计算机组成与设计 Lab1

10192100571 俞辰杰

实验目的

在此实验中，运用C++来模拟RISCV的指令级别的单周期循环，使用一些最简单的指令集（见下表）来模拟一些简单的RISCV的操作。在模拟运行的过程中，涉及到ALU的计算，PC地址的跳转，寄存器的读取，立即数的计算。

通过各个部分独立的实现，组合的运行，加深对于计算机底层设备的运行逻辑和设计方式

能够在用户层面加深对于RISCV指令集在机器层面上的操作和理解。

通过编写机器码，能够理解对于RISCV指令集不同类型指令的组成形式和解释方法。

Name	Format Type	Opcode (Binary)	Func 3(Binary)	Func 7(Binary)
add	R-Type	0110011	000	0000000
sub	R-Type	0110011	000	0100000
addi	I-Type	0010011	000	
and	R-Type	0110011	111	0000000
or	R-Type	0110011	110	0000000
xor	R-Type	0110011	100	0000000
beq	SB-Type	1100011	000	
jal	UJ-Type	1101111		
ld	I-Type	0000011	011	
sd	S-Type	0100011	011	

实验步骤

- **如果需要测试自己的代码需要自行更改imem.txt以及在代码中配置寄存器初始值以及数据内存存储值，代码内修改部分如下：**
  - 寄存器堆RF初始化寄存器初始值

```
RF(){
    Registers.resize(32);
    // 有32个寄存器
    Registers[0] = bitset<64>(0);
    // 0号寄存器一直为全零

    /*================= 修改Register ================*/
    Registers[10] = bitset<64>("1");
    // x10 = 1
    Registers[13] = bitset<64>("110000");
    // x13 = &A[0] = 0x00000030
    Registers[14] = bitset<64>("10000");
    // x14 = &B[0] = 0x00000010
    Registers[28] = bitset<64>("11110");
    // x28 = i = 30
    Registers[29] = bitset<64>("1");
    // x29 = j = 1
    /*================= 修改Register ================*/
}
```

  - 数据内存DMem初始化数据内存初始值

```
DataMem(){
    DMem.resize(MemSize);
    ifstream dmem;
    string line;
    int i = 0;
    dmem.open("dmem.txt");
    if (dmem.is_open()){
        while (getline(dmem, line)){
            DMem[i] = bitset<8>(line.substr(0, 8));
            i++;
        }
        /*================= 修改DMem ================*/
        DMem[16 + 0 * 8 + 7] = bitset<8>("11111111");
        // DMem[16] = B[0] = 6
        DMem[16 + 1 * 8 + 6] = bitset<8>("1");
        DMem[16 + 1 * 8 + 7] = bitset<8>("00110111");
        // B[1] = 137

        DMem[48 + 0 * 8 + 7] = bitset<8>("0");
        // DMem[55] = A[0] = 0
        DMem[48 + 27 * 8 + 6] = bitset<8>("1");
        DMem[48 + 27 * 8 + 7] = bitset<8>("00110111");
        // A[27] = 137
        DMem[48 + 28 * 8 + 7] = bitset<8>("11100");
```

```
        // A[28] = 28
        DMem[48 + 29 * 8 + 7] = bitset<8>("11101");
        // A[29] = 29
        /*================= 修改DMem ================*/
    }
    else
        cout << "Unable to open file";
    dmem.close();
}
```

- **如果不进行修改的话，可以直接运行RISC-V.cpp，运行代码如下CCM.txt所示：**

样例程序的C语言代码（**不使用样例的测试代码，因为样例测试代码并不包含beq和jal指令**），RISCV汇编代码，以及对于的指令集的机器码存放于此

```
/*============ 源测试C程序 ============*/
while(B[1] != A[i - j]){
    j += 1;
}
A[j] = B[0];

/*========== 简化得到以下C程序 ==========*/
while(true){
    if(B[1] == A[i - j]){
        break;
    }else{
        j += 1;
    }
}
A[j] = B[0];
```

此外，寄存器初始化和数据内存在代码中进行设定

```
x10 : 1
x12 : addr_temp
x13 : A[]
x14 : B[]
x27 : data_temp2
x28 : i（i = 30）
x29 : j（j = 1）
x30 : data_temp
```

对应的riscv指令及32位指令码

```
Loop:   sub x30, x28, x29                       // compute i-j
00      0100000 11101 11100 000 11110 0110011
        add x12, x30, x30                       // multiply by 8
04      0000000 11110 11110 000 01100 0110011
        add x12, x12, x12
08      0000000 01100 01100 000 01100 0110011
        add x12, x12, x12
12      0000000 01100 01100 000 01100 0110011
        add x12, x12, x13
16      0000000 01101 01100 000 01100 0110011
        ld x30, 0(x12)                          // x30 load A[i-j]
20      000000000000  01100 011 11110 0000011
        ld x27, 8(x14)                          // x31 load B[1]
24      000000001000  01110 011 11011 0000011
        beq x27, x30, Exit      (40 - 28 = 12 = 000000000110[0])
28      0 000000 11110 11011 000 0110 0 1100111
        add x29, x29, x10                       // j += 1
32      0000000 01010 11101 000 11101 0110011
        jal x0, Loop            (0 - 36 = -36 = 11111111111111101110[0])
36      1 1111101110 1 11111111 00000 1101111
Exit:   ld x30, 0(x14)                          // x30 = B[0]
40      000000000000  01110 011 11110 0000011
        add x12, x29, x29                       // j * 8
44      0000000 11101 11101 000 01100 0110011
        add x12, x12, x12
48      0000000 01100 01100 000 01100 0110011
        add x12, x12, x12
52      0000000 01100 01100 000 01100 0110011
        add x12, x12, x13
56      0000000 01101 01100 000 01100 0110011
        sd x30, 0(x12)                          // A[j] = x30
60      0000000 11110 01100 011 00000 0100011
        end
64      1111111 11111 11111 111 11111 1111111
```

```
01000001110111100000111100110011
00000001111011110000011000110011
00000000110001100000011000110011
00000000110001100000011000110011
00000000110101100000011000110011
00000000000001100011111100000011
00000000100001110011110110000011
00000001111011011000011001100111
00000000101011101000111010110011
11111101110111111111000001101111
00000000000001110011111100000011
00000001110111101000011000110011
00000000110001100000011000110011
00000000110001100000011000110011
00000000110101100000011000110011
00000001111001100011000000100011
11111111111111111111111111111111
```

实验结果

对于给定的测试例子，在RFresult.txt中，可以看到寄存器的最终结果：

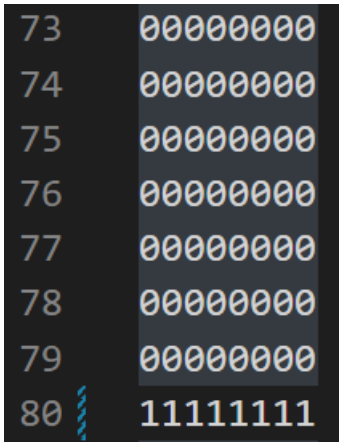
```
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000001 x10 = 1
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001001000 x12 = &A[3]
00000000000000000000000000000000000000000000000000000000110000 x13 = &A[0]
00000000000000000000000000000000000000000000000000000000010000 x14 = &B[0]
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000100110111 x27 = B[1]
00000000000000000000000000000000000000000000000000000000011110 x28 = i
00000000000000000000000000000000000000000000000000000000000011 x29 = j
00000000000000000000000000000000000000000000000000000011111111 x30 = A[j]
00000000000000000000000000000000000000000000000000000000000000000
```

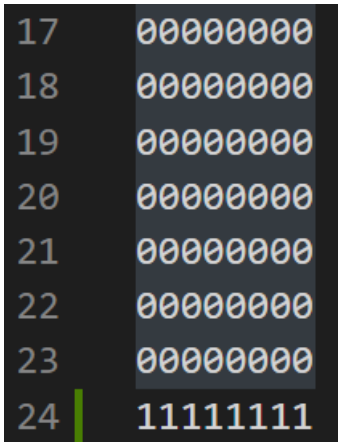
在dmemresult.txt中，可以跟踪数据内存的数据：

按照预期 $A[j]$ 和 $B[0]$ 应该有相同的数据，$A[i-j]=A[27]$ 和 $B[1]$应该有相同的结果

根据上方寄存器的结果，我们反向追踪数据内存的位置，其中：

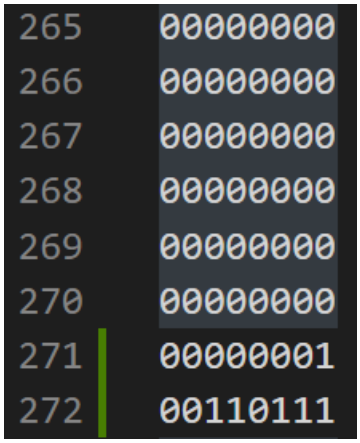
- $A[j]$ 应该在 **1001000b + 111b= 79** 的内存位置
- $B[0]$ 应该在 **10000b + 111b = 23** 的内存位置

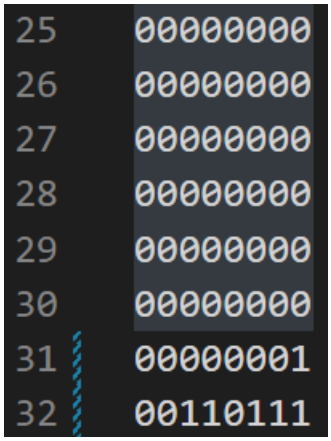




$A[j]$ **和** $B[0]$ **有相同的结果**

- $A[27]$ 应该在 **110000b + 11011000b + 111b = 100001111b = 271** 的内存位置
- $B[1]$ 应该在 **11000b + 111b = 11111b = 31** 的内存位置





$A[27]$ **和** $B[1]$ **有相同的结果**

在实验过程中，我们跟踪输出了每条指令所**读取的寄存器内容、立即数数字，执行的操作类型，PC的目标地址，读写操作的目标地址。**

由于执行条数过多，故不全部展示，如果修改代码，可以通过此方法在运行过程中及时观察运行过程是否符合逻辑，下图各种类型操作分别展示一次。

```
================================================
Instruction:01000001110111100000111100110011
Regsier1: 28
00000000000000000000000000000000000000000000000000000000011110
Register2: 29
00000000000000000000000000000000000000000000000000000000000010
Regsier target: 30
00000000000000000000000000000000000000000000000000000000000000
Immediate Number:
00000000000000000000000000000000000000000000000000000000000000
SUBU result:
00000000000000000000000000000000000000000000000000000000011100
Regsier1: 28
00000000000000000000000000000000000000000000000000000000011110
Register2: 29
00000000000000000000000000000000000000000000000000000000000010
Regsier target: 30
00000000000000000000000000000000000000000000000000000000011100
PC point to 4
Finish A Instruction
================================================
```

图1 数据计算R-type指令结果

```
================================================
Instruction:00000000000011000111111100000011
Regsier1: 12
00000000000000000000000000000000000000000000000000000100001000
Register2: 0
00000000000000000000000000000000000000000000000000000000000000
Regsier target: 30
00000000000000000000000000000000000000000000000000000000000000
Immediate Number:
00000000000000000000000000000000000000000000000000000000000000
sw/lw address:
00000000000000000000000000000000000000000000000000000100001000
Load start at 264
MemoryAccess:
00000000000000000000000000000000000000000000000000000100110111
Regsier1: 12
00000000000000000000000000000000000000000000000000000100001000
Register2: 0
00000000000000000000000000000000000000000000000000000000000000
Regsier target: 30
00000000000000000000000000000000000000000000000000000100110111
PC point to 24
Finish A Instruction
================================================
```

图2 数据内存读写sd/ld指令结果

```
==================================================
Instruction:00000001111011011000011001100111
Regsier1: 27
00000000000000000000000000000000000000000000000000000100110111
Register2: 30
00000000000000000000000000000000000000000000000000000100110111
Immediate Number:
00000000000000000000000000000000000000000000000000000000000000
Branch/Jump
Regsier1: 27
00000000000000000000000000000000000000000000000000000100110111
Register2: 30
00000000000000000000000000000000000000000000000000000100110111
Success Branch
addressExtend:
00000000000000000000000000001100
PC point to 40
Finish A Instruction
==================================================
```

图3 分支跳转beq指令结果

```
==================================================
Instruction:11111101110111111111000001101111
Regsier1: 0
00000000000000000000000000000000000000000000000000000000000000
Register2: 0
00000000000000000000000000000000000000000000000000000000000000
Regsier target: 0
00000000000000000000000000000000000000000000000000000000000000
Immediate Number:
00000000000000000000000000000000000000000000000000000100100100
store (PC + 4):
00000000000000000000000000000000000000000000000000000101000000
Regsier1: 0
00000000000000000000000000000000000000000000000000000000000000
Register2: 0
00000000000000000000000000000000000000000000000000000000000000
Regsier target: 0
00000000000000000000000000000000000000000000000000000101000000
Success Jump
addressExtend:
11111111111111111111111111011100
PC point to 0
Finish A Instruction
==================================================
```

图4 直接跳转jal指令结果

代码分析

```
class RF{
// 寄存器堆 Register File
public:
    bitset<64> ReadData1, ReadData2;
    RF()
    {
        Registers.resize(32);
        // 有32个寄存器
        Registers[0] = bitset<64>(0);
        // 0号寄存器一直为全零

        Registers[10] = bitset<64>("1");
        // x10 = 1
        Registers[13] = bitset<64>("110000");
        // x13(保存数组A的起始地址) = &A[0] = 0x00000030
        Registers[14] = bitset<64>("10000");
        // x14 = &B[0] = 0x00000010
        Registers[28] = bitset<64>("11110");
        // x28 = i = 30
        Registers[29] = bitset<64>("1");
        // x29 = j = 1
    }

    void ReadWrite(bitset<5> RdReg1, bitset<5> RdReg2, bitset<5> WrtReg,
bitset<64> WrtData, bitset<1> WrtEnable)
    // WrtEnable 0 is read else is read and write
    {
        // 通过5位的RdReg，来偏移获取寄存器内的数据
        // 无论如何，都会把源寄存器的内容读出，如果有写使能的话，可以向目的寄存器写入目的
数据
        ReadData1 = this->Registers[RdReg1.to_ulong()];
        ReadData2 = this->Registers[RdReg2.to_ulong()];

        // 输出寄存器Rs1,Rs2内的数据
        cout << "Register1: " << RdReg1.to_ulong() << endl;
        cout << ReadData1 << endl;
        cout << "Register2: " << RdReg2.to_ulong() << endl;
        cout << ReadData2 << endl;
        if (WrtEnable.to_ulong()){
            // 如果有有写使能的话，向寄存器里保存数据
            this->Registers[WrtReg.to_ulong()] = WrtData;
            cout << "Regsier target: " << WrtReg.to_ulong() << endl;
            cout << WrtData << endl;
        }
        // 防止x0被修改，所以每次调用最后都需要复位x0
        Registers[0] = bitset<64>(0);
    }

    void OutputRF()
    {
        ofstream rfout;
        rfout.open("RFresult.txt", std::ios_base::app);
        if (rfout.is_open())
```

```
        {
            rfout << "A state of RF:" << endl;
            for (int j = 0; j < 32; j++)
            {
                rfout << Registers[j] << endl;
            }
        }
        else
            cout << "Unable to open file";
        rfout.close();
    }

private:
    vector<bitset<64>> Registers;
};

```

```cpp
class ALU{
// 算数运算单元
public:
    bitset<64> ALUresult;
    /* 根据主函数，不同的操作有着不同的ALUop
        ALUOP:
            000 add
            001 sub
            010 and
            011 or
            100 xor
            101 sw/lw
            110 jal
            111 none
    */
    void ALUOperation(bitset<3> ALUOP, bitset<64> oprand1, bitset<64> oprand2){
        // TODO: implement!
        // 通过计算得到的结果放在ALUresult之中
        switch (ALUOP.to_ulong()){
        case ADDU:
            // 000 add
            ALUresult = bitset<64>(oprand1.to_ullong() + oprand2.to_ullong());
            cout << "ADDU result: " << endl;
            cout << ALUresult << endl;
            break;
        case SUBU:
            // 001 sub
            ALUresult = bitset<64>(oprand1.to_ullong() - oprand2.to_ullong());
            cout << "SUBU result: " << endl;
            cout << ALUresult << endl;
            break;
        case AND:
            // 010 and
            ALUresult = bitset<64>(oprand1.to_ullong() & oprand2.to_ullong());
            cout << "AND result: " << endl;
            cout << ALUresult << endl;
            break;
        case OR:
            // 011 or
            ALUresult = bitset<64>(oprand1.to_ullong() | oprand2.to_ullong());
            cout << "OR result: " << endl;
            cout << ALUresult << endl;
            break;
        case XOR:
            // 100 xor
            ALUresult = bitset<64>(oprand1.to_ullong() ^ oprand2.to_ullong());
            cout << "XOR result: " << endl;
            cout << ALUresult << endl;
            break;
        case 5:
```

```
            // 101 sw/lw, 这里开始拿到的都是 寄存器的起始地址oprand1和立即数的地址偏移oprand2 做计算得到的目的地址
            ALUresult = bitset<64>(oprand1.to_ullong() + oprand2.to_ullong());
            cout << "sw/lw address: " << endl;
            cout << ALUresult << endl;
            break;
        case 6:
            // 110 jal Jype没有reg1，所以oprand1是空的，oprand2是PC的地址，我们需要再加4
            ALUresult = bitset<64>(oprand2.to_ullong() + 4);
            cout << "store (PC + 4): " << endl;
            cout << ALUresult << endl;
            break;
        case 7:
            // 111 不知道做啥
            cout << "Branch/Jump" << endl;
            break;
        default:
            cout << "ERROR" << endl;
            break;
        }
    }
};

```

```cpp
class INSMem{
// 指令寄存器 Instruction Memory
public:
    bitset<32> Instruction;
    INSMem()
    {
        IMem.resize(MemSize);
        ifstream imem;
        string line;
        int i = 0;
        imem.open("imem.txt");
        if (imem.is_open())
        {
            while (getline(imem, line))
            {
                IMem[i] = bitset<8>(line.substr(0, 8));
                i++;
            }
        }
        else
            cout << "Unable to open file";
        imem.close();
    }

    bitset<32> ReadMemory(bitset<32> ReadAddress)
    {
        // (Read the byte at the ReadAddress and the following three byte).
        // 从PC一共读取 32 位， 读取到的指令就是所谓的 R-type，I-type等等
        bitset<8> adr;
        string ans;
        for (int i = 0; i < 4; i++)
        {
            // 由于单行是8位存储，所以循环读取4次可以将32位的指令读取
            adr = IMem[ReadAddress.to_ulong() + i];
            ans += adr.to_string();
        }
        cout << "Instruction:" << ans << endl;
        this->Instruction = bitset<32>(ans);
        return Instruction;
    }

private:
    vector<bitset<8>> IMem;
};

```

```cpp
class DataMem{
// 数据寄存器 Data Memory
public:
    bitset<64> readdata;

    DataMem()
    {
        DMem.resize(MemSize);
        ifstream dmem;
        string line;
        int i = 0;
        dmem.open("dmem.txt");
        if (dmem.is_open())
        {
            while (getline(dmem, line))
            {
                DMem[i] = bitset<8>(line.substr(0, 8));
                i++;
            }
            DMem[16 + 0 * 8 + 7] = bitset<8>("11111111");
            // DMem[16] = B[0] = 6
            DMem[16 + 1 * 8 + 6] = bitset<8>("1");
            DMem[16 + 1 * 8 + 7] = bitset<8>("00110111");
            // B[1] = 0x00 00 00 00 00 00 01 37

            DMem[48 + 0 * 8 + 7] = bitset<8>("0");
            // DMem[48] = A[0]
            DMem[48 + 27 * 8 + 6] = bitset<8>("1");
            DMem[48 + 27 * 8 + 7] = bitset<8>("00110111");
            // A[27] = 0x00 00 00 00 00 00 01 37
            DMem[48 + 28 * 8 + 7] = bitset<8>("11100");
            // A[28] = 28
            DMem[48 + 29 * 8 + 7] = bitset<8>("11101");
            // A[29] = 29
        }
        else
            cout << "Unable to open file";
        dmem.close();
    }

    // 只有load或store指令的时候处理地址
    bitset<64> MemoryAccess(bitset<64> Address, bitset<64> WriteData,
bitset<1> readmem, bitset<1> writemem)
    {
        // address 是 （reg1 + imm) 的地址；writedata 是 reg2
        if (readmem.to_ulong())
        {
            // isLoad的情况
            // address 是源地址， writedata是空
            // 从address的地址读数据，写入this->readdata，所以下一步
myRF.ReadWrite的时候需要使用readdata而不是ALUresult
            cout << "Load start at " << Address.to_ulong() << endl;
            bitset<8> adr;
            string ans;
            for (int i = 0; i < 8; i++)
            {
```

```
                adr = this->DMem[Address.to_ulong() + i];
                // 拼接得到8字节的readdata
                ans += adr.to_string();
            }
            cout << "MemoryAccess: " << endl;
            cout << ans << endl;
            this->readdata = bitset<64>(ans);
        }
        if (writemem.to_ulong())
        {
            // isStore的情况
            // address 是 目标地址，writedata是源地址
            // 从writedata的地址读数据，写入address
            bitset<8> data;
            cout << "Store into DMem: " << endl;
            cout << Address.to_ulong() << endl;
            for (int i = 0; i < 8; i++)
            {
                data = bitset<8>(writeData.to_string().substr(i * 8, 8));
                this->DMem[Address.to_ulong() + i] = data;
            }
        }
        return Address;
    }

    void OutputDataMem()
    {
        ofstream dmemout;
        dmemout.open("dmemresult.txt");
        if (dmemout.is_open())
        {
            for (int j = 0; j < 1000; j++)
            {
                dmemout << DMem[j] << endl;
            }
        }
        else
            cout << "Unable to open file";
        dmemout.close();
    }

private:
    vector<bitset<8>> DMem;
};

```

```cpp
int main()
{
    RF myRF;
    ALU myALU;
    INSMem myInsMem;
    DataMem myDataMem;

    // Control Registers
    // PC 初始值是 0x0
    bitset<32> PC;
    bitset<1> wrtEnable;
    bitset<1> isJType;
    bitset<1> isIType;
    bitset<1> isLoad;   // I-type
    bitset<1> isStore;  // S-type
    bitset<1> isBranch; // SB-type
    bitset<1> isRType;
    bitset<3> aluOp;

    while (1)
    {
        // 1. Fetch Instruction
        bitset<32> instruction = myInsMem.ReadMemory(PC);

        // If current insturciton is "11111111111111111111111111111111", then break;
        if (myInsMem.Instruction.to_ulong() == 0xffffffff)
        {
            break;
        }

        // decode(Read RF)
        // 判断opcode

        // load 只有reg1，rd，imm，没有reg2，源地址数据就是（&reg1 + imm），目的地址数据是&rd
        isLoad = instruction.to_string().substr(25, 7) == string("0000011");
        // store只有reg1，reg2，imm，没有rd，源地址数据是&reg2，目的地址数据就是（&reg1 + imm）
        // 并且Store不是向寄存器写，而是向数据内存写，所以在执行完myDataMem.MemoryAccess之后，不需要第二次ReadWrite了
        isStore = instruction.to_string().substr(25, 7) == string("0100011");
        // 所以在ALUoperation的时候，只计算 &reg1 + imm，得到ALUresult
        // load  把 ALUresult   赋值给  rd，和Rtype类似
        // store 把 reg2        赋值给  ALUresult

        // 只有目的寄存器和立即数，没有源寄存器12，将此条指令的下一条指令（PC+4）放在目标寄存器rd中，所以需要写使能
        isJType = instruction.to_string().substr(25, 7) == string("1101111");
        isRType = instruction.to_string().substr(25, 7) == string("0110011");
        isBranch = instruction.to_string().substr(25, 7) == string("1100111");
```

```
        isIType = instruction.to_string().substr(25, 5) == string("00100")
||
                  instruction.to_string().substr(25, 5) == string("11000");
        wrtEnable = !(isStore.to_ulong() || isBranch.to_ulong());
        // 如果该条指令不是存储指令或是跳转指令，那么都会有目的寄存器rd；否则没有rd，不
需要写入数据

        // 通过检验funct3和funct7得到ALUop
        if (isRType[0] == 1)
        {
            if (instruction.to_string().substr(17, 3) == string("000"))
            {
                if (instruction.to_string().substr(0, 7) ==
string("0000000"))
                    aluOp = bitset<3>("000"); // add
                else if (instruction.to_string().substr(0, 7) ==
string("0100000"))
                    aluOp = bitset<3>("001"); // sub
            }
            else if (instruction.to_string().substr(17, 3) ==
string("111"))
            {
                aluOp = bitset<3>("010"); // and
            }
            else if (instruction.to_string().substr(17, 3) ==
string("110"))
            {
                aluOp = bitset<3>("011"); // or
            }
            else if (instruction.to_string().substr(17, 3) ==
string("100"))
            {
                aluOp = bitset<3>("100"); // xor
            }
        }
        else if (isStore[0] == 1 || isLoad[0] == 1)
        {
            aluOp = bitset<3>("101"); // sw or lw
        }
        else if (isJType[0] == 1)
        {
            // jal 使用 myRF.ReadWrite 的目的是为了 把下一条指令的位置（PC+4）放在
目的寄存器rd 里面
            // 使用ALUoperation的时候，只要将ALUresult设为（PC+4）即可
            // 接下来要到第5步的时候才能把 pc+4 的地址 赋值给 rd
            aluOp = bitset<3>("110");
        }
        else
        {
            aluOp = bitset<3>("111");
        }

        // 2. Register File Instruction
        myRF.ReadWrite(
            (isJType[0]) ? bitset<5>(string("00000")) : bitset<5>
(instruction.to_string().substr(12, 5)),
            // 找到reg1
```

```
            (isIType[0] || isJType[0] || isLoad[0]) ? bitset<5>
(string("00000")) : bitset<5>(instruction.to_string().substr(7, 5)),
            // 找到reg2
            (isIType[0] || isRType[0] || isJType[0] || isLoad[0]) ?
bitset<5>(instruction.to_string().substr(20, 5)) : bitset<5>
(string("00000")),
            // 找到wrtReg
            bitset<64>(0),
            // wrtData 为 全0
            wrtEnable);
        // 这步进行之后，获得了两个源寄存器的内部数据。
        // 如果之后需要写的话，目的寄存器全部置零备用；否则保持原先的状态

        // 3. Execuete alu operation
        // tmp 存储立即数
        bitset<64> tmp;
        if (isLoad[0] == 1 || isIType[0] == 1)
        {
            // imm[11:0]
            tmp = bitset<64>(instruction.to_string().substr(0, 12));
            // if positive, 0 padded

            if (tmp[20] == true)
            {
                tmp = bitset<64>(string(52, '1') +
tmp.to_string().substr(20, 12));
            }
        }
        else if (isStore[0] == 1)
        {
            // imm[11:5] rs2 rs1 010 imm[4:0]
            tmp = bitset<64>(instruction.to_string().substr(0, 7) +
instruction.to_string().substr(20, 5));
            // if positive, 0 padded

            if (tmp[20] == true)
            {
                tmp = bitset<64>(string(52, '1') +
tmp.to_string().substr(20, 12));
            }
        }
        else if (isJType[0] == 1)
        {
            if (PC[31] == true)
                // 这里的立即数只获得了PC的位置，在ALUopreation的时候需要再加4
                tmp = bitset<64>(string(32, '1') + PC.to_string());
                // R[rd] = PC + 4
            else
                tmp = bitset<64>(string(32, '0') + PC.to_string());
        }
        cout << "Immediate Number: " << endl;
        cout << tmp << endl;
        myALU.ALUOperation(aluOp, myRF.ReadData1, (isIType[0] || isJType[0]
|| isLoad[0] || isStore[0]) ? tmp : myRF.ReadData2);
        // 使用源寄存器1。如果是IJS的类型的话，使用立即数，否则使用源寄存器2
        // 无论得到的是 结算结果 还是 目的地址 都放在结果ALUresult之中

        // 4. Read/Write Mem(Memory Access)
```

```
        // load:  使用 myALU.ALUresult 和 isload
        // store: 使用 myRF.ReadData2 和 isStore
        myALU.ALUresult = myDataMem.MemoryAccess(myALU.ALUresult,
myRF.ReadData2, isLoad, isStore);

        // 5. Register File Update(Write Back)
        myRF.ReadWrite(
            (isJType[0]) ? bitset<5>(string("00000")) : bitset<5>
(instruction.to_string().substr(12, 5)),
            (isJType[0] || isIType[0] || isLoad[0]) ? bitset<5>
(string("00000")) : bitset<5>(instruction.to_string().substr(7, 5)),
            (isIType[0] || isRType[0] || isJType[0] || isLoad[0]) ?
bitset<5>(instruction.to_string().substr(20, 5)) : bitset<5>
(string("00000")),
            isLoad[0] ? myDataMem.readdata : myALU.ALUresult,
            wrtEnable);

        // Update PC
        if (isBranch[0] && myRF.ReadData1 == myRF.ReadData2)
        {
            cout << "Success Branch" << endl;
            // 如果相同，跳转PC += 指定大小；不同则进行下一个指令，PC += 4，处理的
branch指令都是 beq
            bitset<32> addressExtend;
            // imm[12|10:5] rs2 rs1 000 imm[4:1|11]
            if (instruction[31] == true)
                addressExtend = bitset<32>(string(19, '1') +
instruction.to_string().substr(0, 1) + instruction.to_string().substr(24,
1) + instruction.to_string().substr(1, 6) +
instruction.to_string().substr(20, 4) + string("0"));
            else
                addressExtend = bitset<32>(string(19, '0') +
instruction.to_string().substr(0, 1) + instruction.to_string().substr(24,
1) + instruction.to_string().substr(1, 6) +
instruction.to_string().substr(20, 4) + string("0"));
            // 最后加一位0是为了偶跳转
            cout << "addressExtend:" << endl;
            cout << addressExtend << endl;
            PC = bitset<32>(PC.to_ulong() + addressExtend.to_ulong());
            cout << "PC point to " << PC.to_ulong() << endl;
        }
        else if (isJType[0])
        {
            cout << "Success Jump" << endl;
            bitset<32> addressExtend;
            // imm[20|10:1|11|19:12]
            if (instruction[31] == true){
                addressExtend = bitset<32>(string(11, '1') +
instruction.to_string().substr(0, 1) + instruction.to_string().substr(12,
8) + instruction.to_string().substr(11, 1) +
instruction.to_string().substr(1, 10) + string("0"));
            }else{
                addressExtend = bitset<32>(string(11, '0') +
instruction.to_string().substr(0, 1) + instruction.to_string().substr(12,
8) + instruction.to_string().substr(11, 1) +
instruction.to_string().substr(1, 10) + string("0"));
            }
            cout << "addressExtend:" << endl;
```

```
            cout << addressExtend << endl;
            PC = bitset<32>(PC.to_ulong() + addressExtend.to_ulong());
            cout << "PC point to " << PC.to_ulong() << endl;
        }
        else
        {
            PC = bitset<32>(PC.to_ulong() + 4);
            cout << "PC point to " << PC.to_ulong() << endl;
        }

        myRF.OutputRF(); // dump RF;
        cout << "Finish A Instruction" << endl;
        cout << "=================================================" <<
endl;
    }
    cout << "Successful!!!" << endl;
    myDataMem.OutputDataMem(); // dump data mem

    return 0;
}

```