigned int reg_write;

    unsigned int jump;
    unsigned int jal;
    unsigned int jr;

    unsigned int alu_src;
    unsigned int alu_op;

    unsigned int mem_to_reg;
    unsigned int mem_write;

    unsigned int branch;
    unsigned int extra;

    unsigned int* opcode;
}CU;

void CU_operation(CU* cu);
```

opcode는 IR레지스터의 opcode변수의 포인터값을 할당한다.

CU_operation함수를 통해 opcode값에 따라 시그널값들이 변경된다.

---

register

```
typedef struct _register_file{
    unsigned int* read_reg1;
    unsigned int* read_reg2;
    unsigned int* write_reg;
    unsigned int write_data;
    unsigned int read_data1;
    unsigned int read_data2;
    unsigned int* reg_write;
}Register_file;

typedef struct _Instruction_register{
    unsigned int data;
    unsigned int opcode;
    unsigned int rs;
    unsigned int rt;
    unsigned int rd;
    unsigned int shamt;
    unsigned int funct;
    unsigned int imm;
    unsigned int addr;
}Ins_reg;

typedef struct _reg{
    unsigned int PC;
    Ins_reg IR;
    unsigned int gp_reg[32];
    Register_file reg_file;
}Register;

void Reg_file_operation(Register* reg, unsigned int* mux_out);
```

레지스터 파일의 read_reg1과 read_reg2는 각각 IR레지스터의 rs ,rt 의 포인터값을 할당한다.

레지스터 파일의 write_reg는 Reg_file_operation함수의 인자인 mux_out값을 할당한다.

mux_out값은 write_reg_mux의 결과이다.

레지스터 파일의 reg_write는 control unit의 reg_write 변수의 포인터값을 할당한다.

Reg_file_operation함수를 통하여 레지스터파일의 read_data1과 read_data2의 값을 gp_reg[*read_reg1] 과 gp_reg[*read_reg2]값으로 변경한다.

---

memory

```
typedef struct _Memory{
    char* mem;
    unsigned int text_area_start;
    unsigned int text_area_end;
```

```
    unsigned int data_area_start;
    unsigned int data_area_end;
}Memory;

void init_memory(Memory* memory,unsigned int size,int text_area_start,int
text_area_end,int data_area_start,int data_area_end);
int load_program(Memory* memory,char* dir);
int load_word(unsigned int address, char* mem);
unsigned int fetch_ins_mem(Memory* memory,unsigned int address);
unsigned int fetch_data_mem(Memory* memory,unsigned int address);

void store_data_mem(Memory* memory,unsigned int address,unsigned int value,unsigned int
count);
```

init함수를 통하여 메모리의 크기와 명령어 영역의 범위와 데이터 영역의 범위를 결정할 수 있다.

load program 함수를 통하여 dir 위치에 존재하는 bin파일을 메모리에 로드한다. 만약 리턴값이 1이면 파일입출력 실패이다.

load word 함수를 통하여 1바이트 배열에서 4바이트 값을 읽어온다.

fetch_ins_mem함수와 fetch_data_mem함수는 각자의 영역에서 해당하는 주소의 값을 가져오는 함수로 해당영역 밖이면 가져오지 않는다.

store_data_mem함수는 해당하는 주소에 값을 저장하는 함수로 1,2,4바이트를 저장할 수 있으며 데이터 영역 밖이라면 저장되지 않는다.

---

cpu

```
typedef struct _CPU{
    Register reg;
    ALU alu;
    ALU_CU acu;
    CU cu;

    unsigned int* write_reg_mux[3];
    unsigned int* alu_src_mux[4];
    unsigned int alu_result_mux[2];
    unsigned int load_mux[4];
    unsigned int pc_mem_mux[2];
    unsigned int branch_mux[4];
    unsigned int branch_taken_mux[2];
    unsigned int jump_mux[2];
    unsigned int jump_reg_mux[2];

    unsigned int pc_adder;
    unsigned int branch_adder;
    unsigned int jal_adder;
```

```
    unsigned int zero_extened_data;
    unsigned int sign_extened_data;
    unsigned int lui_shifted_data;

}CPU;


void init_cpu(CPU* cpu);
void fetch(CPU* cpu, Memory* mem);
void decode(CPU* cpu);
void execute(CPU* cpu);
void memory_operation(CPU* cpu, Memory* mem);
void writeback(CPU* cpu, Memory* mem);
void PC_operation(CPU* cpu);
```

cpu는 모든 장치가 모이는 곳으로 모든 unit과 adder,mux들이 멤버로 존재한다.

init_cpu함수는 위의 장치들을 연결하고 값을 초기화 하는 함수이다.

- fetch

```
void fetch(CPU* cpu, Memory* mem){
    cpu->reg.IR.data = fetch_ins_mem(mem, cpu->reg.PC);

    cpu->pc_adder = cpu->reg.PC + 4;
}
```

메모리로 부터 명령어를 읽어 IR레지스터에 저장하고 pc_adder 값을 설정한다.

- decode

```
void decode(CPU* cpu){
    R_type tmp_ins_R = *((R_type*)&(cpu->reg.IR.data));
    I_type tmp_ins_I = *((I_type*)&(cpu->reg.IR.data));
    J_type tmp_ins_J = *((J_type*)&(cpu->reg.IR.data));

    cpu->reg.IR.opcode = tmp_ins_R.opcode;
    cpu->reg.IR.rs = tmp_ins_R.rs;
    cpu->reg.IR.rt = tmp_ins_R.rt;
    cpu->reg.IR.rd = tmp_ins_R.rd;
    cpu->reg.IR.shamt = tmp_ins_R.shamt;
    cpu->reg.IR.funct = tmp_ins_R.funct;
    cpu->reg.IR.imm = tmp_ins_I.imm;
    cpu->reg.IR.addr = tmp_ins_J.addr;

    CU_operation( &(cpu->cu) );
    ACU_operation( &(cpu->acu) );
    Reg_file_operation(  &(cpu->reg) , (cpu->write_reg_mux[ cpu->cu.reg_dst ]) );
```

```
    cpu->sign_extened_data = cpu->reg.IR.imm & 0x8000 ? cpu->reg.IR.imm |
0xffff0000 : cpu->reg.IR.imm & 0x0000ffff;
    cpu->zero_extened_data = cpu->reg.IR.imm & 0x0000ffff;
    cpu->lui_shifted_data = cpu->reg.IR.imm<<16;

    cpu->branch_adder = cpu->pc_adder + (cpu->sign_extened_data << 2);
}
```

명령어를 해석하여 시그널들을 만들어낸다. 레지스터 파일의 입출력을 조정하며 extension과 shiter값을 미리 설정해놓는다.

- execute

```
void execute(CPU* cpu){
    ALU_operation( &(cpu->alu) , cpu->alu_src_mux[ cpu->cu.alu_src ]);
}
```

신호들과 입력값을 이용하여 ALU연산을 진행한다.

- memory operation

```
void memory_operation(CPU* cpu, Memory* mem){
    if(cpu->reg.IR.data == 0)
        return;
    if(cpu->cu.mem_write)
        store_data_mem(mem, cpu->alu.result, cpu->reg.reg_file.read_data2, cpu-
>cu.extra);
}
```

만약 mem_write신호, 즉 메모리에 데이터를 저장해야한다면 함수를 호출하여 메모리에 값을 저장한다

이때 메모리에 값을 얼마만큼 저장할지 (sw ,sh, sb)에 대한 구분은 control unit의 extra 신호를 이용한다.

- writeback (with pc operation)

```
void writeback(CPU* cpu, Memory* mem){
    if(cpu->reg.IR.data == 0){
        PC_operation(cpu);
        return;
    }

    unsigned int lw = fetch_data_mem(mem, cpu->alu.result);
    unsigned int lh = (((cpu->cu.extra)>>2)&1) ? ((int)lw >> 16) : (lw >> 16);
    unsigned int lb = (((cpu->cu.extra)>>2)&1) ? ((int)lw >> 24) : (lw >> 24);

    cpu->load_mux[0] = lb;
    cpu->load_mux[1] = lh;
    cpu->load_mux[2] = 0;
    cpu->load_mux[3] = lw;
```

```
    unsigned int load_mux_out = cpu->load_mux[ (cpu->cu.extra & 0x3) ];

    cpu->alu_result_mux[0] = cpu->alu.result;
    cpu->alu_result_mux[1] = load_mux_out;

    unsigned int alu_result_mux_out = cpu->alu_result_mux[ cpu->cu.mem_to_reg ];

    cpu->jal_adder = cpu->pc_adder + 4;

    cpu->pc_mem_mux[0] = alu_result_mux_out;
    cpu->pc_mem_mux[1] = cpu->jal_adder;

    unsigned int pc_mem_mux_out = cpu->pc_mem_mux[ cpu->cu.jal ];
    cpu->reg.reg_file.write_data = pc_mem_mux_out;

    if( *(cpu->reg.reg_file.reg_write) && *(cpu->reg.reg_file.write_reg)){
         cpu->reg.gp_reg[ *(cpu->reg.reg_file.write_reg) ] = cpu-
>reg.reg_file.write_data; //memory write
    }

    PC_operation(cpu);
}
```

```
void PC_operation(CPU* cpu){
    cpu->branch_mux[0] = cpu->alu.alu_flag.zero;
    cpu->branch_mux[1] = !cpu->alu.alu_flag.zero;
    cpu->branch_mux[2] = cpu->alu.alu_flag.lez;
    cpu->branch_mux[3] = cpu->alu.alu_flag.gtz;

    unsigned int branch_mux_out = cpu->cu.branch ? cpu->branch_mux[cpu->cu.extra &
0x3] : 0; //pcsrc

    cpu->branch_taken_mux[0] = cpu->pc_adder;
    cpu->branch_taken_mux[1] = cpu->branch_adder;
    unsigned int branch_taken_mux_out = cpu->branch_taken_mux[ branch_mux_out ];

    cpu->jump_mux[0] = branch_taken_mux_out;
    cpu->jump_mux[1] = ((cpu->pc_adder)&0xf0000000) | (cpu->reg.IR.addr << 2);
    unsigned int jump_mux_out = cpu->jump_mux[ cpu->cu.jump ];

    cpu->jump_reg_mux[0] = jump_mux_out;
    cpu->jump_reg_mux[1] = cpu->alu_result_mux[cpu->cu.mem_to_reg];
    unsigned int jump_reg_mux_out = cpu->jump_reg_mux[ cpu->cu.jr ];
    cpu->reg.PC = jump_reg_mux_out;
}
```

writeback함수에서는 load해온 값을 범용 레지스터에 저장해야하거나 범용 레지스터에 alu_reault를 저장해야하는 경우 control unit의 reg_write신호를 확인하여 범용 레지스터에 값을 저장한다.

이때 PC 레지스터의 업데이트는 PC_operation에서 진행한다.

이때 분기 종류의 구분이나 load명령어의 구분은 control unit의 extra 신호를 이용한다.

빌드 환경

Ubuntu

> Ubuntu 22.04.2 LTS (GNU/Linux 5.15.0-69-generic aarch64)
>
> gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0

M2 MacBook Air

> Apple clang version 14.0.0 (clang-1400.0.29.202)
>
> Target: arm64-apple-darwin22.3.0
>
> Thread model: posix

Make command

> make main -> build the execution program
>
> make clean -> clean the object files

```
jojeonghui@MacBook-AirM2 CA2 % make main
gcc -Wall -c src/alu.c -o src/alu.o
gcc -Wall -c src/acu.c -o src/acu.o
gcc -Wall -c src/cu.c -o src/cu.o
gcc -Wall -c src/memory.c -o src/memory.o
gcc -Wall -c src/register.c -o src/register.o
gcc -Wall -c src/cpu.c -o src/cpu.o
ar rv ./src/cycle.a ./src/alu.o ./src/acu.o ./src/cu.o ./src/memory.o ./src/register.o ./src/cpu.o
ar: creating archive ./src/cycle.a
a - ./src/alu.o
a - ./src/acu.o
a - ./src/cu.o
a - ./src/memory.o
a - ./src/register.o
a - ./src/cpu.o
gcc -Wall -o main main.c -L. ./src/cycle.a
jojeonghui@MacBook-AirM2 CA2 %
```

make main

```
jojeonghui@MacBook-AirM2 CA2 % make clean
rm -f ./src/alu.o ./src/acu.o ./src/cu.o ./src/memory.o ./src/register.o ./src/cpu.o ./src/cycle.a main
jojeonghui@MacBook-AirM2 CA2 %
```

make clean

결과

```
jojeonghui@MacBook-AirM2 CA2 % ./main
choose input file(1~5) :
```

인풋 파일 선택

```
[PC Update] PC <- 0x00000028 = 0x00000024+4

Cycle[2675] (PC: 0x28)
[Fetch instruction] 0x00001025   pc: 0x28
[Decode instruction] type: R
        opcode: 0x0, rs: 0x0 (R[0]=0x0), rt: 0x0 (R[0]=0x0), rd: 0x2 (2), shamt: 0x0, funct: 0x25
[Write Back] R[2] <- 0x0
[PC Update] PC <- 0x0000002c = 0x00000028+4

Cycle[2676] (PC: 0x2c)
[Fetch instruction] 0x03c0e825   pc: 0x2c
[Decode instruction] type: R
        opcode: 0x0, rs: 0x1e (R[30]=0xffffd8), rt: 0x0 (R[0]=0x0), rd: 0x1d (29), shamt: 0x0, funct: 0x25
[Write Back] R[29] <- 0xffffd8
[PC Update] PC <- 0x00000030 = 0x0000002c+4

Cycle[2677] (PC: 0x30)
[Fetch instruction] 0x8fbf0024   pc: 0x30
[Decode instruction] type: I
        opcode: 0x23, rs: 0x1d (R[29]=0xffffd8), rt: 0x1f (R[31]=0x24), imm: 0x24
[Load] R[31] <- Mem[0x00fffffc] = 0xffffffff
[PC Update] PC <- 0x00000034 = 0x00000030+4

Cycle[2678] (PC: 0x34)
[Fetch instruction] 0x8fbe0020   pc: 0x34
[Decode instruction] type: I
        opcode: 0x23, rs: 0x1d (R[29]=0xffffd8), rt: 0x1e (R[30]=0xffffd8), imm: 0x20
[Load] R[30] <- Mem[0x00fffff8] = 0x0
[PC Update] PC <- 0x00000038 = 0x00000034+4

Cycle[2679] (PC: 0x38)
[Fetch instruction] 0x27bd0028   pc: 0x38
[Decode instruction] type: I
        opcode: 0x9, rs: 0x1d (R[29]=0xffffd8), rt: 0x1d (R[29]=0xffffd8), imm: 0x28
[Write Back] R[29] <- 0x1000000
[PC Update] PC <- 0x0000003c = 0x00000038+4

Cycle[2680] (PC: 0x3c)
[Fetch instruction] 0x03e00008   pc: 0x3c
[Decode instruction] type: R
        opcode: 0x0, rs: 0x1f (R[31]=0xffffffff), rt: 0x0 (R[0]=0x0), rd: 0x0 (0), shamt: 0x0, funct: 0x8
[PC Update] PC <- 0xffffffff = R[31]: 0xffffffff


==========================PROGRAM RESULT============================
Return value (R[2]) :                   0
Total Cycle :                           2680
Executed 'R' instruction :              547
Executed 'I' instruction :              1752
Executed 'J' instruction :              109
Number of Branch Taken :                109
Number of Memory Access Instruction :   1095
====================================================================
jojeonghui@MacBook-AirM2 CA2 %
```

결과

느낀점

이번 과제를 하면서 생각보다 어려운 점이 많았다. 구조체 비트필드를 사용하여 구조를 짜려하니 비트필트 변수는 포인터값 지정이 안되어 처음에 의도한 장치 연결에 부합하지 않았고 장치들이 조건에 따라 필요한 길을 찾아가는것이 아닌 일단 모든 연산을 해놓고 MUX를 통해 원하는 값을 빼오는것을 구현하다 보니 어려운점이 많았다(명령어의 종류와 상관없이 writeback 단계에서 메모리를 읽기때문에 위치가 읽는 위치가 메모리 최대치보다 높을 경우 segfault11 에러가 발생했다. 메모리 범위 연산과 메모리 크기를 늘리는것으로 수정함). 보완해야 할점또한 많은데 예외처리가 만족스럽지 못했다. 특히나 datapath에 대한 의문점이 들었는데 j타입 명령어들은 이론적으로는 alu연산을 하지 말아야하는데 datapath 그림으로는 alu로 가는 길을 막지도 않을 뿐더러 alu에서 j타입을 감지해서 스킵하지 않기때문에 나중에 mux에서 어차피 alu연산결과를 안쓰기 때문에 상관없는건지, 아니면 애초부터 alu 연산을 막아야하는지에 대한 고민이 들었다. 이 프로젝트에서의 결론은 alu 연산결과를 j타입에서 결과적으로 사용하지 않기때문에 연산을 막아두지는 않았다.

난이도와는 상관없이 대학교에 들어와서 받은 과제중 가장 재미있고 꽤나 오래걸린(구조를 많이 갈아엎음) 과제였다.