<Project 2. RISC-V simulator 구현하기>

2020112099 송민지

수업 시간에 배운 single cycle 프로그램을 리눅스 개발환경에서 직접 구현해보고자 하였다. single cycle은 크게 다섯 단계의 과정을 거치면서 진행이 되는데, instruction 메모리 상의 명령어를 읽는 Instruction Fetch(IF) 단계, 가져온 instruction을 해석하는 Instruction Decode(ID) 단계, 해석된 instruction을 바탕으로 계산을 하는 Execute(EX) 단계, 데이터 메모리에 접근하는 Memory(MEM) 단계, 메모리에 저장된 값을 읽어오는 Write back(WB) 단계로 분류된다.

싱글 사이클을 통해 많은 instruction을 수행할 수 있지만, 그 중에서 대표적인 몇 가지를 뽑아서 구현하고자 하였다. R-type에 속하는 add, I-type에 속하는 addi, ld, jalr, S-type에 속하는 sd, SB-type에 속하는 beq, UJ-type에 속하는 jal을 구현하였다.

우선 연산을 수행하기 위해 instruction 메모리상의 명령어를 읽어온다. instruction 메모리상의 명령어는 pc(program counter)에 저장이 되어있으며, 이 명령어를 찾아온 후에는 다음 주소를 가리키기 위하여 pc+4를 해준다(IF). 어떤 type의 어떤 연산을 할지 결정하기 위해서는 opcode의 값을 이용하기로 했다. instruction의 젤 아래에 있는 7 비트의 opcode를 10진수로 받아와서 그 값에 따라 특정 연산을 하기로 하였다(ID). <그림 0.1>은 2진수(배열)를 10진수로 받아오는 함수이고, <그림 0.2>는 Instruction Fetch를 나타낸 함수이며, <그림 0.3>는 opcode를 단순히 2진수의 형태로 받아온 것이다.

```
//binary to decimal
int bintodec(int bin[], int cnt)
{
        int cur=0;
        int dec=0;
        for(int i=cnt; i>=0 ; i--)
        {
                if(bin[i]==1)
                        dec += 1 << cur;
                cur++;
        }
        return dec;
}
```

<그림0.1 - 2진수를 10진수로 바꿔주는 함수>

```
void fetch()
{
    x=inst_mem[pc/4];
    pc+=4;
}
```

<그림0.2 - IF>

```
for(int i=25;i<32;i++)
        op[i-25]=bin[i];
```

<그림0.3 - opcode>

1. add

add는 R-type의 연산자로, 레지스터값 두 개(rs1, rs2)를 더하여 또 다른 레지스터(rd)에 저장하는 방법의 연산이다. add의 opcode는 0110011이고, 이를 10진수로 나타내면 51이다. 따라서 opcode가 51이라면 EX에서 add 연산을 수행하도록 한다.

Instruction에서 각 비트 별로 나눠서 위치에 맞춰서 피연산자 rs1과 rs2를 받아오고(ID), 이 받아온 레지스터들을 EX 단계에서 둘이 더해준 후 temp에 저장을 해준다(EX). 이후, temp에 저장을 해준 연산 결과를 write back 단계에서 또 다른 레지스터(rd)에 저장을 해준다(WB). 이때, add 연산에서는 메모리에 대한 접근이 불필요하다. 또한, x0은 레지스터 상에서 상수 0으로 고정되어있기 때문에 x0에는 값이 들어가면 안된다. 따라서 결과 값을 저장할 레지스터(rd)가 0이 아닌 경우에만 이러한 연산이 실행되고, 0인 경우에는 연산값이 저장되지 않고 0이 된다. 아래 그림들은 add 연산에서 수행되는 ID, EX, WB 과정을 나타낸 코드이다.

```
if(bintodec(op,6)==51)
{
        //R-type: add
        //int func[7], rs2[5], rs1[5], func3[3], rd[5];

        for(int i=0;i<7;i++)
                func7[i]=bin[i];

        for(int i=7;i<12;i++)
                rs2[i-7]=bin[i];

        for(int i=12;i<17;i++)
                rs1[i-12]=bin[i];

        for(int i=17;i<20;i++)
                func3[i-17]=bin[i];

        for(int i=20;i<25;i++)
                rd[i-20]=bin[i];
}
```

<그림1.1 - add 연산에서의 ID 과정>

```
if(bintodec(op,6)==51)//add
        temp=regs[bintodec(rs2,4)]+regs[bintodec(rs1,4)];
```

<그림1.2 - add 연산에서의 EX 과정>

```
if(bintodec(op,6)==51 || bintodec(op,6)==19)//add, addi
{
        if(bintodec(rd,4)==0)
                regs[bintodec(rd,4)]=0;
        else
                regs[bintodec(rd,4)]=temp;
}
```

<그림1.3 - add 연산에서의 WB 과정>

2. addi

addi는 I-type의 연산자로, 레지스터 값 하나(rs1)와 immediate 값을 더하여 또 다른 레지스터(rd)에 저장하는 방법의 연산이다. addi의 opcode는 0010011이고, 이를 10진수로 나타내면 19이다. 따라서 opcode가 19라면 EX에서 addi 연산을 수행하도록 한다.

우선 instruction에서 각 비트 별로 나눠서 위치에 맞춰서 피연산자 rs1과 상수 값 immediate를 받아온다, 이때 받아오는 상수 값은 양수가 될 수도 있고 음수가 될 수도 있는데, 만약 받아온 상수 값이 양수일 경우에는 임시 저장소인 tfi에 바로 저장을 해주고, 받아온 상수 값이 음수일 경우, 보수 처리 방식을 이용하여 연산해야 하는데, 0은 1로, 1은 0으로 바꿔준 후, 1을 더해주고, 이 값에 마이너스 부호를 붙여서 임시 저장소인 tfi에 넣어준다(ID). 이러한 tfi값과 rs1을 더하여 temp에 잠시 저장해준다(EX). 이후, temp에 저장된 연산 결과를 write back 단계에서 또 다른 레지스터(rd)에 저장을 해준다(WB). 이때, addi 연산에서도 add 연산에서와 마찬가지로 메모리에 대한 접근이 불필요하며, 값을 저장할 레지스터(rd)가 0이 아니어야 한다. 아래 그림들은 addi 연산에서 수행되는 ID, EX, WB 과정을 나타낸 코드이다.

```
if(bintodec(op,6)==19)
{
    //I-type: addi
    //int imme[12], rs2[5], rs1[5], func3[3], rd[5];

    if(bin[0]==0)
    {
        for(int i=0;i<12;i++)
            imme[i]=bin[i];

        tfi=bintodec(imme,11);
    }

    else if(bin[0]==1)
    {
        for(int i=0;i<12;i++)
        {
            if(bin[i]==0)
                imme[i]=1;
            else if(bin[i]==1)
                imme[i]=0;
        }

        tfi_temp=bintodec(imme,11)+1;
        tfi=-tfi_temp;
    }

    for(int i=12;i<17;i++)
        rs1[i-12]=bin[i];

    for(int i=17;i<20;i++)
        func3[i-17]=bin[i];

    for(int i=20;i<25;i++)
        rd[i-20]=bin[i];
}
```

<그림2.1 - addi 연산에서의 IF 과정>

```
if(bintodec(op,6)==19)//addi
    temp=regs[bintodec(rs1,4)]+tfi;
```

<그림2.2 - addi 연산에서의 EX 과정>

```
if(bintodec(op,6)==51 || bintodec(op,6)==19)//add, addi
{
    if(bintodec(rd,4)==0)
        regs[bintodec(rd,4)]=0;
    else
        regs[bintodec(rd,4)]=temp;
}
```

<그림2.3 - addi 연산에서의 WB 과정>

3. ld

ld는 I-type의 연산자로, 메모리에 저장된 값을 레지스터(rs1)에서 immediate에 가져와서 읽고, 계산 값을 또 다른 레지스터(rd)에 넣는 방식의 연산이다. 주로 sd와 함께 쓰이는 instruction이다. ld의 opcode는 0000011이고, 이를 10진수로 나타내면 3이다. 따라서 opcode가 3라면 EX에서 ld 연산을 수행하도록 한다.

우선 instruction에서 각 비트 별로 나눠서 위치에 맞춰서 피연산자 rs1과 상수 값 immediate를 받아온다, 만약 받아온 상수 값이 양수일 경우, tfi에 상수 값을 바로 저장해주고, 받아온 상수 값이 음수일 경우, 0은 1로, 1은 0으로 바꿔준 후 1을 더해주고, 이 값에 마이너스 부호를 붙인 값을 tfi에 저장해줘야 한다(ID). 이렇게 저장된 tfi를 rs1과 더하여 그 값을 temp에 저장해둔다(EX). 이후, temp에 저장된 결과를 data_mem에 넣고 이 값을 다시 temp로 지정한 후(MEM), 다른 레지스터(rd)에 이 값을 넣어야 한다. 아래 그림들은 ld 연산에서 수행되는 ID, EX, MEM, WB 과정을 나타낸 코드이다.

```
if(bintodec(op,6)==3)
{
    //I-type: ld

    if(bin[0]==0)
    {
        for(int i=0;i<12;i++)
            imme[i]=bin[i];

        tfi=bintodec(imme,11);
    }

    else if(bin[0]==1)
    {
        for(int i=0;i<12;i++)
        {
            if(bin[i]==0)
                imme[i]=1;
            else
                imme[i]=0;
        }
        tfi_temp=bintodec(imme, 11)+1;
        tfi=-tfi_temp;
    }

    for(int i=12;i<17;i++)
        rs1[i-12]=bin[i];

    for(int i=17;i<20;i++)
        func3[i-17]=bin[i];

    for(int i=20;i<25;i++)
        rd[i-20]=bin[i];
}
```

<그림3.1 - ld 연산에서의 ID 과정>

```
if(bintodec(op,6)==3)//ld
    temp=regs[bintodec(rs1,4)]+tfi;
```

<그림3.2 - ld 연산에서의 EX 과정>

```
if(bintodec(op,6)==3)//ld
    temp=data_mem[temp];
```

<그림3.3 - ld 연산에서의 MEM 과정>

```
if(bintodec(op,6)==3) //ld
{
    if(bintodec(rd,4)==0)
        regs[bintodec(rd,4)]=0;
    else
        regs[bintodec(rd,4)]=temp;
}
```

<그림3.4 - ld 연산에서의 WB 과정>

4. sd

sd는 S-type의 연산자로, ld와 반대로 레지스터에 저장된 값을 메모리에 저장하는 방식의 연산이다. sd의 opcode는 0100011이고, 이를 10진수로 나타내면 35이다. 따라서 opcode가 35라면 EX에서 sd 연산을 수행하도록 한다.

우선 instruction에서 각 비트 별로 나눠서 위치에 맞춰서 피연산자 rs1과 rs2, 그리고 상수 값 immediate를 받아온다(ID), 만약 받아온 상수 값이 양수일 경우, tfi에 상수 값을 바로 저장해주고, 받아온 상수 값이 음수일 경우, 0은 1로, 1은 0으로 바꿔준 후 1을 더해주고, 이 값에 마이너스 부호를 붙인 값을 tfi에 저장해줘야 한다(ID). 이렇게 저장된 tfi를 rs1과 더하여 그 값을 temp에 저장해둔다(EX). 이후, temp에 저장된 결과를 data_mem에 넣고 레지스터 값(rs2)을 이 값을 넣어야 한다. sd는 메모리에 저장하는 것이기 때문에 WB 과정을 필요로 하지 않는다. 아래 그림들은 ld 연산에서 수행되는 ID, EX, MEM 과정을 나타낸 코드이다.

```
if(bintodec(op,6)==35)
{
    //S-type: sd
    if(bin[0]==0)
    {
        for(int i=0;i<7;i++)
            imme[i]=bin[i];
        for(int i=20;i<25;i++)
            imme[i-13]=bin[i];
        tfi=bintodec(imme,11);
    }

    else if(bin[0]==1)
    {
        for(int i=0;i<7;i++)
        {
            if(bin[i]=0)
                imme[i]=1;
            else
                imme[i]=0;
        }

        for(int i=20;i<25;i++)
        {
            if(bin[i]=0)
                imme[i-13]=1;
            else
                imme[i-13]=0;
        }

        tfi_temp=bintodec(imme,11)+1;
        tfi=-tfi_temp;
    }

    for(int i=7;i<12;i++)
        rs2[i-7]=bin[i];
    for(int i=12;i<17;i++)
        rs1[i-12]=bin[i];
    for(int i=17;i<20;i++)
        func3[i-17]=bin[i];
}
```

<그림4.1 - sd 연산에서의 ID 과정>

```
if(bintodec(op,6)==35)//sd
    temp=regs[bintodec(rs1,4)]+tfi;
```

<그림4.2 - sd 연산에서의 EX 과정>

```
if(bintodec(op,6)==35) //sd
    data_mem[temp]=regs[bintodec(rs2,4)];
```

<그림 4.3 - sd 연산에서의 MEM 과정>

5. beq

beq는 SB-type의 연산자로, rs1과 rs2가 같다면 다음 pc가 pc+4의 위치가 아니라 pc+immediate의 위치로 이동하는 연산자이다. beq의 opcode는 1100011이고, 이를 10진수로 나타내면 99이다. 따라서 opcode가 99라면 EX에서 beq 연산을 수행하도록 한다.

우선 instruction에서 각 비트 별로 나눠서 위치에 맞춰서 피연산자 rs1과 rs2, 그리고 상수값 immediate를 받아온다. immediate를 받아올 때, pc 점프는 짝수번째로만 가능하기 때문에, 32bit를 64bit로 바꿔주기 위해서 한비트씩 왼쪽으로 이동, 즉 곱하기 2를 해준다. (ID), 받아온 후 rs2의 값과 rs1의 값이 같은지 다른지를 확인하기 위해 둘을 빼준다. 둘을 뺐을 때의 값이 0이라면 두 레지스터 안의 값은 같기 때문에 pc+immediate를 하고, 이 값이 다르다면 pc는 원래대로 pc+4를 해줘야 한다. 여기서 주의할 점은, 현재의 pc를 이용하여 비교해야하기 때문에, 다음 위치를 가리키는 pc가 아닌, pc-4로 해야 한다(EX). beq에서는 값을 저장하는 것이 아닌 pc의 위치를 변경하는 것이므로, MEM, WB 단계가 불필요하다.
아래 그림들은 beq 연산에서 수행되는 ID, EX 과정을 나타낸 코드이다.

```
if(bintodec(op, 6)==99)
{
    //SB-type: beq
    if(bin[0]==0)
    {
        imme[0]=0;
        imme[1]=bin[24];
        for(int i=1;i<7;i++)
            imme[i+1]=bin[i];
        for(int i=20;i<24;i++)
            imme[i-12]=bin[i];
        tfi=bintodec(imme,11)*2;
    }

    else if(bin[0]==1)
    {
        imme[0]=0;
        if(bin[24]==1)
            imme[1]=0;
        else
            imme[1]=1;
        for(int i=1;i<7;i++)
        {
            if(bin[i]==1)
                imme[i+1]=0;
            else
                imme[i+1]=1;
        }

        for(int i=20;i<24;i++)
        {
            if(bin[i]==1)
                imme[i-12]=0;
            else
                imme[i-12]=1;
        }

        tfi_temp=bintodec(imme,11)+1;
        tfi=-(tfi_temp*2);
    }
```

```
for(int i=7;i<12;i++)//24-20
    rs2[i-7]=bin[i];

for(int i=12;i<17;i++)//19-15
    rs1[i-12]=bin[i];

for(int i=17;i<20;i++)
    func3[i-17]=bin[i];
```

<그림5.1 - beq 연산에서의 ID 과정>

```
if(bintodec(op, 6)==99)//beq
{
    temp=regs[bintodec(rs2,4)]-regs[bintodec(rs1,4)];

    if(temp==0)
    {
        pc-=4;
        pc+=tfi;
    }
}
```

<그림5.2 - beq 연산에서의 EX 과정>

6. jal

jal은 UJ-type의 연산자로, x1에 돌아올 pc값, 즉 현재 위치를 저장한 후 pc+immediate의 위치로 이동하는 연산자이다. jal의 opcode는 1101111이고, 이를 10진수로 나타내면 111이다. 따라서 opcode가 111라면 EX에서 jal 연산을 수행하도록 한다.

instruction에서 각 비트 별로 나눠서 위치에 맞춰서 피연산자 rs1과 상수 값 immediate를 받아온다. immediate를 받아올 때, pc 점프는 beq과 마찬가지로 짝수 번째로만 가능하므로, 32bit를 64bit로 바꿔주기 위해서 한비트씩 왼쪽으로 이동, 즉 곱하기 2를 해준다(ID), 이후, pc 값을 rd의 위치에 저장해주고, 이동하고자 하는 주소, 즉 pc+immediate로 위치를 이동한다. 여기서 주의할 점은, 현재의 pc 정보를 이용해야하기 때문에, 다음 위치를 가리키는 pc가 아닌, pc-4를 한 번 해주어야 한다(EX). jal에서도 beq에서와 마찬가지로 특정 값을 저장하는 것이 아닌 pc의 위치를 변경하여 이동하는 것이므로, MEM, WB 단계가 불필요하다. 아래 그림들은 jal 연산에서 수행되는 ID, EX 과정을 나타낸 코드이다.

```
if(bintodec(op, 6) ==111)
{
    //UJ-type: jal -> imme2[20]

    if(bin[0]==0)
    {
        imme2[0]=bin[0];

        for(int i=1;i<11;i++)
            imme2[i+9]=bin[i];

        imme2[9]=bin[11];

        for(int i=12;i<20;i++)
            imme2[i-11]=bin[i];

        tfi=bintodec(imme2,19)*2;
    }
    else if(bin[0]==1)
    {
        imme2[0]=0;

        for(int i=1;i<11;i++)
        {
            if(bin[i]==1)
                imme2[i+9]=0;
            else
                imme2[i+9]=1;
        }

        if (bin[11]==1)
            imme2[9]=0;
        else
            imme2[9]=1;

        for(int i=12;i<20;i++)
        {
            if(bin[i]==1)
                imme2[i-11]=0;
            else
                imme2[i-11]=1;
        }

        tfi_temp=bintodec(imme2,19)+1;
        tfi=-(tfi_temp*2);
    }

    for(int i=20;i<25;i++)
        rd[i-20]=bin[i];
}
```

<그림6.1 - jal 연산에서의 ID 과정>

```
if(bintodec(op, 6)==111)//jal
{
    regs[bintodec(rd,4)]=pc;
    pc-=4;
    pc+=tfi;
}
```

<그림6.2 - jal 연산에서의 EX 과정>

7. jalr

jalr은 I-type의 연산자로, x1에 돌아올 pc값, 즉 현재 위치를 저장한 후 rs1+immediate의 위치로 이동하는 연산자이다. jalr의 opcode는 1100111이고, 이를 10진수로 나타내면 103이다. 따라서 opcode가 103이라면 EX에서 jalr 연산을 수행하도록 한다.

instruction에서 각 비트 별로 나눠서 위치에 맞춰서 피연산자 rs1과 상수 값 immediate를 받아온다. immediate를 받아올 때, pc 점프는 beq이나 jal과 마찬가지로 짝수 번째로만 가능하므로, 32bit를 64bit로 바꿔주어야 한다. 그러므로 한비트씩 왼쪽으로 이동, 즉 곱하기 2를 해준다(ID), 이후, pc 값을 rd의 위치에 저장해주고, 이동하고자 하는 주소, 즉 rs1+immediate로 위치를 이동한다. 여기서 주의할 점은, 현재의 pc 정보를 이용해야하기 때문에, 다음 위치를 가리키는 pc가 아닌, pc-4를 한 번 해주어야 한다(EX). 또, rd 값이 0인경우는 다시 말하면 레지스터 x0를 뜻하는 경우인데, x0는 무조건 0이어야하기 때문에, 값이 들어가지 않도록 해야한다. jalr에서도 jal이나 beq에서와 마찬가지로 특정 값을 저장하는 것이 아닌 pc의 위치를 변경하여 이동하는 것이므로, MEM, WB 단계가 불필요하다. 아래 그림들은 jalr 연산에서 수행되는 ID, EX 과정을 나타낸 코드이다.

```
if(bintodec(op,6)==103)
{
    //I-type: jalr

    if(bin[0]==0)
    {
        for(int i=0;i<12;i++)
        {
            imme[i]=bin[i];
        }
        tfi=bintodec(imme,11)*2;
    }

    else if(bin[0]==1)
    {
        for(int i=0;i<12;i++)
        {
            if(bin[i]==1)
                imme[i]=0;
            else
                imme[i]=1;
        }

        tfi_temp=bintodec(imme,11)+1;
        tfi=-(tfi_temp*2);
    }

    for(int i=12;i<17;i++)
        rs1[i-12]=bin[i];

    for(int i=17;i<20;i++)
        func3[i-17]=bin[i];

    for(int i=20;i<25;i++)
        rd[i-20]=bin[i];
}
```

<그림7.1 - jalr 연산에서의 ID 과정>

```
if(bintodec(op,6)==103)//jalr -rd:x0, rs1:x31
{
    pc-=4;

    if(bintodec(rd,4)==0)
        regs[bintodec(rd,4)]=0;
    else
        regs[bintodec(rd,4)]=pc+4;

    pc=regs[bintodec(rs1,4)]+tfi;
}
```

<그림7.2 - jalr 연산에서의 EX 과정 >