《计算机组成原理与接口技术实验》

实验报告

（实验二）

学 院 名 称：数据科学与计算机学院

专业（班级）：16 软件工程三（6）班

学 生 姓 名：王迎旭

学　　号：16340226

时　　间：2018 年 5 月 25 日

成 绩 ：

实验二 ： 单周期CPU设计与实现

一. 实验目的

1. 掌握单周期 CPU 数据通路图的构成、原理及其设计方法；
2. 掌握单周期 CPU 的实现方法，代码实现方法；
3. 认识和掌握指令与 CPU 的关系；
4. 掌握测试单周期 CPU 的方法；
5. 掌握单周期 CPU 的实现方法。

二. 实验内容

设计一个单周期 CPU，该 CPU 至少能实现以下指令功能操作。指令与格式如下：

==> 算术运算指令

（1）**add rd , rs, rt** （**说明：以助记符表示，是汇编指令；以代码表示，是机器指令**）

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

功能：rd←rs + rt。**reserved 为预留部分，即未用，一般填“0”。**

（2）addi rt , rs ,**immediate**

000001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs + (sign-extend)**immediate**；**immediate** 符号扩展再参加“加”运算。

（3）sub rd , rs , rt

000010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs - rt

==> 逻辑运算指令

（4）ori rt , rs ,**immediate**

010000	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs | (zero-extend)**immediate**；**immediate** 做“0”扩展再参加“或”运算。

（5）and rd , rs , rt

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

功能：rd←rs & rt；逻辑与运算。

（6）or rd , rs , rt

010010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

==>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）slti rt, rs,immediate 带符号

011011	rs(5 位)	rt(5 位)	**immediate**(16 位)

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

==> 存储器读/写指令

（9）sw rt ,**immediate**(rs) 写存储器

100110	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：memory[rs+ (sign-extend)**immediate**]←rt；**immediate** 符号扩展再相加。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。

（10）lw rt , **immediate**(rs) 读存储器

100111	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt ← memory[rs + (sign-extend)**immediate**]；**immediate** 符号扩展再相加。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

（11）beq rs,rt,**immediate**

110000	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

特别说明：**immediate 是从 PC+4 地址开始和转移到的指令之间指令条数。immediate** 符号扩展之后左移 2 位再相加。为什么要左移 2 位？由于跳转到的指令地址肯定是 4 的倍数（每条指令占 4 个字节），最低两位是“00”，因此将 **immediate** 放进指令码中的时候，是右移了 2 位的，也就是以上说的“指令之间指令条数”。

（12）bne rs,rt,**immediate**

110001	rs(5 位)	rt(5 位)	**immediate**

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

特别说明：与 beq 不同点是，不等时转移，相等时顺序执行。

==>跳转指令

（13）j addr

111000	addr[27..2]

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

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 位地址了，剩下最高 4 位由 pc+4 最高 4 位拼接上。

==> 停机指令

（14）halt

111111	00000000000000000000000000(26 位)

功能：停机；不改变 PC 的值，PC 保持不变。

三. 实验原理

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期（**如果晶振的输出没有经过分频就直接作为 CPU 的工作时钟，则时钟周期就等于振荡周期。若振荡周期经二分频后形成时钟脉冲信号作为 CPU 的工作时钟，这样，时钟周期就是振荡周期的两倍。**）

CPU 在处理指令时，一般需要经过以下几个步骤：

(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。

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

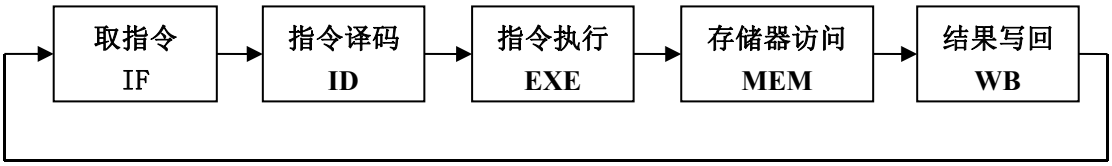
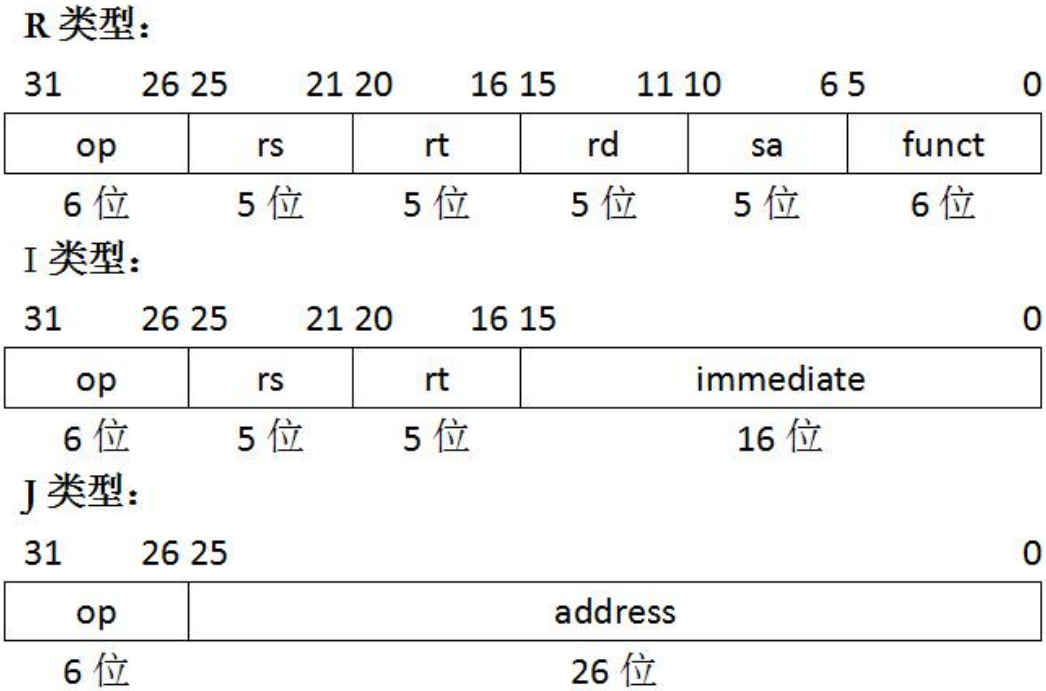


图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：



其中，

op：为操作码；

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

rt：可读可写。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：只写。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与操作码配合使用；

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(Load)/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。

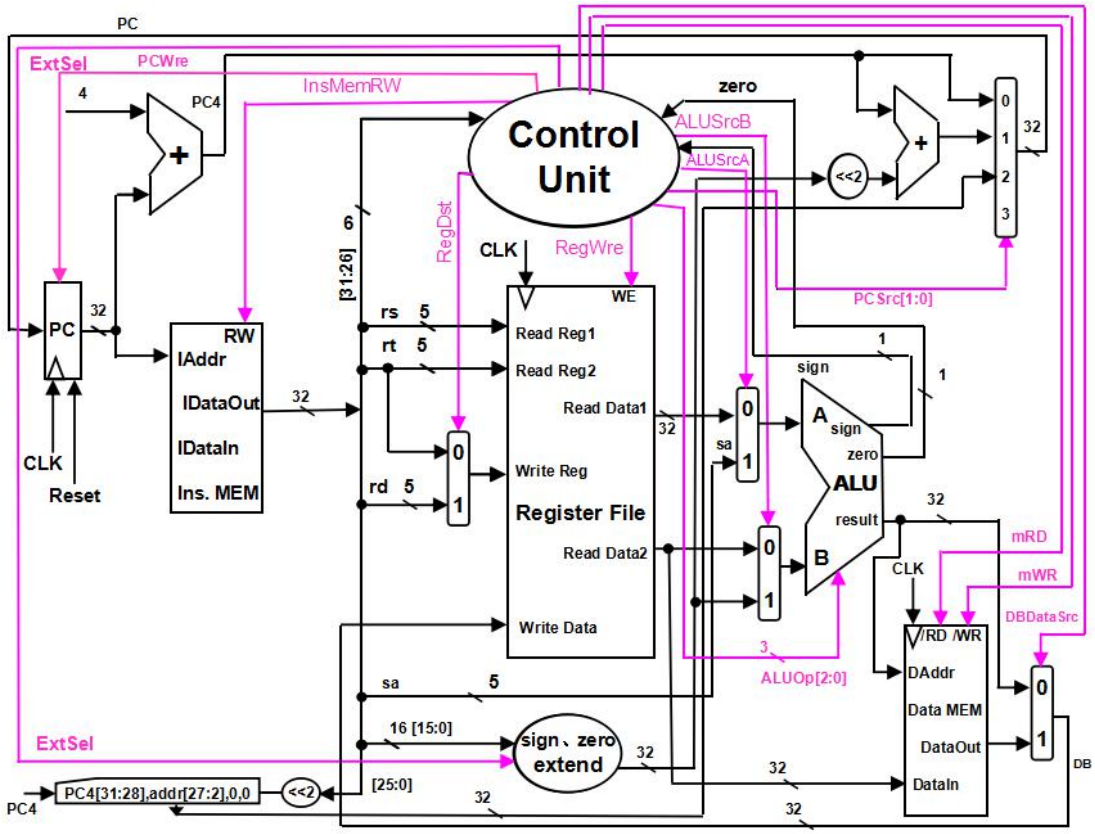


图 2　单周期 CPU 数据通路和控制线路图

图 2 是一个简单的基本上能够在单周期 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，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，先给出寄存器地址，读操作时，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，在时钟边沿触发将数据写入寄存器。图中控制信号作用如表 1 所示，表 2 是 ALU 运算功能表。

表 1 控制信号的作用

控制信号名	状态“0”	状态“1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指	来自移位数 sa，同时，进行

	令：add、sub、addi、or、and、ori、beq、bne、slti、sw、lw	(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、or、and、sll、beq、bne	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、slti、sll	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、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	输出高阻态	**读数据存储器，相关指令：lw**
mWR	无操作	**写数据存储器，相关指令：sw**
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addi、ori、lw、slti	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、sll
ExtSel	(zero-extend)**immediate**（0 扩展），相关指令：ori	(sign-extend)**immediate**（符号扩展），相关指令：addi、slti、sw、lw、beq、bne
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slti、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<－{(pc+4)[31:28],addr[27:2],2{0}}，相关指令：j； 11：未用	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

相关部件及引脚说明：

Instruction Memory：指令存储器，

Iaddr，指令存储器地址输入端口

IDataIn，指令存储器数据输入端口（指令代码输入端口）

IDataOut，指令存储器数据输出端口（指令代码输出端口）

RW，指令存储器读写控制信号，为 0 写，为 1 读

Data Memory：数据存储器，

Daddr，数据存储器地址输入端口

DataIn，数据存储器数据输入端口

DataOut，数据存储器数据输出端口

/RD，数据存储器读控制信号，为 0 读

/WR，数据存储器写控制信号，为 0 写

Register File：寄存器组

Read Reg1，rs 寄存器地址输入端口

Read Reg2，rt 寄存器地址输入端口

Write Reg，将数据写入的寄存器端口，其地址来源 rt 或 rd 字段

Write Data，写入寄存器的数据输入端口

Read Data1，rs 寄存器数据输出端口

Read Data2，rt 寄存器数据输出端口

WE，写使能信号，为 1 时，在时钟边沿触发写入

ALU： 算术**逻辑单元**

result，ALU 运算结果

zero，运算结果标志，结果为 0，则 zero=1；否则 zero=0

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 = B<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y= (A<B) ?1: 0	比较 A 与 B 不带符号
110	**Y=(((rega<regb) && (rega[31] == regb[31])) ||((rega[31] ==1 && regb[31] == 0))) ? 1:0**	比较 A 与 B 带符号
111	Y = A ⊕ B	异或

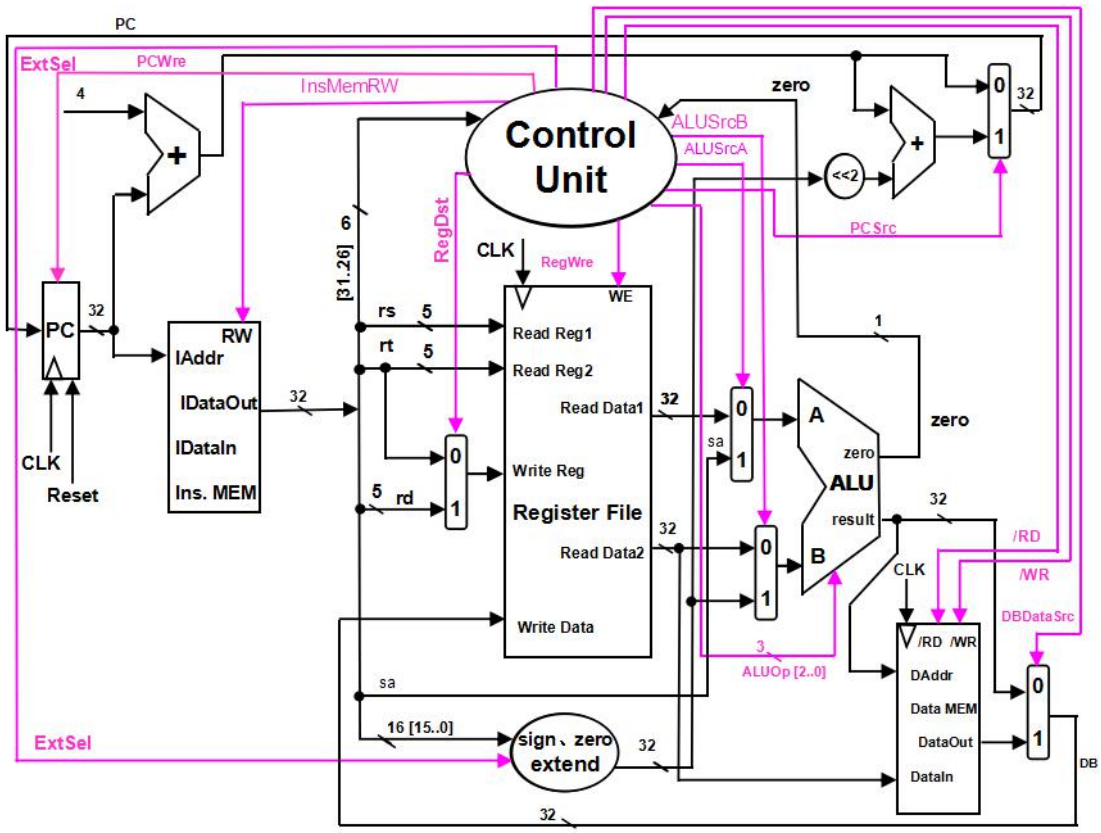
需要说明的是以上数据通路图是根据要实现的指令功能的要求画出来的，同时，还必须确定 ALU 的运算功能(当然，以上指令没有完全用到提供的 ALU 所有功能，但至少必须能实现以上指令功能操作)。从数据通路图上可以看出控制单元部分需要产生各种控制信号，当然，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出以上表 1，这样，从表 1 可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系表（留给学生完成），再根据关系表可以写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五. 实验过程与结果

.1. CPU原理结构图



2. 设计思路与方法

整个实验的核心便是上面的数据通路图。用模块化的思想方法，不难将上面的数据通路图进行分解，得到 7 个模块。

对于每一个模块，我们只关心如何从他的输入得到他的输出，并将其中的逻辑过程转换为代码。例如，对于立即数扩展模块，我们只关心如何从 16 位的立即数输入和 ExtSel 信号输入这两个条件，确定该模块的输出：32 位的扩展后的立即数。根据所学的知识，不难确定其中的逻辑。当 ExtSel 为 0 时，对 16 位的立即数进行零扩展；当 ExtSel 为 1 时，对 16 位的立即数进行符号扩展。于是，这个模块的内容，便是这个逻辑转换而成的代码。

分解后的模块相互独立，并不能驱动 CPU 运转。为此，还要设计一个 CPU 顶层模块，将这 7 个模块通过线，有条不紊地连接起来，形成流水线，提高吞吐量。同时，还可以暴露各种控制单元信号和线的值，便于调试和验证。

为进行测试，还需要设计一个测试模块。测试模块实例化 CPU，传入时钟信号和 Reset 信号，可以从这里观测各个信号的取值情况，绘制波形图。

- 控制信号与指令之间的关系表

	op	zero	PCWre	PCSrc	InsMemRW	RegWre	RegDst	ALUSrcA	ALUSrcB	ExtSel	RD	WR	DBDataSrc	ALUOp
add	000000	X	1	0	1	1	1	0	0	X	1	1	0	000

addi	000001	X	1	0	1	1	0	0	1	1	1	1	0	000
sub	000010	X	1	0	1	1	1	0	0	X	1	1	0	001
ori	010000	X	1	0	1	1	0	0	1	0	1	1	0	100
and	010001	X	1	0	1	1	1	0	0	X	1	1	0	101
or	010010	X	1	0	1	1	1	0	0	X	1	1	0	100
sll	011000	X	1	0	1	1	1	1	0	0	1	1	0	011
sw	100110	X	1	0	1	0	X	0	1	1	1	0	X	000
lw	100111	X	1	0	1	1	0	0	1	1	0	1	1	000
beq	110000	0	1	1	1	0	X	0	0	1	1	1	X	001
beq	110000	1	1	1	1	0	X	0	0	1	1	1	X	001
halt	001010	X	0	0	1	0	X	X	X	X	1	1	X	X

通过这个关系表，我们可以很清晰地看出，对于每条指令操作码，控制单元需要发出怎样的信号和怎样的 ALU 操作码。

- **各模块剖析（附关键代码与代码思路）**

  - PC 模块

在我的设计中,PC 模块的输入是时钟信号 CLK、重置信号 Reset、控制信号 PCWre、控制信号PCSrc、立即数immediate、j指令跳转地址addr、输出是PC的次态。

PC 的改变在时钟信号的上升沿进行与Reset的下降沿，故将 PC 的改变放在 always 块里进行。当 Reset 信号为0时候，对系统进行重置，将 address 重置为 0。

```
35      always @(posedge clk or negedge Reset)
36          begin
37              if (Reset == 0) begin
38                  address = 0;
```

当 Reset 不为零，PCWre 信号有效时，就要按照不同的PCWrc值对 PC 进行改变。一般 PCWre 信号都是有效的，这样 PC 才能不停地变化，不停的获取新的指令。而当停机时，PCWre 信号无效，输出的指令地址便不再改变。

PrcWre	PrcSrc	Reset	PC(次态)
x	x	0	0
0	x	1	不变
1	00	1	PC+4
1	01	1	PC+4+(sign-extend)immediate
1	10	1	{(PC+4)[31..28],address[27..2],0,0}
1	11	1	不变

在PCWre有效之后，根据上方表格进一步对PCSrc指令进行选择，从而选出对应的跳转方式，求出PC的次态。

```
case(PCSrc)
    2'b00: begin
        //普通+4求PC
        address = address + 4;
    end
    2'b01: begin
        //beq、bne类跳转指令求PC
        address = address + 4 + immediate*4;
    end
    2'b10: begin
        pc4 = address + 4;
        //由实验报告中提到的j跳转地址求值
        address = {pc4[31:28],addr[25:0],1'b0,1'b0};
    end
    default: begin
        address = address + 4;
    end
endcase
```

■ 指令存储器模块（InstructionMemory）

在我的设计中，该模块的输入是32位指令地址 PC，输出是输入的 PC 分段拆分之后数据集合。

```
24      input [31:0] pc,
25      output [5:0] opCode,
26      output [4:0] rs,
27      output [4:0] rt,
28      output [4:0] rd,
29      output [15:0] immediate,
30      output [5:0] sa,
31      output [25:0] addr
```

在指令存储器中，有一个长度为8位的储存器组（有 128 个储存器）Memory，其中每 4 个储存器储存一条指令，总共储存 17 条指令。

```
34      reg [7:0] memory[0:127];
35      reg [31:0] address;
36      reg [31:0] instruction;
```

先把指令txt中的指令读入memory中，随后再将memory中的指令按位的高低拼凑起来放入临时变量instruction中用于系统读取与返回，指令地址的高位放在instruction

的高位，指令地址的地位放在instruction的低位。

```
37    initial begin
38        $readmemb("C:/Users/dell/Desktop/CPU/instructions.txt", memory);
39        instruction = 0;
40    end
41
42    always @(pc) begin
43        address = pc[6:2] << 2;
44        instruction = (memory[address]<<24)+
45        (memory[address+1]<<16)+
46        (memory[address+2]<<8)+
47        memory[address+3];
48    end
```

在完成这一步之后，接着把存放在instruction中的指令，按照I,J，R型指令的规定，按照顺序拆分，依次赋给模块开始申请的需要输出的值。

```
50     assign opCode = instruction[31:26];
51     assign rs = instruction[25:21];
52     assign rt = instruction[20:16];
53     assign rd = instruction[15:11];
54     assign immediate = instruction[15:0];
55     assign sa = instruction[10:6];
56     assign addr = instruction[25:0];
```

■ 控制单元模块（CtrlUnit）

控制单元，根据指令发出针对其余模块的控制信号，以使得其余模块按照指令正常工作。控制信号与指令的真值表如下：

OpCode	指令	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	InsMemRW
000000	add	1	0	0	0	1	1
000001	addi	1	0	1	0	1	1
000010	sub	1	0	0	0	1	1
010000	ori	1	0	1	0	1	1
010001	and	1	0	0	0	1	1
010010	or	1	0	0	0	1	1
011000	sll	1	1	0	0	1	1
011011	slti	1	0	1	0	1	1
100110	sw	1	0	1	x	0	1
100111	lw	1	0	1	1	1	1
110000	beq	1	0	0	x	0	1
110001	bne	1	0	0	x	0	1
111000	j	1	x	x	x	0	1
111111	halt	0	x	x	x	0	1

OpCode	指令	mRD	mWR	RegDst	ExtSel	PCSrc	ALUOp
000000	add	0	0	1	x	00	000
000001	addi	0	0	0	1	00	000
000010	sub	0	0	1	x	00	001
010000	ori	0	0	0	0	00	011
010001	and	0	0	1	x	00	100
010010	or	0	0	1	x	00	011
011000	sll	0	0	1	x	00	010
011011	slti	0	0	0	1	00	110
100110	sw	0	1	x	1	00	000
100111	lw	1	0	0	1	00	000
110000	beq	0	0	x	1	00(zero=0); 01(zero=1)	001
110001	bne	0	0	x	1	00(zero=1); 01(zero=0)	001
111000	j	0	0	x	x	10	010
111111	halt	0	0	x	x	xx	xxx

在我的设计中，控制单元模块输入部分是来自指令存储器的 Op ，来自ALU的 zero ，输出是PCWre、ALUSrcA、ALUSrcB、DBDataSrc、RegWre、RD、WR、RegDst、ExtSel、PCSrc（2 位）、ALUOP（3 位）

```
23  module ControlUnit(
24      input [5:0] opCode,
25      input zero, output Reset, output PCWre,
26      output ALUSrcA,output ALUSrcB,
27      output DBDataSrc,output RegWre,output InsMemRW,
28      output mRD,output mWR,output RegDst,
29      output ExtSel, output [1:0] PCSrc, output [2:0] ALUOp
30      );
```

对这些部分的输出数据赋值的代码，由于太过冗长，就不再在这里截图。

■ 寄存器组模块（RegisterFile）

这个模块输入部分是：clk ， RegWre,RegDst,opCode,rs,rt,rd,immediate, DBDataSrc ， dataFromALU ， dataFromRW;

输出是 ReadData1，ReadData2;

这个模块用于模拟 MIPS 架构中的 32 个寄存器，需要注意的一点是在写入数据时要避免访问 0 号寄存器。读取寄存器数据时，无需时钟信号，根据传入 RegWre,RegDst 等信号，选取不同的行为模式，完成对寄存器组中的申请的需要赋值的数据的赋值，进而实现我们在输出波形上可以看到读操作数的功能。

```
39      wire [4:0] WriteReg;
40      wire [31:0] WriteData;
41
42      assign WriteReg = RegDst ? rd : rt;
43      assign WriteData = DBDataSrc ? dataFromRW : dataFromALU;
44
45      reg [31:0] register[0:31];
46      integer i;
47      initial begin
48          for (i = 0; i < 32; i = i+1) register[i] <= 0;
49      end
50
51      assign ReadData1 = (opCode == 6'b011000) ? immediate[10:6] : register[rs];
52      assign ReadData2 = register[rt];
```

然而当我们往寄存器写数据时候，就要在时钟的下降沿完成这项工作。

```
55      always @(negedge clk) begin
56          if (WriteReg && WriteData) register[WriteReg] = WriteData;
57      end
```

■ 符号位或零扩展模块（SignZeroExtend）

根据拓展选择信号，选择拓展方式：符号位拓展或零拓展，并对 16 位立即数进行拓展，输出 32 位数。其中，拓展选择信号为 0 时，进行零拓展，为 1 时，进行符号位拓展。

这个模块的输入是：16 位立即数，信号 ExtSel
输出是：32 位数据 Out

```
module signZeroExtend(
    input [15:0] immediate,
    input ExtSel,
    output [31:0] out
);
assign out[15:0] = immediate;
//通过信号选择决定扩展方式
assign out[31:16] = ExtSel ? (immediate[15] ? 16'hffff :
16'h0000) : 16'h0000;
endmodule
```

■ 算逻运算单元模块（ALU）

算术逻辑运算单元，根据控制信号的不同，选择对输入操作数进行不同的算术或逻辑运算，得到结果。算术功能包括加，减和移位，输出包含结果与符号位；逻辑功能包括或，与，比较和异或，输出包含结果与标志位。具体功能与控制信号对应如下表：

ALUOp[2..0]	功能	描述
000	Y = A + B	加
001	Y = A - B	减
010	Y = B<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y= (A<B) ?1: 0	比较 A 与 B 不带符号
110	Y=(((rega<regb) && (rega[31] == regb[31])) ||((rega[31] ==1 && regb[31] == 0))) ? 1:0	比较 A 与 B 带符号
111	Y = A ⊕ B	异或

这个模块的输入是：两个 32 位操作数， 32 位扩展数字，偏移量，与 ALUop 选择数据，ALUSrcA 与 ALUSrcB

输出是：32 位的运算结果 result 与运算结果是否为 0 的判断 zero

在ALU模块的always循环中，通过判断ALUop中的数据来选择实现不同的逻辑功能。

```
module ALU(
    input [31:0] ReadData1,
    input [31:0] ReadData2,
    input [31:0] extend,
    input [5:0] sa,
    input [2:0] ALUOp,
    input ALUSrcA,
    input ALUSrcB,
    output reg [31:0] result,
    output reg zero
);
wire [31:0] A;
wire [31:0] B;
//完成取数操作
assign A = ALUSrcA ? sa : ReadData1;
assign B = ALUSrcB ? extend : ReadData2;
//实现 ALU 功能
always @(ReadData1 or ReadData2 or extend or ALUSrcA or ALUSrcB
or ALUOp or A or B)
    begin
        case(ALUOp)
            3'b000: begin
                result = A + B;
                zero = (result == 0) ? 1 : 0;
            end
            3'b001: begin
                result = A - B;
                zero = (result == 0) ? 1 : 0;
            end
            3'b010: begin
                result = B << A;
                zero = (result == 0) ? 1 : 0;
            end
            3'b011: begin
                result = A | B;
                zero = (result == 0) ? 1 : 0;
            end
            3'b100: begin
                result = A & B;
                zero = (result == 0) ? 1 : 0;
            end
```

```
            3'b101: begin
                result = (A < B) ? 1 : 0;
                zero = (result == 0) ? 1 : 0;
            end
                3'b110: begin
                result = (((A < B) && (A[31] == B[31])) ||((A[31]
            == 1 && B[31] == 0))) ? 1 : 0;
                zero = (result == 0) ? 1 : 0;
            end
            3'B111: begin
                result = A ^~ B;
                zero = (result == 0) ? 1 : 0;
            end
        endcase
    end
endmodule
```

■ 数据存储器模块（DataMemory）

这个模块的输入：时钟信号CLK，nRD，nWR，32位的DAddr，32位的DataIn

输出：DataOut

数据存储器中，采用 8 位一字节，大端模式模拟内存。根据图中真值表，当WR信号为 0 时，RD 信号为 1 时，进行写数据操作。当WR信号为 1 时，RD信号为 0 时进行读操作。

RD	WR	DataOut	功能
0	1	Memory[DAddr]	读数据，DataOut=Memory[DAddr]
1	0	1	写数据，Memory[DAddr]=DataIn
1	1	1	无

```
module DataMemory(
    input clk,
    input nRD,
    input nWR,
    input [31:0] DAddr,
    input [31:0] DataIn,
    output reg[31:0] DataOut
);
//模拟内存
```

```
//申请 128 个 8 位存储单元
reg [7:0] memory[0:127];
reg [31:0] address;
//完成读操作
always @(nRD) begin
    if (nRD == 0) begin
        address = (DAddr << 2);
        DataOut = (memory[address]<<24)+
        (memory[address+1]<<16)+
        (memory[address+2]<<8)+
        memory[address+3];
    end
end
integer i;
initial begin
for (i = 0; i < 128; i = i+1) memory[i] <= 0;
end
// 完成写操作
always @(negedge clk)
begin
    if (nWR == 0) begin
        address = DAddr << 2;
        memory[address] = DataIn[31:24];
        memory[address+1] = DataIn[23:16];
        memory[address+2] = DataIn[15:8];
        memory[address+3] = DataIn[7:0];
    end
end
endmodule
```

以上就是我设计的CPU的7个组成模块，接下来对自己的组合逻辑进行测试，测试指令为老师给的17条32位待测指令。

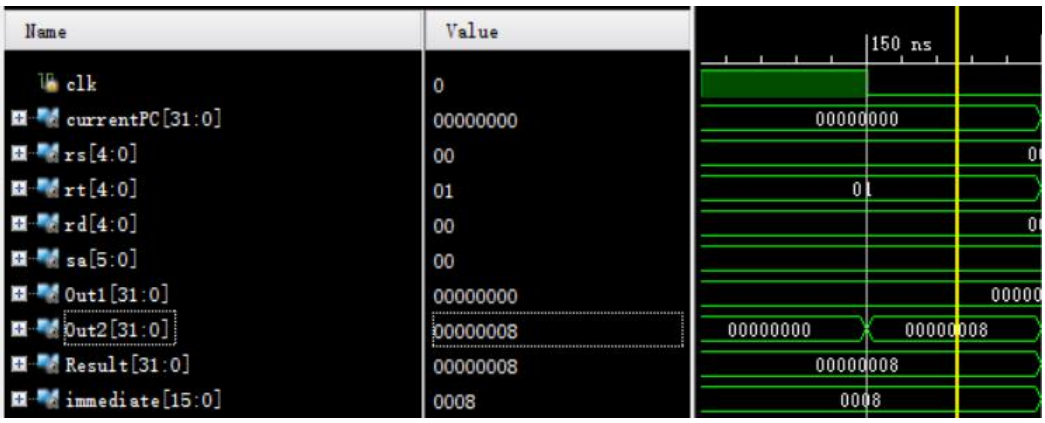
3.指令测试

- 待测数据

将老师所给的待测指令按顺序提取出，先转换成32位二进制指令，随后再加上注释方便自己进行debug操作。

```
//0 addi $1, $0, 8  ($1 = 8                    = 00001000)
00000100 00000001 00000000 00001000
//4 ori  $2, $0, 2  ($2 = 2                    = 00000010)
01000000 00000010 00000000 00000010
//8 add  $3, $2, $1 ($3 = 10                   = 00001010)
00000000 01000001 00011000 00000000
//C sub  $5, $3, $2 ($5 = 8                    = 00001000)
00001000 01100010 00101000 00000000
//10 and  $4, $5, $2 ($4 = 0                    = 00000000)
01000100 10100010 00100000 00000000
//14 or   $8, $4, $2 ($8 = 2                    = 00000010)
01001000 10000010 01000000 00000000
//18 sll  $8, $8, 1  ($8 = 4                    = 00000100)
01100000 00001000 01000000 01000000
//1C bne  $8, $1, -2 (first time $8 = 4  second time $8 = 8 )
11000101 00000001 11111111 11111110
//20 slti $6, $2, 8  ($6 = 1                    = 00000001)
01101100 01000110 00000000 00001000
//24 slti $7, $6, 0  ($7 = 0                    = 00000000)
01101100 11000111 00000000 00000000
//28 addi $7, $7, 8  ($7 = 8                    = 00001000)
00000100 11100111 00000000 00001000
//2C beq  $7, $1,-2 (=,转28)
11000000 00100111 11111111 11111110
//30 sw   $2,4($1)
10011000 00100010 00000000 00000100
//34 lw   $9,4($1)
10011100 00101001 00000000 00000100
//38 j  0x00000040
11100000 00000000 00000000 00010000
//3C addi $10,$0,10
00000100 00001010 00000000 00001010
//40 halt
11111100 00000000 00000000 00000000
```

■ 指令1： addi $1, $0, 8



解释：

① currentPC 代表当前指令的地址，对应的 00000000 显示当前地址的值

② rs 为 0 号寄存器,对应的值 Out1 为 0 ;rt 为 1 号寄存器,对应的值 Out2

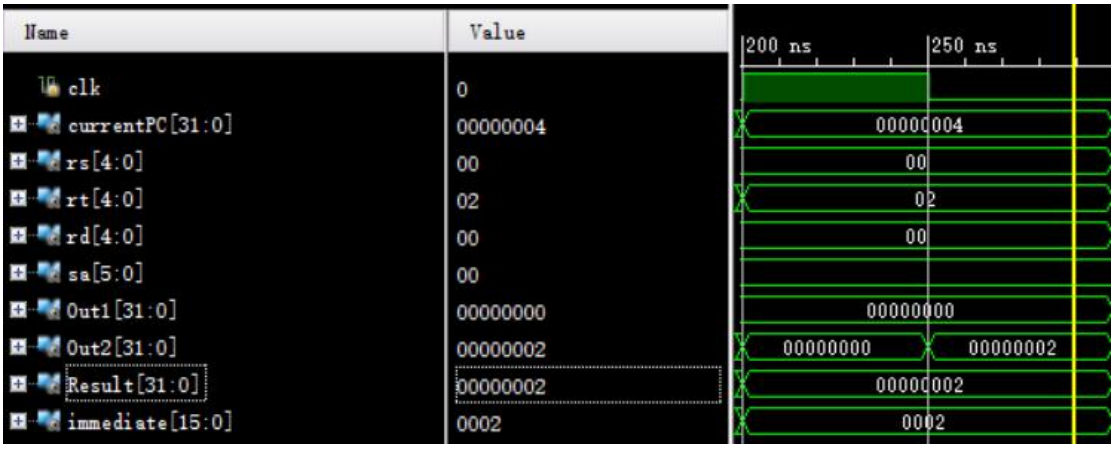
在时钟下降沿时由 0 变为 8 ；rd ， sa在这里都是 0 ，这一条指令没有用到这两个数据段

③ result 代表运算结果为 8

④ immediate 代表立即数为 8

⑤ 经过分析，可以证明指令执行正确($1 = $0 + 8 = 0 + 8 = 8)。

■ 指令2：ori $2, $0, 2



解释：

① currentPC 代表当前指令的地址，对应的 00000004 显示当前地址的值

② rs 为 0 号寄存器,对应的值 Out1 为 0 ;rt 为 2 号寄存器,对应的值 Out2

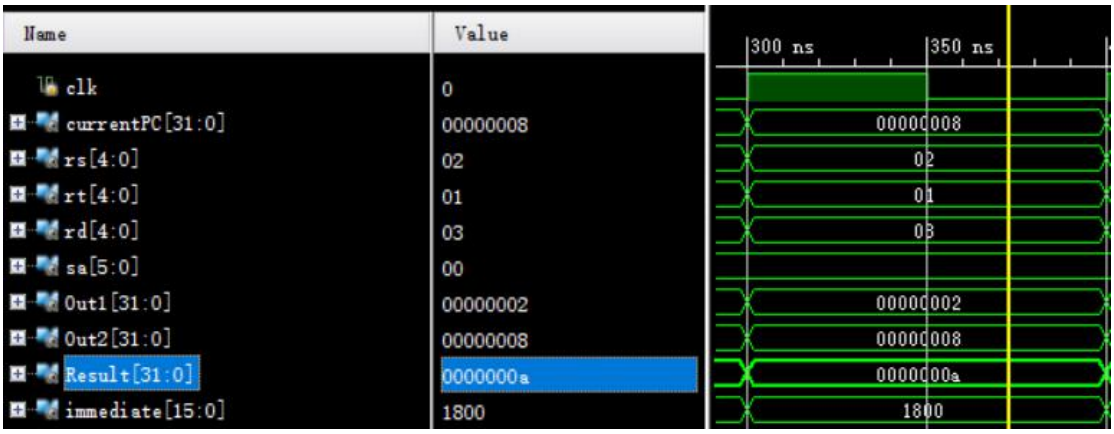
在时钟下降沿时由 0 变为 2 ；rd ， sa在这里都是 0 ，这一条指令没有用到这两个数据段

③ result 代表运算结果为 2

④ immediate 代表立即数为 2

⑤ 经过分析，可以证明指令执行正确($2 = $0 | 2 = 0 | 2 = 0000 | 0010 = 2)。

■ 指令3：add $3, $2, $1



解释：

① currentPC 代表当前指令的地址，对应的 00000008 显示当前地址的值

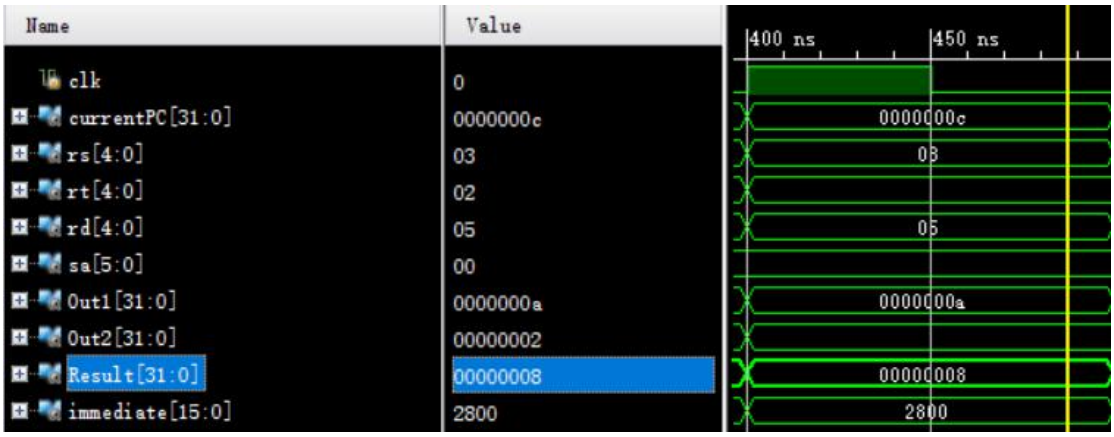
② rs 为 2 号寄存器，对应的值 Out1 为 2 ；rt 为 1 号寄存器，对应寄存器的值为 8 ；rd 为 3 号寄存器，对应的值result 为 10（也即16进制的a）；sa在这里是 0 ，这一条指令没有用到这个数据段；

③ result 代表运算结果为 10

④ immediate在这里是1800，这是由于在提取指令时候，把这个段的指令提取了出来，但是并没有用到。

⑤ 经过分析，可以证明指令执行正确($3 = $2 + $1 = 2 + 8 = 10)。

■ 指令4：sub $5, $3, $2



解释：

① currentPC 代表当前指令的地址，对应的 0000000c 显示当前地址的值

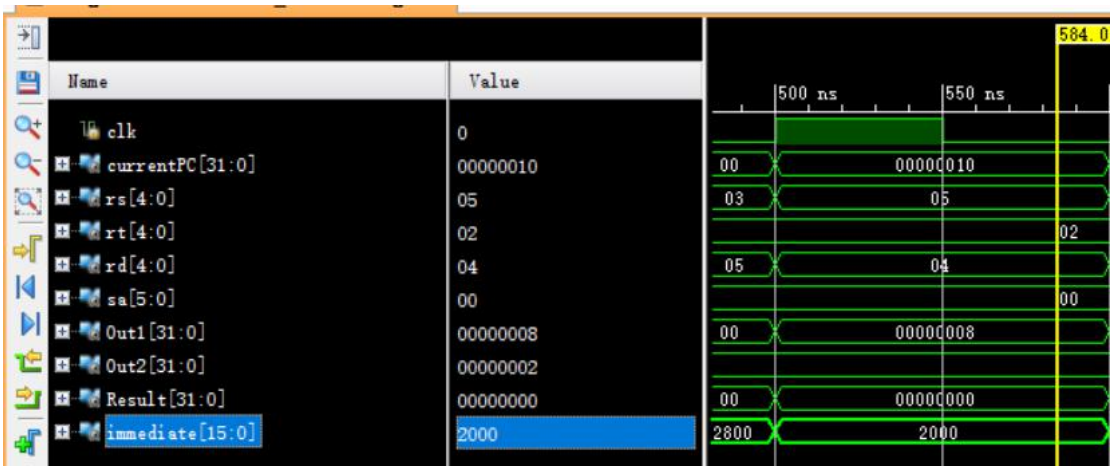
② rs 为 3 号寄存器，对应的值 Out1 为 10（也即十六进制的a）；rt 为 2 号寄存器，对应寄存器的值为 2 ；rd 为 5 号寄存器，对应的值result 为 8 ；sa在这里是 0 ，这一条指令没有用到这个数据段；

③ result 代表运算结果为 8

④ immediate在这里是2800，这是由于在提取指令时候，把这个段的指令提取了出来，但是并没有用到。

⑤ 经过分析，可以证明指令执行正确(sub $5, $3, $2 = 10 - 2 = 8)。

■ 指令5：and $4, $5, $2



解释：

① currentPC 代表当前指令的地址，对应的 00000010 显示当前地址的值

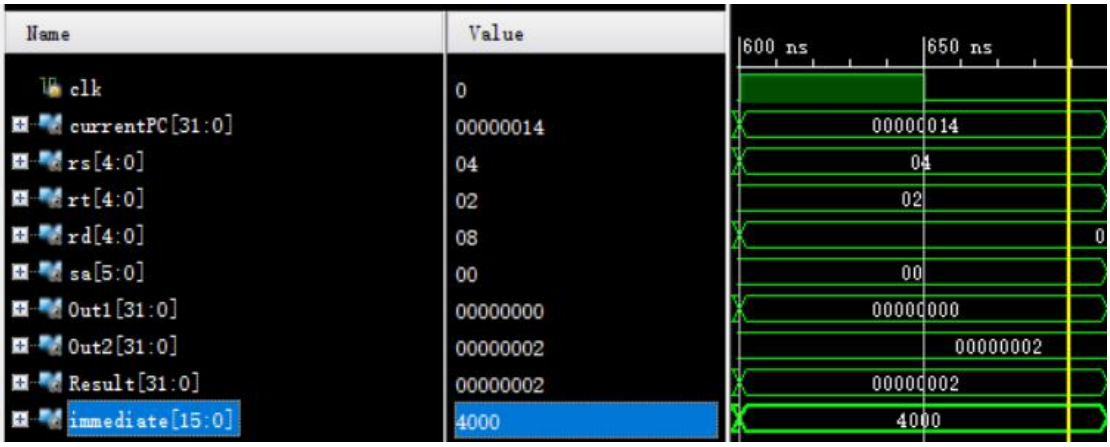
② rs 为 5 号寄存器，对应的值 Out1 为 8（也即十六进制的a）；rt 为 2 号寄存器，对应寄存器的值为 2；rd 为 4 号寄存器，对应的值result 为 0；sa在这里是 0，这一条指令没有用到这个数据段；

③ result 代表运算结果为 0

④ immediate在这里是2000，这是由于在提取指令时候，把这个段的指令提取了出来，但是并没有用到。

⑤ 经过分析，可以证明指令执行正确(and $4, $5, $2 = 8 & 2 = 1000 & 0010 = 0000)。

■ 指令6：or $8, $4, $2



解释：

① currentPC 代表当前指令的地址，对应的 00000014 显示当前地址的值

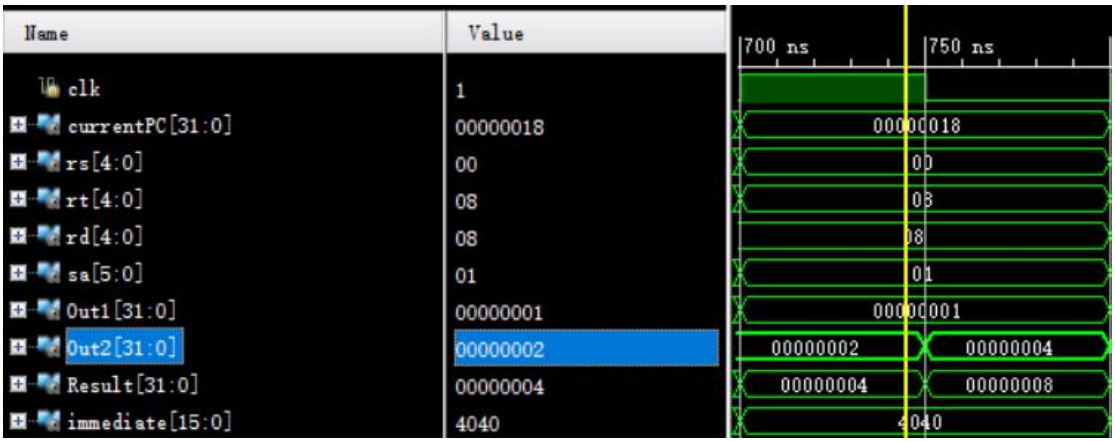
② rs 为 4 号寄存器，对应的值 Out1 为 0（也即十六进制的a）；rt 为 2 号寄存器，对应寄存器的值为 2；rd 为 8 号寄存器，对应的值result 为 2；sa在这里是 0，这一条指令没有用到这个数据段；

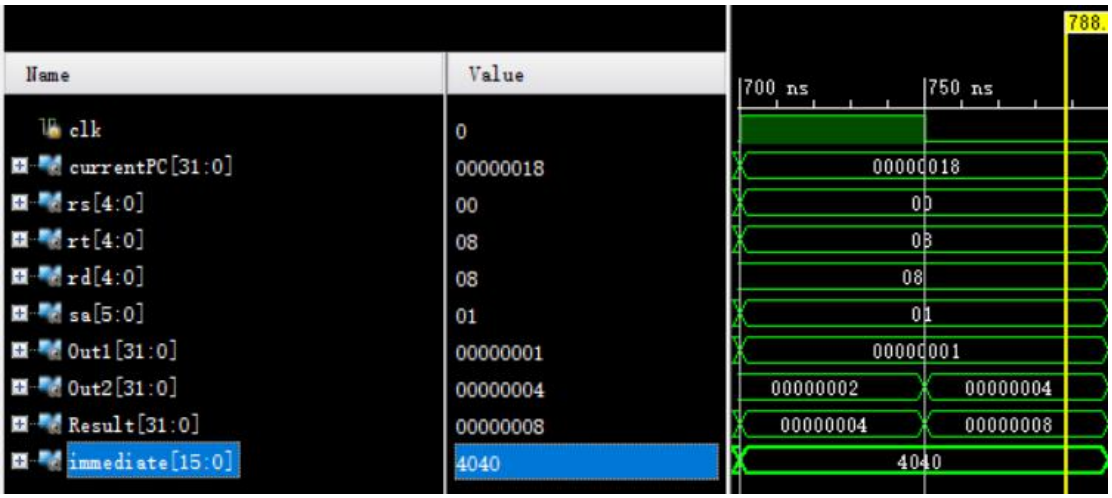
③ result 代表运算结果为 2

④ immediate在这里是4000，这是由于在提取指令时候，把这个段的指令提取了出来，但是并没有用到。

⑤ 经过分析，可以证明指令执行正确(or $4, $5, $2 = 0 | 2 = 0000 | 0010 = 0010)。

■ 指令7：sll $8, $8, 1





解释：

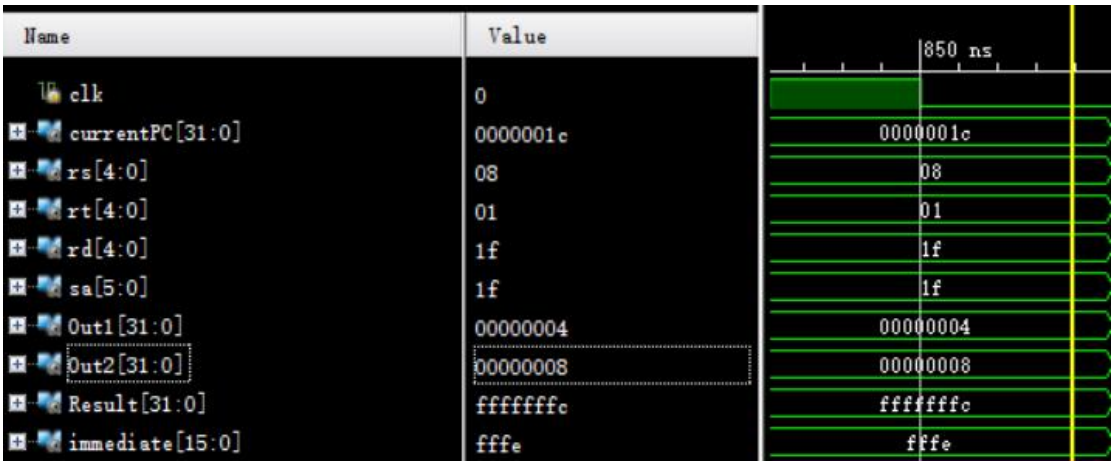
① currentPC 代表当前指令的地址，对应的 00000018 显示当前地址的值

② rs 为 0 号寄存器 ；rt 为 8 号寄存器，对应寄存器的值为 2 ，在时钟下降沿被写为 4 ，虽然第二幅图中result，显示为 8 ，但是并没有被写入，所以$8仍为 4 ，从下一条指令也可以证明 ；sa为 01 ，代表偏离量为 1 。

③ immediate在这里是4040，这是由于在提取指令时候，把这个段的指令提取了出来，但是并没有用到。

④ 经过分析，可以证明指令执行正确(sll $8, $8, 1 = 2 << 1 = 4)。

■ 指令8：bne $8, $1, -2

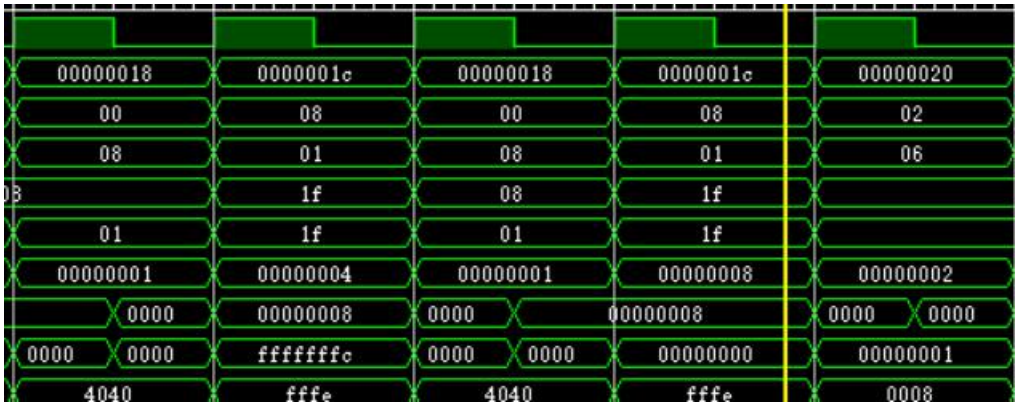


解释：

① currentPC 代表当前指令的地址，对应的 0000001c 显示当前地址的值

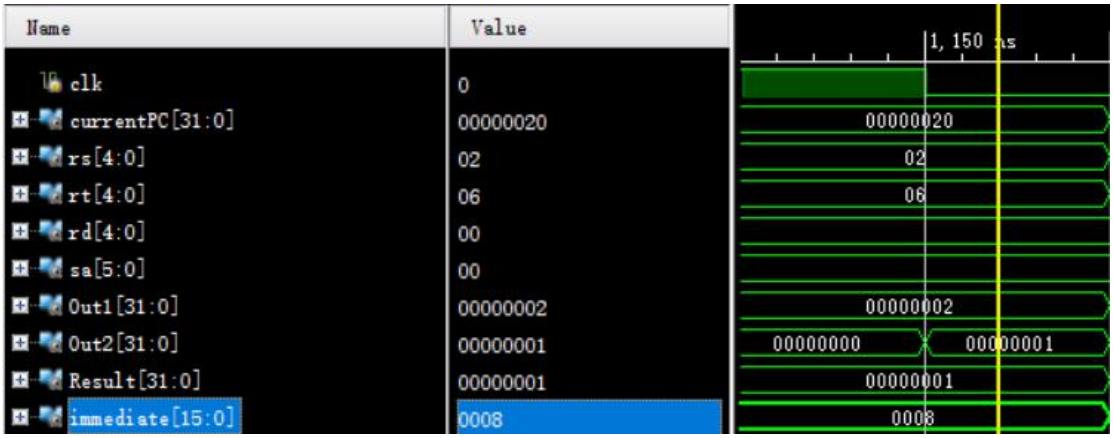
② rs 为 8 号寄存器 ,对应寄存器的值为 4 ；rt 为 1 号寄存器，对应寄存器的值为 8 ，rd与sa这里没有用到，所以值的大小可以忽略。

③ immediate在这里是fffe，这是由于在提取指令时候，把这个段的指令提取了出来，但是并没有用到。



④ 分析(bne $8, $1, -2 ； $8 = 4 ≠ $1 = 8 ， 所以PC - 2 ，也即跳转回 sll $8, $8, 1这条指令所在的地址，从下图可以看出，CPU重新返回到了地址18，然后运行sll $8, $8, 1 这条指令，在时钟下降沿将 $8 的值改为 8 ，然后再次回到 1c ，这时候 $8 中存的内容为 8 ，此时 $8 与 $1相等，所以bne $8, $1, -2 这条指令不成立，所以下一次的地址变为 20 ）

■ 指令9：slti $6, $2, 8



解释：

① currentPC 代表当前指令的地址，对应的 00000020 显示当前地址的值

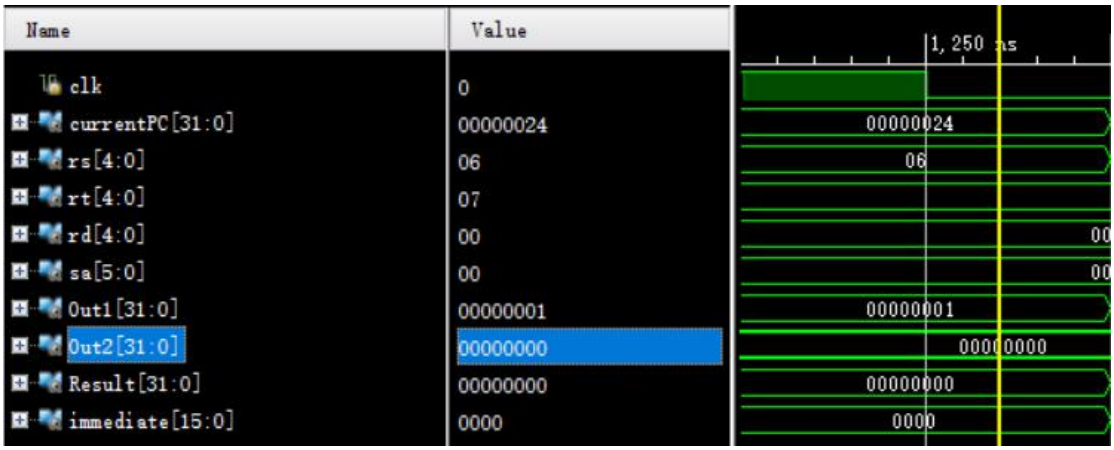
② rs 为 2 号寄存器，对应的值 Out1 为 2；rt 为 6 号寄存器，从上面的指令中，在时钟下降沿，6号寄存器的值从0改为 1

③ result 代表运算结果为 1

④ immediate在这里是0008，代表待比较立即数8

⑤ 经过分析，可以证明指令执行正确(slti $6, $2, 8 $6 = 2 < 8 = 1)。

■ 指令10：slti $7, $6, 0



解释：

① currentPC 代表当前指令的地址，对应的 00000024 显示当前地址的值

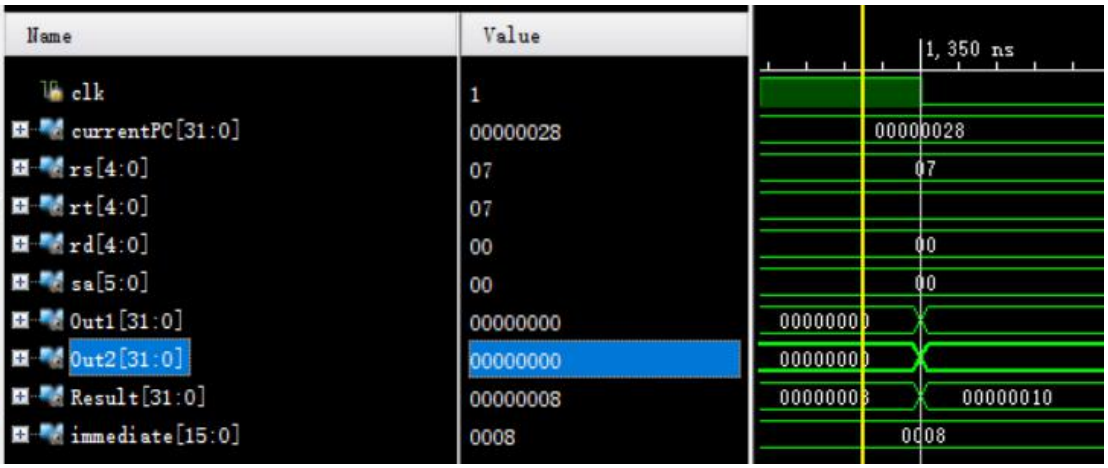
② rs 为 6 号寄存器，对应的值 Out1 为 1 ；rt 为 7 号寄存器，从上面的指令中，我们可以得知 $6 的值为 0；由于这里 $6 = 1 > 0 ,所以$7 = 0 不发生改变

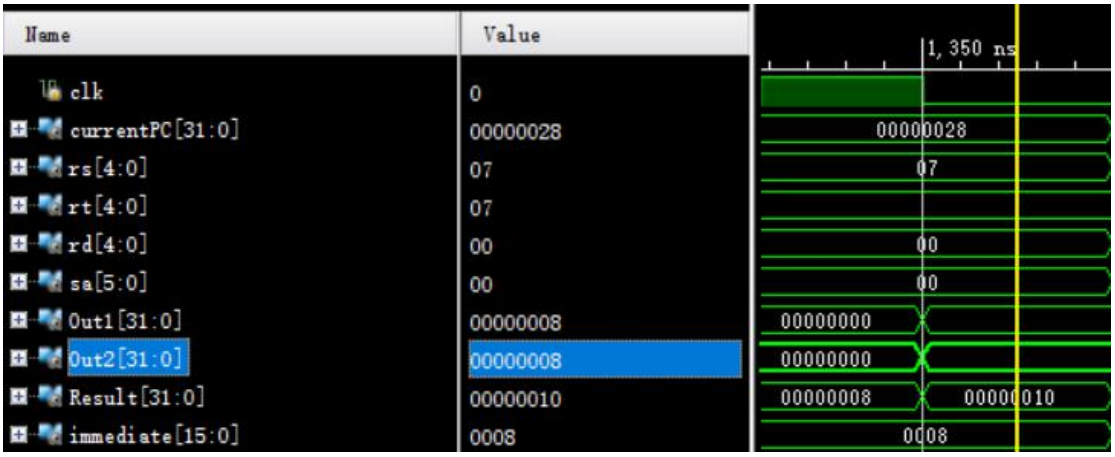
③ result 代表运算结果为 0

④ immediate在这里是0000，代表立即数0

⑤ 经过分析，可以证明指令执行正确(slti $7, $6, 0 $7 = 1 < 0 = 0)。

■ 指令11：$7, $7, 8





解释：

① currentPC 代表当前指令的地址，对应的 00000028 显示当前地址的值

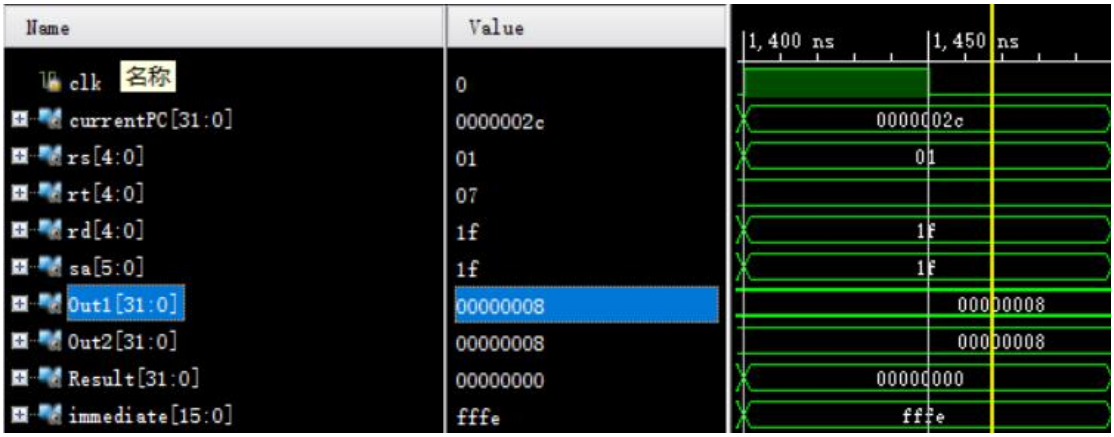
② rs 为 7 号寄存器,对应的值 Out1 为 0 ;rt 为 7 号寄存器,对应值Out2 为 0 ; sa 与 rd 并没有被用到 ; Result在时钟下降沿被写入到 $7 中 ，从第二个图中我们可以看出。

③ result 代表运算结果为 8

④ immediate在这里是0008，代表立即数8

⑤ 经过分析，可以证明指令执行正确($7, $7, 8 $7 = 0 + 8 = 8)。

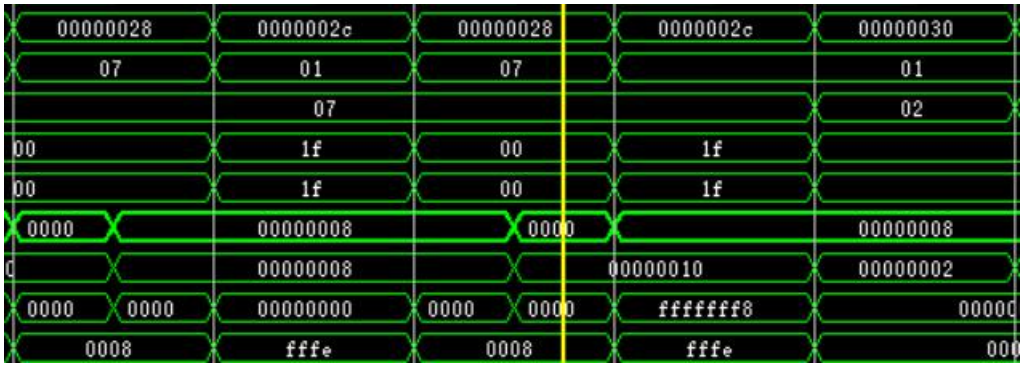
■ 指令12：beq $7, $1,-2



解释：

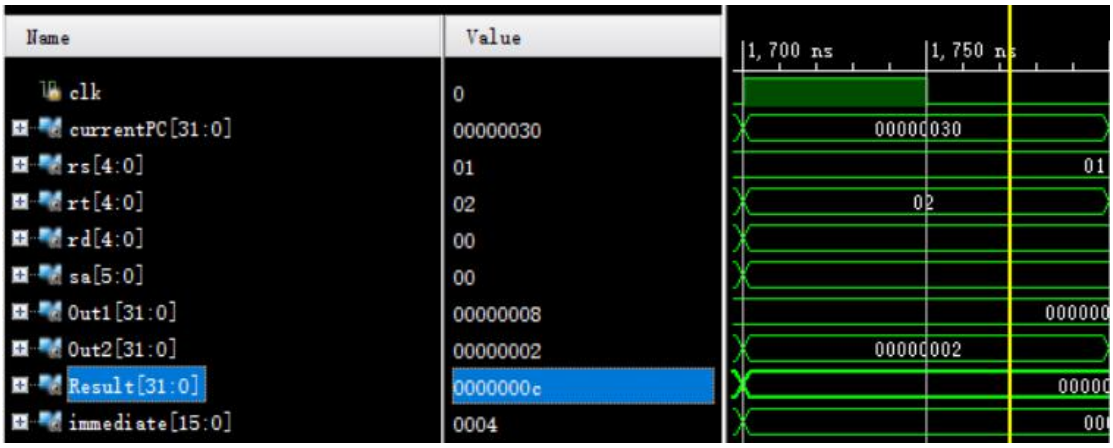
① currentPC 代表当前指令的地址，对应的 0000002c 显示当前地址的值

② rs 为 1 号寄存器，对应的值 Out1 为 8 ；rt 为 7 号寄存器，对应值Out2 为 8 ； sa 与 rd 并没有被用到



③ 分析(beq $7, $1, -2 ； 由于$7 = 8 == $1 = 8 ，所以这条指令成立，所以PC - 2 ，也即跳转回addi $7, $7, 8这条指令所在的地址，从下图可以看出，CPU重新返回到了地址28，然后运行addi $7, $7, 8 这条指令，在时钟下降沿将 $7 的值改为 16 ，然后再次回到 1c ，这时候 $7 中存的内容为 16 ，此时 $7 与 $1不相相等，所以beq $7, $1,-2这条指令不成立，所以下一次的地址变为 30)

■ 指令13：sw $2,4($1)

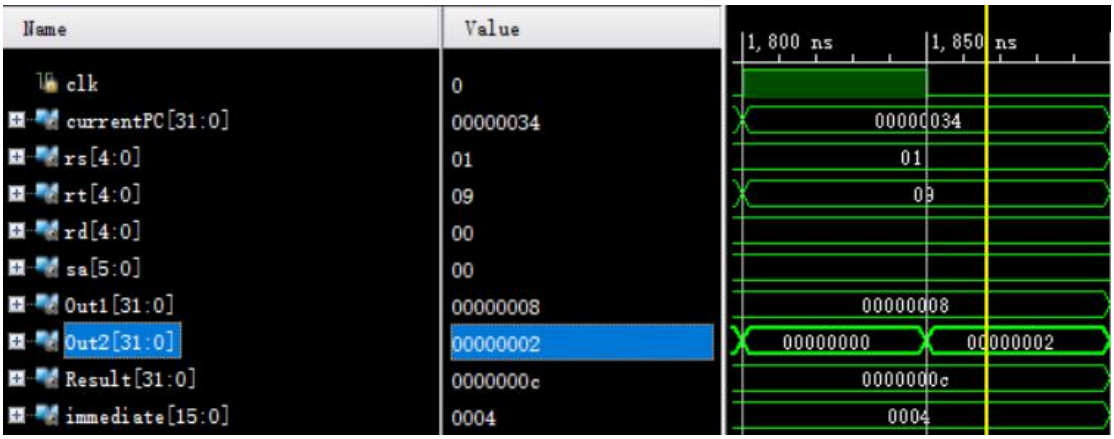


解释：

① currentPC 代表当前指令的地址，对应的 00000030 显示当前地址的值

② 立即数immediate为 4 。Rs 为 1 号寄存器，对应值Out1为 8 ；Rt为 2寄存器，对应值Out2为2。ALU的运算结果为12。由于采用8位一字节的大端存储模式，所以在12，13，14，15号内存中写入数据 2 (memory[$1 + (sign-extend)immediate] = $2 => memory[8 + 4] = 2 => memory[12:15] = 2)。

■ 指令14：lw $9,4($1)

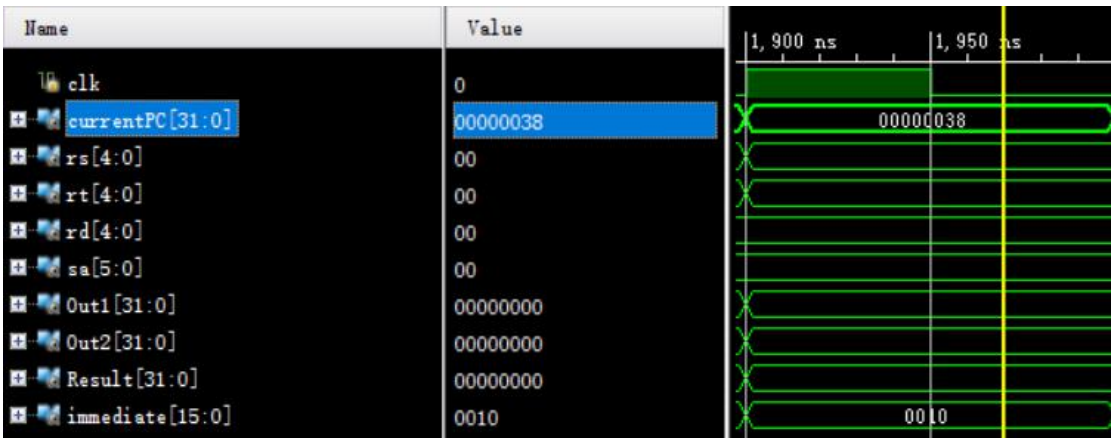


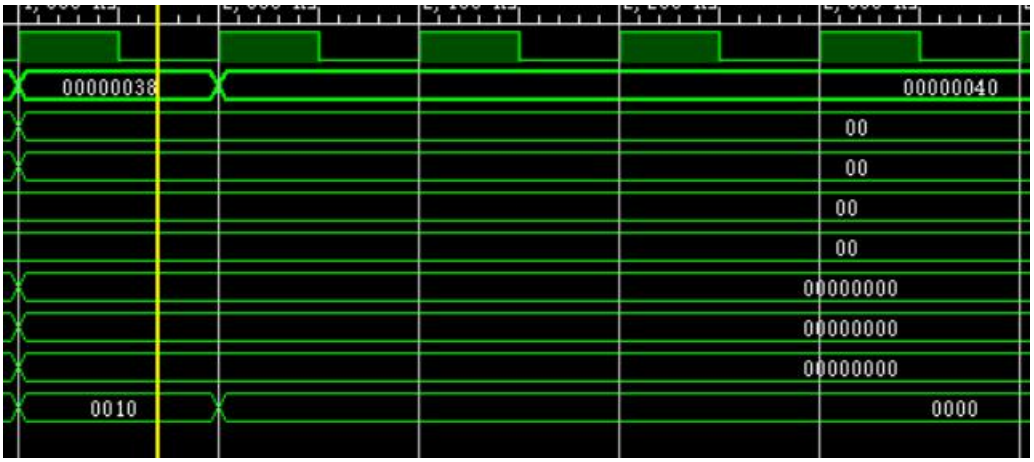
解释：

① currentPC 代表当前指令的地址，对应的 00000034 显示当前地址的值

②立即数immediate 为 4 。Rs 为1号寄存器，对应值Out1 为8；rt 为 9 号寄存器，对应值Out 2在时钟下降沿由 0 变为 2。ALU 的运算结果为 12。由于采用 8 位一字节的大端模式存储数据，所以在12，13，14，15 号内存读出数据 2，写入 rt，9号寄存器，指令执行正确($9 = memory[$1 + (sign-extend)immediate] => $9 = memory[8 + 4] => $9 = memory[12:15] = 2)。

■ 指令15：j 0x00000040





解释：

① currentPC 代表当前指令的地址，对应的 00000038 显示当前地址的值

② 这里由于是跳转指令，所以直接观察下一条指令的地址，可以发现是00000040，所以运行过程是正确的

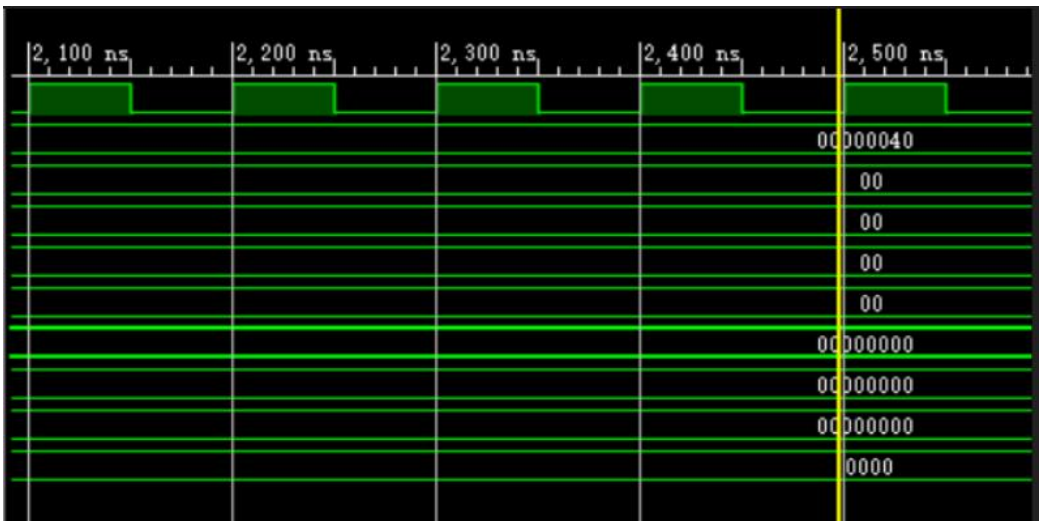
③ 由于是j型跳转指令，所以并不涉及寄存器的使用，所以这里的数据值都为0.

■ 指令16：addi $10,$0,10

由于上一条指令直接跳到0x00000040地址，所以0x0000003c地址的指令 addi $10,$0,10被跳过

■ 指令17：halt

由于是停机指令，所以地址不再发生变化，如图：



- **烧板测试（部分指令）**

注：图片按照顺时针方向，依次为：
PC：nextPC
rs：RsData
rt：RTData，
ALU的运算结果Result

- 指令1： addi $1, $0, 8

- 指令3： add $3, $2, $1

- 指令8： bne $8,$1,-2 (≠,转18)

- 指令13： sw $2,4($1)

- 指令14：lw $9,4($1)

六. 实验心得

先讲讲这次完成这次CPU设计的体会吧。

在这次实验中，模块化思想被得到充分的运用，这也就使得我能够专注于某个模块的抽象逻辑，不必关心太多，同时模块化的思想也体现了计组理论课中的一句话，简单源于规整，只有把层次理清楚了，模块划分清楚了，那么接下来的工作自然就会变得有条理一些，那么自己在动手设计时候，设计过程会变得相对简单一些。

接下来说说问题。

碰到的第一个问题是，读完实验要求，感到十分迷茫，只知道老师布置去做单周期CPU，但是如何下手去做，是没有一点思路的。随后在翻阅资料时候就接触到那张CPU结构图，第一次看到那张图时候，看着密密麻麻的连线以及奇奇怪怪的标注，自己完全没有办法捕捉到图里面的有效信息。后来，在第二周的实验课上，我与老师进行了沟通讲出了心中的疑问并寻求了帮助，老师给出的意见是将CPU结构图进行模块化分解，然后逐一完成每一块的设计，

再使用顶层模块将每一块拼接起来。听取了老师的建议之后，我开始着手将原理结构图分解，在多方面的查阅资料之后，最后定下自己设计的基调——将整个CPU一分为七，包含ALU,PC,CU,指令存储器，寄存器组，零或符号位扩展，数据存储器，最后再设计一个顶层模块，将分开的模块串联。

第二个碰到的问题，就是在仿真过程中如何解决 reg 格式和 wire 格式的赋值问题。reg 格式不能用于 assign 的赋值， assign 只能对 Wire 类型的变量使用，而在 initial 段和 always 段进行赋值操作时又不允许使用 assign ，只能通过 reg 格式才能赋值。所以要想仿真时通过编译，就需要注意区分 <= 和 assign 哪种赋值方式具有异步的特征。

第三个碰到的问题是，对使用if-else语句的使用。我们在编写高级语言程序时候，往往是如果有一个if就可以完成任务，不必去考虑一定要加上else，不会出现有if没有else配对程序就会运行出错的情况，但是在verilog中就不行，比如我在设计ALU模块的时候，在指令选择时候第一次就没有加上else这个选项，导致了我的仿真信号一直有一段是红色的，一直debug了好几个小时都没有解决，最后还是舍友帮我发现了问题，并帮我找到了verilog教程里面的提示信息，if-else必须一一配对，不能只含有单个if。后来我还了解到一个信息就是，使用case语句时候必须要有default语句与之对应，否则可能会产生锁存器，导致程序进入死循环状态。

最后一个问题就是写板子的问题了，一开始遇到的问题是无法将仿真文件生成二进制bit文件，一直都是提示生成文件逻辑组合error，后来与舍友讨论之后，发现是代码的架构问题，随后又重新拉取了一个新的文件架构，顺利的解决了生成二进制文件的问题。在烧板完成之后，进行测试时候发现，每当我按动板子上的键位时候，LED屏的数字显示会变得不稳定，不会稳定的显示某一个数字。一开始我以为是接触不良的问题，多次使用充电宝还有自己的计算机当做电源进行供给，都是同样的会抖动。后来询问班里的几个做好的同学，他们告诉我是需要处理按键防抖动。网上的某个博客给出的解决方案如下：按键的防抖动可以采用在时钟上升沿时，对按键正信号或负信号进行取样，如果取样周期达到了预设周期，则输出正信号或负信号，这样可以避免按键信号的抖动问题，且输出的信号则一定是稳定状况下的按键信号。 按照他的说法，我进行了尝试，然后再次实验解决了问题。