微处理器实验报告

实验一：汇编器

班级：物联1601

学号：201608010315

姓名：刘祺

实验目标

设计一个 RISC-V 基本整数指令集汇编器，能够实现汇编指令向二进制的转化。

实验要求

采用 C/C++编写程序

汇编器的输入是模拟的汇编指令文件

汇编器的输出是汇编指令经过汇编之后的二进制指令文件

实验内容

1.汇编器简介

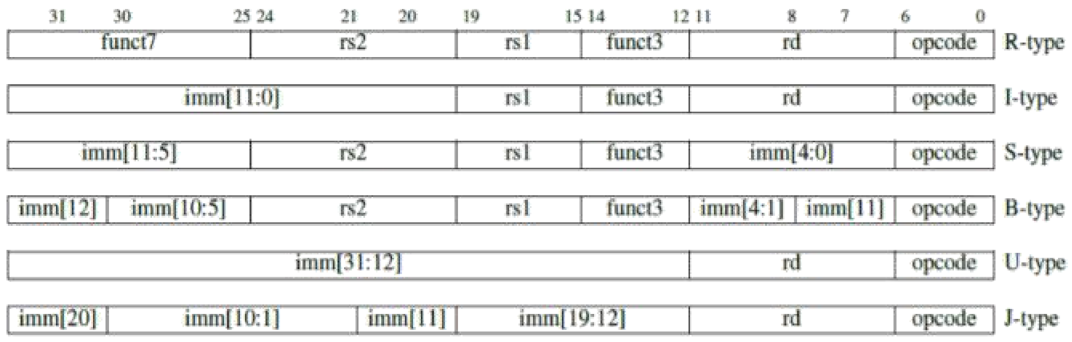
汇编器（Assembler）是将汇编语言翻译为机器语言的程序。一般而言，汇 编生成的是目标代码，需要经链接器（Linker）生成可执行代码才可以执行。

汇编语言是一种以处理器指令系统为基础的低级语言，采用助记符表达指令操作码，采用标识符表示指令操作数。作为一门语言，对应于高级语言的编译器，需要一个“汇编器”来把汇编语言原文件汇编成机器可执行的代码。

2.RISC- V 指令集内容

我们在这里编写的是 RV32I 指令集，其包含了六种基本指令格式，分别是：用于寄存器-寄存器操作的 R 类型指令，用于短立即数和访存 load 操作的 I 型指令，用于访存 store 操作的 S 型指令，用于条件跳转操作的 B 类型指令，用于长立即数的 U 型指令和用于无条件跳转的 J 型指令。

3.RISC- V 指令集编码格式


31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

4.RISC- V 指令

Category Name	**Fmt**	**RV32I Base**	
Shifts			
Shift Left Logical	R	SLL	rd,rs1,rs2
Shift Left Log.Imm.	I	SLLI	rd,rs1,shamt
Shift Right Logical	R	SRL	rd,rs1,rs2
Shift Right Log.Imm.	I	SRLI	rd,rs1,shamt
Shift Right Arithmetic	R	SRA	rd,rs1,rs2
Shift Right Arith.Imm.	I	SRAI	rd,rs1,shamt
Arithmetic			
ADD	R	ADD	rd,rs1,rs2
ADD Immediate	I	ADDI	rd,rs1,imm
SUBtract	R	SUB	rd,rs1,rs2
Load Upper Imm	U	LUI	rd,imm
Add Upper Imm to PC	U	AUIPC	rd,imm
Logical			
XOR	R	XOR	rd,rs1,rs2
XOR Immediate	I	XORI	rd,rs1,imm
OR	R	OR	rd,rs1,rs2

OR Immediate	I	ORI	rd,rs1,imm

AND	R	AND	rd,rs1,rs2
AND Immediate	I	ANDI	rd,rs1,imm
Category Name	**Fmt**	**RV32I Base**	
Compare			
Set <	R	SLT	rd,rs1,rs2
Set < Immediate	I	SLTI	rd,rs1,rs2
Set < Unsigned	R	SLTU	rd,rs1,rs2
Set < Imm Unsigned	I	SLTIU	rd,rs1,imm
Branches			
Branch =	B	BEQ	rs1,rs2,imm
Branch ≠	B	BNE	rs1,rs2,imm
Branch <	B	BLT	rs1,rs2,imm
Branch ≥	B	BGE	rs1,rs2,imm
Branch < Unsigned	B	BLTU	rs1,rs2,imm
Branch ≥ Unsigned	B	BGEU	rs1,rs2,imm
Jump & Link			
J&L	J	JAL	rd,imm
Jump & Link Register	I	JALR	rd,rs1,imm
Synch			
Synch thread	I	FENCE	
Synch Instr & Data	I	FENCE.I	
Environment			
CALL	I	ECALL	
BREAK	I	EBREAK	
Control Status Register(CSR)			
Read/Write	I	CSRRW	rd,csr,rs1
Read & Set Bit	I	CSRRS	rd,csr,rs1
Read & Clear Bit	I	CSRRC	rd,csr,rs1
Read/Write Imm	I	CSRRWI	rd,csr,imm
Read & Set Bit Imm	I	CSRRSI	rd,csr,imm
Read & Clear Bit Imm	I	CSRRCI	rd,csr,imm
Loads			
Load Byte	I	LB	rd,rs1,imm
Load Halfword	I	LH	rd,rs1,imm
Load Byte Unsigned	I	LBU	rd,rs1,imm
Load Half Unsigned	I	LHU	rd,rs1,imm
Load Word	I	LW	rd,rs1,imm
Stores			
Store Byte	S	SB	rs1,rs2,imm
Store Halfword	S	SH	rs1,rs2,imm
Store Word	S	SW	rs1,rs2,imm

汇编器程序框架

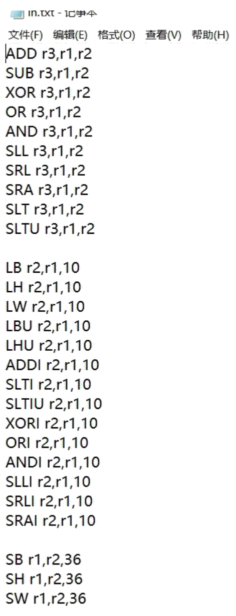
我们将模拟器的框架设计如下：

```
<标号>:add x1, x2, x3
<标号>:10101010…

汇编程序文件  file.asm
一行一个汇编语句

初始化地址计数器  addr_counter = 0;
while(file.asm 没有到文件尾) {
    读入一行
    while(读入的是纯标号且不是文件尾) { 继续读一行 }

    拆开行，得到标号（有可能没有），操作码或者伪指令助记符，操作数

    if(有标号) { 记下标号和当前地址计数器的值，保存到符号表；
        查看未决汇编语句是否需要这个标号，并解决
    }
    if(操作码助记符) {
        生成操作码编码;
```

```
            操作数 -> 寄存器编号或者立即数

            if(操作数是标号) { 查找符号表，如果查到，计算得到偏移量；
                如果没查到，记下当前汇编语句和地址
                }

            生成指令的二进制表示 }

        else(伪指令助记符) { 根据伪指令含义执行相应转换 }

    }
```

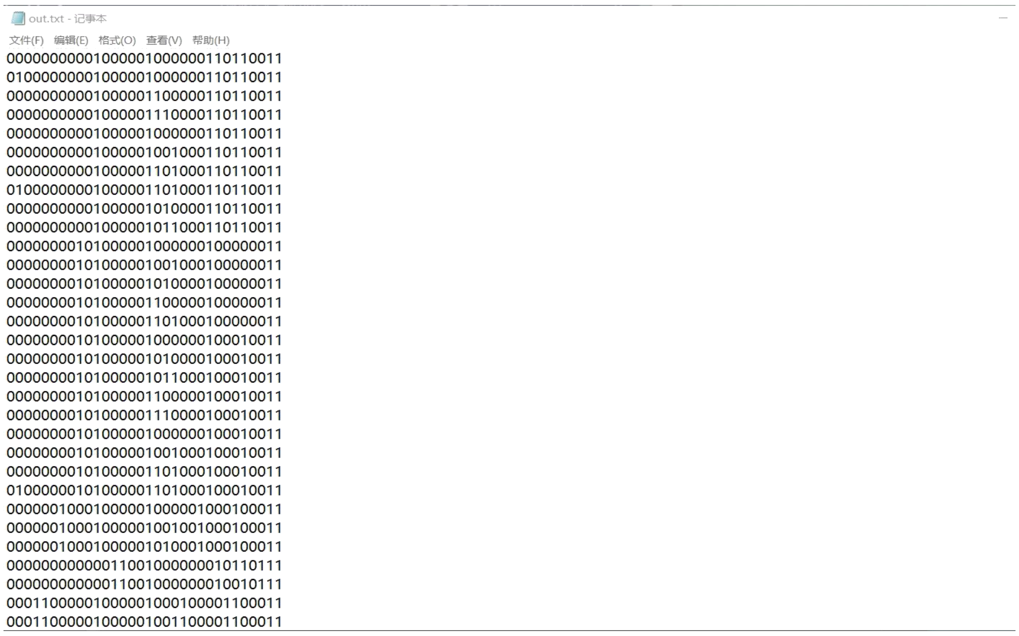
测试

模拟器输入如下



```
ADD r3,r1,r2
SUB r3,r1,r2
XOR r3,r1,r2
OR r3,r1,r2
AND r3,r1,r2
SLL r3,r1,r2
SRL r3,r1,r2
SRA r3,r1,r2
SLT r3,r1,r2
SLTU r3,r1,r2

LB r2,r1,10
LH r2,r1,10
LW r2,r1,10
LBU r2,r1,10
LHU r2,r1,10
ADDI r2,r1,10
SLTI r2,r1,10
SLTIU r2,r1,10
XORI r2,r1,10
ORI r2,r1,10
ANDI r2,r1,10
SLLI r2,r1,10
SRLI r2,r1,10
SRAI r2,r1,10

SB r1,r2,36
SH r1,r2,36
SW r1,r2,36
```

将结果输入至 out.txt 文件中：



```
00000000001000001000000110110011
01000000001000001000000110110011
00000000001000001100000110110011
00000000001000001110000110110011
00000000001000001000000110110011
00000000001000001001000110110011
00000000001000001101000110110011
01000000001000001101000110110011
00000000001000001010000110110011
00000000001000001011000110110011
00000000101000001000000100000011
00000000101000001001000100000011
00000000101000001010000100000011
00000000101000001100000100000011
00000000101000001101000100000011
00000000101000001000000100010011
00000000101000001010000100010011
00000000101000001011000100010011
00000000101000001100000100010011
00000000101000001110000100010011
00000000101000001000000100010011
00000000101000001001000100010011
00000000101000001101000100010011
01000000101000001101000100010011
00000010001000001000001000100011
00000010001000001001001000100011
00000010001000001010001000100011
00000000000011001000000010110111
00000000000011001000000010010111
00011000001000001000100001100011
00011000001000001001100001100011
```

分析和结论

从汇编器实现了对输入汇编指令的读入、汇编、输出操作。

实验心得体会

这个实验的设计思路是维护一个映射表，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。

实验二：存储器

实验目标

设计一个简单存储器，实现存储器的功能

实验要求

采用 VHDL 或 Verilog 语言

自定义存储器的输入和输出

实现存储器的存储功能

实验内容

1.存储器简介

存储器是用来存储程序和各种数据信息的记忆部件。存储器可分为主存储器(简称主存或内存)和辅助存储器(简称辅存或外存)两大类。和 CPU 直接交换信息的是主存。

主存的工作方式是按存储单元的地址存放或读取各类信息，统称访问存储器。主存中汇集存储单元的载体称为存储体，存储体中每个单元能够存放一串二进

码表示的信息,该信息的总位数称为一个存储单元的字长。存储单元的地址与存 储在其中的信息是一一对应的, 单元地址只有一个, 固定不变, 而存储在其中的信息是可以更换的。

指示每个单元的二进制编码称为地址码。寻找某个单元时，先要给出它的地址码。暂存这个地址码的寄存器叫存储器地址寄存器(MAR)。为可存放从主存的 存储单元内取出的信息或准备存入某存储单元的信息,还要设置一个存储器数据寄存器(MDR)。

2.存储器的特点

① 设置多个存储器并且使他们并行工作。本质:增添瓶颈部件数目,使它们并行工作，从而减缓固定瓶颈。

② 采用多级存储系统，特别是 Cache 技术，这是一种减轻存储器带宽对系统性能影响的最佳结构方案。本质:把瓶颈部件分为多个流水线部件,加大操作时间的重叠、提高速度，从而减缓固定瓶颈。

③ 在微处理机内部设置各种缓冲存储器,以减轻对存储器存取的压力。增加 CPU 中寄存器的数量也可大大缓解对存储器的压力。本质：缓冲技术，用于减 缓暂时性瓶颈。

存储器程序框架

考虑到存储器的主要原理就是读取、存储数据。

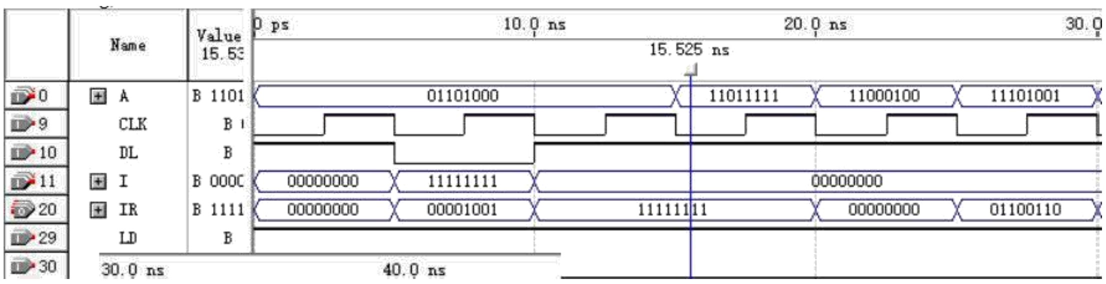
我们将模拟器的框架设计如下:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mem is
                port(
                        addrbus: in std_logic_vector(31 downto 0);
                        databus: inout std_logic_vector(31 downto 0);
                        read: in std_logic;
                        write: in std_logic
                        );
end entity;
architecture mem_behav of mem is
                type memtype is array(natural range<>) of std_logic_vector(7 downto 0);
                signal memdata: memtype(4095 downto 0) := (
                        0 => X"04",
                        1 => X"00",
                        2 => X"00",
                        3 => X"00",
                        4 => X"08",
                  5 => X"00",
                        6 => X"00",
                        7 => X"00",
                        others => X"11"
                );
begin
do_read: process(addrbus, read)
                                variable i: integer;
                begin
                        i := to_integer(unsigned(addrbus));
                        if (read='1') then
                                -- assume little-endian
        databus <= memdata(i+3) & memdata(i+2) & memdata(i+1) & memdata(i);
                        else
                                databus <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                        end if;
                end process do_read;
end;
```

```
--定义主体
entity mem is
port(
    --定义接口
    --定义使能端
);
end entity;

--定义结构
architecture mem_behav of mem is
    type memtype is array(natural range<>) of std_logic_vector(7 downto
    0);
    --初始化
    signal memdata: memtype(4095 downto 0) := (
        0 => X"04",
        1 => X"00",
        2 => X"00",
        3 => X"00",
        4 => X"08",
        5 => X"00",
        6 => X"00",
        7 => X"00",
        others => X"11"
    );
--主函数
begin
    do_read: process(addrbus, read)
    variable i: integer;
    begin
```

```
        i := to_integer(unsigned(addrbus));
        if (read='1') then

            databus <= memdata(i+3) & memdata(i+2) & memdata(i+1) &
        memdata(i); else
            databus <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        end if;
    end process do_read;

end;
```

仿真结果如下：



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。

实验心得与体会

通过本次实验复习了 RAM 的原理以及它的读写机制和过程。

实验三：模拟器

实验目标

设计一个 CPU 模拟器，能模拟 CPU 指令集的功能

实验要求

采用 C/C+＋编写程序

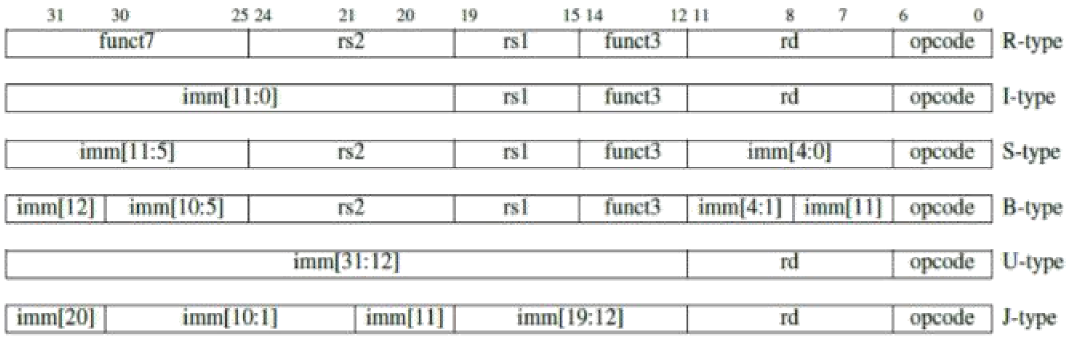
模拟器的输入是二进制的机器指令文件

模拟器的输出是 CPU 各个寄存器的状态和相关的存储器单元状态

.RISC-V 指令集内容

我们在这里编写的是 RV32I 指令集，其包含了六种基本指令格式，分别是于寄存器操作的 R类型指令，用于短立即数和访存 load 操作的 I 型指令，用于访存 store 操作的 S 型指令，用于条件跳转操作的 B 类型指令，用于长立即数的 U 型指令和用于无条件跳转的 J 型指令。

RISC- V 指令集编码格式



4.RISC- V 指令

Category Name	**Fmt**	**RV32I Base**	
Shifts			
Shift Left Logical	R	SLL	rd,rs1,rs2
Shift Left Log.Imm.	I	SLLI	rd,rs1,shamt
Shift Right Logical	R	SRL	rd,rs1,rs2
Shift Right Log.Imm.	I	SRLI	rd,rs1,shamt
Shift Right Arithmetic	R	SRA	rd,rs1,rs2
Shift Right Arith.Imm.	I	SRAI	rd,rs1,shamt
Arithmetic			
ADD	R	ADD	rd,rs1,rs2
ADD Immediate	I	ADDI	rd,rs1,imm
SUBtract	R	SUB	rd,rs1,rs2
Load Upper Imm	U	LUI	rd,imm
Add Upper Imm to PC	U	AUIPC	rd,imm
Logical			
XOR	R	XOR	rd,rs1,rs2
XOR Immediate	I	XORI	rd,rs1,imm
OR	R	OR	rd,rs1,rs2
OR Immediate	I	ORI	rd,rs1,imm

4.RISC- V 指令

Category Name	Fmt	RV32I Base	
Shifts			
Shift Left Logical	R	SLL	rd,rs1,rs2
Shift Left Log.Imm.	I	SLLI	rd,rs1,shamt
Shift Right Logical	R	SRL	rd,rs1,rs2
Shift Right Log.Imm.	I	SRLI	rd,rs1,shamt
Shift Right Arithmetic	R	SRA	rd,rs1,rs2
Shift Right Arith.Imm.	I	SRAI	rd,rs1,shamt
Arithmetic			
ADD	R	ADD	rd,rs1,rs2
ADD Immediate	I	ADDI	rd,rs1,imm
Add Upper Imm to PC	U	AUIPC	rd,imm
Logical			
XOR	R	XOR	rd,rs1,rs2
XOR Immediate	I	XORI	rd,rs1,imm
OR	R	OR	rd,rs1,rs2
OR Immediate	I	ORI	rd,rs1,imm
AND	R	AND	rd,rs1,rs2
AND Immediate	I	ANDI	rd,rs1,imm

Category Name	**Fmt**	**RV32I Base**	
Compare			
Set<	R	SLT	rd,rs1,rs2
Set<Immediate	I	SLTI	rd,rs1,rs2
Set<Unsigned	R	SLTU	rd,rs1,rs2
Set<Imm Unsigned	I	SLTIU	rd,rs1,imm
Branches			
Branch=	B	BEQ	rs1,rs2,imm
Branch≠	B	BNE	rs1,rs2,imm
Branch<	B	BLT	rs1,rs2,imm
Branch≥	B	BGE	rs1,rs2,imm
Branch<Unsigned	B	BLTU	rs1,rs2,imm
Branch≥ Unsigned	B	BGEU	rs1,rs2,imm
Jump& Link			
J&L	J	JAL	rd,imm
Jump&Link Register	I	JALR	rd,rs1,imm
Synch			
Synch thread	I	FENCE	
Synch Instr&Data	I	FENCE.I	
Environment			
CALL	I	ECALL	
BREAK	I	EBREAK	
Control Status Register(CSR)			
Read/Write	I	CSRRW	rd,csr,rs1
Read&Set Bit	I	CSRRS	rd,csr,rs1
Read&Clear Bit	I	CSRRC	rd,csr,rs1
Read/Write Imm	I	CSRRWI	rd,csr,imm
Read&Set Bit Imm	I	CSRRSI	rd,csr,imm
Read&Clear Bit Imm	I	CSRRCI	rd,csr,imm
Loads			
Load Byte	I	LB	rd,rs1,imm
Load Halfword	I	LH	rd,rs1,imm
Load Byte Unsigned	I	LBU	rd,rs1,imm
Load Half Unsigned	I	LHU	rd,rs1,imm
Load Word	I	LW	rd,rs1,imm
Stores			
Store Byte	S	SB	rs1,rs2,imm
Store Halfword	S	SH	rs1,rs2,imm
Store Word	S	SW	rs1,rs2,imm

1. 模拟器架构：

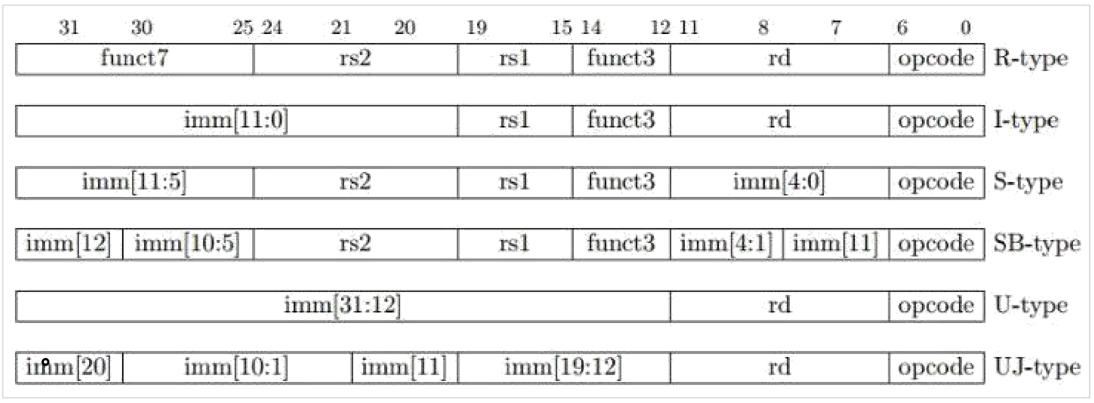
Program（）：负责通过移位运算将指令写入寄存器。

Decode（）：负责译码。

main（）:根据译码结果选择相应的模块执行。

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。



31	30　25	24　21	20	19　15	14　12	11　8	7	6　0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	SB-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	UJ-type

主要代码：

内存分配和管理

```
def allocMem(s,M):
    M=[0 for i in range(s)]
    MSize=s
    return M,MSizedef freeMem(M):
    del M[:]
```

数据的存储和读取

```
def readByte(M,address):
    if address >= MSize:
         print("ERROR: Address out of range in readByte")
         return 0
    return int('0x'+M[address],16)   #int 型 def
writeByte(M,address,data):        #data 为 int 型
    data="{0:02x}".format(data)       #data 为 str 型数值为 16 进制
    if address >= MSize:
         print("ERROR: Address out of range in writeByte")
         return 0
    M[address]=data                         #M[]为 str 型 def
readHalfWord(M,address):
    if address >= MSize-WORDSIZE/2:
         print("ERROR: Address out of range in readHalfWord")
         return 0
    return int('0x'+M[address+1]+M[address],16)def
writeHalfWord(M,address,data):
    data="{0:04x}".format(data)
    if address >= MSize-WORDSIZE/2:
         print("ERROR: Address out of range in writeHalfWord")
         return 0
    M[address+1]=data[0:2]
    M[address]=data[2:4]def readWord(M,address):
    if address >= MSize:
         print("ERROR: Address out of range in readWord")
         return 0
    return
int('0x'+M[address+3]+M[address+2]+M[address+1]+M[address],16)
def writeWord(M,address,data):
    data="{0:08x}".format(data)
    if address >= MSize:
         print("ERROR: Address out of range in writeWord")
         return 0
    M[address+3]=data[0:2]
    M[address+2]=data[2:4]
```

```
    M[address+1]=data[4:6]
    M[address]=data[6:8]
```

存储器的设计以及指令的初始存储

```
def Mem(M):
    writeWord(M,0,(0xfffff<<12)|(2<<7)|(opcode['LUI']))
    writeWord(M,4,(1<<12)|(5<<7)|(opcode['AUIPC']))
    writeWord(M,8,(0x20<<25)|(5<<20)|(opcode['SW']))
    writeWord(M,12,(0x400<<20)|(3<<7)|(opcode['LB']))
    writeWord(M,16,(0x400<<20)|(7<<7)|(opcode['LBU']))
    writeWord(M,20,(2<20)|(0x8<<7)|(opcode['BGE']))
    writeWord(M,28,(0x8<<20)|(3<<15)|(8<<7)|(opcode['SLTIU']))
    writeWord(M,32,(0x2<<20)|(0x2<<15)|(9<<7)|(opcode['SRAI']))

    writeWord(M,36,(0x400)<<20|(1<<15)|(4<<7)|(opcode['JALR']))
    writeWord(M,40,(0x20<<25)|(7<<20)|(9<<7)|(opcode['SH']))
    writeWord(M,44,(4<<20)|(1<<15)|(0x8<<7)|(opcode['BGEU']))
    writeWord(M,48,(0x400<<20)|(2<<15)|(4<<7)|(opcode['ORI']))
    writeWord(M,52,(4<<20)|(2<<15)|(9<<7)|(opcode['SUB']))

    writeWord(M,56,(1<<31)|(8<<20)|(opcode['BLTU']))
    writeWord(M,60,(0x20<<25)|(8<<20)|(opcode['SB']))
    writeWord(M,64,(0x100<<20)|(3<<15)|(9<<7)|(opcode['XORI']))
    writeWord(M,68,(3<<20)|(1<<15)|(10<<7)|(opcode['ADD']))

writeWord(M,72,(1<<31)|(1<<23)|(1<<22)|(1<<12)|(7<<7)|(opcode['J
AL']))

    writeWord(M,0,0x0013ab73)
    writeWord(M,4,0x0013db73)
    writeWord(M,8,0x0013fb73)
    writeWord(M,12,0x0000100f)
    writeWord(M,16,0x00100073)
```

指令的译码

```
def decode(instruction,R):                #instruction 为 int 型
```

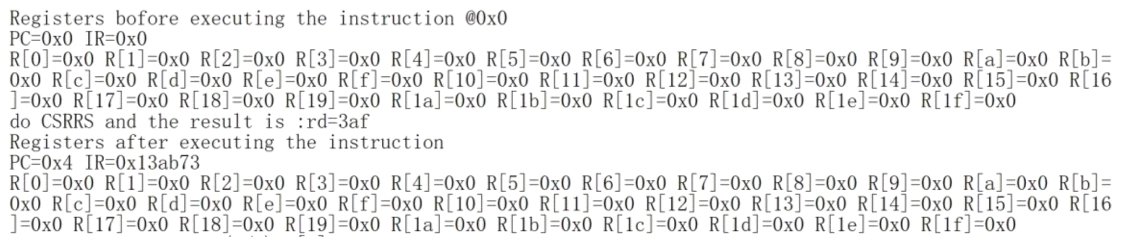
```
    opcd=instruction&0xfe00707f
    rd=(instruction&0x0f80)>>7
    rs1=(instruction&0xf8000)>>15
    zimm=rs1
    rs2=(instruction&0x1f00000)>>20
    shamt=rs2
    imm11_0i=instruction>>20
    csr=instruction>>20
    imm11_5s=instruction>>25
    imm4_0s=(instruction>>7)&0x01f
    imm12b=instruction>>31
    imm10_5b=(instruction>>25)&0x3f
    imm4_1b=(instruction&0x0f00)>>8
    imm11b=(instruction&0x080)>>7
    imm31_12u=instruction>>12
    imm20j=instruction>>31
    imm10_1j=(instruction>>21)&0x3ff
    imm11j=(instruction>>20)&1
    imm19_12j=(instruction>>12)&0x0ff
    pred=(instruction>>24)&0x0f
    succ=(instruction>>20)&0x0f

    src1=R[rs1]
    src2=R[rs2]
  Imm11_0ItypeZeroExtended=imm11_0i& 0x0fff
    Imm11_0ItypeSignExtended=imm11_0i
    Imm11_0StypeSignExtended=(imm11_5s<<5)|imm4_0s

Imm12_1BtypeZeroExtended=imm12b&0x00001000|(imm11b<<11)|(i
mm10_5b<<5)|(imm4_1b<<1)

Imm12_1BtypeSignExtended=imm12b&0xfffff000|(imm11b<<11)|(im
m10_5b<<5)|(imm4_1b<<1)
```

```
    Imm31_12UtypeZeroFilled=instruction&0xfffff000

Imm20_1JtypeSignExtended=(imm20j&0xfff00000)|(imm19_12j<<12)|(imm11j<<11)|(imm10_1j<<1)

Imm20_1JtypeZeroExtended=(imm20j&0x00100000)|(imm19_12j<<12)|(imm11j<<11)|(imm10_1j<<1)
    return
opcd,rd,rs1,zimm,rs2,shamt,imm11_0i,csr,imm11_5s,imm4_0s,imm12b,imm10_5b,imm4_1b,\

imm11b,imm31_12u,imm20j,imm10_1j,imm11j,imm19_12j,pred,succ,src1,src2,Imm11_0ItypeZeroExtended,\

Imm11_0ItypeSignExtended,Imm11_0StypeSignExtended,Imm12_1BtypeZeroExtended,Imm12_1BtypeSignExtended,\

Imm31_12UtypeZeroFilled,Imm20_1JtypeSignExtended,Imm20_1JtypeZeroExtended
```

测试与运行截图

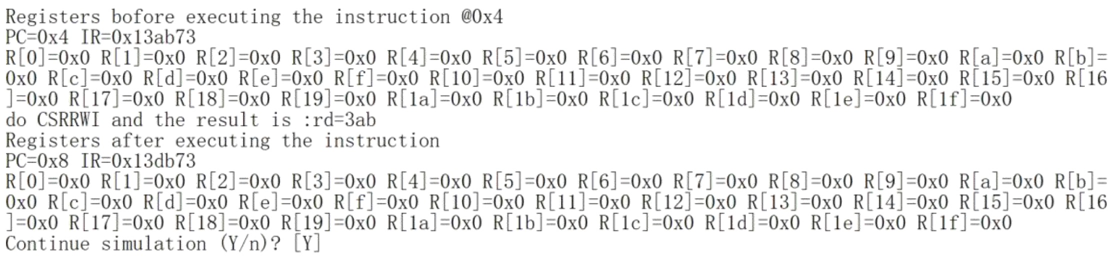
模拟器运行的截图如下

第一条指令运行输出：

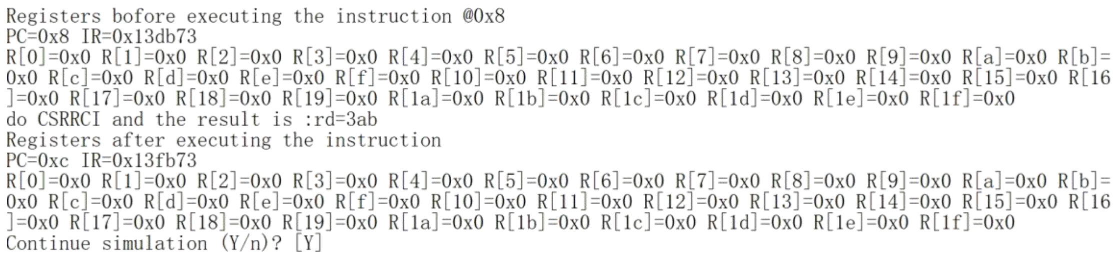


```
Registers bofore executing the instruction @0x0
PC=0x0 IR=0x0
R[0]=0x0 R[1]=0x0 R[2]=0x0 R[3]=0x0 R[4]=0x0 R[5]=0x0 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=
0x0 R[c]=0x0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16
]=0x0 R[17]=0x0 R[18]=0x0 R[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0
do CSRRS and the result is :rd=3af
Registers after executing the instruction
PC=0x4 IR=0x13ab73
R[0]=0x0 R[1]=0x0 R[2]=0x0 R[3]=0x0 R[4]=0x0 R[5]=0x0 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=
0x0 R[c]=0x0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16
]=0x0 R[17]=0x0 R[18]=0x0 R[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0
```

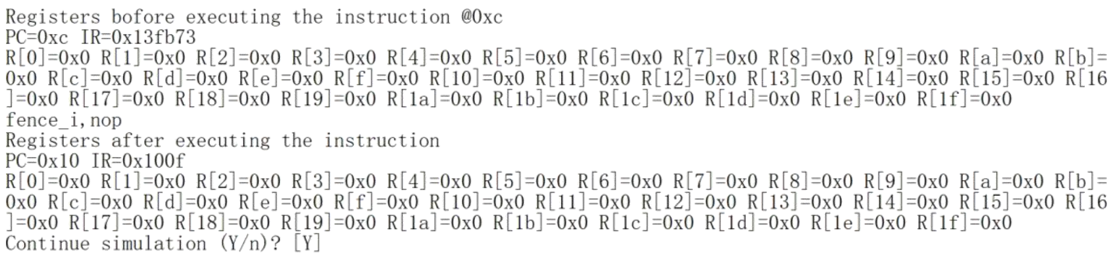
第二条指令运行输出：


```
Registers bofore executing the instruction @0x4
PC=0x4 IR=0x13ab73
R[0]=0x0 R[1]=0x0 R[2]=0x0 R[3]=0x0 R[4]=0x0 R[5]=0x0 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=
0x0 R[c]=0x0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16
]=0x0 R[17]=0x0 R[18]=0x0 R[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0
do CSRRWI and the result is :rd=3ab
Registers after executing the instruction
PC=0x8 IR=0x13db73
R[0]=0x0 R[1]=0x0 R[2]=0x0 R[3]=0x0 R[4]=0x0 R[5]=0x0 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=
0x0 R[c]=0x0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16
]=0x0 R[17]=0x0 R[18]=0x0 R[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0
Continue simulation (Y/n)? [Y]
```

第三条指令运行输出：


```
Registers bofore executing the instruction @0x8
PC=0x8 IR=0x13db73
R[0]=0x0 R[1]=0x0 R[2]=0x0 R[3]=0x0 R[4]=0x0 R[5]=0x0 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=
0x0 R[c]=0x0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16
]=0x0 R[17]=0x0 R[18]=0x0 R[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0
do CSRRCI and the result is :rd=3ab
Registers after executing the instruction
PC=0xc IR=0x13fb73
R[0]=0x0 R[1]=0x0 R[2]=0x0 R[3]=0x0 R[4]=0x0 R[5]=0x0 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=
0x0 R[c]=0x0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16
]=0x0 R[17]=0x0 R[18]=0x0 R[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0
Continue simulation (Y/n)? [Y]
```

第四条指令运行输出：


```
Registers bofore executing the instruction @0xc
PC=0xc IR=0x13fb73
R[0]=0x0 R[1]=0x0 R[2]=0x0 R[3]=0x0 R[4]=0x0 R[5]=0x0 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=
0x0 R[c]=0x0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16
]=0x0 R[17]=0x0 R[18]=0x0 R[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0
fence_i,nop
Registers after executing the instruction
PC=0x10 IR=0x100f
R[0]=0x0 R[1]=0x0 R[2]=0x0 R[3]=0x0 R[4]=0x0 R[5]=0x0 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=
0x0 R[c]=0x0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16
]=0x0 R[17]=0x0 R[18]=0x0 R[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0
Continue simulation (Y/n)? [Y]
```

分析与总结

模拟器实现了对二进制指令文件的读入、指令功能的模拟，CPU 和存储器状态的输出。

实验四：cpu

实验要求

硬件设计采用 VHDL 或 Verilog 语言，软件设计采用 C/C++或 SystemC 语言，其它语言例如 Chisel、MyHDL 等也可选

实验报告采用 markdown 语言，或者直接上传 PDF 文档

实验最终提交所有代码和文档

实验内容

RISC-V 指令集介绍

RV32I 指令集包含了六种基本指令格式，分别是：

R 类型指令：用于寄存器到寄存器操作

I 类型指令：用于短立即数和访存 load 操作

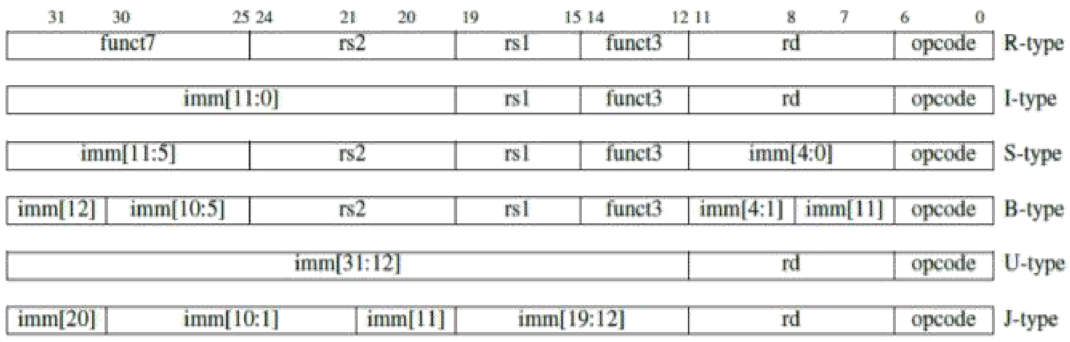
S 类型指令：用于访存 store 操作

B 类型指令：用于条件跳转操作

U 类型指令：用于长立即数

J 类型指令：用于无条件跳转

RISC-V 指令集编码格式



31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

4.RISC- V 指令

Category **Name**	**Fmt**	**RV32I Base**	
Shifts			
Shift Left Logical	R	SLL	rd,rs1,rs2
Shift Left Log.Imm.	I	SLLI	rd,rs1,shamt
Shift Right Logical	R	SRL	rd,rs1,rs2
Shift Right Log.Imm.	I	SRLI	rd,rs1,shamt
Shift Right Arithmetic	R	SRA	rd,rs1,rs2
Shift Right Arith.Imm.	I	SRAI	rd,rs1,shamt
Arithmetic			
ADD	R	ADD	rd,rs1,rs2
ADD Immediate	I	ADDI	rd,rs1,imm
SUBtract	R	SUB	rd,rs1,rs2
Load Upper Imm	U	LUI	rd,imm
Add Upper Imm to PC	U	AUIPC	rd,imm
Logical			
XOR	R	XOR	rd,rs1,rs2
XOR Immediate	I	XORI	rd,rs1,imm
OR	R	OR	rd,rs1,rs2
OR Immediate	I	ORI	rd,rs1,imm

Category Name	Fmt	RV32I Base	
Shifts			
Shift Left Logical	R	SLL	rd,rs1,rs2
Shift Left Log.Imm.	I	SLLI	rd,rs1,shamt
Shift Right Logical	R	SRL	rd,rs1,rs2
Shift Right Log.Imm.	I	SRLI	rd,rs1,shamt
Shift Right Arithmetic	R	SRA	rd,rs1,rs2
Shift Right Arith.Imm.	I	SRAI	rd,rs1,shamt
Arithmetic			
ADD	R	ADD	rd,rs1,rs2
ADD Immediate	I	ADDI	rd,rs1,imm
Add Upper Imm to PC	U	AUIPC	rd,imm
Logical			
XOR	R	XOR	rd,rs1,rs2
XOR Immediate	I	XORI	rd,rs1,imm
OR	R	OR	rd,rs1,rs2
OR Immediate	I	ORI	rd,rs1,imm
AND	R	AND	rd,rs1,rs2
AND Immediate	I	ANDI	rd,rs1,imm

Category **Name**	**Fmt**	**RV32I Base**	
Compare			
Set<	R	SLT	rd,rs1,rs2
Set< Immediate	I	SLTI	rd,rs1,rs2
Set< Unsigned	R	SLTU	rd,rs1,rs2
Set< Imm Unsigned	I	SLTIU	rd,rs1,imm
Branches			
Branch=	B	BEQ	rs1,rs2,imm
Branch≠	B	BNE	rs1,rs2,imm
Branch<	B	BLT	rs1,rs2,imm
Branch≥	B	BGE	rs1,rs2,imm
Branch< Unsigned	B	BLTU	rs1,rs2,imm
Branch≥ Unsigned	B	BGEU	rs1,rs2,imm
Jump& Link			
J& L	J	JAL	rd,imm
Jump& Link Register	I	JALR	rd,rs1,imm
Synch			
Synch thread	I	FENCE	
Synch Instr& Data	I	FENCE.I	
Environment			
CALL	I	ECALL	
BREAK	I	EBREAK	
Control Status Register(CSR)			
Read/ Write	I	CSRRW	rd,csr,rs1
Read& Set Bit	I	CSRRS	rd,csr,rs1
Read& Clear Bit	I	CSRRC	rd,csr,rs1
Read/ Write Imm	I	CSRRWI	rd,csr,imm
Read& Set Bit Imm	I	CSRRSI	rd,csr,imm
Read& Clear Bit Imm	I	CSRRCI	rd,csr,imm
Loads			
Load Byte	I	LB	rd,rs1,imm
Load Halfword	I	LH	rd,rs1,imm
Load Byte Unsigned	I	LBU	rd,rs1,imm
Load Half Unsigned	I	LHU	rd,rs1,imm
Load Word	I	LW	rd,rs1,imm
Stores			
Store Byte	S	SB	rs1,rs2,imm
Store Halfword	S	SH	rs1,rs2,imm
Store Word	S	SW	rs1,rs2,imm

Cpu代码：

```
libra
ry
ieee
;
    use ieee.std_logic_1164.all;
    use ieee.numeric_std.all;

    --use ieee.std_logic_arith.all;
    --use IEEE.std_logic_unsigned.ALL;
    --use ieee.std_logic_arith.all;

    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);
    inst_read: out std_logic;
    data_addr: buffer std_logic_vector(31 downto 0);
    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;

    architecture cpu_simple_behav of my_cpu is
```

```vhdl
-- utype instructions, using opcode
constant utype_lui: std_logic_vector(6 downto 0) := B"0110111";
constant utype_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";

-- btype branches, using opcode, funct3
constant btype_branch: std_logic_vector(6 downto 0) := B"1100011";
constant btype_beq: std_logic_vector(2 downto 0) := B"000";
constant btype_bne: std_logic_vector(2 downto 0) := B"001";
constant btype_blt: std_logic_vector(2 downto 0) := B"100";
constant btype_bge: std_logic_vector(2 downto 0) := B"101";
constant btype_bltu: std_logic_vector(2 downto 0) := B"110";
constant btype_bgeu: std_logic_vector(2 downto 0) := B"111";
-- Itype branches, using opcode, funct3
constant Itype_branch: std_logic_vector(6 downto 0) := B"0000011";
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";

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);

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);
```

```vhdl
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 utype_full_imm31_0:std_logic_vector(31 downto 0);

signal itype_imm11_0: std_logic_vector(11 downto 0);
signal itype_all_imm: std_logic_vector(32 downto 0);

signal btype_imm12_1: std_logic_vector(12 downto 1);

signal ltype_imm11_0: std_logic_vector(11 downto 0);

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);

signal load_addr: std_logic_vector(31 downto 0);
signal load_data: std_logic_vector(31 downto 0);
signal store_addr: std_logic_vector(31 downto 0);

signal branch_target: std_logic_vector(31 downto 0);
signal branch_taken: std_logic;


function bool2logic32(b: boolean) return std_logic_vector is
begin
if b then
```

```
return X"00000001";
else
return X"00000000";
end if;
end;
function signext8to32(b: std_logic_vector(7 downto 0)) return
std_logic_vector is
variable t: std_logic_vector(31 downto 0);
begin
t(7 downto 0) := b;
t(31 downto 8) := (others=>b(7));
return t;
end;
function signext16to32(h: std_logic_vector(15 downto 0)) return
std_logic_vector is
variable t: std_logic_vector(31 downto 0);
begin
t(15 downto 0) := h;
t(31 downto 16) := (others=>h(15));
return t;
end;
begin
-- 组合逻辑部分
-- instruction fetch
inst_addr <= pc; -- 取指地址
inst_read <= '1' when reset = '0' else '0'; -- 当 reset 无效时发出指令读取
信号;
ir <= inst; -- 当前指令
-- 数据访问


-- store_addr <= ...
data_addr <= load_addr when opcode=itype_load else
store_addr;
```

```
data_read <= '1' when opcode=itype_load else '0'; -- 当 reset 无效时发
出指令读取信号;
-- data_write <= ...
load_data <= data when funct3=itype_lw else
signext8to32(data(7 downto 0)) when funct3=itype_lb else
signext16to32(data(15 downto 0)) when funct3=itype_lh else
X"000000" & data(7 downto 0) when funct3=itype_lbu else
X"0000" & data(15 downto 0) when funct3=itype_lhu else
X"00000000";
-- data <= ...
-- *******************BY QLM decode directly from the instruction
opcode <= ir(6 downto 0);
rd <= ir(11 downto 7);
rs1 <= ir(19 downto 15);
rs2 <= ir(24 downto 20);
funct3 <= ir(14 downto 12);
funct7 <= ir(31 downto 25);
-- *******************BY QLM decode directly from the instruction
--BY QLM:the value in register 1 and 2
rs1_data <= regs( to_integer( unsigned(rs1)) );
rs2_data <= regs( to_integer( unsigned(rs2)) );

jal_imm20_1 <= ir(31) & ir(19 downto 12) & ir(20) & ir(30 downto 21);
jal_offset(20 downto 0) <= jal_imm20_1 & '0';
jal_offset(31 downto 21) <= (others=>jal_imm20_1(20)); --signed
extend

--****************BY QLM: the exact immediate values of specific
types DIRECTLY decent from the instructions**************
utype_imm31_12 <= ir(31 downto 12);

itype_imm11_0 <= ir(31 downto 20);
itype_all_imm(31 downto 12) <= (others=>itype_imm11_0(11));
load_addr <=
std_logic_vector(to_signed((to_integer(signed(rs1_data)) +
to_integer(signed(itype_imm11_0))),32)); --jalr

btype_imm12_1 <= ir(31) & ir(7) & ir(30 downto 25) & ir(11 downto
8);

Itype_imm11_0<=ir(31 downto 20);


--****************BY QLM: the exact immediate values of specific
types DIRECTLY decent from the instructions**************
-- ......
-- R-type ALU operations
--rs
rtype_alu_result <=
std_logic_vector( to_unsigned( ( to_integer( unsigned(rs1_data)) +
to_integer( unsigned(rs2_data)) ) , 32 ) )
when funct3 = rtype_addsub and funct7 = rtype_add else
std_logic_vector( to_unsigned( ( to_integer( unsigned(rs1_data)) -
to_integer( unsigned(rs2_data)) ) , 32 ) )
when funct3 = rtype_addsub and funct7 = rtype_sub else
std_logic_vector( shift_left(unsigned(rs1_data) ,
to_integer(unsigned(rs2_data) ) ) )
when funct3 = rtype_sll else
X"00000001"
when(signed(rs1_data) < signed(rs2_data)) and funct3 = rtype_slt else
X"00000001"
when(unsigned(rs1_data) < unsigned(rs2_data)) and funct3 =
rtype_sltu else
rs1_data xor rs2_data
when funct3 = rtype_xor else
std_logic_vector(shift_right(unsigned(rs1_data),
to_integer(unsigned(rs2_data))))
when funct3 = rtype_srlsra and funct7 = rtype_srl else
```

```
std_logic_vector(shift_right(signed(rs1_data),
to_integer(signed(rs2_data))))
when funct3 = rtype_srlsra and funct7 = rtype_sra else
rs1_data or rs2_data
when funct3 = rtype_or else
rs1_data and rs2_data
when funct3 = rtype_and else
X"00000000"; -- default ALU result
--BY QLM: using opcode to decide which value shall be put into
rd_data
utype_full_imm31_0 <= utype_imm31_12 & X"000"
when opcode = utype_lui or opcode = utype_auipc;

rd_data <= rtype_alu_result
when opcode = rtype_alu else

std_logic_vector( to_unsigned( (to_integer(unsigned(pc))+4) , 32 ) )
when opcode = jtype_jal or opcode = itype_jalr else

utype_full_imm31_0
when opcode = utype_lui else

std_logic_vector( to_unsigned( ( to_integer( unsigned(utype_full_imm3
1_0)) + to_integer( unsigned(pc)) ) , 32 ) )
when opcode = utype_auipc else

mems(to_integer(unsigned(data_addr)))
when opcode=itype_load else
X"00000000"; -- default rd data

rd_write <= '1' when opcode = rtype_alu --Qlm: write opration signal
or opcode=utype_lui
or opcode=utype_auipc
or opcode=jtype_jal
or opcode=itype_load;

write_avi <= rd_write; --Qlm: debug,watch if rd_write is avilable

-- 分支指令
branch_target(12 downto 0) <= btype_imm12_1 & '0' ;
branch_target(31 downto 14) <= ( others => btype_imm12_1(12) );
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';
--load

-- 下一条指令地址
next_pc <=
std_logic_vector( to_unsigned( (to_integer(unsigned(pc))+to_integer(
unsigned(jal_offset))) , 32 ))
when opcode = jtype_jal else --JAL inst add imm and pc

load_addr
when opcode = itype_jalr else --JALR inst

std_logic_vector( to_unsigned( (to_integer(unsigned(pc))+to_integer(
unsigned(branch_target))) , 32 ))
when opcode = btype_branch and branch_taken ='1' else

std_logic_vector(to_unsigned(to_integer(unsigned(pc)) + 4,32)); -- 需
补充其它情况


is_jal <= '1' when opcode = jtype_jal else --Qlm: debug watch if jal
```

```
instrution avilable
'0';
-- ...... (其它组合逻辑)

-- 时序逻辑部分
-- pc
pc_update: process(clk)
begin
if(rising_edge(clk)) then
if(reset='1') then
pc <= X"00000000"; -- 当 reset 信号有效时，pc 被重置为 0
else
pc <= next_pc;
end if;
end if;
end process pc_update;

-- regs
reg_update: process(clk)
variable i: integer;
variable k: integer;
begin
i := to_integer(unsigned(rd));

if(rising_edge(clk)) then
if(reset='1') then
-- reset all regs to 0 except reg[0]
for k in 1 to 31 loop
regs(k) <= X"00000000"; -- reset to 0
end loop;
-- regs(0) <= X"00000001";
-- regs(1) <= X"00000003";
-- regs(2) <= X"00000100";
-- mems(0) <= X"00000006";
-- mems(1) <= X"00000007";
-- mems(2) <= X"00000008";
--BY QLM:when the write signal is available, put the result for
example,the result of add inst into the register(i)
elsif(rd_write='1' and i /= 0) then
regs(i) <= rd_data;
reg_val <= regs(i);

if(funct3 = rtype_addsub and funct7 = rtype_add) then
is_alu <= '1';
end if;

opcode_val <= opcode;

end if;

end if;
end process reg_update;

end;
```

测试与运行：

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。

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)) +

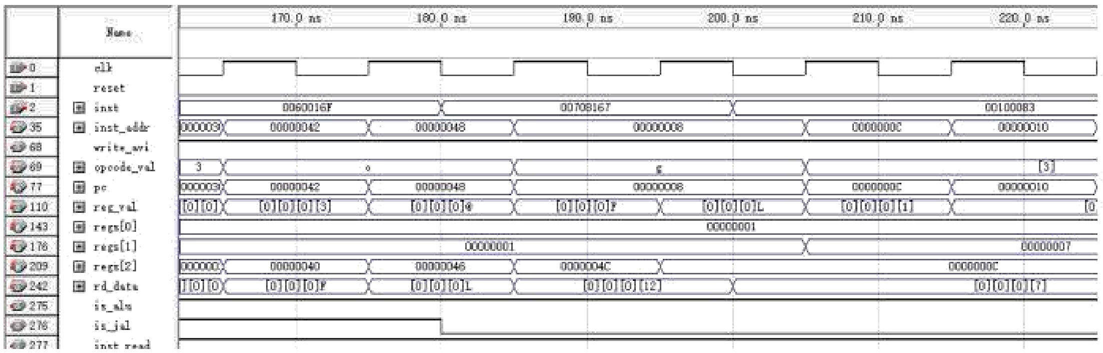
to_integer(signed(itype_imm11_0)))

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：



跳转指令是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。

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(31downto14)<=(others=>btype_imm12_1(12)); 跳转条件:

branch_taken<='1'when(rs1=rs2andfunct3=btype_beq) or (rs1 /= rs2 and funct3 = btype_bne)

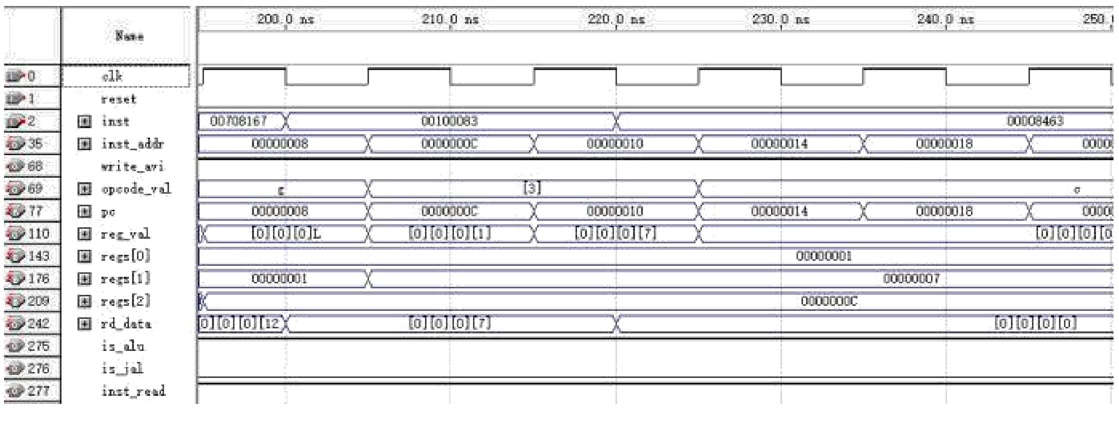
or (signed(rs1) < signed(rs2) and funct3 = btype_blt)

or(unsigned(rs1)<unsigned(rs2)andfunct3=btype_bltu) or (signed(rs1) > signed(rs2) and funct3 = btype_bge)

or(unsigned(rs1)>unsigned(rs2)andfunct3=btype_bgeu) else '0';

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

仿真结果:


Name
200.0 ns
210.0 ns
220.0 ns
230.0 ns
240.0 ns
250.
0 clk
1 reset
2 inst 00708167 00100083 00008463
35 inst_addr 00000008 0000000C 00000010 00000014 00000018
68 write_avi
69 opcode_val [3]
77 pc 00000008 0000000C 00000010 00000014 00000018
110 reg_val [0][0][0]L [0][0][0][1] [0][0][0][7]
143 regs[0] 00000001
176 regs[1] 00000001 00000007
209 regs[2] 0000000C
242 rd_data [0][0][0][12 [0][0][0][7] [0][0][0][0]
275 is_alu
276 is_jal
277 inst_read

实验与总结：

　这次实验，更加了解RISC-V指令的使用和框架的理解，代码是网上开源的，但进行了一定的修改，主要是对代码的理解。