컴퓨터 구조 Project2_Report

컴퓨터학부 심화컴퓨터전공
2020114658 이윤서

기본 구조 설계

1. 비트 연산자를 활용하여, **instruction** 을 **decode** 하자.
    a. 이러면 16 진수만을 활용하여 **simulator** 를 완성할 수 있다.
2. 주어진 연산 (**add, addi, jal, jalr, beq, sd, ld**) 에서 필요 없는 **funct3**, **funct7** 부분은 **decode** 에서 제외한다.
3. **immediate** 값을 **decode** 할 때 주의하자.
    a. 보수 처리, **sign-extended** 가 필요하다.
        i. **AND** 연산을 통해 제일 앞에 위치한 부호 비트를 알아내고,
        음수라면 **OR** 연산으로 1 을 채워준다.
        ii. **imm** 은 **int** 형 (**4byte**)로 선언하고 그에 맞게 **1** 을 채워준다.
    b. **1 bit shift right** 를 잊지 말자.
        i. **UJ-type, SB-type** 은 **imm** 값을 넣을 때 1 비트를 남겨두고 **shift** 연산한다.
    c. 인덱스 순서에 주의하자.
        i. **UJ-type, SB-type** 은 순서가 뒤죽박죽이다. **[1]** 부터 차례대로 넣자.
4. **imm** 을 PC 연산에 사용하는 **instruction** 일 시에, **sizeof(int)** 로 나누어 **PC** 와 크기를 맞춰줘야 한다.
    a. PC 를 인덱스 겸용으로 사용하여 **debug** 할 때에 편리하게 만들자.
    b. 출력할 때엔 **pc * 4** 를 하여 **byte** 단위로 출력한다.

코드 리뷰

1. 변수 선언

```
// for saving each instruction
long instruction, opcode, rs1, rs2, rd;
int imm;
```

decode 하기 위한 변수를 선언한다.

2. fetch 함수 구현

```
// fetch an instruction from a instruction memory
void fetch()
{
    instruction = inst_mem[pc]; // fetch an instruction
    if (instruction == 0)       // if there is no more instruction, exit
        exit(0);
    pc++; // increase Program Counter
}
```

pc 값이 가리키는 instruction 을, .hex 파일의 값들을 저장한 inst_mem 배열에서 가져와 instruction 변수에 넣음.
이때의 instruction 에는 16 진수 값이 들어있다.
만약 instruction 에 0 이 들어있다면, (전역변수로 선언한 배열은 모두 0 으로 초기화가 되기 때문) 더 이상의 instruction 이 없다는 뜻이므로 프로그램을 종료한다.
실행되어야 할 instruction 이 변수에 들어갔다면 pc 값을 1 증가시킨 뒤 decode 함수로 넘어간다.

3. decode 함수 구현

```
imm = 0;                                // initialize immediate value
opcode = instruction & 0x7f;            // 0x7f = 1111111'b
rd = (instruction & 0xf80) >> 7;        // 0xf80 = 111110000000'b
rs1 = (instruction & 0xf8000) >> 15;    // 0xf8000 = 11111000000000000000'b
rs2 = (instruction & 0x1f00000) >> 20; // 0x1f00000 = 1111100000000000000000000'b
```

각 type 별로 decode 를 해주기 전에 공통인 부분은 각 변수에 넣어둔다.
AND 연산을 하면 1 과 연산 되는 bit 부분만 결과값에 저장이 되고 나머지는 0 으로 저장되기 때문에, instruction 에서 빼내고자 하는 값이 있는 위치가 1 인 16 진수와 AND 연산을 한 후, 자릿수에 맞게 shift 하면 올바르게 저장된다.

immediate 값은 type 별로 위치와 순서가 다르므로 decode 함수에서 0 으로 초기화한 후, switch-case 문을 사용하여 각 opcode 에 맞게 분류하여 저장한다. 또한 지역변수 temp 를 선언하여, jal 연산 때에 imm 값을 편리하게 계산할 수 있도록 하였다.

```
unsigned int temp;

switch (opcode)
{

// add (R-type)
case 0x33:
    break;

// addi (I-type)
case 0x13:
    imm = (instruction & 0xfff00000) >> 20; // make immidiate value

    // if negative value, make 1111 ... (imm) by OR
    if (imm & 0x800)
        imm = imm | 0xfffff000;
    break;
```

이 때, type 별로 저장되는 imm 크기에 따라 imm 의 앞자리가 1 인 경우, 즉 음수가 보수 처리 되어 저장된 imm 일 경우, 부호 확장을 해주어야 한다. OR 연산을 통해, int 자료형의 크기인 4byte 에 맞추어 imm 값 앞을 모두 1 로 채워준다.

```
// jal (UJ-type)
case 0x6f:

    temp = instruction >> 12;
    imm += (temp & 0x7fe00) >> 8; // [10:1]
    imm += (temp & 0x100) << 3;   // [11]
    imm += (temp & 0xff) << 12;   // [19:12]
    imm += (temp & 0x80000) << 1; // [20]

    // if negative value, make 1111 ... (imm) by OR
    if (imm & 0x100000)
        imm = imm | 0xffe00000;
    break;
```

immediate 값이 분리되어 있는 경우, 덧셈 연산을 통해 계산하므로 순서에 맞추어 연산한다.
또한 UJ-type 과 SB-type 은 shift right 1 bit 연산이 있으므로 끝에 한 bit 를 남기고 shift 해야한다.
마지막 bit 를 0 으로 하기 위함이기 때문에, AND 연산에서 이미 0 이 되어있으므로 다른 분기문은 필요없다.

jalr, beq, ld, sd 도 같은 원리로 fetch 를 완료한다.

3. exe 함수 구현

```
switch (opcode)
{
// add (R-type)
case 0x33:
    regs[rd] = regs[rs1] + regs[rs2];
    break;

// addi (I-type)
case 0x13:
    regs[rd] = regs[rs1] + imm;
    break;

// jal (UJ-type)
case 0x6f:
    regs[rd] = pc * 4;
    pc = (pc - 1) + imm / sizeof(int);
    break;
```

```
// jalr (I-type)
case 0x67:
    regs[rd] = pc * 4;
    pc = (regs[rs1] + imm) / sizeof(int);
    break;

// beq (SB-type)
case 0x63:
    if (regs[rs1] == regs[rs2])
        pc = (pc - 1) + imm / sizeof(int);
    break;
}
```

각 opcode 에 맞는 연산을 수행하기 위해 switch-case 문을 사용한다. PC 값 조정에 유의하며 각 case 별로 연산을 수행하는데, 이때에 memory 에 접근을 해야하는 sd / ld 는 다음 함수에서 처리한다.

4. mem 함수 구현

```c
// access the data memory
void mem()
{
    // ld (I-type)
    if (opcode == 0x03)
        regs[rd] = data_mem[(regs[rs1] + imm) / sizeof(int)];
}
```

ld 의 경우 데이터가 저장된 배열에 접근하여 레지스터에 저장하는 연산이기 때문에, mem 함수에서 처리한다.

5. wb 함수 구현

```c
// write result of arithmetic operation or data read from the data memory if required
void wb()
{
    // sd (S-type)
    if (opcode == 0x23)
        data_mem[(regs[rs1] + imm) / sizeof(int)] = regs[rs2];
}
```

sd 의 경우에도 데이터가 저장된 배열에 레지스터 값을 쓰는 연산이기 때문에, wb 함수에서 처리한다.

실행 결과

```
s2020114658@snow-ubuntu:~/ComputerArchitecture$ ./riscv_sim runme.hex 1
-------------------------------------------------
Clock cycles = 140
PC         = 88

x0   = 0
x1   = 12
x2   = 800
x3   = 0
x4   = 0
x5   = 0
x6   = 45
x7   = 0
x8   = 0
x9   = 10
x10  = 55
x11  = 0
x12  = 0
x13  = 0
x14  = 0
x15  = 0
x16  = 0
x17  = 0
x18  = 0
x19  = 0
x20  = 0
x21  = 0
x22  = 0
x23  = 0
x24  = 0
x25  = 0
x26  = 0
x27  = 0
x28  = 0
x29  = 0
x30  = 0
x31  = 0
```