《计算机组成原理实验》

实验报告

（实验三）

学 院 名 称 ： 数据科学与计算机学院

学 生 姓 名 ： 吴沐晓

学 号 ： 15352339

专业（班级）： 15 移动 3 班

时 间 ： 2017 年 6 月 2 日

成 绩 ：

实验三 ：多周期CPU设计

一. 实验目的

1、认识和掌握多周期数据通路原理及其设计方法。
2、掌握多周期 CPU 的实现方法，代码实现方法。
3、掌握多周期 CPU 的测试方法。

二. 实验内容

设计一个多周期 CPU，该 CPU 至少能实现以下指令功能操作。需设计的指令与格式如下：（**说明：操作码按照以下规定使用，都给每类指令预留扩展空间。**）

==>算术运算指令

（1）add rd, rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd<－rs + rt

（2）sub rd, rs, rt

000001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

完成功能：rd<－rs － rt

（3）addi rt, rs, **immediate**

000010	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt<－rs + (sign-extend)**immediate**

==>逻辑运算指令

（4）or rd, rs, rt

010000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd<－rs | rt

（5）and rd, rs, rt

010001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd<－rs & rt

（6）ori rt, rs, **immediate**

010010	rs(5 位)	rt(5 位)	**immediate**

功能：rt<－rs | (zero-extend)**immediate**

==>移位指令

（7）sll rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved

功能：rd<－rt<<(zero-extend)sa，左移 sa 位 ，(zero-extend)sa

==>比较指令

（8） slt rd, rs, rt 带符号数

100110	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：if (rs<rt)　rd =1 else　rd=0，具体请看表 2 ALU 运算功能表，带符号

（9）　sltu　rd, rs, rt　　不带符号数

100111	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：if (rs<rt)　rd =1 else　rd=0，具体请看表 2 ALU 运算功能表，不带符号

==>存储器读写指令

（10）sw　rt, **immediate**(rs)

110000	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：memory[rs+ (sign-extend)**immediate**]<－rt。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

（11）lw　rt, **immediate**(rs)

110001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt <－ memory[rs + (sign-extend)**immediate**]。即读取 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数，然后保存到 rt 寄存器中。

==>分支指令

(12) beq　rs,rt, **immediate** (说明：**immediate 从 pc+4 开始和转移到的指令之间间隔条数**)

110100	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if(rs=rt) pc <－pc + 4 + (sign-extend)**immediate** <<2 else pc <－pc + 4

(13) bne　rs,rt, **immediate** (说明：**immediate 从 pc+4 开始和转移到的指令之间间隔条数**)

110101	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if(rs!=rt) pc <－pc + 4 + (sign-extend)**immediate** <<2 else pc <－pc + 4

==>跳转指令

（14）j　addr

111000	addr[27..2]

功能：pc <－{(pc+4)[31..28],addr[27..2],0,0}，跳转。

说明：由于 MIPS32 的指令代码长度占 4 个字节，所以指令地址二进制数最低 2 位均为 0，将指令地址放进指令代码中时，可省掉！这样，除了最高 6 位操作码外，还有 26 位可用于存放地址，事实上，可存放 28 位地址，剩下最高 4 位由 pc+4 最高 4 位拼接上。

（15）jr　rs

111001	rs(5 位)	未用	未用	reserved

功能：pc <－ rs，跳转。

==>调用子程序指令

（16）jal　addr

111010	addr[27..2]

功能：调用子程序，pc <－ {(pc+4)[31..28],addr[27..2],0,0}；$31<－pc+4，返回地

址设置；子程序返回，需用指令 jr $31。跳转地址的形成同 j addr 指令。

==>停机指令

（17）halt (停机指令)

111111	00000000000000000000000000(26 位)

不改变 pc 的值，pc 保持不变。

三. 实验原理

多周期 CPU 指的是将整个 CPU 的执行过程分成几个阶段，每个阶段用一个时钟去完成，然后开始下一条指令的执行，而每种指令执行时所用的时钟数不尽相同，这就是所谓的多周期 CPU。CPU 在处理指令时，一般需要经过以下几个阶段:

(1) 取指令(IF)：根据程序计数器 pc 中的指令地址，从存储器中取出一条指令，同时，pc 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 pc，当然得到的“地址”需要做些变换才送入 pc。

(2) 指令译码(ID)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

(3) 指令执行(EXE)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

(4) 存储器访问(MEM)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(5) 结果写回(WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

实验中就按照这五个阶段进行设计，这样一条指令的执行最长需要五个(小)时钟周期才能完成，但具体情况怎样？要根据该条指令的情况而定，有些指令不需要五个时钟周期的，这就是多周期的 CPU。

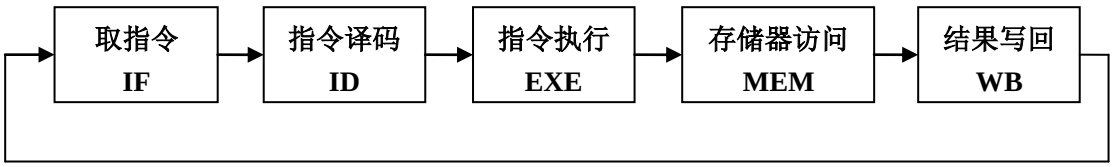
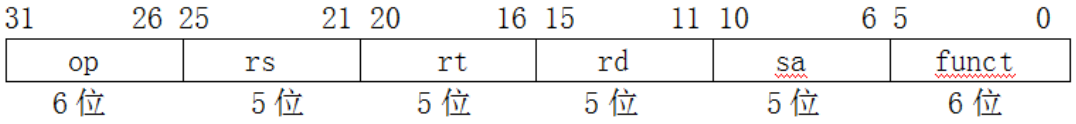


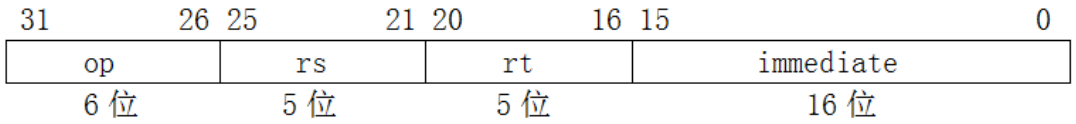
图 1　多周期 CPU 指令处理过程

MIPS 指令的三种格式:

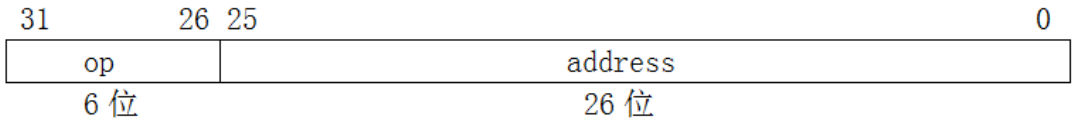
R 类型：



I 类型：



J 类型：



其中，

op：为操作码；

rs：为第 1 个源操作数寄存器，寄存器地址（编号）是 00000~11111，00~1F；

rt：为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd：为目的操作数寄存器，寄存器地址（同上）；

sa：为位移量（shift amt），移位指令用于指定移多少位；

funct：为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能；

immediate：为 16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载（Laod）/数据保存（Store）指令的数据地址字节偏移量和分支指令中相对程序计数器（PC）的有符号偏移量；

address：为地址。

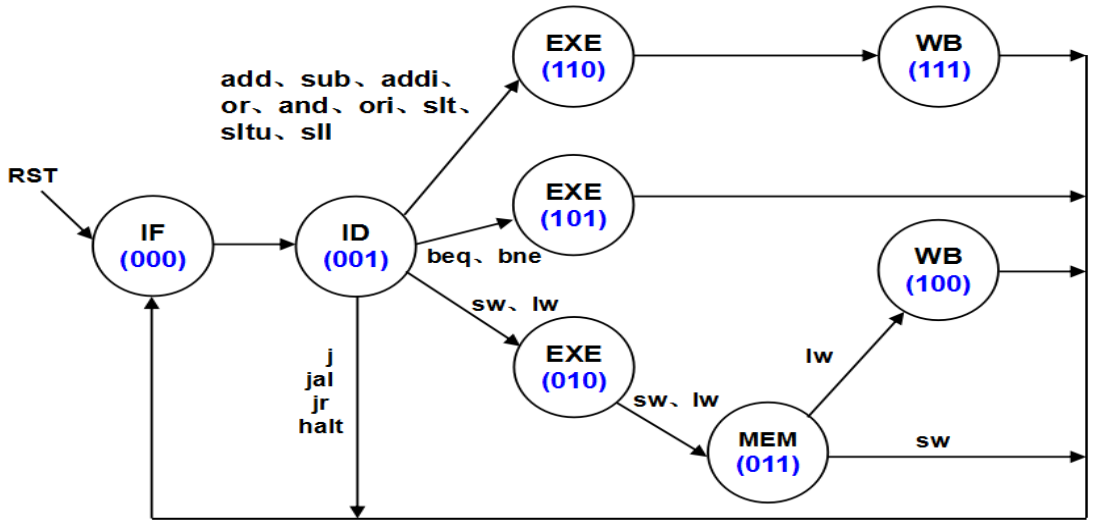


图 2 多周期 CPU 状态转移图

状态的转移有的是无条件的，例如从 IF 状态转移到 ID 就是无条件的；有些是有条件的，例如 ID 或 EXE 状态之后不止一个状态，到底转向哪个状态由该指令功能，即指令操作码决定。每个状态代表一个时钟周期。

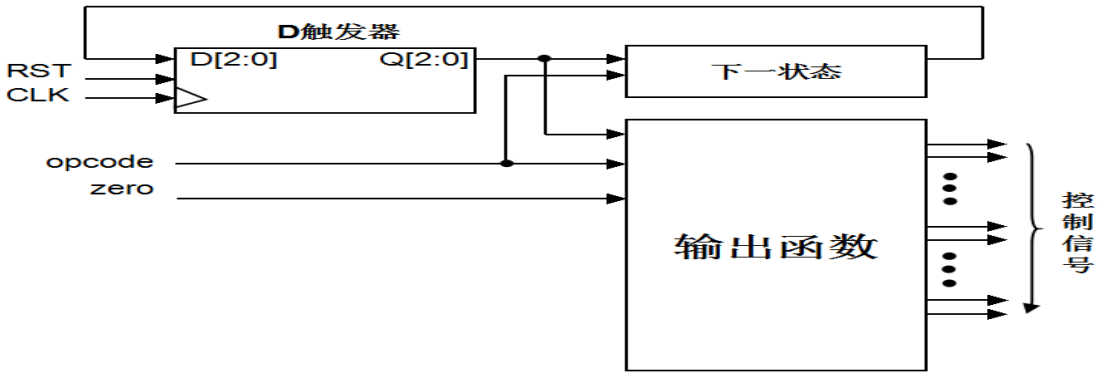


图 3 多周期 CPU 控制部件的原理结构图

图 3 是多周期 CPU 控制部件的电路结构，三个 D 触发器用于保存当前状态，是时序逻辑电路，RST 用于初始化状态“000“，另外两个部分都是组合逻辑电路，一个用于产生下一个阶段的状态，另一个用于产生每个阶段的控制信号。从图上可看出，下个状态取决于指令操作码和当前状态；而每个阶段的控制信号取决于指令操作码、当前状态和反映运算结果的状态 zero 标志等。

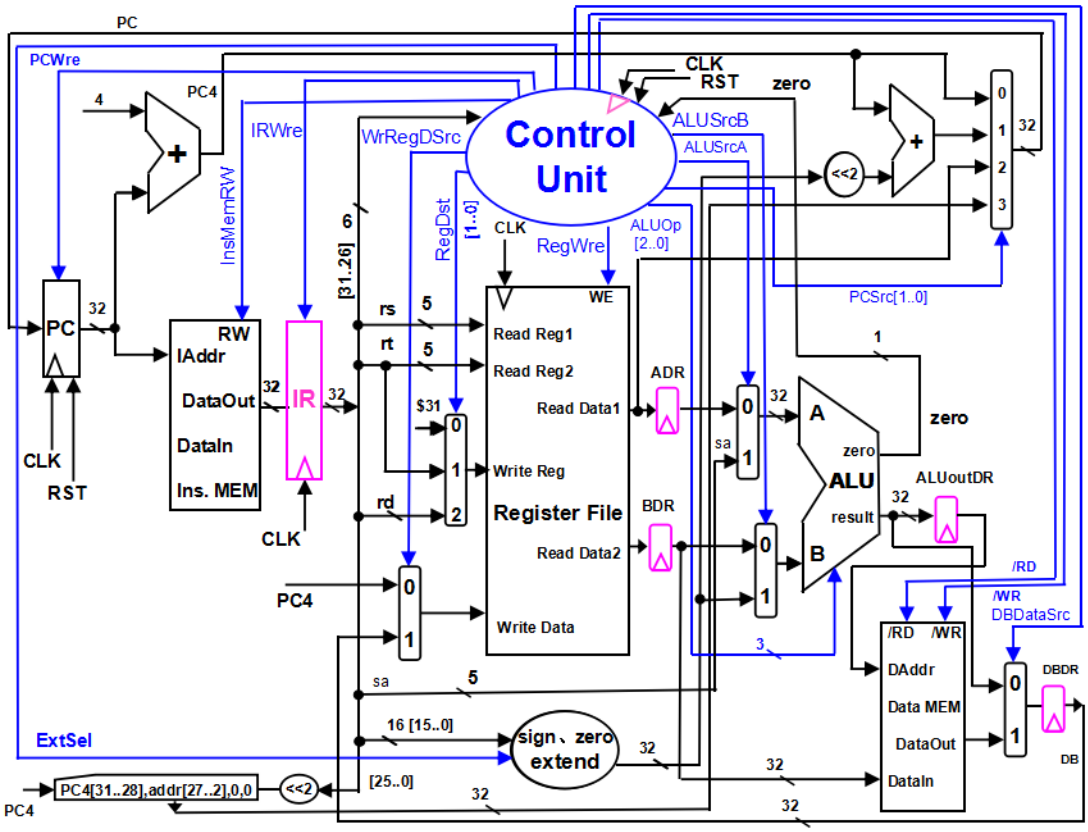


图 4 多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在多周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出地址，然后由读或写信号控制操作。对于寄存器组，读操作时，给出寄存器地址（编号），输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发写入。图中控制信号功能如表 1 所示，表 2 是 ALU 运算功能表。

特别提示，图上增加 IR 指令寄存器，目的是使指令代码保持稳定，pc 写使能控制信号 PCWre，是确保 pc 适时修改，原因都是和多周期工作的 CPU 有关。ADR、BDR、ALUoutDR、DBDR 四个寄存器**不需要写使能信号，**其作用是切分数据通路，将大组合逻辑切分为若干个小组合逻辑，大延迟变为多个分段小延迟。

表 1 控制信号作用

控制信号名	状态“0”	状态“1”
RST	对于 PC，初始化 PC 为程序首地址	对于 PC，PC 接收下一条指令地址
PCWre	PC 不更改，相关指令：halt，另外，除‘000’状态之外，其余状态慎改 PC 的值。	PC 更改，相关指令：除指令 halt 外，另外，在‘000’状态时，修改 PC 的值合适。
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addi、or、and、ori、beq、bne、slt、sltu、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {27(0),sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、addi、or、and、ori、beq、bne、slt、sltu、sll	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、lw、sw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、sub、addi、or、and、ori、slt、sltu、sll、move	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、j、sw、jr、halt	寄存器组寄存器写使能，相关指令：add、sub、addi、or、and、ori、slt、sltu、sll、lw、jal
WrRegData	写入寄存器组寄存器的数据来自 pc+4(pc4），相关指令：jal，写 $31	写入寄存器组寄存器的数据来自存储器、寄存器组寄存器和 ALU 运算结果，相关指令：add、addi、sub、or、and、ori、slt、sltu、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
/RD	读数据存储器，相关指令：lw	输出高阻态
/WR	写数据存储器，相关指令：sw	无操作
IRWre	IR(指令寄存器)不更改	IR 寄存器写使能。向指令存储器发出读指令代码后，这个信号也接着发出，在时钟上升沿，IR 接收从指令存储器送来的指令代码。与每条指令都相关。
ExtSel[1..0]	00：(zero-extend)sa，相关指令：sll 01：(zero-extend)**immediate**，相关指令：ori 10：(sign-extend)**immediate**，相关指令：addi、lw、sw、beq 11：未用 值得注意的是这个信号与原实验原理的有所不同	
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slt、sltu、sll、sw、lw、beq(zero=0)、bne(zero=1)； 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bne(zero=0)；	

	10：pc<－rs，相关指令：jr； 11：pc<－{pc(31..28],addr[27..2],0,0}，相关指令：j、jal；
RegDst[1..0]	写寄存器组寄存器的地址，来自： 00：0x1F($31)，相关指令：jal，用于保存返回地址（$31<-pc+4）； 01：rt 字段，相关指令：addi、ori、lw； 10：rd 字段，相关指令：add、sub、or、and、slt、sltu、sll； 11：未用；
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	Y = A + B	加
001	Y = A － B	减
010	Y= (A<B) ?1: 0	比较 A 与 B 不带符号

011	if (rega<regb &&((rega[31] == 0 && regb[31]==0) || (rega[31] == 1 && regb[31]==1))) result = 1;
	else if (rega[31] == 0 && regb[31]==1) result = 0;
	else if (rega[31] == 1 && regb[31]==0) result = 1;
	else result = 0;
		比较 A 与 B
		带符号

100	Y = B<<A	B 左移 A 位
101	Y = A ∨ B	或
110	Y = A ∧ B	与
111	Y = A ⊕ B	异或

四. 实验器材

电脑一台、Xilinx Vivado 软件一套。

五. 实验分析与设计

模块化设计：

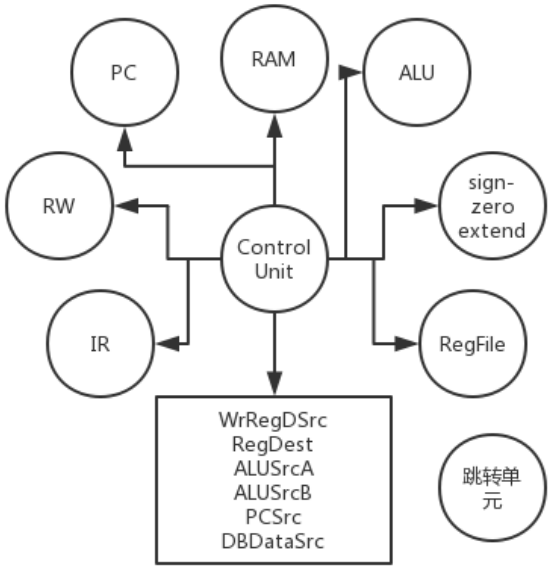


图5多周期CPU模块化设计

除了控制电路外，本次实验共有8个大模块。方框中的是6个多路选择器。接下来我将对其中8个大模块的设计进行详细分析。

<1>控制电路

下面将列出控制电路的真值表，控制电路模块实际上实现的就是一个查表的功能。

根据数据通路图可以知道，控制单元的功能是接收一个6位的操作码（opCode）和一个标志符（zero）作为输入，在CLK的激励RST的控制下遵循实验原理中的表1控制信号的规范，输出PCWre、ALUSrcB等控制信号。各控制信号的作用定义也在实验原理的控制信号作用表（表1）给出。为了达到控制各指令的目的，其中模块内部是根据opCode与控制电路中的当前状态curState的逻辑运算来实现的

举一个例子来说：例如lw $8 1($2)指令的实现。根据图2多周期CPU状态转移图我们可以得到sw指令共需要经过IF，ID，EXE3（010），MEM，WB2（100）五个状态。这是目前来说经历了最多状态的指令。下面将分阶段对sw指令进行分析。

Stage	Ins	Zero	PCWre	ALUSrcB	DBDataSrc	RegWre	WrRegData	InsMemRW	/RD	/WR	IRWre	ExtSel	PCSrc [1..0]	RegDst [1..0]	ALUOp [2..0]
IF	x	x	1	x	x	0	x	1	0	1	1	xx	xx	xx	xxx
ID	j	x	0	x	x	0	x	x	0	1	0	xx	11	xx	xxx
	jal	x	0	x	x	1	0	x	0	1	0	xx	11	00	xxx
	jr	x	0	x	x	0	x	x	0	1	0	xx	10	xx	xxx
	halt	x	0	x	x	0	x	x	0	1	0	xx	xx	xx	xxx
exe1	add	x	0	0	x	0	x	x	0	1	0	xx	xx	xx	000
	sub	x	0	0	x	0	x	x	0	1	0	xx	xx	xx	001
	addi	x	0	1	x	0	x	x	0	1	0	10	xx	xx	000
	or	x	0	0	x	0	x	x	0	1	0	xx	xx	xx	101
	and	x	0	0	x	0	x	x	0	1	0	xx	xx	xx	110
	ori	x	0	1	x	0	x	x	0	1	0	01	xx	xx	101
	move	x	0	0	x	0	x	x	0	1	0	xx	xx	xx	000
	slt	x	0	0	x	0	x	x	0	1	0	xx	xx	xx	010
	sltu	x	0	0	x	0	x	x	0	1	0	xx	xx	xx	011
	sll	x	0	1	x	0	x	x	0	1	0	00	xx	xx	100
exe2	beq	0	0	0	x	0	x	x	0	1	0	10	00	xx	001
	bnq	1	0	0	x	0	x	x	0	1	0	10	00	xx	001
	beq	1	0	0	x	0	x	x	0	1	0	10	01	xx	001
	bnq	0	0	0	x	0	x	x	0	1	0	10	01	xx	001
exe3	sw	x	0	1	x	0	x	x	0	1	0	10	xx	xx	000
	lw	x	0	1	x	0	x	x	0	1	0	10	xx	xx	000
mem	sw	x	0	x	x	0	x	x	1	0	0	10	00	xx	xxx
	lw	x	0	x	x	0	x	x	0	1	0	10	xx	xx	xxx
wb1	add	x	0	x	0	1	1	x	0	1	0	xx	00	10	xxx
	sub	x	0	x	0	1	1	x	0	1	0	xx	00	10	xxx
	addi	x	0	x	0	1	1	x	0	1	0	xx	00	01	xxx
	or	x	0	x	0	1	1	x	0	1	0	xx	00	10	xxx
	and	x	0	x	0	1	1	x	0	1	0	xx	00	10	xxx
	ori	x	0	x	0	1	1	x	0	1	0	xx	00	01	xxx
	Slt/ sltu	x	0	x	0	1	1	x	0	1	0	xx	00	10	xxx
	sll	x	0	x	0	1	1	x	0	1	0	xx	00	10	xxx
wb2	lw	x	0	x	1	1	1	x	0	1	0	xx	00	01	xxx

表3控制信号与指令的关系表

IF：取指令，取得下一条指令的地址。如果上调指令没有跳转则为PC<-PC+4

PCWre = 1;

ID：指令译码。取得指令的内容，在本次中为c4480001。在IR模块中将其拆分为不同的代码块。IRWre = 1

op(6)	rs(5)	rs(5)	Immediate(16)
110001	00010	01000	0000 0000 0000 0001

EXE3：计算rs+ (sign-extend)immediate

ALUOp = 3'b000;

MEM：储存器访问取得memory[rs + (sign-extend)immediate]

WR = 0; RD = ~WR;

WB2：写回rt <— memory[rs + (sign-extend)immediate]

RegWre = 1;

<2>PC

PC 的作用是随时钟获取指令在 RW 中的地址。若 Reset==1&&PCwre==0,每次到时钟的下 降沿的时候 PC 会取得 PCInput 的地址。PCInput是由四路选择器PCSrc决定的， 有四种情况。

```
wire [31:0] branchPC;
assign branchPC = (Ext << 2) + nextPC;
always @(*) begin
    case(PCSrcIn)
    2'b00:PCSrcOut <= nextPC;
    2'b01:PCSrcOut <= branchPC;
    2'b10:PCSrcOut <= ReadData1;
    2'b11:PCSrcOut <= JOut;
    endcase
end
```

<3>指令存储器RW

实际上指令存储器RW和内存RAW是一个模块。但是为了简化试验，我们把指令存储器单独分开一个模块。试验的最后我们需要一个简单的程序来验证各条指令的正确性。而指令存储器中就要预先加载这些指令。在本次实验中通过一条语句来实现：

```
initial       $readmemb ("C:/Users/Yan/Desktop/ECOP/instructions.txt", RM);
```

在IF状态时，遇到时钟上升沿的时候指令存储器RW读取对应地址的指令并输出到IR部件中。

<4>寄存器IR

图上增加IR指令寄存器，目的是使指令代码保持稳定。同时在ID状态时由IRWre控制IR进行译码工作。本次实验中代码指令种类增加，相比之前的单周期CPU中的指令多了J型指令。将指令分为不同的指令块供后序模块选择使用。

```
always @(InsOut) begin
        op[5:0] <= InsOut[31:26];
        rs[4:0]   <= InsOut[25:21];
        rt[4:0]   <= InsOut[20:16];
        rd[4:0]   <= InsOut[15:11];
        sa[4:0] <= InsOut[10:6];
        immediate[15:0] <= InsOut[15:0];
        addressJmp[25:0] <= InsOut[25:0];
end
```

<5>寄存器 RegFile

RegFile 中有 32 个 32 位的寄存器。这一点我们通过一条语句实现：

```
 reg [31:0]   register [1:31];
```

其中值得一提的是零号寄存器。零号寄存器是一个为零只读寄存器。为了避免错误，在本次实验中实际上没有真实的零号寄存器。每当要访问零号寄存器的时候实际上是直接返回数值零而不是返回零号寄存器里的值。

对于其他寄存器我们可以正常的进行读写操作。在EXE状态中如果中完成的。一般情况下

写寄存器是在WB状态下完成的。但是对于jal指令有所例外。Jal指令要将当前的PC+4存储到$31中。由于jal指令只经历IF，ID两个状态，所以jal中的写$31号寄存器是在ID阶段中完成的。这一点在控制电路中要额外注意。

<6>ALU32

ALU32 实现的是对两个输入进行逻辑运算。在EXE状态中会使用到。具体是什么逻辑运算是由控制电路决定的。 控制电路 ALUop 和逻辑运算的关系已经在表 2 ALU 运算功能表中给出。实现上可以通过一个计算机组成原理实验 简单的 case 语句来完成。

```
always @( ALUopCode or rega or regb ) begin
        case (ALUopCode)
          3'b000 : Result = rega + regb;
          3'b001 : Result = rega - regb;
          3'b010 : Result = (rega<regb)?1:0;
          3'b011 : begin
                if (rega<regb &&(( rega[31] == 0 && regb[31]==0)  || (rega[31] == 1 && regb[31]==1)))  Result = 1;
                else if (rega[31] == 0 && regb[31]==1)  Result = 0;
                else if ( rega[31] == 1 && regb[31]==0)  Result = 1;
                else Result = 0;  end
          3'b100 : Result = regb<<rega ;
          3'b101 : Result = rega|regb;
          3'b110 : Result = rega&regb;
          3'b111 : Result = rega^regb;
          default : begin
                    Result = 8'h00000000;
                    $display (" no match");
            end
        endcase
    end
```

<7>内存RAM

本次RAM的代码基本与单周期CPU的代码是一致的。在MEM状态时会访问或修改RAM。

在计算机中RAM是以一个字节来作为存储单位的，在本次实验中通过一个一条语句来实现。对于内存我们可以像对寄存器一样的操作方法来实现读写。值得一提的是读是可以不用放在 always 中，而写是需要时钟同步。

```
 reg [7:0] RAM [0:60]; //存储
assign DataOut[7:0]   = (RD==0)?RAM[address]:8'bz;
assign DataOut[15:8]  = (RD==0)?RAM[address + 1]:8'bz;
assign DataOut[23:16] = (RD==0)?RAM[address + 2]:8'bz;
assign DataOut[31:24] = (RD==0)?RAM[address + 3]:8'bz;
always@(*) begin
   if( WR==0 ) begin
       RAM[address+3] <= writeData[31:24];
       RAM[address+2] <= writeData[23:16];
       RAM[address+1] <= writeData[15:8];
       RAM[address]   <= writeData[7:0];
   end
end
```

<8>zero-sign extend

拓展器根据 ExtSel 对输入的立即数做零拓展或者符号拓展。与单周期CPU不同的是这次的有sa和immediate两种输入，其中sa是零拓展immediate可能是符号拓展或者零拓展。可能会出错的是符号拓展是根据输入 16 位数的正负进行一拓展或者零拓展。

<9>多路选择器

多路选择器应该是简单的。但实际上我也遇到了一个问题，有时使用assign赋值变量却不会发生改变，于是我都放置在了always@(*)中解决这个问题。举例：DBDataSrc。

```
always@(*) begin
    DBDataSrcOut <= (DBDataSrcIn==1)?DataOut:ALU; end
```

测试程序设计

测试程序全部重新设计。

程序概述：将$1-$7通过ALU指令赋值1-7，将$8赋值为-1（用于slt和sltu两条指令的比较），$9用作存储比较结果的寄存器。将slt、sltu和beq、bnq两两组合变成小于则跳转指令用作检查slt、sltu的两种比较情况和beq、bnq是否跳转。穿插检测j，jar，jr指令。

地址	汇编程序	指令代码				
		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)	十六进制代码
0x00000000	addi $3,$0,3	000010	00000	00011	0000 0000 0000 0011	8030003
0x00000004	ori $5,$0,5	010010	00000	00101	0000 0000 0000 0101	48050005
0x00000008	and $1,$5,$3	010001	00101	00011	0000 1000 0000 0000	44a30800
0x0000000C	sub $2,$5,$3	000001	00101	00011	0001 0000 0000 0000	4a31000
0x00000010	add $4,$3,$1	000000	00011	00001	0010 0000 0000 0000	612000
0x00000014	sll $6 $3 1	011000	00000	00011	00110 00001 000000	60033040
0x00000018	or $7,$6,$3	010000	00110	00011	0011 1000 0000 0000	40c33800
0x0000001C	addi $8,$0,-1	000010	00000	01000	1111 1111 1111 1111	808ffff
0x00000020	slt $9 $8 $1（slt置1）	100110	01000	00001	0100 1000 0000 0000	99014800
0x00000024	bne $9 $1 10（bne不跳）	110101	01001	00001	0000 0000 0000 1010	d521000a
0x00000028	Sltu $9 $8 $1（sltu置0）	100111	01000	00001	0100 1000 0000 0000	9d014800
0x0000002C	bne $9 $1 6（bne跳）	110101	01001	00001	0000 0000 0000 0110	d5210006
0x00000030	lw $8 1($2)	110001	00010	01000	0000 0000 0000 0001	c4480001
0x00000034	jar 0x00000054	111010	0000 0000 0000 0000 0000 0101 01			E8000015
0x00000038	slt $9 $8 $1（slt置0）	100110	01000	00001	0100 1000 0000 0000	99014800
0x0000003C	beq $9 $1 4（beq不跳）	110100	01001	00001	0000 0000 0000 0100	d1210004
0x00000040	sltu $9 $8 $2 (sltu 置1)	100111	01000	00010	0100 1000 0000 0000	9D024800
0x00000044	beq $9 $1 2（beq跳）	110100	01001	00001	0000 0000 0000 0010	d1210002
0x00000048	sw $1 1($2)	110000	00010	00001	0000 0000 0000 0001	c0410001
0x0000004c	j 0x00000030	111000	0000 0000 0000 0000 0000 0011 00			E000000c
0x00000050	halt	111111	00000	00000	0000 0000 0000 0000	fc000000
0x00000054	jr $31	111001	11111 00000 0000 0000 0000 0000			e7e00000

表4检测程序代码

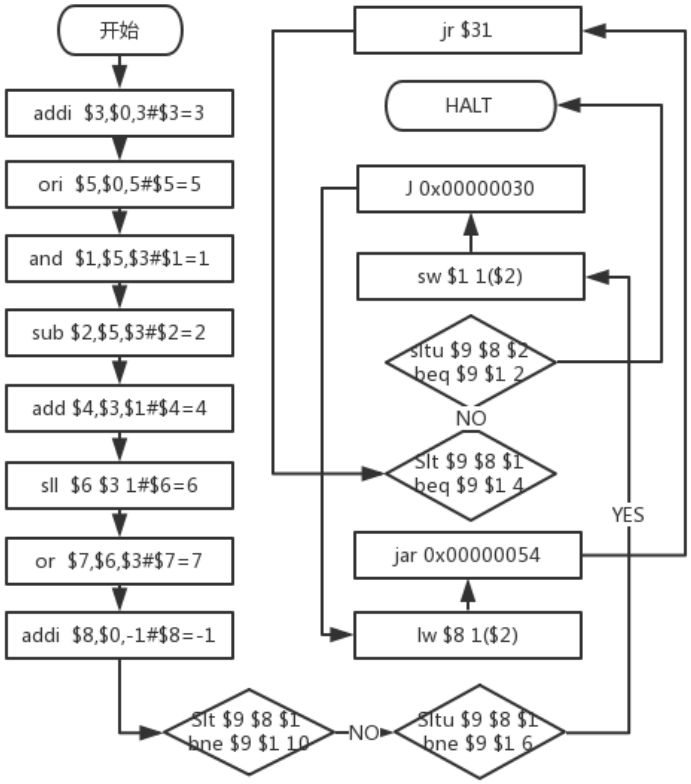
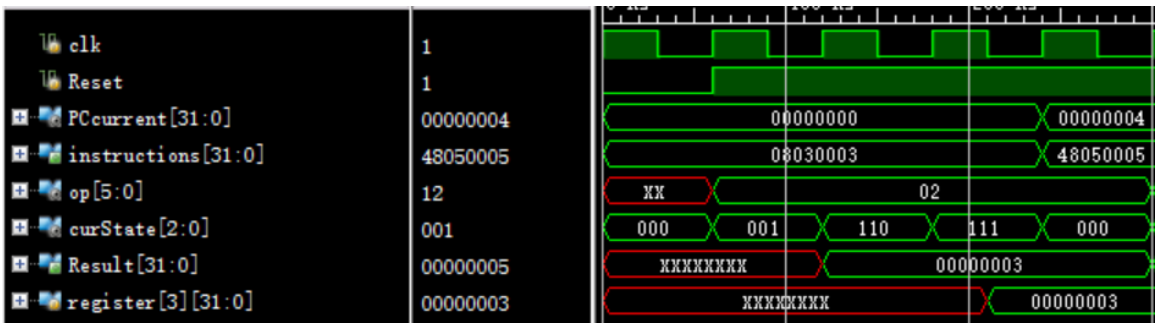


图6测试程序正确运行流程图

代码正确性分析

0x00000000	addi $3, $0, 3	8030003



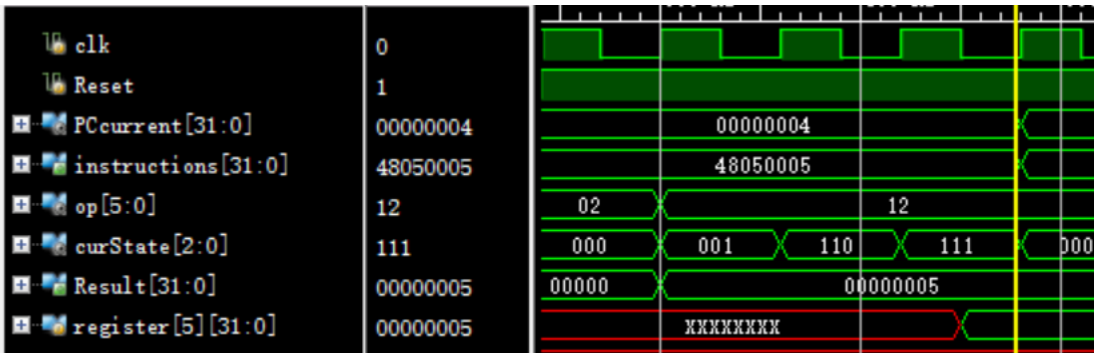
IF：成功读取地址0的指令8030003与预期一致

ID：成功进行指令译码其中opCode为02

EXE：成功计算出了结果3

WB：成功写回register[3]=3

0x00000004	ori $5,$0,5	48050005



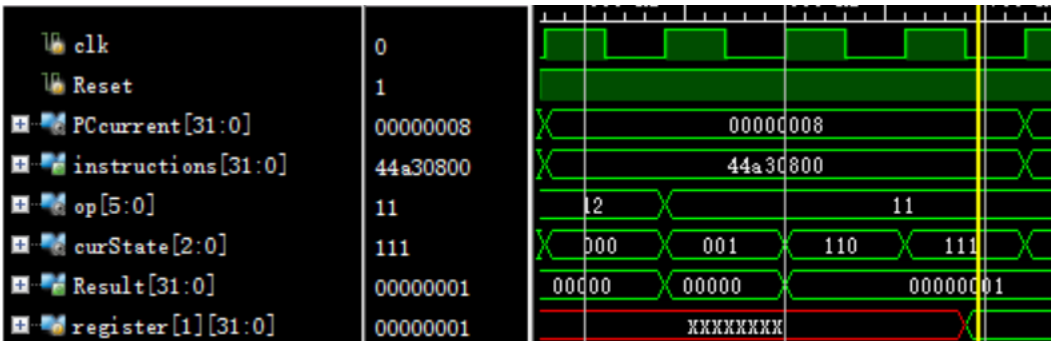
IF：成功读取地址04的指令48050005与预期一致

ID：成功进行指令译码其中opCode为12

EXE：成功计算出了结果5

WB：成功写回register[5]=5

0x00000008	and $1,$5,$3	44a30800



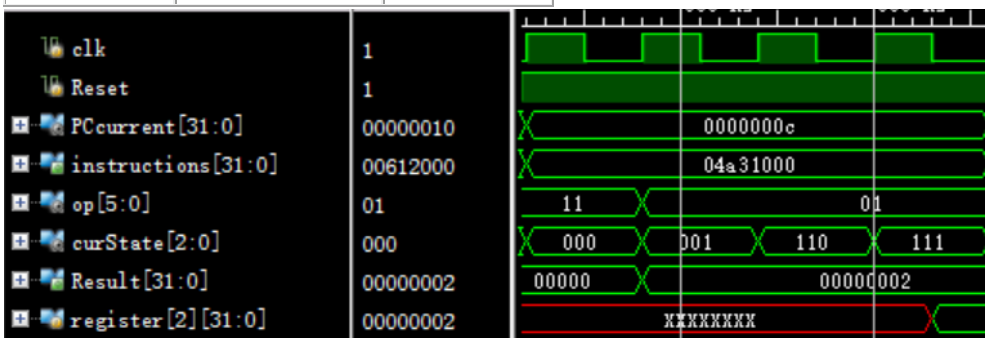
IF：成功读取地址08的指令44a30800与预期一致

ID：成功进行指令译码其中opCode为11

EXE：成功计算出了结果1

WB：成功写回register[1]=1

0x0000000C	sub $2,$5,$3	4a31000



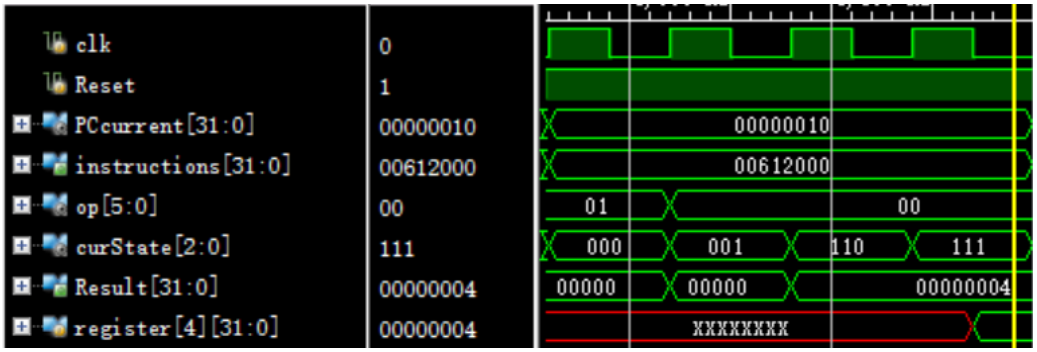
IF：成功读取地址c的指令4a31000与预期一致

ID：成功进行指令译码其中opCode为01

EXE：成功计算出了结果2

WB：成功写回register[2]=2

0x00000010	add $4,$3,$1	612000



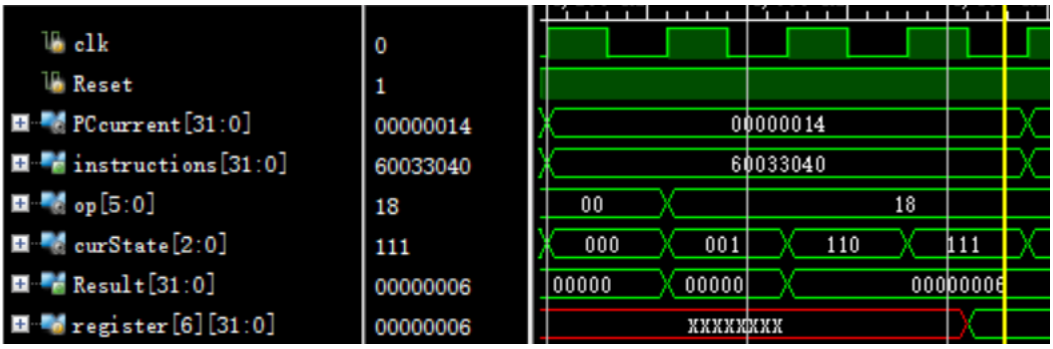
IF：成功读取地址10的指令612000与预期一致

ID：成功进行指令译码其中opCode为00

EXE：成功计算出了结果4

WB：成功写回register[4]=4

0x00000014	sll $6 $3 1	60033040



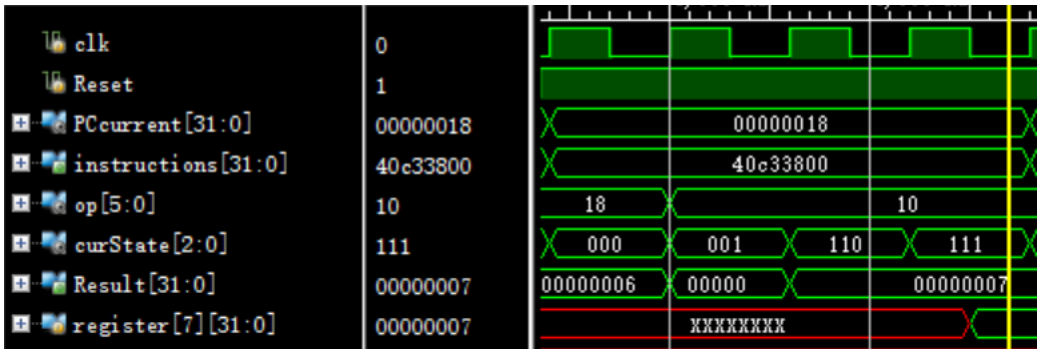
IF：成功读取地址14的指令60033040与预期一致

ID：成功进行指令译码其中opCode为18

EXE：成功计算出了结果6

WB：成功写回register[6]=6

0x00000018	or $7,$6,$3	40c33800



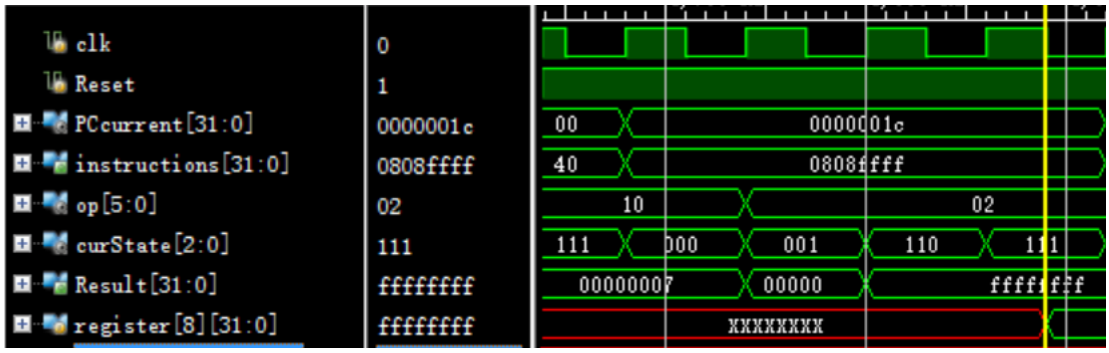
IF：成功读取地址18的指令40c33800与预期一致

ID：成功进行指令译码其中opCode为10

EXE：成功计算出了结果3

WB：成功写回register[3]=3

0x0000001C	addi $8,$0,-1	808ffff



IF：成功读取地址1c的指令808ffff与预期一致

ID：成功进行指令译码其中opCode为02

EXE：成功计算出了结果ffffffff

WB：成功写回register[8]=ffffffff

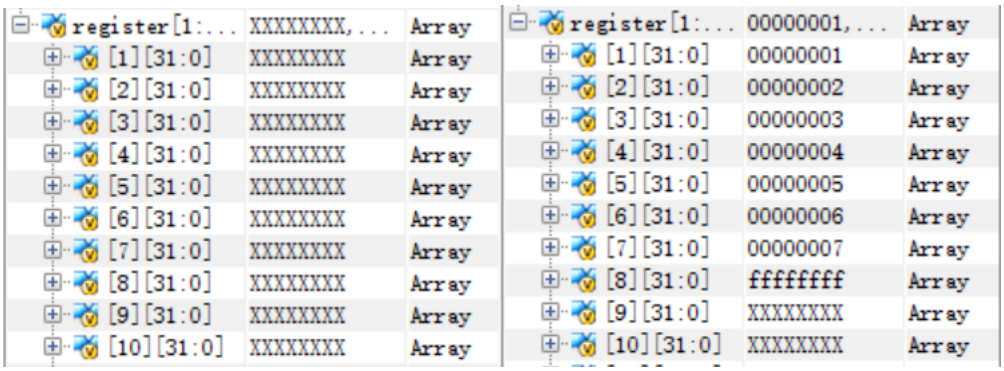
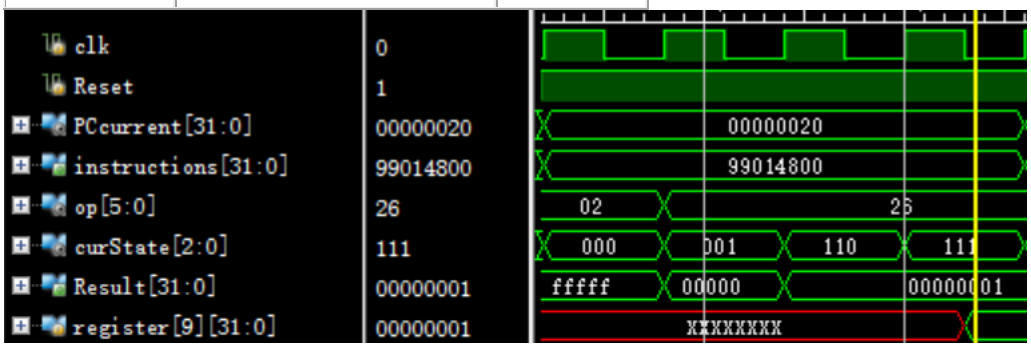


图7原始寄存器和运行至此的寄存器

0x00000020	slt $9 $8 $1 （slt 置 1）	99014800



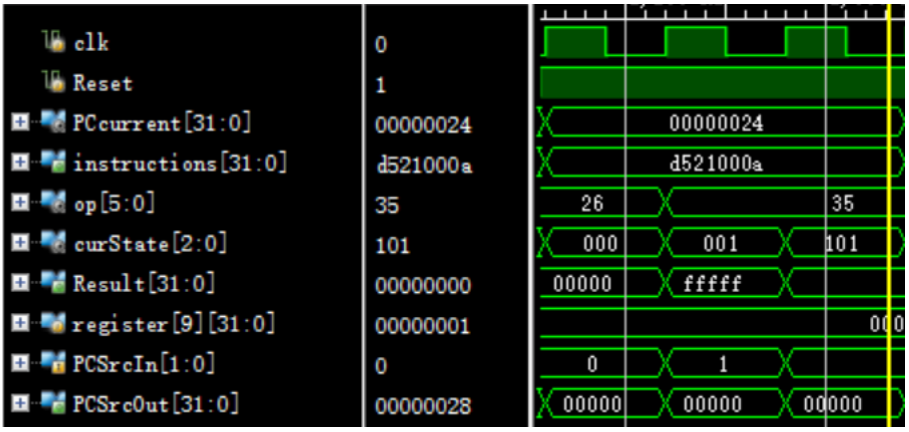
IF：成功读取地址20的指令99014800与预期一致

ID：成功进行指令译码其中opCode为25

EXE：比较得到的结果为真

WB：成功写回register[9]=1

0x00000024	bne $9 $1 10（bne 不跳）	d521000a

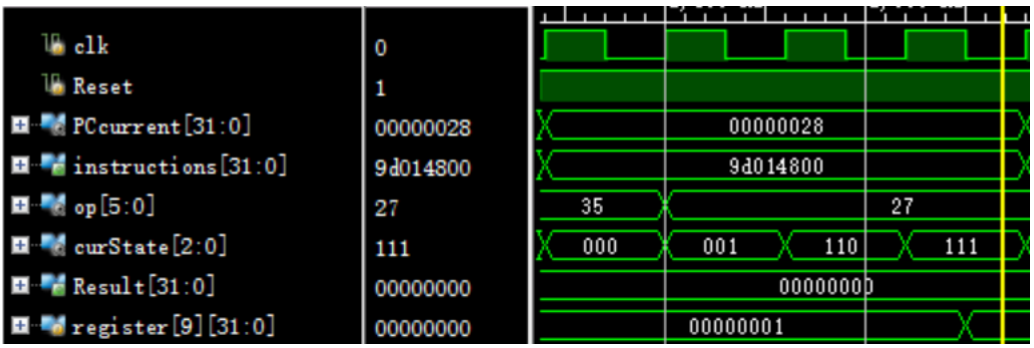


IF：成功读取地址24的指令d521000a与预期一致

ID：成功进行指令译码其中opCode为35

EXE：取得register[9]为1，不跳转，PCSrc置0，下一条指令是28

0x00000028	Sltu $9 $8 $1（sltu 置 0）	9d014800



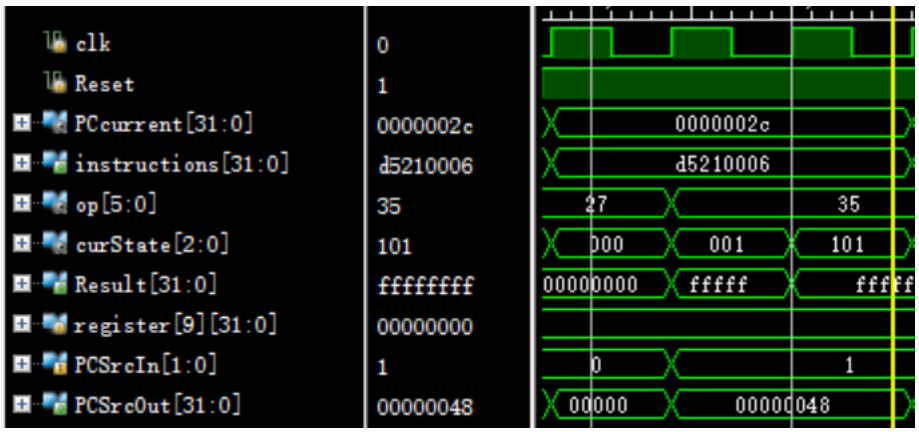
IF：不跳转成功，读取地址28的指令9d014800与预期一致

ID：成功进行指令译码其中opCode为27

EXE：比较得到的结果为假

WB：成功写回register[9]=0

0x0000002C	bne $9 $1 6（bne 跳）	d5210006

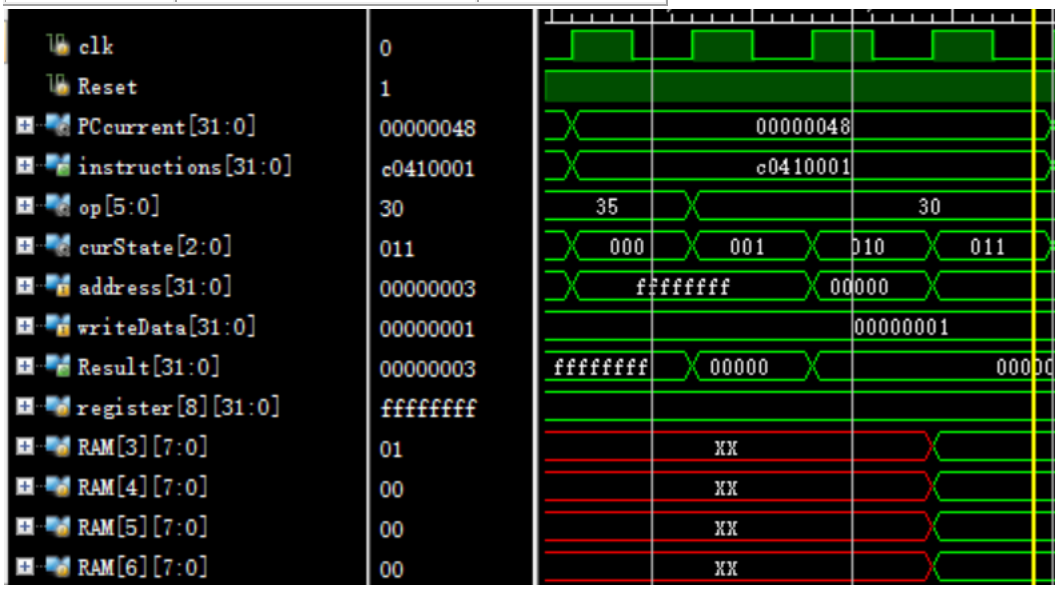


IF：成功读取地址2c的指令d5210006与预期一致

ID：成功进行指令译码其中opCode为35

EXE：取得register[9]=0，跳转，PCSrc置1，下一条指令是48

0x00000048	sw $1 1($2)	c0410001



IF：跳转成功，读取地址48的指令c0410001与预期一致

ID：成功进行指令译码其中opCode为30

EXE:取得register[1]中的1

MEM：将RAM[3,4,5,6] = register[1]中的1

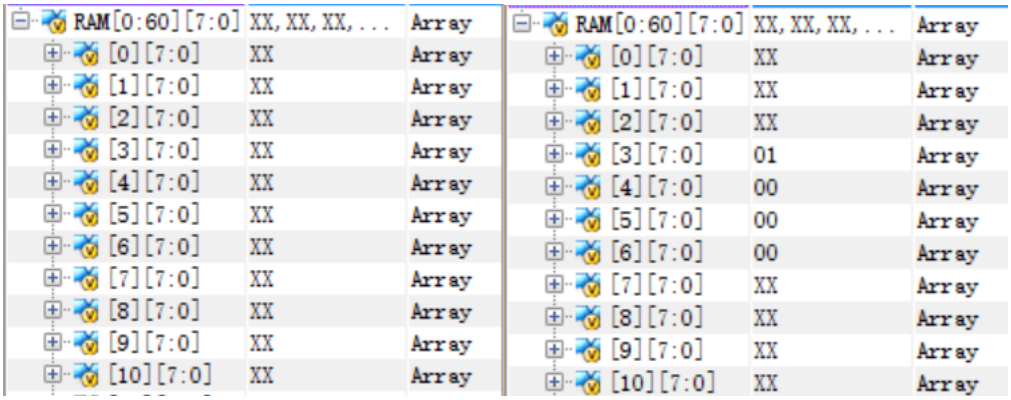
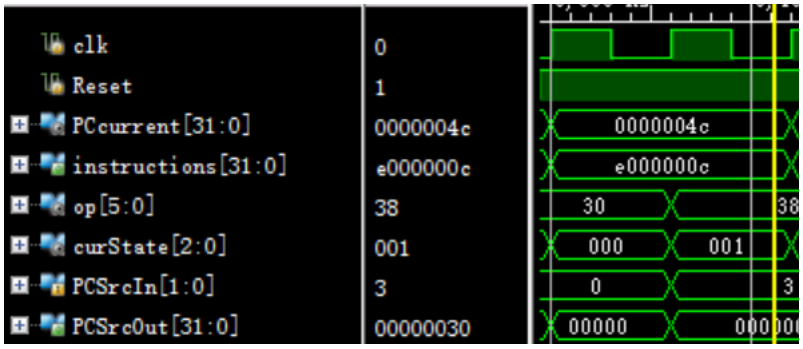


图8原始内存和运行sw后内存

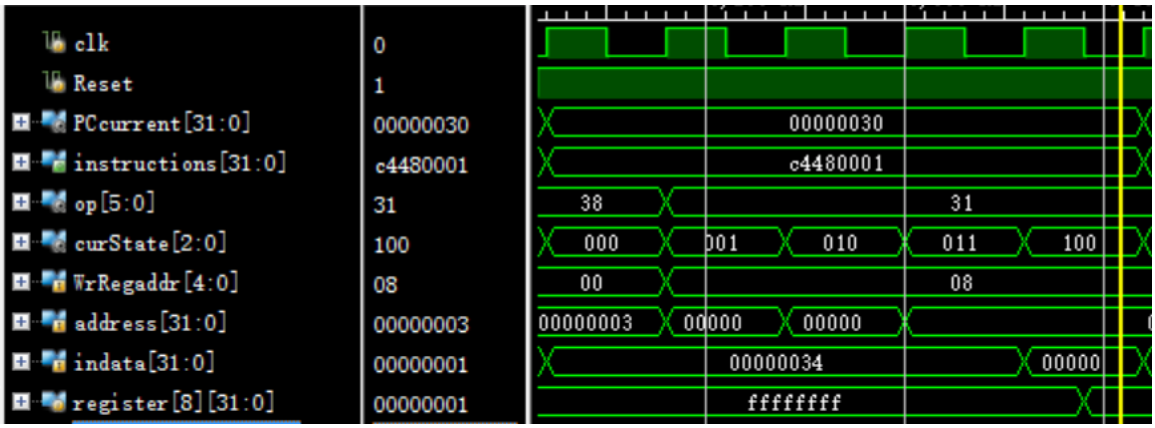
0x0000004c	j 0x00000030	E000000c



IF：成功读取地址4c的指令E000000c与预期一致

ID：成功进行指令译码其中opCode为38,将PCSrcIn置为11，PCSrcOut取30

0x00000030	lw $8 1($2)	c4480001

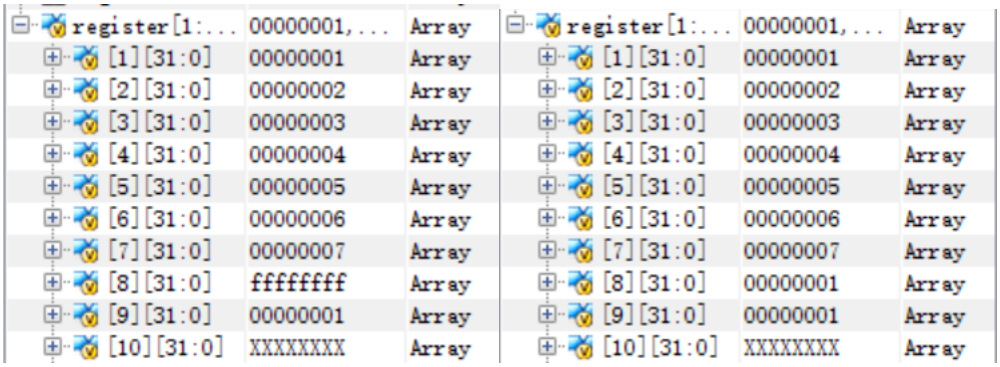


IF：跳转成功，读取地址30的指令c4480001与预期一致

ID：成功进行指令译码其中opCode为31

EXE：计算RAM的地址得到3

MEM：将RAM[3,4,5,6] 赋给 register[8]

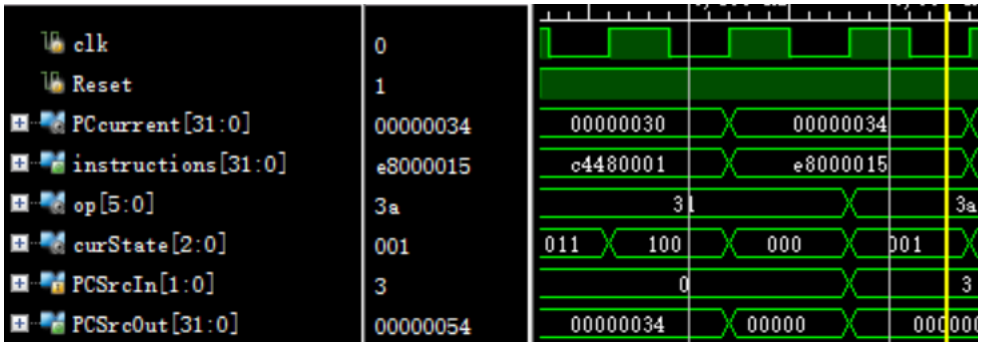


register[1:...	00000001,...	Array
[1][31:0]	00000001	Array
[2][31:0]	00000002	Array
[3][31:0]	00000003	Array
[4][31:0]	00000004	Array
[5][31:0]	00000005	Array
[6][31:0]	00000006	Array
[7][31:0]	00000007	Array
[8][31:0]	ffffffff	Array
[9][31:0]	00000001	Array
[10][31:0]	XXXXXXXX	Array

register[1:...	00000001,...	Array
[1][31:0]	00000001	Array
[2][31:0]	00000002	Array
[3][31:0]	00000003	Array
[4][31:0]	00000004	Array
[5][31:0]	00000005	Array
[6][31:0]	00000006	Array
[7][31:0]	00000007	Array
[8][31:0]	00000001	Array
[9][31:0]	00000001	Array
[10][31:0]	XXXXXXXX	Array

图9运行lw前的寄存器运行lw后的寄存器

0x00000034	jar 0x00000054	E8000015



IF：成功读取地址34的指令E8000015与预期一致

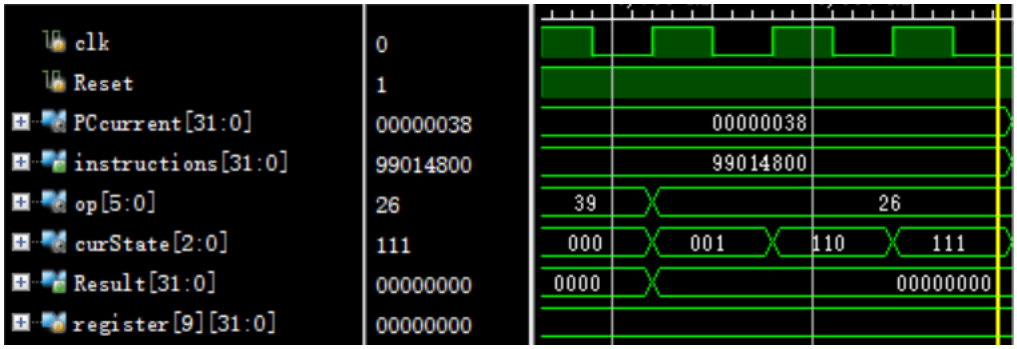
ID：成功进行指令译码其中opCode为3a，nextPC存入$31将PCSrcIn置为11，PCSrcOut取54

0x00000054	jr $31	e7e00000

IF：成功跳转，读取地址4c的指令E000000c与预期一致

ID：成功进行指令译码其中opCode为38,将PCSrcIn置为11，PCSrcOut取$31=38

0x00000038	slt $9 $8 $1 （slt置0）	99014800



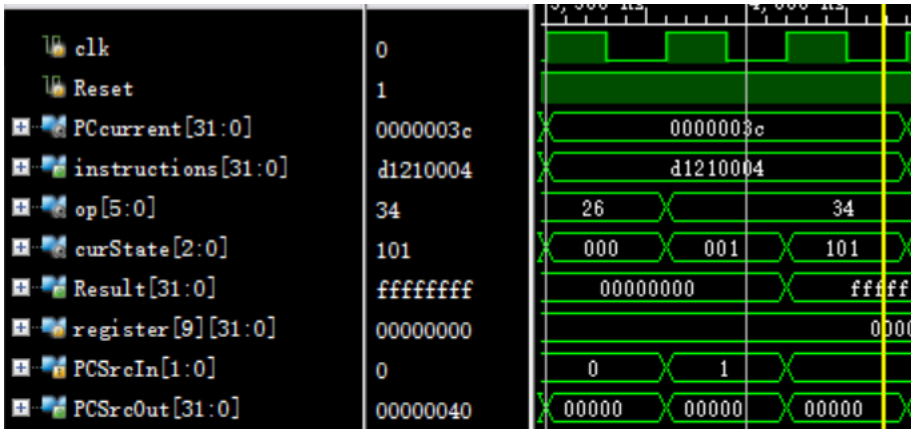
IF：成功跳转，读取地址38的指令99014800与预期一致

ID：成功进行指令译码其中opCode为26

EXE：比较得到的结果为假

WB：成功写回register[9]=0

0x0000003C	beq $9 $1 4（beq不跳）	d1210004

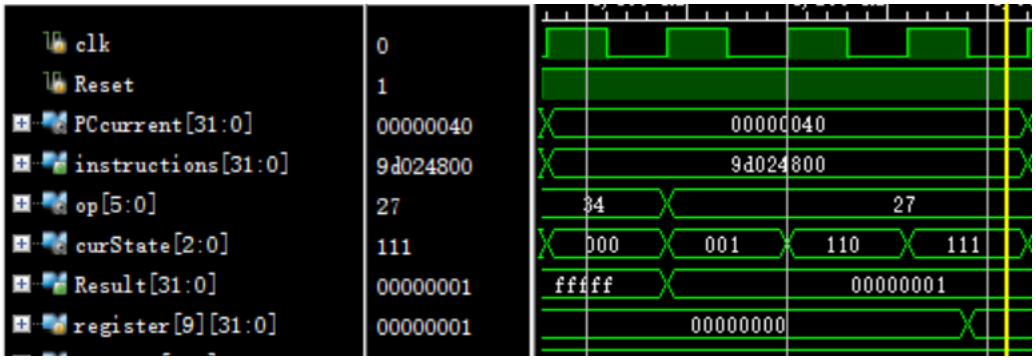


IF：成功读取地址3c的指令d1210004与预期一致

ID：成功进行指令译码其中opCode为34

EXE：取得register[9]=0，不跳转，PCSrc置0，下一条指令是40

0x00000040	sltu $9 $8 $2(sltu 置1)	9D024800



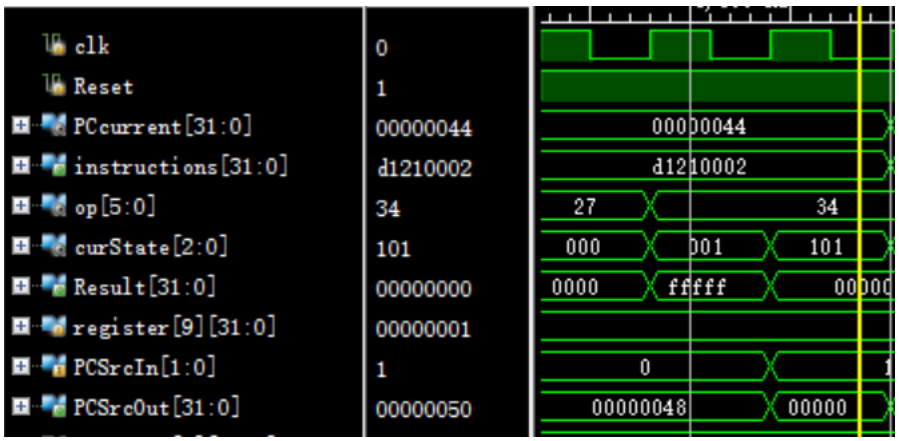
IF：成功读取地址40的指令9D024800与预期一致

ID：成功进行指令译码其中opCode为27

EXE：比较得到的结果为真

WB：成功写回register[9]=1

0x00000044	beq $9 $1 2（beq跳）	d1210002

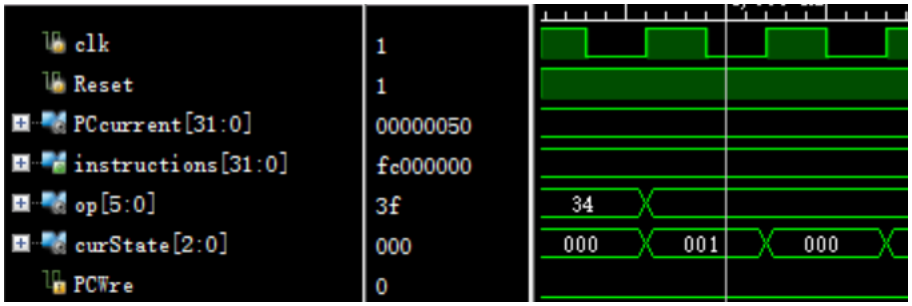


IF：成功读取地址44的指令d1210002与预期一致

ID：成功进行指令译码其中opCode为34

EXE：取得register[9]=1，跳转，PCSrc置1，下一条指令是50

0x00000050	halt	fc000000



IF：成功读取地址44的指令dfc000000与预期一致

ID：成功进行指令译码其中opCode为3f

其中PCWre一直为零，PC不更改进入停机状态。测试程序结束。

六、实验心得

遇到的问题

（1） 控制电路中的两个状态curState和nextState

在一开始我写控制电路的状态转换时将其直接理解成一个时序电路。回想到上学期学的数字电路设计中时序电路直接用一个state就可以完成改变，完全不需要两个变量来表示当前状态和下一步状态，所以我只设置了一个变量state。之后又紧接着根据state来发出控制信号。这就造成了一个问题，当时钟上升沿到来的时候，state改变，那控制信号到底是根据改变前还是改变后的来发出呢？这就造成了混乱。之后我把curState和nextState两个变量分开，这样就解决了问题。

（2） jar如何把nextPC存入$31

jar指令是特殊的。一般的写回都在WB状态中。但是jar只有IF和ID两个状态，没有WB状态的它是如何写回的呢？CPU为此给jar开了两个小后门。一是在RegDst三选一选择器中直接有$31的地址。而是WrRegDSrc中直接提供了PC4。这两个部件是的jar可以在ID状态时就把PC4写入$31。

本次实验总结

本次的实验虽然是在单周期上进行更改，但是我觉得难度还是要大一些的。其中最麻烦的是有些部件是跟单周期CPU的类似但是却不完全相同。这给我造成了很大的困扰。本来以为不会有错的单周期中CPU的代码，只是因为一个小小的地方就会出错。这种错误是很难找出来的。

另外我可以改进的是我的模块可以减少。本次实验中每出现一个模块，无论是选择器还是重要部件，我都会为将其模块化。这导致我的子模块非常多，找错误也很麻烦。其实有一些选择器就可以融入到其他模块中不用单独模块化。

本学期实验课总结

本学期实验课总共有三次实验的作业，虽然次数不多但是每一次的含金量真不少。

第一次算是小试牛刀，迈出了实操汇编语言的第一步。对于刚接触MIPS的我们来说增加了我们对汇编语言的熟悉程度。

第二次实验作业是单周期CPU。总的来说我觉得是不难的。说是CPU的设计其实设计图也都已经给我们了，我们能找到的资源也非常丰富，所以在本次实验中我的进程是比较顺利的。这样以来在不那么困难的情况下我们也算是真正了解了CPU底层的构造。

第三次实验作业就是多周期CPU了。这次我觉得是最难的一次作业，同时我们制作的CPU也更加接近真实的CPU情况。如果直接要求我们做多周期CPU那一定是非常困难的，但是在第二次铺垫之后完成多周期CPU，这样我们就能更加得感受到它们的相似和不同。这也是我这次实验中收获最多的。

所以总地来说，我喜欢这种层层递进的，却不失挑战的学习方式。我收获颇丰，很开心能来上您的实验课。