湖南大學

HUNAN UNIVERSITY

微处理器设计
实验报告

姓　名：吕志恒

班　级：智能 1602

学　号：201608010719

时　间：2019-12-10

一、RV32I 的汇编器

1. 设计思路：

首先将指令用 vector 这个数据结构映射成（指令名称，指令类型，16 进制表示）的形式，具体的实现方式是先将映射关系通过 map 实现，然后将 map 存入 vector。最后用指令去查该表得到对应的 16 进制表达，使用 vector 的原因是因为其查询复杂度较低同时可以有效节省空间的使用。接着转换成机器能识别的二进制表示。用到的主要数据结构有：std：：map，std：：vector。

2. 汇编器架构：

汇编器由以下部分组成：

main：读取测试的指令；

helper：分离指令成若干部分；

assembler：负责将指令转换成二进制形式；

instruction_set：负责维护一个映射表，同时返回查询结果。（指令名称，指令类型，16 进制表示）

3. 关键代码说明：

（1）建立映射表：（只贴部分代码）

```
std::unordered_map<std::string, riscv_asm::instruction_set_op> riscv_asm::instruction_set_op_factory() {
    std::unordered_map<std::string, instruction_set_op> result;

    std::vector<instruction_set_op> data = {
        {"ecall",  'N', 0x00000073},
        {"ebreak",  'N', 0x00100073},

        {"add",  'R', 0x00000033},
        {"sub",  'R', 0x40000033},
        {"sll",  'R', 0x00001033},
        {"slt",  'R', 0x00002033},
        {"sltu", 'R', 0x00003033},
```

这里返回的类型是一个 map，可以理解为 vector 存储的对象是 map。查询代码如下：

```
    for (instruction_set_op& instr : data) {
        result[instr.name] = instr;
    }

    return result;
```

（2）分离指令：

```
std::vector<std::string> riscv_asm::helper::split(const std::string& input, const std::string& pattern) {
    std::vector<std::string> result;

    size_t current = 0;
    size_t old;

    do {
        old = current;
        current = input.find_first_of(pattern, old + 1);
        result.push_back(input.substr(old, current - old));
        current = input.find_first_not_of(pattern, current);
    } while (current != std::string::npos);

    return result;
}
```

函数的返回值是 vector 类型，函数有两个参数，一个是输入的指令，一个分割标志符，比如指令的逗号或者指令尾部的结束符，用于分离判断。函数执行逻辑如下：在没有判断到指令结束前，不断从指令截取字符 push 到一个 vector，然后判断 vector 顶部是不是一个分隔符，如果是则说明截取成功能，这个时候 current 的值为 nop 退出分离同时返回分离结果。比如指令 add ，x1，x2 分离后就是 add（只执行一次该函数）

（3）汇编转换

这里我用了一个 switch-case，目的是根据指令的类型调用不同类型的函数，如 R 类型函数，则用 R 类型的处理函数。如图：

```cpp
uint32_t riscv_asm::assembler::assemble_single_instruction(const std::string& instr) {
    auto parsedInstr = helper::split(instr, " ,");
    if (!parsedInstr.empty()) {  //空置处理
        auto op = m_instructionSet.find(parsedInstr[0]);
        if (op == m_instructionSet.end()) {
            return 0;
        }

        switch (op->second.type) {
        case 'N':
            return build_N_Type(parsedInstr);//如果是N类型，则用N类型的处理函数。
        case 'R':
            return build_R_Type(parsedInstr);
        case 'I':
            return build_I_Type(parsedInstr);
        case 'S':
            return build_S_Type(parsedInstr);
        case 'B':
            return build_B_Type(parsedInstr);
        case 'U':
            return build_U_Type(parsedInstr);
        case 'J':
            return build_J_Type(parsedInstr);
        case 'W':
            return build_W_Type(parsedInstr);
        default:
            return 0;
        }
    }
}
```

函数的具体实现：

```cpp
uint32_t riscv_asm::assembler::build_N_Type(std::vector<std::string>& input) {
    return m_instructionSet[input[0]].opcode;
}
```

```cpp
uint32_t riscv_asm::assembler::build_R_Type(std::vector<std::string>& input) {
    if (input.size() < 4) {
        return 0;
    }

    return
        parse_register(input[1]) << 7 |
        parse_register(input[2]) << 15 |
        parse_register(input[3]) << 20 |
        m_instructionSet[input[0]].opcode;
}
```

```cpp
uint32_t riscv_asm::assembler::build_I_Type(std::vector<std::string>& input) {
    if (input.size() < 4) {
        return 0;
    }

    return
        parse_register(input[1]) << 7 |
        parse_register(input[2]) << 15 |
        parse_immediate(input[3]) << 20 |
        m_instructionSet[input[0]].opcode;
}
```

其它具体实现详见代码。实现的思路都是先根据指令的类型，然后再去获取对应的位置的内容写入到寄存器里面去。parse_register 是一个解析函数，作用是提取指定位置的内容。其实现如下：

```
uint32_t riscv_asm::assembler::parse_register(const std::string& reg) {
    //only x-style
    if (reg.length() < 2) {
        return 0;
    }

    return reg[1] - 48;
}
```

有个立即数解析的函数也是类似的原理，如下：

```
uint32_t riscv_asm::assembler::parse_immediate(const std::string& imm) {
    auto symbol = m_symbolTable.find(imm);
    if (symbol != m_symbolTable.end()) {
        return symbol->second - m_currentAddress;
    } else {
        return std::stoul(imm, 0, 0);
    }
}
```

汇编处理函数：

```
std::vector<uint32_t> riscv_asm::assembler::assemble(const std::string& input) {
    std::vector<uint32_t> result;
    auto lines = helper::split(input, ";\n");//从读取的指令分离出操作名称
    m_currentAddress = 0;//当前操作地址

    for (auto& line : lines) {
        auto symbol = line.find(":");
        if (symbol != std::string::npos) {
            m_symbolTable[line.substr(0, symbol)] = m_currentAddress;

            if (line.find_first_not_of(" ;\n\t") != std::string::npos) {
                result.push_back(assemble_single_instruction(line.substr(symbol + 1)));
                m_currentAddress += 4;
            }
        } else {
            result.push_back(assemble_single_instruction(line));
            m_currentAddress += 4;
        }

    }

    return result;
}
```

该函数的返回值是 vector，其作用是将输入的指令处理成：（指令名称，指令类型）的形式，用于后面查询。assemble_single_instruction 函数是前面提到的根据指令的类型调用相应的函数处理。

（4）main 函数：读入指令，生成相应的 hex 文件并输出。

```
int main(int argc, char **argv) {
    std::string input =
        "add x1, x0, x0;"
        "addi x2, x0, 10;"
        "loop:"
        "addi x1, x1, 1;"
        "blt x1, x2, loop;"
        "lui x6, 0x20;"
        "addi x1, x0, 0xFF;"
        "sw x1, x6, -4;"
        "lw x3, x6, -4;"
        "lh x4, x6, -4;"
        "lb x5, x6, -4;"

        "end:"
        "jal x0, end;"
    ;

    riscv_asm::assembler asmConverter;

    auto code = asmConverter.assemble(input);

    for (uint32_t c : code) {
        std::cout << std::hex << c << std::endl;
    }

    return 0;
}
```

4. 运行结果：运行环境：Windows7+CODEBLOCK

（1）测试 div 运算：

汇编程序如下：

```
main:
    addi    $a0, $zero, 8
    addi    $a1, $zero, 2
    jal   mydiv

    li  $v0, 10          # 调用退出命令
    SYSCALL                  # 程序停止处
#mydiv的具体实现
mydiv:
    add   $t1, $zero, $zero # i = 0

mydiv_test:
    slt   $t0, $a0, $a1     # if ( a < b )
    bne   $t0, $zero, mydiv_end # then get out of here
    sub   $a0, $a0, $a1     # else, a = a - b
    addi    $t1, $t1, 1     # and   i = i + 1
    j     mydiv_test        # let's test again

mydiv_end:
    add $a1, $zero, $a0     # rest   = a
    add $a0, $zero, $t1     # result = i
    jr  $ra
```

输出的 bin 文件如下：

```
1305 8000 9305 2000 ef00 c000 9308 a000
7300 0000 3303 0000 b322 b500 6398 0200
3305 b540 1303 1300 6ff0 1fff b305 a000
3305 6000 6780 0000
```

（2）测试 multi 运算：

汇编程序如下：

```
    li  $s2, 20
    li  $s1, 30
    li  $s0, 0
whileLoop:
    ble  $s2, $0, endWhile
    add  $s0, $s0, $s1
    addi $s2, $s2, -1
    j    whileLoop
endWhile:
    li  $v0, 10
    SYSCALL
```

生成的 **bin** 文件如下：

```
1309 4001 9304 e001 1304 0000 6358 2001
3304 9400 1309 f9ff 6ff0 5fff 9308 a000
7300 0000
```

反编译验证：可以看到寄存器的值确实被改变了。

```
x1:     0x00000000
x2:     0x00000000
x3:     0x00000000
x4:     0x00000000
x5:     0x00000000
x6:     0x00000000
x7:     0x00000000
x8:     0x00000258
x9:     0x0000001e
x10:    0x00000000
x11:    0x00000000
x12:    0x00000000
x13:    0x00000000
x14:    0x00000000
x15:    0x00000000
x16:    0x00000000
x17:    0x0000000a
x18:    0x00000000
x19:    0x00000000
x20:    0x00000000
x21:    0x00000000
x22:    0x00000000
x23:    0x00000000
x24:    0x00000000
x25:    0x00000000
x26:    0x00000000
x27:    0x00000000
x28:    0x00000000
x29:    0x00000000
x30:    0x00000000
x31:    0x00000000
```

（3）测试 SUM 运算：

汇编程序如下：

```
main:
    li $v0, 5          # enter the value
    SYSCALL

    move $a2, $v0 #a7
    jal sum


    li $v0, 1
    SYSCALL

    li $v0, 10
    SYSCALL

sum:
    li $a0, 0
loop:
    ble $a2, $zero, stop
    add $a0, $a2, $a0
    addi $a2, $a2, -1
    j loop

stop:
    addi $a1, $a1, 4
    jr $ra
```

该段程序的作用是计算前 n 和，比如 sum（4）=10（亦即 1+2+3+4）

生成的 bin 文件如下：

```
9308 5000 7300 0000 1386 0800 9705 0000
6f00 4001 9308 1000 7300 0000 9308 a000
7300 0000 1305 0000 6358 c000 3305 a600
1306 f6ff 6ff0 5fff 9385 4500 6780 0500
```

减法和加法相反不在列出。

5. 遇到的问题：

在 kdevelop4 上面成功编译，但运行的时候提示 fail。后来询问老师后可以尝试升级一下 G++版本，但并没有效果，后来改到 Windows 是平台，修改了使用的数据结构，不再用原来的高级数据结构。同时修改输出为 bin 文件。

6. 实验心得：

RISCV 汇编器的设计其实整体难度还好，设计思路比较简单的一种是维护一个映射表，然后通过解析输入的指令去查相应的映射表，最后拼出一段二进制序列。本人采用的正是这种实现方法，但前期踩了很多坑，因为贪方便用了高级的 C++数据结构，结果编译器不支持，调试了很久花费了很多时间。

二、RV32I 的模拟器设计

1. 设计思路：

为了节省时间，采用夏季小学期基于 C++写的模拟器，同时尝试修改它的输入为 bin 文件，配合刚才写的汇编模拟器，从而双向验证汇编模拟器和 RV32I 模拟器的正确性。设计的总体思路还是分为三个阶段：取指、译码、执行。取指：用 WriteWord()函数将待执行的指令存入存储器，通过 IR 取址 。译码：对于每条取出来的 IR 进行译码，按操作码进行分类 。执行：分析操作码不同功能执行 switch 语句。

2. 模拟器架构：

Program（）：负责通过移位运算将指令写入寄存器。

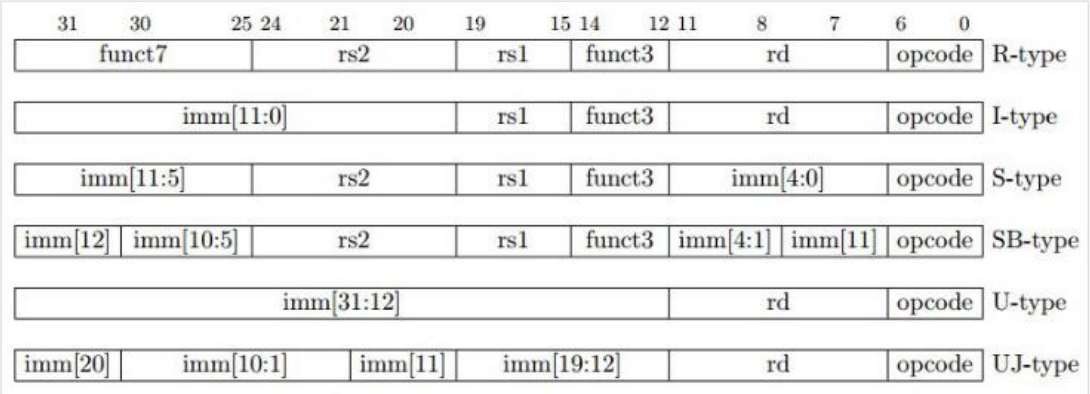
Decode（）：负责译码。

main（）:根据译码结果选择相应的模块执行。

3. 关键代码说明：

（1）取指：

RV32I 的指令长度为 32 位，并且需要在内存中对齐存储，并且是小端存储。一共有 6 种指令格式：R、I、S、U 以及变种 SB、UJ，如下图所示。



所以取指函数可以这样写，针对不同类型的指令：只展示部分，详见源码。

```
void Program(){
    /*U类指令*/
    WriteWord(0,(0x12345<<12)|(1<<7)|(LUI));
    WriteWord(4,(0x2<<12)|(2<<7)|(ALUPC));
    /*J类指令*/
    WriteWord(8,(0<<31)|(4<<21)|(0<<20)|(0<<12)|(3<<7)|(JAL));
    WriteWord(16,(12<<20)|(5<<15)|(0<<12)|(4<<7)|(JALR));
    /*B类指令*/

    WriteWord(24,(0<<31)|(0<<25)|(6<<20)|(5<<15)|(0<<12)|(4<<8)|(0<<7)|(BType));

    WriteWord(32,(0<<31)|(0<<25)|(6<<20)|(3<<15)|(1<<12)|(6<<8)|(0<<7)|(BType));

    WriteWord(44,(0<<31)|(0<<25)|(3<<20)|(6<<15)|(4<<12)|(4<<8)|(0<<7)|(BType));

    WriteWord(52,(0<<31)|(0<<25)|(6<<20)|(3<<15)|(5<<12)|(4<<8)|(0<<7)|(BType));

    WriteWord(60,(0<<31)|(0<<25)|(3<<20)|(6<<15)|(6<<12)|(4<<8)|(0<<7)|(BType));

    WriteWord(68,(0<<31)|(0<<25)|(6<<20)|(3<<15)|(7<<12)|(4<<8)|(0<<7)|(BType));
```

WirteWord 具体实现如下：

```
void WriteWord(uint32_t addr,uint32_t data){
    if(addr>=Msize-wordsize) {
        cout<<"ERROR:地址范围超出内存容量"<<endl;
        return;
    }
    *((uint32_t*)&(M[addr]))=data;
}
```

针对不同的数据类型，还有 WriteByte、Write2Byte，有写的函数肯定也有读的函数，如下：

```
void WriteByte(uint32_t addr,char data){
    if(addr>=Msize) {
        cout<<"ERROR:地址范围超出内存容量"<<endl;
        return;
    }
    M[addr]=data;
}

int32_t Read2Byte(uint32_t addr,bool flag){
    if(addr>=Msize-wordsize/2) {
        cout<<"ERROR:地址范围超出内存容量"<<endl;
        return 0;
    }
    if (flag==1) //返回有符号的
        return *((int16_t*)&(M[addr]));
    else
        return *((uint16_t*)&(M[addr]));
}

void Write2Byte(uint32_t addr,uint32_t data){ /
    if(addr>=Msize-wordsize/2) {
        cout<<"ERROR:地址范围超出内存容量"<<endl;
        return;
    }
    *((uint16_t*)&(M[addr]))=data;
}
int32_t ReadWord(uint32_t addr){
    if(addr>=Msize-wordsize) {
        cout<<"ERROR:地址范围超出内存容量"<<endl;
        return 0;
    }
    return *((int32_t*)&(M[addr]));
}
```

实现的思路都是根据当前的数据类型和操作地址，然后计算出存放的位置，最后返回它在内存的位置。

（2）译码：

译码的思路是通过取出 IR 寄存器的特定位置的内容，如取 opcode。主要是通过与或运算，然后将相应的变量赋值即可。

部分代码如下：

```
void Decode(unsigned int IR){
    opcode= IR & 0x7f;
    rd= (IR>>7)& 0x1f;
    r1= (IR>>15)&0x1f;
    r2= (IR>>20)&0x1f;
    func3=(IR>>12)&0x7;
    func7=(IR>>25)&0x7f;

    imm31_12U = (IR>>12)& 0xfffff;

    imm31J=(IR>>31) & 1;
    imm30_21J=(IR>>21) & 0x3ff;
    imm20J=(IR>>20) &1;
    imm19_12J=(IR>>12) & 0xff;

    imm31_20JR=IR>>20;
    /*B类指令*/
    imm31B=imm31J;
    imm30_25B=(IR>>25)& 0x3f;
    imm11_8B=(IR>>8)&0xf;
    imm7B=(IR>>7)&0x1;
    /*L类指令*/
    imm31_20L=IR>>20;
    /*S类指令*/
    imm31S=imm31J;
    imm30_25S=(IR>>25)&0x3f;
    imm11_7S=(IR>>7) & 0x1f;
    /*I类指令*/
    imm_sign_31_20I=(int)IR>>20;
    shamt=(IR>>20)&0x1f;
```

（3）执行：

执行的分支放在 main 函数里面，通过译码的结果来确定要执行的分支。部分代码如下：

```
Decode(IR) ;
switch(opcode){
    case LUI:{
        cout<<"执行LUI指令:将立即数作为高20位，低12位用0填充，结果放进rd寄存器"<<endl;
        R[rd]=imm31_12U_0;
        break;
    }
    case ALUPC:{
        cout<<"执行ALUPC指令：将立即数作为高20位，低12位用0填充，结果加上此时PC值放入rd寄存器，PC值本身不变"<<endl;
        R[rd]=imm31_12U_0+PC;
        break;
    }
    case JAL:{
        cout<<"执行JAL指令：将立即数有符号扩展*2+pc作为新的pc值，并将原pc+4放进rd寄存器"<<endl;
        R[rd]=PC+4;
        nextPC=PC+imm_sign_31_12J*2;
        break;
    }
    case JALR:{
        cout<<"执行JALR指令：将指令高12位作为立即数有符号扩展*2+r1作为新的pc值，并将原pc+4放进rd"<<endl;
        R[rd]=PC+4;
        nextPC=R[r1]+imm31_20JR*2;
        break;
    }
```

4. 运行结果：

因指令较多，只展示 R、I、S、U、SB、UJ 的若干指令执行结果。

（1）LUI：　（L 类）

操作码：0x37；

测试指令：WriteWord(0,(0x12345<<12)|(1<<7)|(LUI));

指令内容：imm31_12U_0 表示低位用 0 填充。

```
case LUI:{
    cout<<"执行LUI指令:将立即数作为高20位，低12位用0填充，结果放进rd寄存器"<<endl;
    R[rd]=imm31_12U_0;
    break;
}
```

```
----------在执行指令前PC=0x0----------
PC=0x0  IR=0x0
32个寄存器值(16进制)分别为:
R[1]=0  R[2]=0  R[3]=0  R[4]=0  R[5]=0  R[6]=0  R[7]=0  R[8]=0  R[9]=0  R[10]=0 R[11]=0 R[12]=0 R[13]=0 R[14]=0 R[15]=0
R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0 R[25]=0 R[26]=0 R[27]=0 R[28]=0 R[29]=0 R[30]=0
R[31]=0 R[32]=0

执行LUI指令:将立即数作为高20位，低12位用0填充，结果放进rd寄存器
----------执行指令后寄存器的值----------
PC=0x4  IR=0x123450b7
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=0  R[3]=0  R[4]=0  R[5]=0  R[6]=0  R[7]=0  R[8]=0  R[9]=0  R[10]=0 R[11]=0 R[12]=0 R[13]=0 R[14]=0
R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0 R[25]=0 R[26]=0 R[27]=0 R[28]=0 R[29]=0
R[30]=0 R[31]=0 R[32]=0
********************************
是否继续执行指令？(y/n)
```

（2）ALUPC：（L 类）

操作码：0x17；

测试指令：WriteWord(4,(0x2<<12)|(2<<7)|(ALUPC));

指令内容：imm31_12U_0 表示低位用 0 填充

```
case ALUPC:{
    cout<<"执行ALUPC指令：将立即数作为高20位，低12位用0填充，结果加上此时PC值放入rd寄存器，PC值本身不变"<<endl;
    R[rd]=imm31_12U_0+PC;
    break;
}
```

```
----------在执行指令前PC=0x4----------
PC=0x4  IR=0x123450b7
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=0  R[3]=0  R[4]=0  R[5]=0  R[6]=0  R[7]=0  R[8]=0  R[9]=0  R[10]=0 R[11]=0 R[12]=0 R[13]=0 R[14]=0
R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0 R[25]=0 R[26]=0 R[27]=0 R[28]=0 R[29]=0
R[30]=0 R[31]=0 R[32]=0

执行ALUPC指令：将立即数作为高20位，低12位用0填充，结果加上此时PC值放入rd寄存器，PC值本身不变
----------执行指令后寄存器的值----------
PC=0x8  IR=0x2117
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004   R[3]=0  R[4]=0  R[5]=0  R[6]=0  R[7]=0  R[8]=0  R[9]=0  R[10]=0 R[11]=0 R[12]=0 R[13]=0
R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0 R[25]=0 R[26]=0 R[27]=0 R[28]=0
R[29]=0 R[30]=0 R[31]=0 R[32]=0
********************************
```

（3）JAL：（J 类）

操作码：0x68；

测试指令：

WriteWord(8,(0<<31)|(4<<21)|(0<<20)|(0<<12)|(3<<7)|(JAL)); 指令内容：imm31_12U_0 表示低位用 0 填充

```
case JAL:{
    cout<<"执行JAL指令：将立即数有符号扩展*2+pc作为新的pc值，并将原pc+4放进rd寄存器"<<endl;
    R[rd]=PC+4;
    nextPC=PC+imm_sign_31_12J*2;
    break;
}
```

```
----------在执行指令前PC=0x8----------
PC=0x8   IR=0x2117
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004   R[3]=0  R[4]=0  R[5]=0  R[6]=0  R[7]=0  R[8]=0  R[9]=0  R[10]=0 R[11]=0 R[12]=0 R[13]=0
R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0 R[25]=0 R[26]=0 R[27]=0 R[28]=0
R[29]=0 R[30]=0 R[31]=0 R[32]=0

执行JAL指令：将立即数有符号扩展*2+pc作为新的pc值，并将原pc+4放进rd寄存器
----------执行指令后寄存器的值----------
PC=0x10   IR=0x8001e8
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004   R[3]=c  R[4]=0  R[5]=0  R[6]=0  R[7]=0  R[8]=0  R[9]=0  R[10]=0 R[11]=0 R[12]=0 R[13]=0
R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0 R[25]=0 R[26]=0 R[27]=0 R[28]=0
R[29]=0 R[30]=0 R[31]=0 R[32]=0
**********
```

（3）BEQ：（B 类）

操作码：0x63

分支功能码：0；

测试指令：

WriteWord(24,(0<<31)|(0<<25)|(6<<20)|(5<<15)|(0<<12)|(4<<8)|(0<<7)|(BType)); //BEQ：r5=r6=0,所以跳转，令立即数=4，PC=24+4*2=32;注意立即数的符号位为 0

指令内容：

```
case BEQ:{
    cout<<"执行BEQ指令：如果r1里值=r2里值，将立即数有符号填充高20位*2+PC作为PC值"<<endl;
    if(R[r1]==R[r2]) {
        nextPC=PC+imm_sign_31_25B_11_7B*2;
    }
    break;
}
```

```
----------在执行指令前PC=0x18----------
PC=0x18  IR=0xc28267
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004       R[3]=c  R[4]=14 R[5]=0  R[6]=0  R[7]=0  R[8]=0  R[9]=0  R[10]=0 R[11]=0 R[12]=0 R[13]=0
R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0 R[25]=0 R[26]=0 R[27]=0 R[28]=0
R[29]=0 R[30]=0 R[31]=0 R[32]=0

执行BEQ指令：如果r1里值=r2里值，将立即数有符号填充高20位*2+PC作为PC值
----------执行指令后寄存器的值----------
PC=0x20  IR=0x628463
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004       R[3]=c  R[4]=14 R[5]=0  R[6]=0  R[7]=0  R[8]=0  R[9]=0  R[10]=0 R[11]=0 R[12]=0 R[13]=0
R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0 R[25]=0 R[26]=0 R[27]=0 R[28]=0
R[29]=0 R[30]=0 R[31]=0 R[32]=0
******************************
```

(4) LH：　（L 类）

操作码：0x03

分支功能码：1；

指令内容：

```
case LH:{
    cout<<"执行LH指令：将指令高12位作为立即数有符号扩展+r1寄存器的值，作为地址，读取存储器相应地址中的2个字节并扩展到32
    R[rd]=(int)Read2Byte(R[r1]+imm31_20L,1);
    break;
}
```

```
----------在执行指令前PC=0x50----------
PC=0x50  IR=0x3f418283
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004       R[3]=c  R[4]=14 R[5]=fffffffe   R[6]=0  R[7]=0  R[8]=0  R[9]=0  R[10]=0 R[11]=0 R[12]=
R[13]=0 R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0 R[25]=0 R[26]=0 R[27]=
R[28]=0 R[29]=0 R[30]=0 R[31]=0 R[32]=0

执行LH指令：将指令高12位作为立即数有符号扩展+r1寄存器的值，作为地址，读取存储器相应地址中的2个字节并扩展到32位放在rd寄
器
----------执行指令后寄存器的值----------
PC=0x54  IR=0x3f419303
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004       R[3]=c  R[4]=14 R[5]=fffffffe   R[6]=ffffff6fe  R[7]=0  R[8]=0  R[9]=0  R[10]=0 R[11]=
R[12]=0 R[13]=0 R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0 R[25]=0 R[26]=
R[27]=0 R[28]=0 R[29]=0 R[30]=0 R[31]=0 R[32]=0
******************************
```

(5) LW：　（L 类）

操作码：0x03

分支功能码：2；

指令内容：

```
case LW:{
    cout<<"执行LW指令：将指令高12位作为立即数有符号扩展+r1寄存器的值，作为地址，读取存储器相应地址中的4个字节放在rd寄存
    R[rd]=ReadWord(R[r1]+imm31_20L);
    break;
}
```

```
----------在执行指令前PC=0x54----------
PC=0x54  IR=0x3f419303
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004      R[3]=c  R[4]=14 R[5]=fffffffe   R[6]=fffff6fe   R[7]=0  R[8]=0  R[9]=0  R[10]=0 R[11]=0
R[12]=0 R[13]=0 R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0 R[25]=0 R[26]=0
R[27]=0 R[28]=0 R[29]=0 R[30]=0 R[31]=0 R[32]=0

执行LW指令：将指令高12位作为立即数有符号扩展+r1寄存器的值，作为地址，读取存储器相应地址中的4个字节放在rd寄存器
----------执行指令后寄存器的值----------
PC=0x58  IR=0x3f41a383
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004      R[3]=c  R[4]=14 R[5]=fffffffe   R[6]=fffff6fe   R[7]=1234f6fe   R[8]=0  R[9]=0  R[10]=0
R[11]=0 R[12]=0 R[13]=0 R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0 R[25]=0
R[26]=0 R[27]=0 R[28]=0 R[29]=0 R[30]=0 R[31]=0 R[32]=0
**********
是否继续执行指令？(y/n)
```

（6）SH：（S 类）

操作码：0x03

分支功能码：0x1；

指令内容：

```
case SH:{
    cout<<"执行SH指令：将立即数有符号扩展32位与r1寄存器相加，作为存储器地址，将r2寄存器中值低16位存进存储器"<<endl;
    Write2Byte(R[r1]+imm_sign_31_25S_11_7S,(R[r2]&0xffff));
    cout<<"执行指令后相应内存值为0x"<<Read2Byte(512,0)<<endl;//便于测试
    break;
}
```

```
----------在执行指令前PC=0x64----------
PC=0x64  IR=0x20758023
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004      R[3]=c  R[4]=14 R[5]=fffffffe   R[6]=fffff6fe   R[7]=1234f6fe   R[8]=fe R[9]=f6fe
R[10]=0 R[11]=0 R[12]=0 R[13]=0 R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0
R[25]=0 R[26]=0 R[27]=0 R[28]=0 R[29]=0 R[30]=0 R[31]=0 R[32]=0

执行SH指令：将立即数有符号扩展32位与r1寄存器相加，作为存储器地址，将r2寄存器中值低16位存进存储器
执行指令后相应内存值为0xf6fe
----------执行指令后寄存器的值----------
PC=0x68  IR=0x20759023
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004      R[3]=c  R[4]=14 R[5]=fffffffe   R[6]=fffff6fe   R[7]=1234f6fe   R[8]=fe R[9]=f6fe
R[10]=0 R[11]=0 R[12]=0 R[13]=0 R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0
R[25]=0 R[26]=0 R[27]=0 R[28]=0 R[29]=0 R[30]=0 R[31]=0 R[32]=0
**********
是否继续执行指令？(y/n)
```

（7）ADDI：　（I 类）

操作码：0x13

分支功能码：0；

指令内容：

```
case ADDI:{
    cout<<"执行ADDI指令:将立即数符号扩展与r1相加，若有溢出省略高位，保留低32位放进rd中"<<endl;
    R[rd]=(R[r1]+imm_sign_31_20I)&0xffffffff;
    break;
}
```

```
----------在执行指令前PC=0x6c----------
PC=0x6c  IR=0x2075a023
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004       R[3]=c  R[4]=14 R[5]=fffffffe   R[6]=fffff6fe   R[7]=1234f6fe   R[8]=fe R[9]=f6fe
R[10]=0 R[11]=0 R[12]=0 R[13]=0 R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0
R[25]=0 R[26]=0 R[27]=0 R[28]=0 R[29]=0 R[30]=0 R[31]=0 R[32]=0

执行ADDI指令:将立即数符号扩展与r1相加，若有溢出省略高位，保留低32位放进rd中
----------执行指令后寄存器的值----------
PC=0x70  IR=0xfff18513
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004       R[3]=c  R[4]=14 R[5]=fffffffe   R[6]=fffff6fe   R[7]=1234f6fe   R[8]=fe R[9]=f6fe
R[10]=b R[11]=0 R[12]=0 R[13]=0 R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0
R[25]=0 R[26]=0 R[27]=0 R[28]=0 R[29]=0 R[30]=0 R[31]=0 R[32]=0
********************************
是否继续执行指令? (y/n)
```

（8）XORI：　（I 类）

分支功能码：0x4；

指令内容：

```
case XORI:{
    cout<<"执行XORI指令:进行异或操作，rd=r1^imm(符号扩展到32位)"<<endl;
    R[rd]=R[r1]^imm_sign_31_20I;
    break;
}
```

```
----------在执行指令前PC=0x78----------
PC=0x78  IR=0xfff1b613
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004       R[3]=c  R[4]=14 R[5]=fffffffe   R[6]=fffff6fe   R[7]=1234f6fe   R[8]=fe R[9]=f6fe
R[10]=b R[11]=0 R[12]=1 R[13]=0 R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0
R[25]=0 R[26]=0 R[27]=0 R[28]=0 R[29]=0 R[30]=0 R[31]=0 R[32]=0

执行XORI指令:进行异或操作，rd=r1^imm(符号扩展到32位)
----------执行指令后寄存器的值----------
PC=0x7c  IR=0xff33c693
32个寄存器值(16进制)分别为:
R[1]=12345000   R[2]=2004       R[3]=c  R[4]=14 R[5]=fffffffe   R[6]=fffff6fe   R[7]=1234f6fe   R[8]=fe R[9]=f6fe
R[10]=b R[11]=0 R[12]=1 R[13]=edcb090d R[14]=0 R[15]=0 R[16]=0 R[17]=0 R[18]=0 R[19]=0 R[20]=0 R[21]=0 R[22]=0 R[23]=0
R[24]=0 R[25]=0 R[26]=0 R[27]=0 R[28]=0 R[29]=0 R[30]=0 R[31]=0 R[32]=0
********************************
```

（9）ADD：（R 类）

分支功能码：0；

FUNC7:0

指令内容：

```
case ADD:{
    cout<<"执行ADD指令：rd=r1+r2，取低32位，忽略溢出"<<endl;
    R[rd]=R[r1]+R[r2];
    break;
}
```

```
执行ADD指令：rd=r1+r2，取低32位，忽略溢出
----------执行指令后寄存器的值----------
PC=0x94   IR=0xa189b3
32个寄存器值(16进制)分别为：
R[1]=12345000   R[2]=2004       R[3]=c  R[4]=14 R[5]=fffffffe   R[6]=fffff6fe   R[7]=1234f6fe   R[8]=fe R[9]=f6fe       R[10]=b R[11]=0 R[
d       R[14]=fffff05  R[15]=4 R[16]=ffff6fe0  R[17]=ffffff6f   R[18]=ffffff6f  R[19]=17        R[20]=0 R[21]=0 R[22]=0 R[23]=0 R[24]=0 R[
=0      R[28]=0 R[29]=0 R[30]=0 R[31]=0 R[32]=0
******************************
```

5. 遇到的问题：

由于是沿用之前的代码，因此并没有遇到太大的问题，只是需要多花一点时间去阅读代码，理顺模拟器的执行逻辑和流程即可。

6. 实验心得：

一开始的时候想着是将程序的输入改成 bin 文件的（之前的 GitHub 上面的模拟器就是输入的 bin 文件）。但那个和我这个有些出入，不适合大改，不然花费的时间很多。尝试一番无果后，无奈沿用原来的将指令写死。

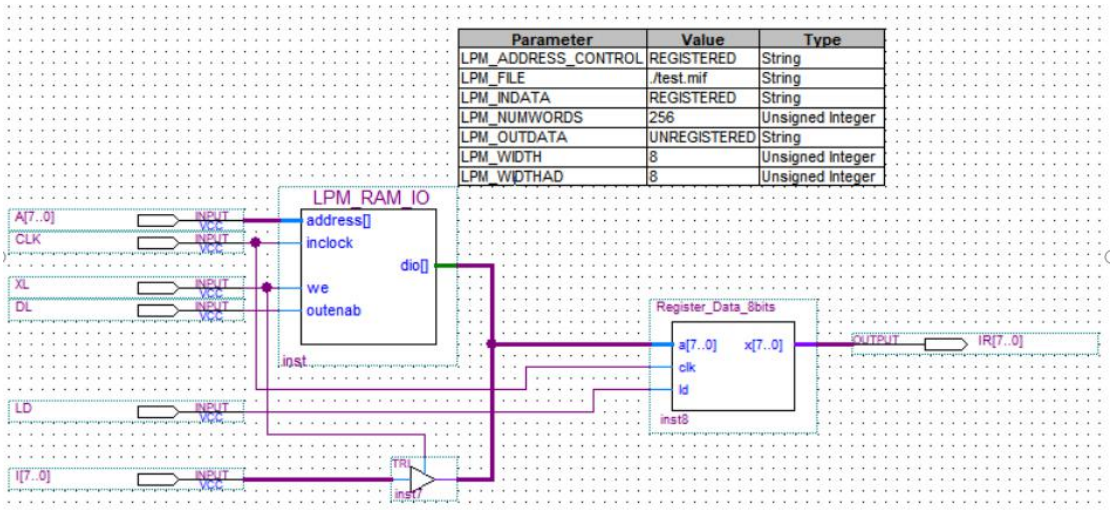
三、简单存储器

1. 设计思路：

由于对 vhdl 语言不是很熟悉，故采用 quarts 自带的 LPM_RAM_IO 定制一个 256*8 的 RAM 进行设计，从而实现读和写的操作。

2. 关键代码说明：

原理图如下：

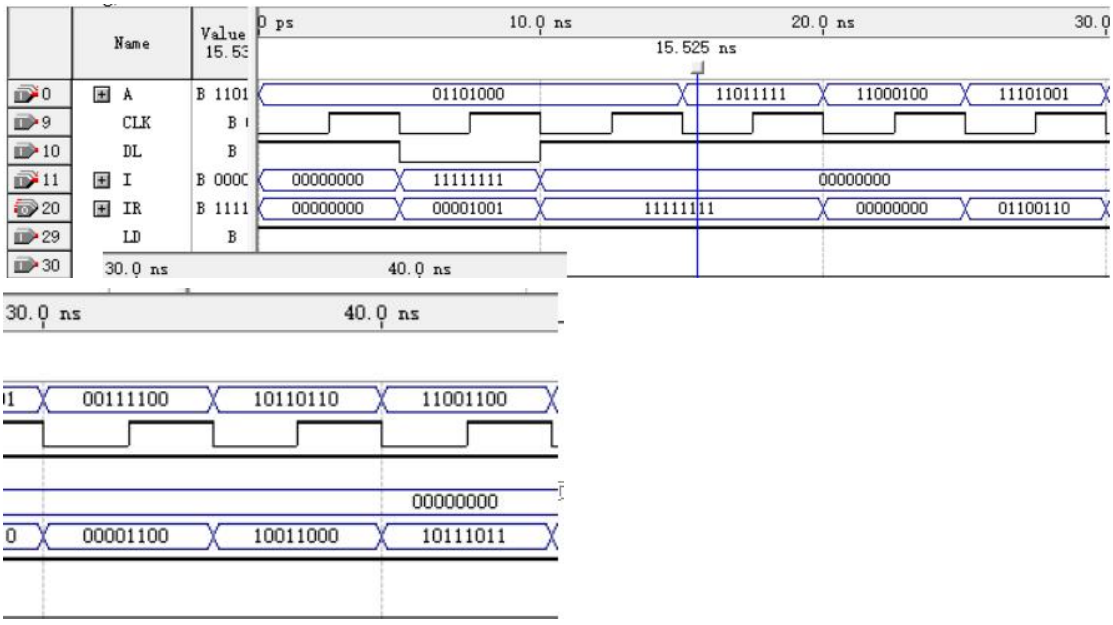


对应的 mif 文件

Addr	+000	+001	+010	+011	+100	+101	+110	+111
0000000	01111111	10000101	10001011	10010010	10011000	10011110	10100100	10101010
0000100	10110000	10110110	10111011	11000001	11000110	11001011	11010000	11010101
0001000	11011001	11011101	11100010	11100101	11101001	11101100	11101111	11110010
0001100	11110101	11110111	11111001	11111011	11111100	11111101	11111110	11111110
0010000	11111111	11111110	11111110	11111101	11111100	11111011	11111001	11110111
0010100	11110101	11110010	11101111	11101100	11101001	11100101	11100010	11011101
0011000	11011001	11010101	11010000	11001011	11000110	11000001	10111011	10110110
0011100	10110000	10101010	10100100	10011110	10011000	10010010	10001011	10000101
0100000	01111111	01111001	01110011	01101100	01100110	01100000	01011010	01010100
0100100	01001110	01001000	01000011	00111101	00111000	00110011	00101110	00101001
0101000	00100101	00100001	00011100	00011001	00010101	00010010	00001111	00001100
0101100	00001001	00000111	00000101	00000011	00000010	00000001	00000000	00000000
0110000	00000000	00000000	00000000	00000001	00000010	00000011	00000101	00000111
0110100	00001001	00001100	00001111	00010010	00010101	00011001	00011100	00100001
0111000	00100101	00101001	00101110	00110011	00111000	00111101	01000011	01001000
0111100	01001110	01010100	01011010	01100000	01100110	01101100	01110011	01111001
1000000	01111111	10000101	10001011	10010010	10011000	10011110	10100100	10101010
1000100	10110000	10110110	10111011	11000001	11000110	11001011	11010000	11010101
1001000	11011001	11011101	11100010	11100101	11101001	11101100	11101111	11110010
1001100	11110101	11110111	11111001	11111011	11111100	11111101	11111110	11111110
1010000	11111111	11111110	11111110	11111101	11111100	11111011	11111001	11110111
1010100	11110101	11110010	11101111	11101100	11101001	11100101	11100010	11011101
1011000	11011001	11010101	11010000	11001011	11000110	11000001	10111011	10110110
1110000	00000000	00000000	00000000	00000001	00000010	00000011	00000101	00000111
1110100	00001001	00001100	00001111	00010010	00010101	00011001	00011100	00100001
1111000	00100101	00101001	00101110	00110011	00111000	00111101	01000011	01001000
1111100	01001110	01010100	01011010	01100000	01100110	01101100	01110011	01111001

3. 运行结果：

波形仿真如下：



2.5ns+:第一个 CLK 时钟上升沿：

DL=1,XL=0,A=01101000,LD=1,I=00000000,输出 IR=00000000,RAM 进行读操作，但因为寄存器是 CLK 上升沿触发，所以 IR 还未发生改变。

5.0ns+：第一个 CLK 时钟下降沿：

DL=0,XL=1,A=01101000,LD=1,I=11111111,输出 IR=00001001,正是 RAM 中地址 A=01101000 所对应的数据。

7.5ns+:第二个 CLK 时钟上升沿：

DL=0,XL=1,A=01101000,LD=1,I=11111111,输出 IR=00001001,RAM 进行写操作，I 的值将写入地址 A 的区域。

10.0ns+:第二个 CLK 时钟下降沿：

DL=1,XL=1,A=01101000,LD=1,I=00000000,输出 IR=11111111，地址 A 和第一个 CLK 下降沿相同，存储的数据却不同，这次数据为第二个 CLK 上升沿写入的 I 值，说明读和写操作正确。

如下图：

CLK	we	outenab	功能
⎍	0	0	Dio<=高阻态Z
⎍	1	0	Dio的数据写入address所指定的存储单元
⎍	0	1	address所指定的存储单元数据从dio输出

4. 遇到的问题：

Mif 文件初始化内容比较多，一开始手动输入后来测试的时候出现值的错误，后来改用 C++的文件输入和输出流去生成 mif 文件的内容。

5. 实验心得：

一开始是打算参考老师的 mem 代码进行实验，但后来配置 GHDL 的时候出现了问题，系统不识别 ghdl 的指令，导致 vhdl 文件不能被编译，同时也无法生成波形文件，后来几经周折决定还是用 quarts 来完成这个实验，通过本次实验复习了 RAM 的原理以及它的读写机制和过程。

四、RV32I 的 CPU 设计

1.设计思路：

将通用寄存器，RAM，ROM 集中设计在了一个代码中，实现 RISV-V 的前 37 条指令。整体分为三个阶段去设计，分别是：取指、译码、执行。

2. CPU 架构：

程序被分为三个部分。取指 译码 执行 一是对输入输出信号、寄存器变量的定义与初始化，二是 获取寄存器变量之后进行指令相 应的计算与赋值，最后是写回操作。

3. 关键代码说明：

（1）指令相关的寄存器，RAM 均使用信号数组实现：

```
type regfile is array(natural range<>) of std_logic_vector(31 downto 0);
signal regs: regfile(31 downto 0);

type memoryfile is array(natural range<>) of std_logic_vector(31 downto 0);
signal mems: memoryfile(3 downto 0);
```

（2）Risc-v 的特点之一是所有涉及到寄存器使用的指令中寄存器的位置都是固定的，而 opcode，func3，func7 则可以帮助我们快速的确认到具体的指令。常量的定义：完整见源码。

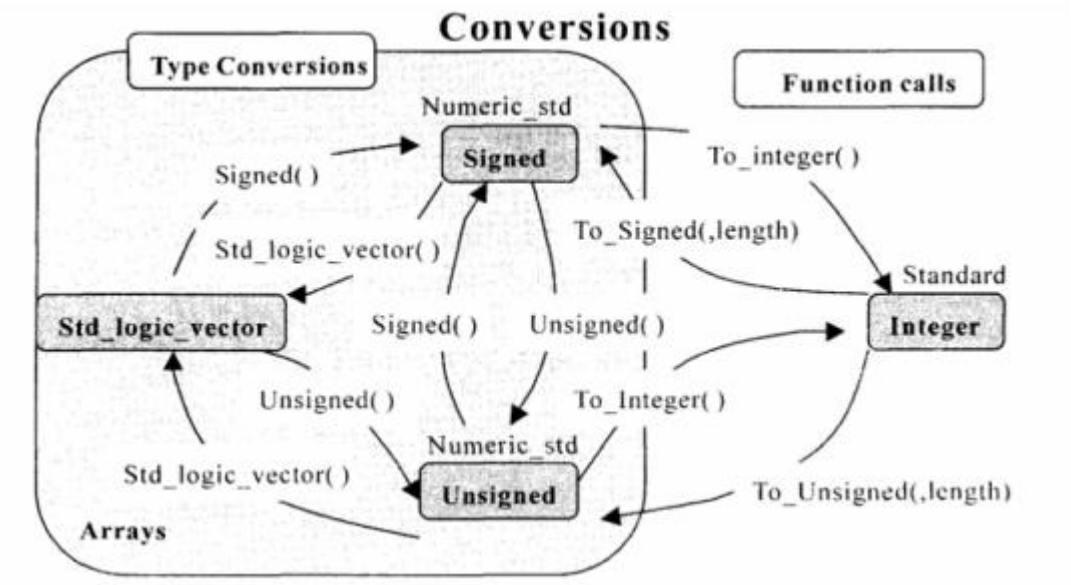
```
-- utype instructions, using opcode
constant rtype_lui: std_logic_vector(6 downto 0) := B"0110111";
constant rtype_auipc: std_logic_vector(6 downto 0) := B"0010111";

-- jtype
constant jtype_jal: std_logic_vector(6 downto 0) := B"1101111";

-- itype load instructions, using opcode, funct3
constant itype_load: std_logic_vector(6 downto 0) := B"0000011";
constant itype_jalr: std_logic_vector(6 downto 0) := B"1100111";
constant itype_lb: std_logic_vector(2 downto 0) := B"000";
constant itype_lh: std_logic_vector(2 downto 0) := B"001";
constant itype_lw: std_logic_vector(2 downto 0) := B"010";
constant itype_lbu: std_logic_vector(2 downto 0) := B"100";
constant itype_lhu: std_logic_vector(2 downto 0) := B"101";
```

```
-- rtype alu operations, using opcode, funct3, funct7
constant rtype_alu: std_logic_vector(6 downto 0) := B"0110011";
constant rtype_addsub: std_logic_vector(2 downto 0) := B"000";
constant rtype_add: std_logic_vector(6 downto 0) := B"0000000";
constant rtype_sub: std_logic_vector(6 downto 0) := B"0100000";
constant rtype_sll: std_logic_vector(2 downto 0) := B"001";
constant rtype_slt: std_logic_vector(2 downto 0) := B"010";
constant rtype_sltu: std_logic_vector(2 downto 0) := B"011";
constant rtype_xor: std_logic_vector(2 downto 0) := B"100";
constant rtype_srlsra: std_logic_vector(2 downto 0) := B"101";
constant rtype_srl: std_logic_vector(6 downto 0) := B"0000000";
constant rtype_sra: std_logic_vector(6 downto 0) := B"0100000";
constant rtype_or: std_logic_vector(2 downto 0) := B"110";
constant rtype_and: std_logic_vector(2 downto 0) := B"111";
```

（3）指令设计，以 R 类指令为例子：ADD，SUB 指令即为算术加减，需要使用到库 ieee.numeric_std.all 其中对加减法的操 作数类型进行了约束，只有整型 integer 才能参与运算，而我们的 std_logic 与整型之间 的转换也需要两次才能做到，首先需要转换成 unsigned 或 signed 类型，接下来再转换 成 integer 类型。其具体的关系可见下图：



故我们的 add 与 sub 指令可以设计为：

```
std_logic_vector( to_unsigned( ( to_integer( unsigned(rs1_data)) +
to_integer( unsigned(rs2_data)) ) , 32 ) )

std_logic_vector( to_unsigned( ( to_integer( unsigned(rs1_data)) -
to_integer( unsigned(rs2_data)) ) , 32 ) )
```

之后是移位指令： 移位在 ieee.numeric_std.all 提供的函数中是 shift_left 与 shift_right，有无符合需要根据 参数而定，移位的指令设计为：

```
std_logic_vector( shift_left(unsigned(rs1_data) , to_integer(unsigned(rs2_data) ) ) )
std_logic_vector(shift_right(unsigned(rs1_data), to_integer(unsigned(rs2_data))))
std_logic_vector(shift_right(signed(rs1_data), to_integer(signed(rs2_data))))
```

按位与或，异或操作：

rs1_data xor rs2_data

when funct3 = rtype_xor else

rs1_data or rs2_data

when funct3 = rtype_or else

rs1_data and rs2_data

when funct3 = rtype_and else

（4）cpu 实体设计：

```
entity my_cpu is
    port(
        clk: in std_logic;
        reset: in std_logic;
        inst: in std_logic_vector(31 downto 0);
        inst_addr: out std_logic_vector(31 downto 0);  -- 指令地址
        inst_read: out std_logic;
        data_addr: buffer std_logic_vector(31 downto 0);  -- 数据地址
        data: inout std_logic_vector(31 downto 0);
        data_read: out std_logic;
        data_write: out std_logic;
        write_avi: out std_logic;
        is_alu: out std_logic;
        reg_val: out std_logic_vector(31 downto 0);
        is_jal: out std_logic;
        opcode_val: out std_logic_vector(6 downto 0)
        );
end entity;
```

（5）CPU 结构体定义：

```
signal rd_write: std_logic;
signal rd_data: std_logic_vector(31 downto 0);

signal opcode: std_logic_vector(6 downto 0);

signal rd: std_logic_vector(4 downto 0);
signal rs1: std_logic_vector(4 downto 0);
signal rs2: std_logic_vector(4 downto 0);
signal rs1_data: std_logic_vector(31 downto 0);
signal rs2_data: std_logic_vector(31 downto 0);

signal funct3: std_logic_vector(2 downto 0);
signal funct7: std_logic_vector(6 downto 0);

signal jal_imm20_1: std_logic_vector(20 downto 1);
signal jal_offset: std_logic_vector(31 downto 0);

signal utype_imm31_12: std_logic_vector(31 downto 12);

signal itype_imm11_0: std_logic_vector(11 downto 0);

signal btype_imm12_1: std_logic_vector(12 downto 1);

signal rtype_alu_result: std_logic_vector(31 downto 0);

signal pc: std_logic_vector(31 downto 0);
signal ir: std_logic_vector(31 downto 0);

signal next_pc: std_logic_vector(31 downto 0);
```

只贴部分内容，结构体声明了计算是需要使用的变量。ir 表示当前执行的指令，pc 表当前的指令的地 址；7 位的 opcode，3 位的 funct3，7 位的 funct7，这三个变量读取 ir 的指令，取到对应的值。寄存器 rd,rs1,rs2 存储 ir 中读取到的对应操作值地址，src1,src2 将 rs1,rs2 中的地址对于 的 reg 中的值转为 32 位保存。

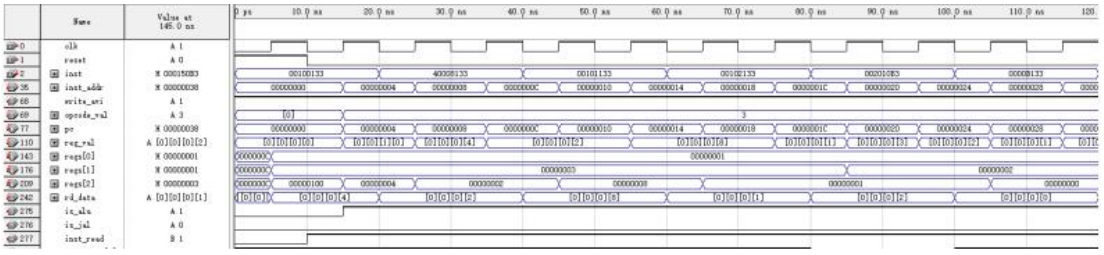
4. 运行结果：

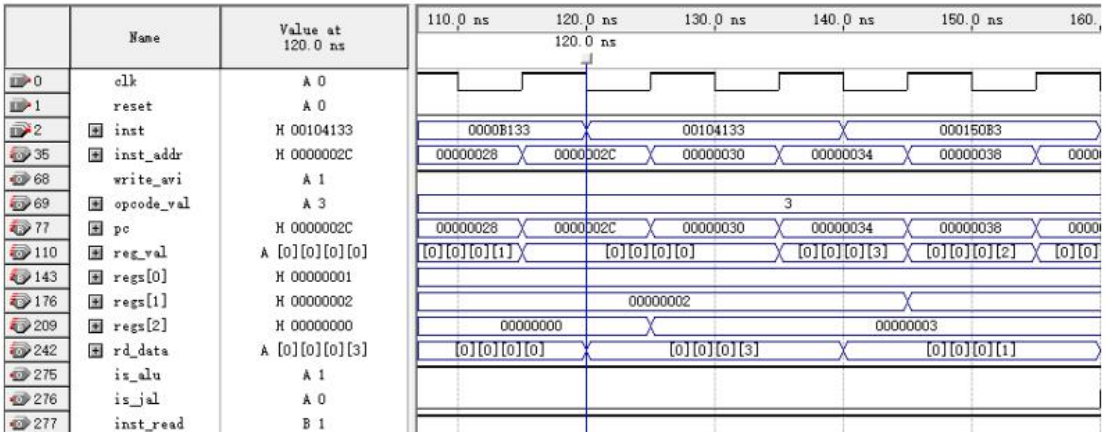
（1）R 类型指令测试：

首先设置测试指令：

		regs(0)	regs(1)	regs(2)	instruction	funct7(7bit	rs2	rs1	funct3(3bits)	rd	opcode(7bits)
		1	3	0x100							
r0+r1=>r2	add 0 1 2	1	3	4	0x00100133	"0000000"	"00001"	"00000"	"000"	"00010"	"0110011"
r1-r0=>r2	sub 1 0 2	1	3	2	0x40008133	"0100000"	"00000"	"00001"	"000"	"00010"	"0110011"
(r1<<r0)=>r2	sll 0 1 2	1	3	8	0x00101133	"0000000"	"00001"	"00000"	"001"	"00010"	"0110011"
	slt 2 1 0	1	3	1	0x00102133	"0000000"	"00001"	"00000"	"010"	"00010"	"0110011"
	sll 0 2 1	1	2	1	0x002010b3	"0000000"	"00010"	"00000"	"001"	"00001"	"0110011"
	sltu 1 0 2	1	2	0	0x0000b133	"0000000"	"00000"	"00001"	"011"	"00010"	"0110011"
	xor 0 1 2	1	2	3	0x00104133	"0000000"	"00001"	"00000"	"100"	"00010"	"0110011"
	srl 2 0 1	1	1	3	0x000150b3	"0000000"	"00000"	"00010"	"101"	"00001"	"0110011"

Quarts 仿真结果如下：





无条件跳转指令 JAL：该指令需要将下一条指令的地址 pc+4 存储到目的寄存器 rd 中，指令中提供一个立即数，该立即数作为偏移量加到当前的 pc 地址上作为下一个 pc 的值。

立即数的获得：

jal_imm20_1 <= ir(31) & ir(19 downto 12) & ir(20) & ir(30 downto 21)；这里需要说明的是指令并不参与组成该立即数的最低位，因为需要保证跳转之后的地 址是 2 的整数倍。指令的 19 到 12 为为 10 downto 1。 寄存器保留下一个地址的 PC 值： rd_data <=

std_logic_vector(to_unsigned((to_integer(unsigned(pc))+4) , 32)) when opcode = jtype_jal or opcode = itype_jalr else 该值保存了跳转命令结束后应该返回的地址。而需要跳转的 pc 值直接存储到 pc 寄存 器中：

next_pc<=std_logic_vector(to_unsigned((to_integer(unsigned(pc))+ to_integer(unsigned(jal_offset))) , 32)) when opcode = jtype_jal 。

与之相似的还有 LOAD 指令，JALR 指令 同样是通过偏移量找到地址，而该地址对应的却是 RAM，而我们的目的则是将 RAM 的特定地址的值加载到目的寄存器 rd 中。 具体的设计如下： Offset 的值获取： itype_imm11_0 <= ir(31 downto 20); 再将其加到源寄存器 rs1 上： load_addr <= std_logic_vector(to_signed((to_integer(signed(rs1)) + to_integer(signed(itype_imm11_0))),32));

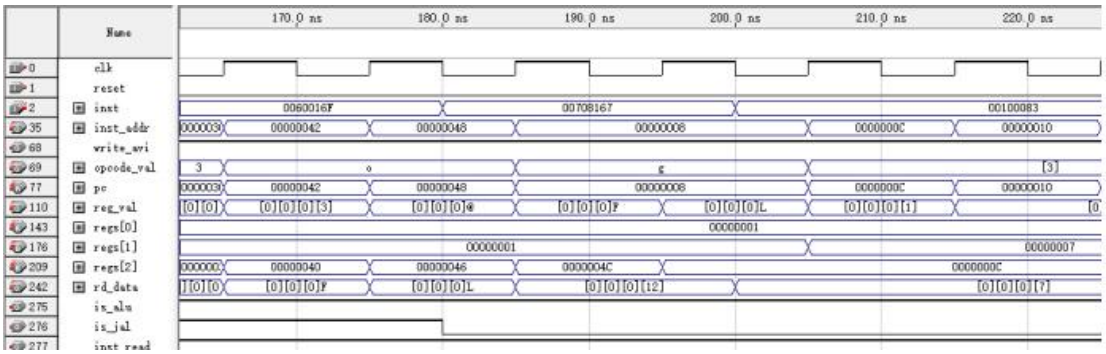
这里考虑到接下来的 s 类型指令也有类似的操作，因此将这个需要装载的地址保存到 一个统一的寄存器中： data_addr <= load_addr when opcode=itype_load else store_addr; 从 RAM 中取得该值并赋值给目的寄存器 rd： rd_data <= mems(to_integer(unsigned(data_addr))) when opcode=itype_load LOAD 指令又能够细分为 LW,LH,LHU,LBL,LBU 五条指令。

仿真结果如下：

测试数据：

jal	1	1	0x40	0x0060016f	"0"	"000000011"	"0"	"00000000"	"00010"	"1101111"
jalr	1	1	0x0c	0x00708167		"000000000111"	"00001"	"000"	"00010"	"1100111"
lload	1	1	8	0x00100083		"000000000001"	"00001"	"000"	"00001"	"0000011"

仿真结果：



（2）B 类型指令测试：

B 类型指令是有条件的跳转指令，当满足条件时将当前 pc 加上偏移量作为下一个 pc。需要满足的条件通过比较两个寄存器的值来决定，BEQ 和 BNE 分别是判断两个源 寄存器 rs1 和 rs2 是/否相等，BLT 和 BLTU 分别使用有符号数和无符号数判断 rs1 小于 rs2；BGE 和 BGEU 则是判断 rs1 大于 rs2。

指令的设计如下：首先得到偏移量值：

btype_imm12_1 <= ir(31) & ir(7) & ir(30 downto 25) & ir(11 downto 8)；由于偏移量时 2 的倍数，这里最低位一定是 0：

branch_target(13 downto 0) <= btype_imm12_1 & '0' ;

branch_target(31 downto 14) <= (others => btype_imm12_1(12));

跳转条件：

branch_taken <='1' when (rs1 = rs2 and funct3 = btype_beq) or (rs1 /= rs2 and funct3 = btype_bne)

or (signed(rs1) < signed(rs2) and funct3 = btype_blt)

or (unsigned(rs1) < unsigned(rs2) and funct3 = btype_bltu)

or (signed(rs1) > signed(rs2) and funct3 = btype_bge)

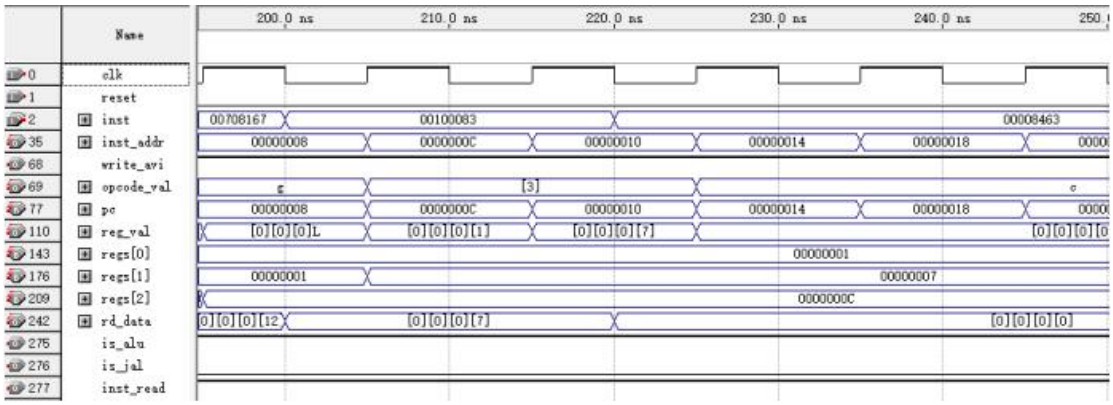
or (unsigned(rs1) > unsigned(rs2) and funct3 = btype_bgeu)

else '0';

pc 的值： next_pc <= when opcode = btype_branch and branch_taken ='1';

测试数据：

仿真结果：



5. 遇到的问题：

借鉴了一下夏季小学期做硬件 CPU 的同学的代码并没有遇到多大的问题，难点在于读懂每段代码的作用以及仿真结果。

6. 实验心得：

由于很久没有接触硬件设计语言了，之前一直都是用 C++写的 cpu 模拟器。硬件理解起来比较难，所以在做它的时候尽管有先前同学的代码，但也花了很多时间去复现他们的结果。