微处理器实验报告

201608010720　　黄梓　　物联 1601

一、实验任务

必做：

1. 完成一个模拟 RISC-V 的基本整数指令集 RV32I 的汇编器设计
2. 完成一个模拟 RISC-V 的基本整数指令集 RV32I 的模拟器设计
3. 完成一个简单存储器的设计
4. 完成一个执行 RISC-V 的基本整数指令集 RV32I 的 CPU 设计（单周期实现）

选做：

1. 完成一个执行 RISC-V 的基本整数指令集 RV32I 的 CPU 设计（流水线实现）
2. 其它经教师确认的任务，例如调试器、编译器，或者 RV32I 之外的扩展指令集 CPU 设计

二、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

三、实验相关知识点

RISC-V 指令集简介

RISC-V 指令集是 UC Berkley 大学设计的第五代开源 RISC ISA, V 也可以认为是允许变种(Variations)和向量(Vector)向量实现，数据的并行加速功能也是明确支持目标，是专用硬件发展的一个重要方向。RISC ISA 相对于成熟的指令集来说有开源、简捷、可扩展、和后发优势（没有历史包袱，可以绕过很多弯路，也不需要考虑兼容历史指令集）等。

指令集分为基本部分和扩展部分，基本部分的指令集所有硬件实现都必须有这一部分实现，而扩展部分则是可选的。扩展部分又分为标准扩展和非标准扩展。例如，乘除法、单双精度的浮点、原子操作就在标准扩展子集中。

“I” 基本整数集，其中包含整数的基本计算、Load/Store 和控制流，所有的硬件实现都必须包含这一部分。

“M” 标准整数乘除法扩展集，增加了整数寄存器中的乘除法指令。

“A” 标准操作原子扩展集，增加对储存器的原子读、写、修改和处理器间的同步。

“F” 标准单精度浮点扩展集，增加了浮点寄存器、计算指令、L/S 指令。

“D” 标准双精度扩展集，扩展双精度浮点寄存器，双精度计算指令、L/S 指令。

I+M+F+A+D 被缩写为 “G” ，共同组成通用的标量指令。

在后续 ISA 的版本迭代过程中，RV32G 和 RV64G 总是保持不变。

基本 RISC-V ISA 具有 32 位固定长度，并且需要 32 位地址对齐。但是也支持变长扩展，要求指令长度为 16 位整数倍，16 位地址对齐。

32 位指令最低 2 位为“11”，而 16 位变长指令可以是“00、01、10”，48 位指令低 5 位位全 1，64 位指令低 6 位全 1。任何长度的指令，如果所有位全 0 或全 1，都认为是非法指令，前者跳入填满 0 的储存区域，后者通常意味着总线或储存器损坏。

另外，RISC-V 默认用小端储存系统，但非标准变种中可以支持大端或者双端储存系统。

另外专业术语：

异常：RISC-V 线程中出现了指令相关的非正常情况。

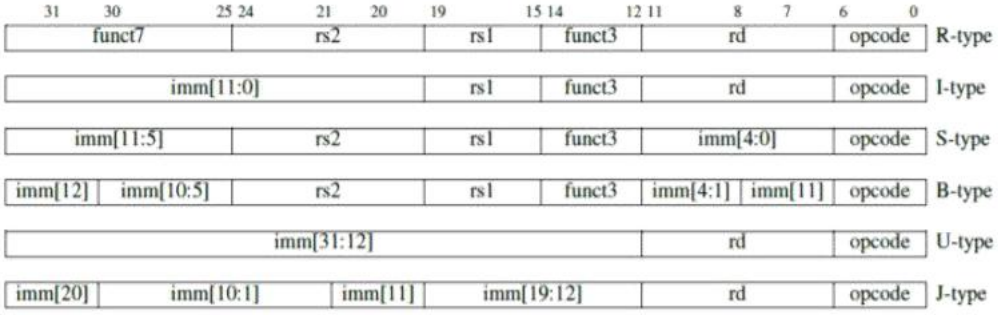
自陷：RISC-V 线程中出现了指令相关的异常情况，控制同步传输到自陷处理函数（一般在高特权环境中执行）。

中断：RISC-V 线程外异步出现了一个事件，如果需要处理则需要选择某条指令来接收，并顺序产生自陷。

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。

RISC-V 指令集编码格式



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

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	FENCEI	
Environment			

四、实验组成

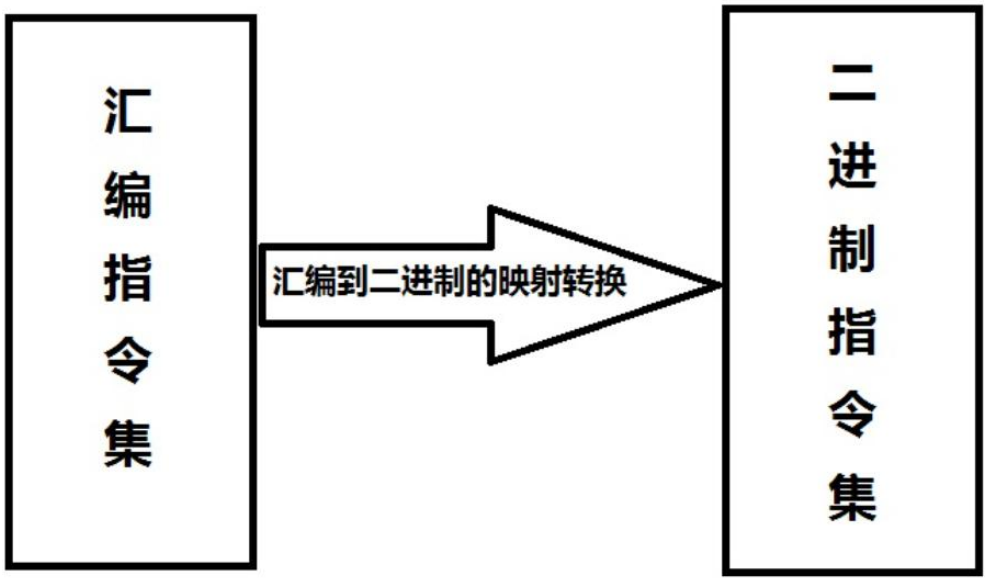
汇编器

1、简单介绍

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。常用的高级语言编译器有 Microsoft 公司的 MASM 系列和 Borland 公司的 TASM 系列编译器，还有一些小公司推出的或者免费的汇编软件包等。

2、相关知识



实现一个汇编器，首先需要定义一个汇编指令集，这里我们还是沿用上文中的汇编指令集。汇编指令与指令之间是一一对应的关系，也就是说是直译的过程。我们的指令集是枚举类型，也是沿用上文源码的指令集。

我们的函数功能是对输入的汇编指令，将其读入，翻译成对应的二进制代码，然后将其输出。

实现汇编器的重点在于理解汇编器的原理，而汇编器的原理就在于定义好汇编指令集、二进制指令集，并且确定好二者之间的映射转换关系。

3、关键代码

```
        string op,s;
        //将结果输入输出文件
        //freopen("out.txt","w",stdout);
        ofstream outFile("out.dat",ios::out | ios::binary);
        while(cin>>op>>s){
        //进行译码过程(下同)
        if(op == "ADD" | op == "SUB" | op == "XOR" | op == "OR" | op == "AND" |
        op == "SLL" | op == "SRL" | op == "SRA" | op == "SLT" | op == "SLTU"){
        getnum(s);
        //输出译码结果(下同)
        //solveR(opcode[op],d[0],d[1],d[2]);
        outFile.write((char*)&solveR(opcode[op],d[0],d[1],d[2]),sizeof(solveR(opcode[op],d[0],d[1],d[2])));
        }
        else if(op == "ADDI" | op == "SLTI" | op == "SLTIU" | op == "XORI" | op == "JALR" |
        op == "ORI" | op == "ANDI" | op == "SLLI" | op == "SRLI" | op == "SRAI" | op == "LB" | op == "LH" | op == "LW" | op == "LBU" | op =
        getnum(s);
        solveI(opcode[op], d[0],d[1],d[2]);
        }
        else if(op == "SB" | op == "SH" | op == "SW"){
        getnum(s);
        solveS(opcode[op],d[0],d[1],d[2]);
        }
        else if(op == "LUI" | op == "AUIPC"){
        getnum(s);
        solveU(opcode[op],d[0],d[1]);
        }
```

```
else if(op == "LUI" | op == "AUIPC"){
getnum(s);
solveU(opcode[op],d[0],d[1]);
}
else if(op == "BEQ" | op == "BNE" | op == "BLT" | op == "BGE" | op == "BLTU" | op == "BGEU"){
getnum(s);
solveSB(opcode[op],d[0],d[1],d[2]);
}
else if(op == "JAL"){
getnum(s);
solveUJA(opcode[op],d[0],d[1]);
}
}
fclose(stdin);
outFile.close();
//fclose(stdout);
return 0;
}
```

实验心得

汇编器实现的关键三点：汇编指令的表示、二进制指令的表示、汇编指令到二进制指令之间的转换三个方面。

在程序中我们对输入的汇编指令是按照空白符间隔的方式进行的汇编代码切分，如果进一步改进，可以对汇编代码进行词法分析，切分出汇编 token——指令码和操作数，然后将指令码和操作数翻译成对应的二进制代码。

模拟器

1、简单介绍

计算机模拟（简称 sim）是利用计算机进行模拟的方法。利用计算机软件开发出的模拟器，可以进行故障树分析、测试 VLSI 逻辑设计等复杂的模拟任务。在优化领域，物理过程的模拟经常与演化计算一同用于优化控制策略。计算机模拟器中有一种特殊类型：计算机架构模拟器，用以在一台计算机上模拟另一台指令不兼容或者体系不同的计算机。阿兰·图灵曾提出：（不同体系的）机器 A 或机器 B 不考虑硬件和速度的限制，在理论上可以用指令实现互相模仿（即图灵机）。然而在现实中，速度和硬件是必须考虑的。

仿真器，或模拟器（英文；emulator、simulator），根据此原理制作的软件又可称为模拟程序，是指主要透过软件模拟硬件处理器的功能和指令系统的程序使计算机或者其他多媒体平台（掌上电脑，手机）能够运在自动化技术、化学工程中同样使用模拟器这一术语。模拟器多用于电视游戏和街机，也有一些用于掌上电脑。模拟器一般需要 ROM 才能执行，ROM 的最初来源是一些原平台的 ROM 芯片，通过一些手段将原程序拷贝下来（这个过程一般称之为“dump”）然后利用模拟器加载这些 ROM 来实现模拟过程。

2、相关知识

本次模拟的 CPU 由指令寄存器 IR、数据寄存器、PC 寄存器、程序状态寄存器 SR、16 个通用寄存器组成。指令寄存器地址总线和数据总线宽度为 16 位，数据总线的地址宽度为 16 位，数据总线的宽度为 8 位。指令寄存器 IR、PC 寄存器宽度为 16 位，16 个通用寄存器组 R0-R15, 对应的宽度为 8 位，对应的地址为 0—15。通用寄存器、程序状态寄存器和数据存储器统一编址，通用寄存器既可以用寄存器号访问，也可以用地址空间的地址访问。

程序状态寄存器的位位置、位名称、读写属性、复位时的值见下图：

程序状态寄存器介绍

位位置	7	6	5	4	3	2	1	0
位名称	I	T	H	S	V	N	Z	C
读写属性	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
复位时	0	0	0	0	0	0	0	0

所对应的含义如下：

I：全局中断允许位：为 1 时，允许中断，否则，禁止中断，CPU 响应中断时，硬件将此位清 0，从中断返回时，将此位置 1

T：位复制存储位：BLD 指令用此位的值与 16 个通用寄存器中的某位交换值

H：半进位标志：即低 4 位是否向高 4 位进位或借位，如果有则为 1，否则，为 0

S：符号标志位：本位是位 N 和位 V 的异或值

V：有符号数溢出标志位

C：无符号数溢出标志位

N：负数标志位：若运算结果是负数，则为 1；否则，为 0

Z：0 标志位：若运算结果是 0，则为 1；否则，为 0

在本次实验中，模拟 CPU 的主要内容为对输入的指令进行模拟，pc+1, IR 寄存器显示当前所执行的指令的二进制码，同时若本次操作对寄存器有影响，也将对相应的寄存器有所影响。本次实验主要展示的指令如下：

加法指令：格式为（Add Rd , Rr）,其中 Rd 为目的操作数寄存器，Rr 为原操作数寄存器，其功能是：Rd <- Rd+Rr，机器码（二进制表示）0000 1100 dddd rrrr，其中，rrrr 为源操作数的寄存器号，dddd 为目的操作数的寄存器号，所影响的标志位 Z,C,N,V,H,S。

减法指令：格式为（Sub Rd , Rr）, 其中 Rd 为目的操作数寄存器，Rr 为原操作数寄存器，其功能是：Rd <- Rd-Rr，机器码（二进制表示）0000 1000 dddd rrrr，其中，rrrr 为源操作数的寄存器号，dddd 为目的操作数的寄存器号，所影响的标志位 Z,C,N,V,H,S。

无符号乘法指令：格式为（Mul Rd, Rr），其中 Rd 为目的操作数寄存器，Rr 为原操作数寄存器，其功能是：R1:R0 <- Rd*Rr，机器码（二进制表示）1001 1100 dddd rrrr，其中，rrrr 为源操作数的寄存器号，dddd 为目的操作数的寄存器号，所影响的标志位 Z,C。

无条件相对跳转指令：格式为（RJMP K），其功能：PC<-PC+K+1，机器码（二进制表示），1100 kkkk kkkk kkkk，其中， kkkk kkkk kkkk 为相对地址，所影响的标志：无

有条件相对跳转指令：格式为（BRMI K），其功能：if (N == 1) PC <- PC + k + 1，机器码（二进制表示），1111 0001 kkkk kkkk ，其中， kkkk kkkk 为相对地址，所影响的标志位：无

数据传送指令：格式为（mov Rd, Rr），其功能是：Rd <-Rr，机器码（二进制表示）， 0010 1100 dddd rrrr，其中，rrrrr 为源操作数的寄存器号，ddddd 为目的操作数的寄存器号，所影响的标志位：无

载入立即数指令: 格式为（ldi Rd, K），其功能是：Rd <-K，机器码（二进制表示） 1110 KKKK dddd KKKK，其中，dddd 为目的操作数的寄存器号，目的寄存器只能是 r8~r15; KKKK KKKK 是立即数。所影响的标志位：无

装载指令：格式为（ld Rd, X），其功能：Rd <-（X），机器码（二进制表示），001 0000 dddd 1100

其中，dddd 为目的操作数的寄存器号，影响的标志位：无，其 X 为 R14 寄存器。

存储指令：格式为（st X, Rr），其功能 (X)<-Rr，器码（二进制表示），001 0010 rrrr 1100，中，rrrr 源操作数的寄存器号， X 为 R14：寄存器，所影响的标志位：无

空操作指令：格式为（nop），其功能 ：不做任何操作，只消耗 CPU 时间，器码（二进制表示）0000 0000 0000 0000，影响的标志位 ：无

3、关键代码

```
//存储器部分
uint32_t Msize=0;
const int wordsize = sizeof(uint32_t); //扩展指令长度 4
char* M;  //定义存储器空间为4096的数组

//分配内存空间
int allocMem(uint32_t s){
        M=new char[s];
        Msize=s;
        return s;
}
//释放内存空间
void freeMem(){
        delete[] M;
        Msize=0;
}


 //!!!!读8,16,32位不同的
int32_t ReadByte(uint32_t addr,bool flag){ //从存储器读8位值
       if(addr>=Msize) {
               cout<<"ERROR:地址范围超出内存容量"<<endl;
               return 0;
       }
       if(flag==1) //此时返回有符号的;
               return M[addr];
       else
               return (unsigned char)M[addr];
}

void WriteByte(uint32_t addr,char data){ //写8位值入存储器
       if(addr>=Msize) {
               cout<<"ERROR:地址范围超出内存容量"<<endl;
               return;
       }
       M[addr]=data;
}
```

```
100
101 int32_t Read2Byte(uint32_t addr,bool flag){ //从存储器读16位值,这里函数类型要用int32_t否则为错的？？
102         if(addr>=Msize-wordsize/2) {
103                 cout<<"ERROR:地址范围超出内存容量"<<endl;
104                 return 0;
105         }
106         if (flag==1) //返回有符号的
107                 return *((int16_t*)&(M[addr]));
108         else
109                 return *((uint16_t*)&(M[addr]));
110 }
111
112 void Write2Byte(uint32_t addr,uint32_t data){ //写值入存储器
113         if(addr>=Msize-wordsize/2) {
114                 cout<<"ERROR:地址范围超出内存容量"<<endl;
115                 return;
116         }
117         *((uint16_t*)&(M[addr]))=data;
118 }
119 int32_t ReadWord(uint32_t addr){ //从存储器读32位值??无符号？
120         if(addr>=Msize-wordsize) {
121                 cout<<"ERROR:地址范围超出内存容量"<<endl;
122                 return 0;
123         }
124         return *((int32_t*)&(M[addr]));
125 }
```

void showRegs(){//hex , oct 二进制，dec 十进制

cout<<"PC=0x"<<hex<<PC<<" "<<"IR=0x"<<IR<<endl; //这里注意！如果用 hex 之后会将之后的所有输出默认为 16 进制

cout<<"32 个寄存器值(16 进制)分别为："<<endl;

for(int i=1;i<33;i++){

cout<<"R["<<dec<<i<<"]="<<hex<<R[i]<<'\t';

}

cout<<endl;

}

```
}
void Decode(unsigned int IR){ //指令译码
        opcode= IR & 0x7f;//0111 1111截取后七位操作码
        rd= (IR>>7)& 0x1f;//将操作码移位7位后截取后5位,这样是32位吗?
        r1=     (IR>>15)&0x1f;
        r2= (IR>>20)&0x1f;
        func3=(IR>>12)&0x7;
        func7=(IR>>25)&0x7f;

        imm31_12U = (IR>>12)& 0xfffff;//取出无符号的前20位(U类)

        imm31J=(IR>>31) & 1; //取出符号位
        imm30_21J=(IR>>21) & 0x3ff;
        imm20J=(IR>>20) &1;
        imm19_12J=(IR>>12) & 0xff;

        imm31_20JR=IR>>20;//JALR取高12位，并且默认移位扩展32位为有符号的
        /*B类指令*/
        imm31B=imm31J;
        imm30_25B=(IR>>25)& 0x3f;//? ? ? 0x3f
        imm11_8B=(IR>>8)&0xf;
        imm7B=(IR>>7)&0x1;
        /*L类指令*/
        imm31_20L=IR>>20; //取高12位，默认扩展32位有符号的??修改成I类类似
        /*C类指令*/
```

4、测试输入

测试输入：

模拟器输入：

void decode(uint32_t instruction) {//decode 是译码的意思，RV32I 指令 4 个字节

// Extract all bit fields from instruction 从指令中提取所有位字段 opcode = instruction & 0x7F;//获取低 7 位，即 0~6 位

rd = (instruction & 0x0F80) >> 7;//获取从低至高第 7~11 位

rs1 = (instruction & 0xF8000) >> 15;//获取第 15~19 位，得到第一个寄存器 zimm = rs1;//zimm 是我们定义的一个 unsigned int，把 rs1 赋值给了它

rs2 = (instruction & 0x1F00000) >> 20;//获取第 20~24 位，得到第二个寄存器 shamt = rs2;//shamt 是我们定义的一个 unsigned int， 把 rs2 赋值给了它

funct3 = (instruction & 0x7000) >> 12;//获取第 12~14 位 funct7 = instruction >> 25;//获取 25~31 位

imm11_0i = ((int32_t)instruction) >> 20;//转化成有符号的再移动，对应着 Itype 类型的地址 csr = instruction >> 20;//获取 20~31 位，应该与上面的 imm11_0i 差不多，不 过是无符号类型的

imm11_5s = ((int32_t)instruction) >> 25;//获取第 25~31 位数据，对应着 Stype 类型的地址

imm4_0s = (instruction >> 7) & 0x01F;//获取第 7~11 位数据，对应 Stype 类型 的地址

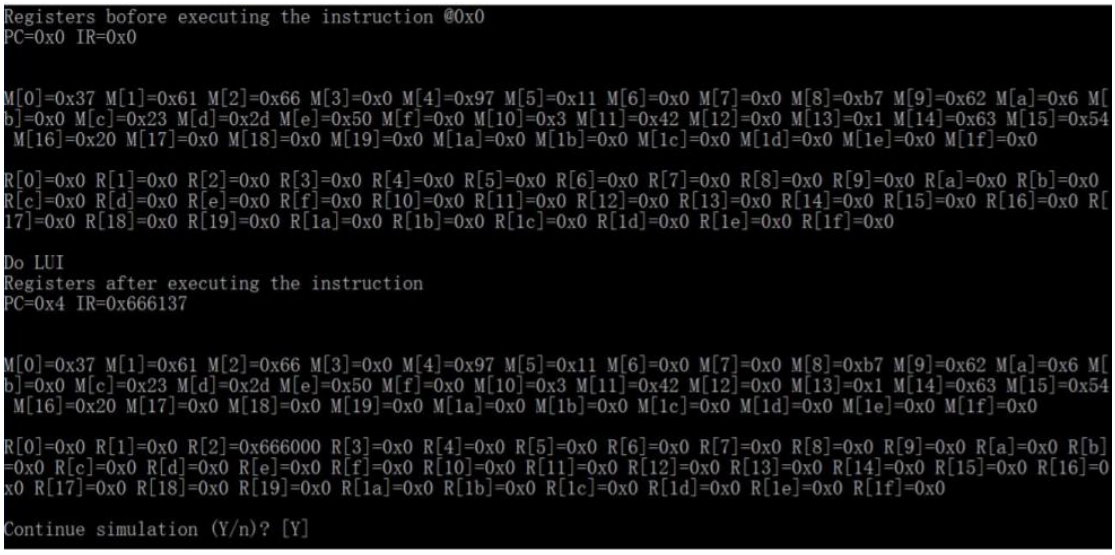
imm12b = ((int32_t)instruction) >> 31;//获取第 31 位数据,对应 Btype 类型的 地址 imm10_5b = (instruction >> 25) & 0x3F;//获取第 25~30 位数据，对应 Btype 类 型的地址

imm4_1b = (instruction & 0x0F00) >> 8;//第 8~11 位，对应 Btype 类型的地址

imm11b = (instruction & 0x080) >> 7;//第 7 位，对应 Btype 类型的地址
imm31_12u = instruction >> 12;//第 12~31 位，对应 Utype 类型的地址
imm20j = ((int32_t)instruction) >> 31;//第 31 位，对应 jtype 类型的地址

imm10_1j = (instruction >> 21) & 0x3FF;//第 21~31 位，对应 jtype 类型的地址

imm11j = (instruction >> 20) & 1;//第 20 位，对应 jtype 类型的地址
imm19_12j = (instruction >> 12) & 0x0FF;//第 12 到 19 位，对应 jtype 类型的 地址 pred = (instruction >> 24) & 0x0F; succ = (instruction >> 20) & 0x0F;



```
Registers bofore executing the instruction @0x4
PC=0x4 IR=0x666137


M[0]=0x37 M[1]=0x61 M[2]=0x66 M[3]=0x0 M[4]=0x97 M[5]=0x11 M[6]=0x0 M[7]=0x0 M[8]=0xb7 M[9]=0x62 M[a]=0x6 M[
b]=0x0 M[c]=0x23 M[d]=0x2d M[e]=0x50 M[f]=0x0 M[10]=0x3 M[11]=0x42 M[12]=0x0 M[13]=0x1 M[14]=0x63 M[15]=0x54
 M[16]=0x20 M[17]=0x0 M[18]=0x0 M[19]=0x0 M[1a]=0x0 M[1b]=0x0 M[1c]=0x0 M[1d]=0x0 M[1e]=0x0 M[1f]=0x0

R[0]=0x0 R[1]=0x0 R[2]=0x666000 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]=0
x0 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 AUIPC
PC = 4
Imm31_12UtypeZeroFilled = 1000
Registers after executing the instruction
PC=0x8 IR=0x1197


M[0]=0x37 M[1]=0x61 M[2]=0x66 M[3]=0x0 M[4]=0x97 M[5]=0x11 M[6]=0x0 M[7]=0x0 M[8]=0xb7 M[9]=0x62 M[a]=0x6 M[
b]=0x0 M[c]=0x23 M[d]=0x2d M[e]=0x50 M[f]=0x0 M[10]=0x3 M[11]=0x42 M[12]=0x0 M[13]=0x1 M[14]=0x63 M[15]=0x54
 M[16]=0x20 M[17]=0x0 M[18]=0x0 M[19]=0x0 M[1a]=0x0 M[1b]=0x0 M[1c]=0x0 M[1d]=0x0 M[1e]=0x0 M[1f]=0x0

R[0]=0x0 R[1]=0x0 R[2]=0x666000 R[3]=0x1004 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
```

```
Registers bofore executing the instruction @0x8
PC=0x8 IR=0x1197


M[0]=0x37 M[1]=0x61 M[2]=0x66 M[3]=0x0 M[4]=0x97 M[5]=0x11 M[6]=0x0 M[7]=0x0 M[8]=0xb7 M[9]=0x62 M[a]=0x6 M[
b]=0x0 M[c]=0x23 M[d]=0x2d M[e]=0x50 M[f]=0x0 M[10]=0x3 M[11]=0x42 M[12]=0x0 M[13]=0x1 M[14]=0x63 M[15]=0x54
 M[16]=0x20 M[17]=0x0 M[18]=0x0 M[19]=0x0 M[1a]=0x0 M[1b]=0x0 M[1c]=0x0 M[1d]=0x0 M[1e]=0x0 M[1f]=0x0

R[0]=0x0 R[1]=0x0 R[2]=0x666000 R[3]=0x1004 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 LUI
Registers after executing the instruction
PC=0xc IR=0x662b7


M[0]=0x37 M[1]=0x61 M[2]=0x66 M[3]=0x0 M[4]=0x97 M[5]=0x11 M[6]=0x0 M[7]=0x0 M[8]=0xb7 M[9]=0x62 M[a]=0x6 M[
b]=0x0 M[c]=0x23 M[d]=0x2d M[e]=0x50 M[f]=0x0 M[10]=0x3 M[11]=0x42 M[12]=0x0 M[13]=0x1 M[14]=0x63 M[15]=0x54
 M[16]=0x20 M[17]=0x0 M[18]=0x0 M[19]=0x0 M[1a]=0x0 M[1b]=0x0 M[1c]=0x0 M[1d]=0x0 M[1e]=0x0 M[1f]=0x0

R[0]=0x0 R[1]=0x0 R[2]=0x666000 R[3]=0x1004 R[4]=0x0 R[5]=0x66000 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0
x0 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
```

简易存储器

1、简单介绍

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。

2、关键代码

```
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;
```

CPU

1、简单介绍

微处理器的主要功能是进行控制部件和算术逻辑部件的功能。

微处理器由一片或少数几片大规模集成电路组成的中央处理器。微处理器能完成取指令、执行指令，以及与外界存储器和逻辑部件交换信息等操作，是微型计算机的运算控制部分。它可与存储器和外围电路芯片组成微型计算机。

微处理器与传统的中央处理器相比，具有体积小、重量轻和容易模块化等优点。微处理器的基本组成部分有：寄存器堆、运算器、时序控制电路，以及数据和地址总线。

微处理器的分类

根据微处理器的应用领域，微处理器大致可以分为三类：通用高性能微处理器、嵌入式微处理器和数字信号处理器、微控制器。

一般而言，通用处理器追求高性能，它们用于运行通用软件，配备完备、复杂的操作系统；嵌入式微处理器强调处理特定应用问题的高性能，主要用于运行面向特定领域的专用程序，配备轻量级操作系统，主要用于蜂窝电话、CD 播放机等消费类家电。

微控制器价位相对较低，在微处理器市场上需求量最大，主要用于汽车、空调、自动机械等领域的自控设备。

CPU 是 Central Processing Unit（中央微处理器）的缩写，它是计算机中最重要的一个部分，由运算器和控制器组成。如果把计算机比作人，那么 CPU 就是人的大脑。CPU 的发展非常迅速，个人电脑从 8088(XT)发展到 Pentium 4 时代，只经过了二十一年的时间。

2、相关知识

1.程序计数器(PC)：包含当前正在执行的指令的地址，当指令被获取之后，一般情况下，指向下一条指令。

2.存储器(Memory):主要有三个作用：

a.存储 CPU 运行的指令

b.保存指令运行过程中的临时变量

c.在指令执行前存放初始化数据

在本示例中，即是指令存放的地方又是操作数所存放的地方

3.指令解码器(Idec):

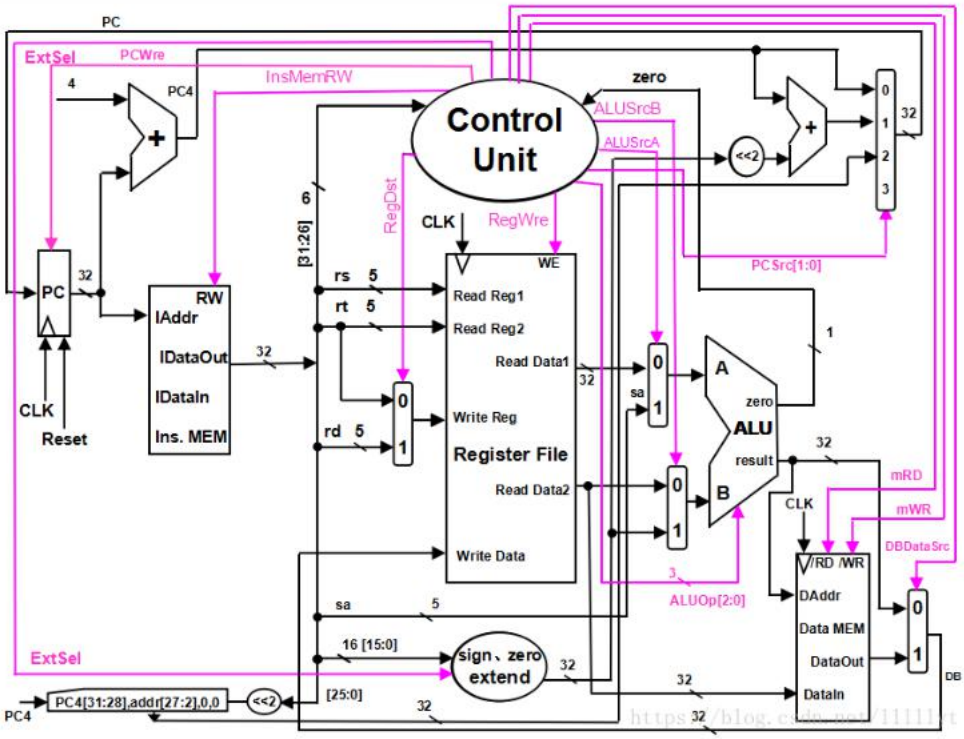
a.将从指令存储器(**本示例中即是从 memory 中读出**)中读出的指令进行翻译，CPU 根据翻译后的代码执行不同的操作

b.将所读取到的指令分为指令码和操作码(本示例中实现 6 条计算机指令，故指令码采取 3 位)

4.算术逻辑单元(ALU)：实现数据的二进制运算及判断标志位(负数标志，溢出标志等)的输出。

5.控制单元(Control): 此部件是本 CPU 的核心模块，控制着整个 CPU 有条不紊地运行，它控制着每个部件何时使能工作，ALU 何时进行什么样的运算，这个功能在 CPU 的顶层原理图体现的淋漓尽致。

单周期CPU数据通路和控制线路图：



3、部分关键代码

```
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;
    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);


ltype_imm11_0<=ir(31 downto 20);




-- 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_intege
r(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_intege
r(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;
```

五、实验体会和总结

微处理器的这次 CPU 课程设计让我更加深入的理解和巩固了 CPU 方面的知识，比如 CPU 中每条指令的执行包括六个阶段：取指令、指令解码、根据操作码决定是否从存储器中读取数据、确定 ALU 的运算、必要时往存储器中写数据、使能 PC 准备开始读取下一条指令。

虽然说，代码不是自己的，而是网上的开源代码，但是经过一段时间的观看和更改，也足以让我收获很多。