《计算机组成原理》实验报告

第 5 次实验：多周期 CPU 的设计

姓名： 张瑞祎

学号：121220139

2012 级 计算机 系 4 班

邮箱：zhangry868@126.com

时间： 2014.4.14

一、 实验目的

1、深入理解 MIPS 指令系统并掌握在多时钟周期 CPU 的设计中，状态机是如何设计的。

2、掌握多时钟周期 CPU 的工作原理与逻辑功能实现。

3、通过对多时钟周期 CPU 的运行状况进行观察和分析，进一步加深理解。

4、实现多周期 CPU 的 50Mhz 时序仿真，完成所有基本指令 26 条，包括过程返回。

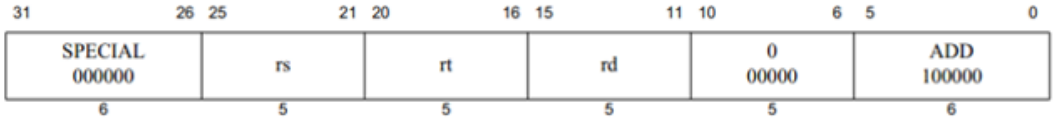
二、 实验设备

1. 搭载 Windows 上的 QuartusII 12.0 和 Modelsin 的计算机
2. Altera2 DE2-70 开发板

三、 实验原理

1. 实验指令

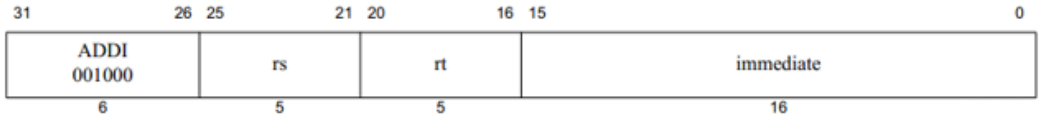
1.1 指令 1： add rd,rs,rt



描述：GPR[rd] <- GPR[rs] + GPR[rt]

32 位数相加，如果发生溢出，会产生自陷。如果没有发生溢出，则结果替换 rd 值。

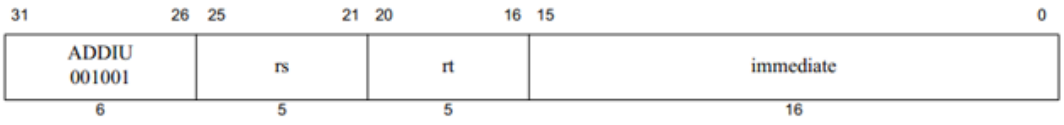
1.2 指令 2： addi rt,rs, imm



描述：GPR[rt] <- GPR[rs] + immediate

16 位立即数加至 32 位寄存器内值来产生一个 32 为结果，如果发生溢出，会产生自陷（异常）。如果没有发生溢出，则结果替换 rt 值。

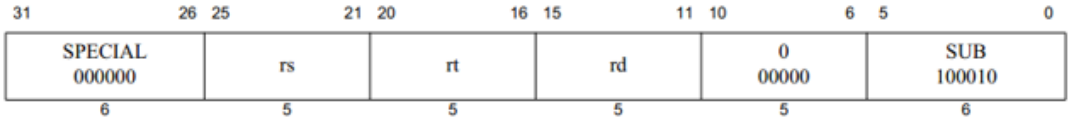
1.3 指令 3： addiu rt,rs, imm



描述：GPR[rt] <- GPR[rs] + immediate

16 位无符号立即数加至 32 位寄存器内值来产生一个 32 为结果，不会发生溢出。

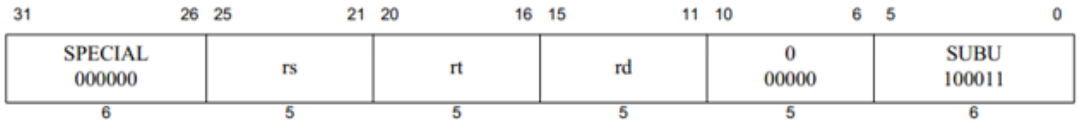
1.4 指令 4： sub rd,rs,rt



GPR[rd] <- GPR[rs] - GPR[rt]

描述：rs 中的 32 位值减去 rt 中的 32 位值来产生一个 32 位的结果，如果发生溢出，目的寄存器内的值将不会被改变，并且产生自陷；若没有溢出，则 32 为结果替换目的寄存器内的值。

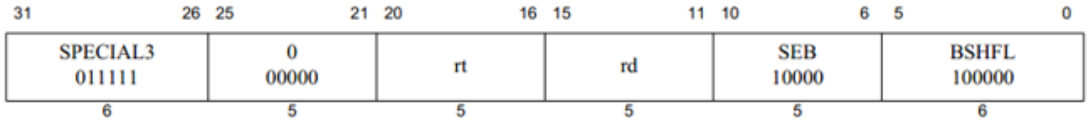
1.5 指令 5： subu rd,rs,rt



GPR[rd] <- GPR[rs] - GPR[rt]

描述：rs 中的 32 位无符号值减去 rt 中的 32 位值来产生一个 32 位的无符号结果，不会发生溢出。

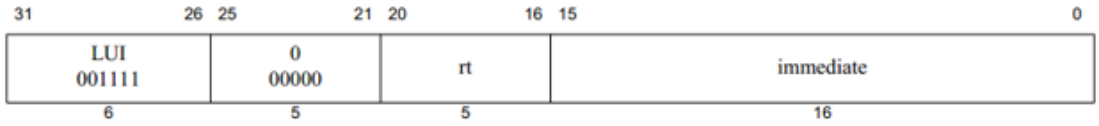
1.6 指令 6： seb rd,rt



GPR[rd] <- SignExtend(GPR[rt]$_{7..0}$)

描述：符号拓展 rt 寄存器内值的低位字节，并将位拓展结果存到 rd 寄存器。

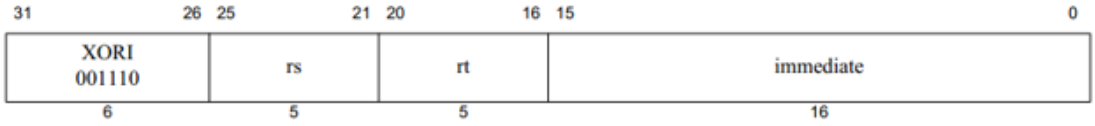
1.7 指令 7： lui rt, imm ///nor



GPR[rt] <- immediate || 0^{16}

描述：将 16 位立即数装入字长的高位（16bit），低 16 位为 0，32 位结果存储到通用寄存器 rt 中。

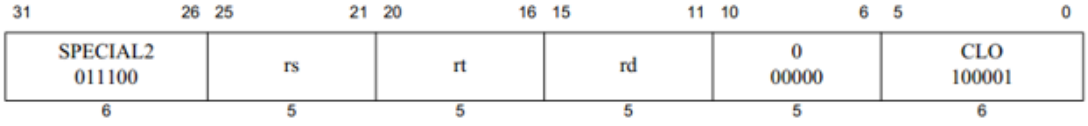
1.8 指令 8： xori rt,rs, imm



GPR[rt] <- GPR[rs] XOR immediate

描述：将寄存器内的值与 16bit 立即数逻辑拓展后的 32 位数进行按位异或，并将结果存到目的寄存器 rt。

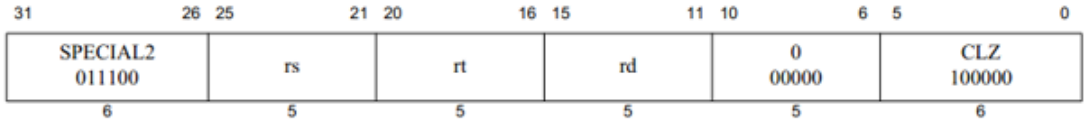
1.9 指令 9： clo rd,rs



描述：GPR[rd] <- count_leading_ones_GPR[rs]

计算前导 1（calculate leading one），计算前导 1 并将结果存入目的寄存器 rd。

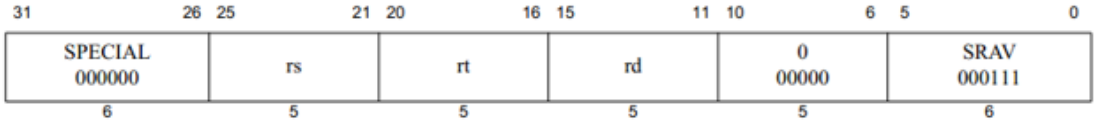
1.10 指令 10：clz rd,rs



描述：GPR[rd] <- count_leading_zeros_GPR[rs]

计算前导 0（calculate leading zero），计算前导 1 并将结果存入目的寄存器 rd。

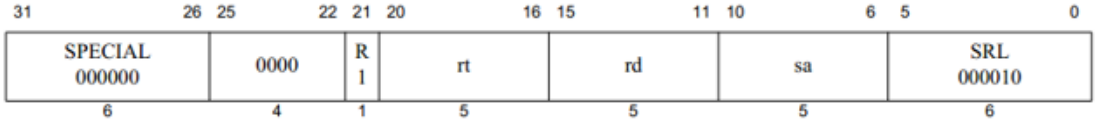
1.11 指令 11：srav rd,rt,rs



描述： GPR[rd] <- GPR[rt] >> rs

寄存器 rt 内的值进行右移，rs 决定移位数，并对高位补符。结果存入目的寄存器 rd。

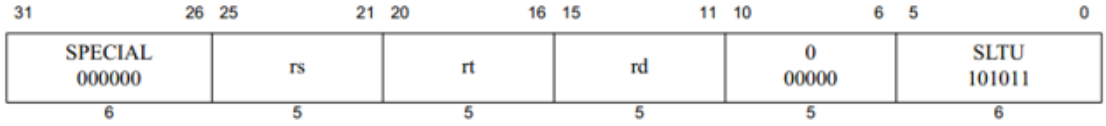
1.12 指令 12：rotr rd,rt,shamt



描述： GPR[rd] <- GPR[rt] <-> (right) sa

寄存器 rt 内的值进行循环右移，sa 决定移位数。结果存入目的寄存器 rd。

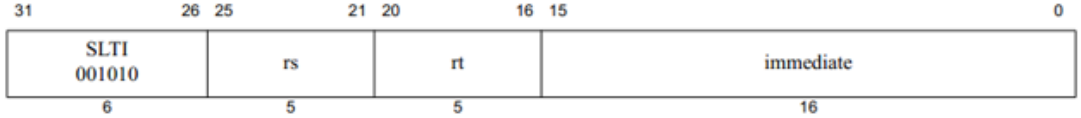
1.13 指令 13：sltu rd,rs,rt



描述：GPR[rd] <- (GPR[rs] < GPR[rt])

记录无符号数比较小于的结果。并将结果存入目的寄存器 rd。

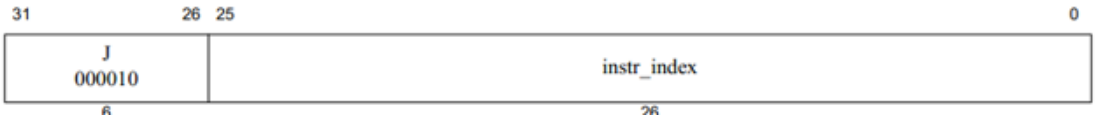
1.14 指令 14：slti rt,rs, imm



描述：GPR[rd] <- (GPR[rs] < immediate)

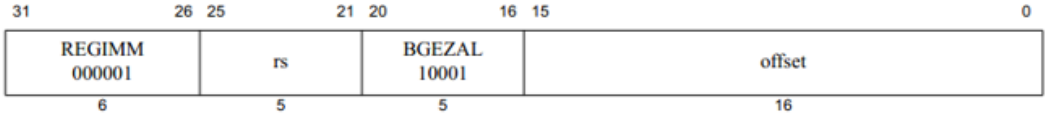
记录无符号数与立即数比较小于的结果。并将结果存入目的寄存器 rd。

1.15 指令 15：j target



描述：无条件跳转指令，256MB 范围内。

1.16 指令 16：bgezal rs,offset

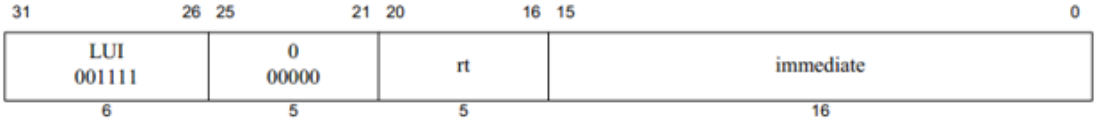


描述：Begzal rs,offset

目的：若寄存器值不小于 0，调用过程分支。调用过程中将返回地址存入 31#寄存器内，返回地址是分支指令后即将被执行的第二条指令。18bit 地址偏移加到 PC 上，如果满足分支条件，则会执行过程调用。

限制:处理器操作将不可预测，如果分支，跳转等指令放置于等待槽中。31#寄存器不能被用作源寄存器 Rs，不然可能发生不可预知的错误。

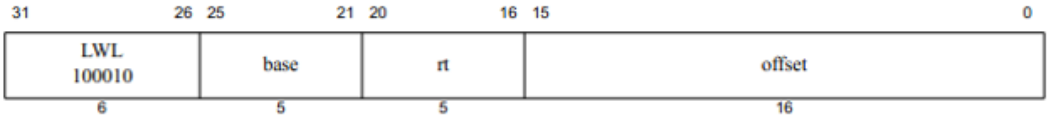
1.17 指令 17： lui rt, imm



GPR[rt] <- immediate || 0^{16}

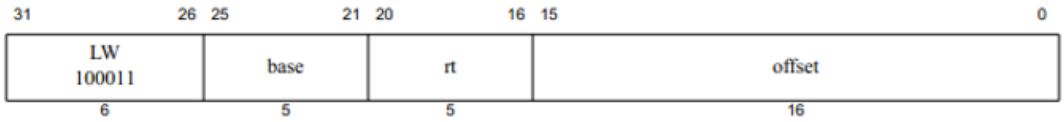
描述：将 16 位立即数装入字长的高位（16bit），低 16 位为 0，32 位结果存储到通用寄存器 rt 中。

1.18 指令 18：lwl rt,offset(base)



目的：从某一内存地址作为有符号值装载字节大端部分到目的寄存器中。

1.19 指令 19：lw rt,offset(base)



描述：LW rt，offset(base)

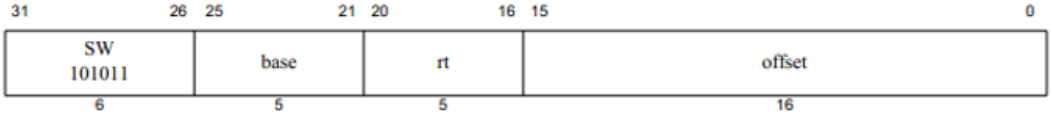
目的：从内存中装载一个 word，以有符号数形式。

GPR[rt] <- memory[GPR[base]+offset]

GPR[base]作为基址，经拓展后的 18bit offset 作为偏移，计算出有效内存地址，并将该地址的数据装入到目的寄存器 Rt。

限制：如果低 2 位地址不为 00，地址错误异常将会触发。

1.20 指令 20：sw rt,offset(base)



描述：SW rt,offset(base)

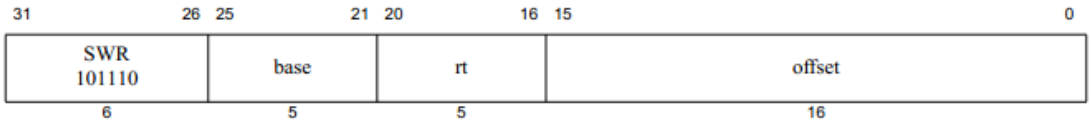
目的：保存一个 Word 到内存中

Memory[GPR[base] + offset] <- GPR[Rt]

GPR[rt]内的 32bit 值被存储到内存中指定的位置上去，16bit 有符号数作为偏移量加到 GPR[base]上获得有效地址。

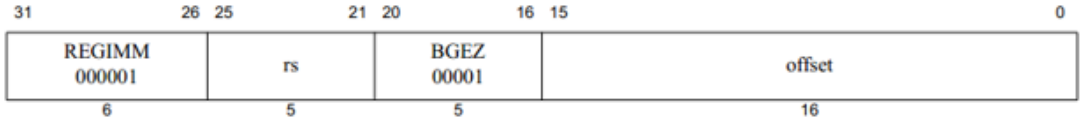
限制：如果低 2 位地址不为 00，地址错误异常将会触发。

1.21 指令 21：swr rt,offset(base)



描述：存储未对齐的内存地址内的一个字的低位的部分。

1.22 指令 22：bgez rs,offset



描述：if GPR[rs] >= 0 then branch

Branch on Greater Than or Equal to Zero，如果 rs 内的值大于等于 0，则发生跳转，下地址由 PC 的值与 18 位有符号偏移相加更新 PC。

2. 实现以上指令的单时钟周期 CPU 电路原理图

3. 指令控制器控制信号表

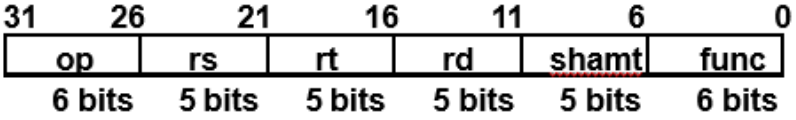
说明：凡是必要的信号全部赋予有效值，对于没有实质影响的信号（不在指令数据通路上），空出，并在赋值时，赋予 0。共分为 13 个状态表，每张表填有若干条指令的控制信号。控制信号表详见附件中的 Excel 文件。

4. MIPS CPU 指令类型

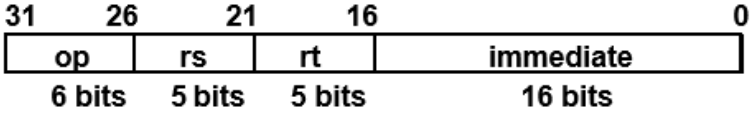
MIPS 是典型的 RISC 处理器，采用 32 位定长指令字，操作码字段也是固定长度，没有专门的寻址方式字段，由指令格式确定各操作数的寻址方式。 指令格式只有三种：

（1） R-型指令中大多数指令序列中 OP 字段为“000000”，操作类型由 func 字段指定，若是双目运算类指令，则 rs 和 rt 的内容分别作为第一和第二源操作数，结果送 rd；若是移位指令，则对 rt 的内容进行移位，结果送 rd，所移位数由 shamt 字段给出。因为一条指令需要左移或右移若干位，所以 MIPS 中移位指令多用桶形移位器以提高速度。本次实验中

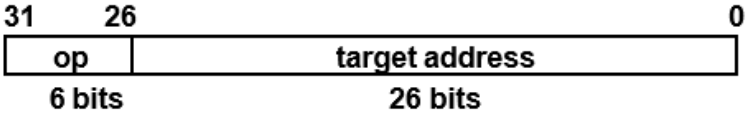
Add,Sub,Subu,Seb,Srav,Rotr,Sltu,Clo,Clz。属于 R-型指令。



（2） I-型指令是立即数型指令，若是双目运算类指令，则将 rs 的内容和立即数分别作为第一源操作数和第二源操作数，结果送 rt；若是条件分支指令，则对 rs 和 rt 内容进行指定的运算，根据运算的结果，决定是否转到目标地址处执行，转移目标地址通过相对寻址方式得到，即将 PC 的内容和立即数符号扩展后的内容相加得到。 Addi, Addiu, Xori, Slti, Bgez,Lui 属于 I-型指令。



（3） J-型指令主要是无条件跳转指令，指令中给出的是 26 位直接地址，只要将当前 PC 的高 4 位拼上 26 位直接地址，最后添两个“0”就可以得到 32 位的跳转目标地址。 J（无条件跳转） 属于 J-型指令。



5.指令执行周期完成的任务

1.取址，PC（程序计数器）模块送地址到指令内存，指令内存输出当前指令。

2.译码，控制器模块读取指令，并对指令进行译码操作，给出控制信号。

3.取数，根据指令中的要求去立即数或从寄存器内取数（包括内存，本实验未涉及）。

4.执行，运算部件进行运算存入结果或进行地址转移。

5.存数，根据控制信号，存储结果到指定位置（溢出不存）。

6.下地址计算，在指令控制模块中完成，包括正常执行，条件转移，无条件跳转。

6.控制信号解析

	信号含义
Rd_write_byte_en[3:0] (Overflow=0)	目的寄存器写使能
Rd_write_byte_en[3:0] (Overflow=1)	目的寄存器写使能
ALUSrcA	ALU 操作数 A　　1（Rs）0（PC_out）
ALUSrcB[2:0]	ALU 操作数 B　　1（4）0（Rt）2（立即数） 3（高 30 位）4（立即数高位 lui）
Ex_top	0（逻辑） 1（算术）
ALU_op[3:0]	ALU 操作选择信号
RegDst[1:0]	目的寄存器选择 0（Rt）1（Rd）2（过程调用考虑）3（暂无）
Shift_amountSrc	移位位数 0：Shamt 1：Rs
ALUShift_sel	1：Shift 0：ALU

Condition[2:0]	条件判断
Shift_op	桶形移位器
PC_source[1:0]	PC 输入来源控制 0(ALU)1(地址寄存器)2(跳转 jump)3（暂无）
Addreg_write_en	地址寄存器写使能
IR_write	指令寄存器写使能
PC_write_cond	条件转移时有效
PC_write	顺序执行是有效
IorD	地址寄存器（1） PC（0）
MemtoReg[1:0]	0（地址寄存器）1（存储器输出移位器）2（ALU）
Mem_byte_write[3:0]	存储器写使能

四、 实验步骤

1. 对单周期 CPU 进行模块化分解将指令分为以下几个功能模块：

a.位拓展模块，向顶层实体输入与立即数有关的内容

b.指令计数器模块，时钟到来时向指令存储传输指令地址

c.存储器模块，内存，从内存中读取指令与数据。

d.指令控制模块，根据输入来决定下地址的值。

e.控制信号模块，对指令进行译码，并生成控制信号。

f.算术逻辑运算单元，对操作数进行算术计算，并输出结果。

g.桶形移位器，进行以为操作。

h.寄存器组，CPU 内的寄存器。

i.寄存器移位输出模块。

j.存储器移位输出模块。

k.指令寄存器模块，存储当前正在执行的指令。

l.地址寄存器模块，暂时存储 ALU 的计算结果。

2. 各个模块的设计思路及部分代码

2.1 位拓展模块

```
module ExNumber(
input [15:0] IR,//Imm,Offset
input Ex_top,
input [2:0] ALU_SrcB,
input [31:0] Rt_out,
output [31:0] B_in
);

wire [31:0]Ex_offset,Imm_ex,addroffset_ex;
assign Ex_offset = {((Ex_top == 1)?{16{IR[15]}}:16'b0),IR[15:0]};//2,Imm
assign Imm_ex = {IR,16'b0};//4,Lui
assign addroffset_ex = {Ex_offset[29:0],2'b00};
MUX8_1_32
```

```
mux8_1_32(ALU_SrcB,Rt_out,32'h4,Ex_offset,addroffset_ex,Imm_ex,32'b0,32
'b0,32'b0,B_in);

endmodule
```

endmodule 说明：模块设计中将第二个操作数的选择设置在此模块内。输出为五个，分别为寄存器输入 Rt_in，地址增量 4，立即数，高 30 位，以及立即数高位装入（lui），对于多余的处理，采取的是赋值为 32’b0。

2.2 指令计数器模块

```
module PC(
input [31:0] PC_in,
output reg[31:0] PC_out,
input Clk,PC_write_en
);

initial PC_out = 32'b0;

always@(negedge Clk)
    if(PC_write_en)
        PC_out <= PC_in;
endmodule
```

说明：在多周期 CPU 中，指令计数器模块增加了使能端，因为只有在第二个公共周期以及跳转指令的特定周期中，PC 的值才允许改变，对于其他情况时，PC 使能端无效，不允许修改 PC 内的值。

在时钟下降沿到来时，若 PC 写使能有效，将 PC_in 数据写入 PC 中，并立即输出在 PC_out 上。采用阻塞赋值方式赋值。另外，为了使 CPU 测试成功，PC 默认起始值为 0，即从第 0 条指令开始执行。并且设置了状态机初始状态。

2.3 存储器模块

使用二维数组存储，按 32bit 字长为单位取址。指令编码预先编好，逐条测试指令执行正确与否。根据 PC 传入值的高 30 位来确定存储器的输出。将输出值送到存储器以为输出器输入端。

2.4 指令控制模块

```
module Instru_Control(
input less,Zero,PC_write_cond,PC_write,
input [2:0] condition,
input [1:0] PC_source,
input [25:0] IR,
input [31:28] PC_out,
input [31:0] ALUShift_out,AddrReg_out,Rs_out,
output [31:0] PC_in,
output PC_write_en
);

wire [31:0] Target_address;
```

```
assign Target_address = {PC_out[31:28],IR[25:0],2'b00};
MUX4_1_IControl
mux4_1_8(PC_source,ALUShift_out,AddrReg_out,Target_address,32'h0,PC_in);
wire condition_out;
MUX8_1_Single
MUX_Con(condition,1'b0,Zero,!Zero,!less,!(less^Zero),less^Zero,less,1'b
1,condition_out);
assign PC_write_en = (condition_out & PC_write_cond)| PC_write;
endmodule
```

说明：指令控制模块，主要有三大类，一是顺序执行，二是分支指令，三是无条件跳转指令。需要注意的是分支指令，需要在 PC+4 的基础上进行操作，以就是说，不论何种情况下，PC+4 一直在执行，虽然无条件转移没有使用当前此值。

其中的 8 选一选择器利用 ALU 传入的 Zero 和 Less 指令来进行条件判断，一旦条件成立，则将 condition_out 置为 1，并将进行由条件转移的下地址计算。在顺序执行情况下，8 选一选择器选择信号为 3’b000。在多周期，顺序执行地址是在指令送指令寄存时由 ALU 计算得到的，指令控制模块决定输入 PC_in 的值，计算过程完全由控制器来控制。指令控制器在转移指令更新 PC 时起控制作用。在条件转移状态下，PC_cond = 1’b1，与 condition 相与后用来确定写使能信号。指令控制模块完成了无条件跳转时地址的计算，作为 PC_in 的一个选择端。

2.5 控制信号模块

控制信号模块是这次多周期设计的核心部分，在此部分的设计中，使用状态机来进行描述，不同类型指令的不同阶段对应不同状态，在部分状态中，控制信号是一致的，而在另一部分的状态中，控制信号因指令的不同而有小小的差别。

在设计这一模块时最重要的是先要把各个信号含义进行清楚地了解，之后根据 MIPS 手册上指令的描述来进行控制信号的判断。个人感觉先进行控制信号填表是十分必要的，这样设计起来更快捷，但是在设计过程中，最好还是在输入时重新对 CPU 数据通路进行梳理，以防发生不必要的错误。

对于控制信号，分为若干大类，前三个周期为公共周期，一类为 R 型指令，他们前六位 op 字段为 000000，直接根据 func 字段判断指令功能。一类为 I 型指令，根据 op 字段判断指令含义，另外还包括 Store 类，load 类，过程分支类，条件转移以及无条件跳转类。另外值得注意的是，要在最后加入 default，以避免这一常见错误，力避生成锁存器。

控制信号在控制其输出时并非全部有用，对于一些无关的信号(比如在计算溢出加时的 shift_op 信号)，我采用的是将其赋为 0，而并没有采用不赋值的策略。对于指令信号中的红线（X），立即就可以判断信号幅值存在问题。

2.6 算术逻辑运算单元（ALU）

沿用了上次实验设计的 MIPS ALU，ALU 全部为组合电路形式，经由许多模块组装而成。

2.7 桶形移位器

沿用了上次实验设计的 MIPS Shifter，上一次设计桶形移位器尝试了多种方法，最终采用的是结构级建模方式实现的桶形移位器，符合桶形移位器的效率要求，为 5 层门电路组成。使用这种方式使我感到在 CPU 中调用时的便捷与高效。

2.8 寄存器组

沿用了上次实验设计的 MIPS Register。在时序仿真期间发生了一个错误，写入和读出出现问题，但在功能仿真下是正常的，经过自己的修改并接受张老师的意见，最终得到的结果正确。

2.9 顶层实体设计

顶层实体将所有的模块进行了组合，特别需要注意各个模块信号的数据位宽，如果出现疏漏将会产生不可预知的错误。我按照实验指导书上的设计图进行了连接。这一模块的设计有一个特点，即 wire 变量特别多和复杂，这对我们的命名提出了要求，对此命名必须有所区分，严格准确，不然很容易出错，而且一旦出错，通过仿真检查起来比较麻烦。

2.10 寄存器移位输出模块

```
module Register_ShiftOutput(
input [31:0] Rt_out,
input [1:0] Mem_addr_in,
input [31:26] IR,
output [31:0] Mem_data_shift
);
wire [2:0] Rt_out_shift_ctr;
wire [31:0] Rt_out_l,Rt_out_r,Rt_out_shift;
assign Rt_out_shift_ctr[2] = (IR[31])&(!IR[30])&(IR[29])
&(((!IR[28])&(IR[27])) | ((IR[27])&(!IR[26])) );
assign Rt_out_shift_ctr[1] = (IR[31])&(!IR[30])&(IR[29])
&(((!IR[28])&(!IR[27])&(IR[26])) | ((IR[28])&(IR[27])&(!IR[26])));//xor better
assign Rt_out_shift_ctr[0] = (IR[31])&(!IR[30])&(IR[29])
&(!IR[28])&(IR[27])&(!IR[26]);

MUX4_1 mux4_1_0(Mem_addr_in[1:0],Rt_out[31:24],8'b0,8'b0,8'b0,Rt_out_l[31:24]);
MUX4_1
mux4_1_1(Mem_addr_in[1:0],Rt_out[23:16],Rt_out[31:24],8'b0,8'b0,Rt_out_l[23:16]);
MUX4_1
mux4_1_2(Mem_addr_in[1:0],Rt_out[15:8],Rt_out[23:16],Rt_out[31:24],8'b0,Rt_out_l[15
:8]);
MUX4_1
mux4_1_3(Mem_addr_in[1:0],Rt_out[7:0],Rt_out[15:8],Rt_out[23:16],Rt_out[31:24],Rt_o
ut_l[7:0]);
MUX4_1
mux4_1_4(Mem_addr_in[1:0],Rt_out[7:0],Rt_out[15:8],Rt_out[23:16],Rt_out[31:24],Rt_o
ut_r[31:24]);
MUX4_1
mux4_1_5(Mem_addr_in[1:0],8'b0,Rt_out[7:0],Rt_out[15:8],Rt_out[23:16],Rt_out_r[23:1
6]);
MUX4_1 mux4_1_6(Mem_addr_in[1:0],8'b0,8'b0,Rt_out[7:0],Rt_out[15:8],Rt_out_r[15:8]);
MUX4_1 mux4_1_7(Mem_addr_in[1:0],8'b0,8'b0,8'b0,Rt_out[7:0],Rt_out_r[7:0]);
```

```
MUX8_1
mux8_1_0(Rt_out_shift_ctr[2:0],Rt_out[7:0],8'b0,Rt_out[15:8],8'b0,Rt_out_l[31:24],R
t_out_l[31:24],Rt_out_r[31:24],8'b0,Mem_data_shift[31:24]);
MUX8_1
mux8_1_1(Rt_out_shift_ctr[2:0],Rt_out[7:0],8'b0,Rt_out[7:0],8'b0,Rt_out_l[23:16],Rt
_out_l[23:16],Rt_out_r[23:16],8'b0,Mem_data_shift[23:16]);
MUX8_1
mux8_1_2(Rt_out_shift_ctr[2:0],Rt_out[7:0],8'b0,Rt_out[15:8],8'b0,Rt_out_l[15:8],Rt
_out_l[15:8],Rt_out_r[15:8],8'b0,Mem_data_shift[15:8]);
MUX8_1
mux8_1_3(Rt_out_shift_ctr[2:0],Rt_out[7:0],8'b0,Rt_out[7:0],8'b0,Rt_out_l[7:0],Rt_o
ut_l[7:0],Rt_out_r[7:0],8'b0,Mem_data_shift[7:0]);
Endmodule
```

按照原理图采用结构级建模来完成，主要在 Lw 指令中使用。

2.11 存储器移位输出模块

与寄存器移位输出模块类似，按照原理图采用结构级建模来完成。

2.12 指令寄存器模块

```
module IRegister(
input [31:0] IR_in,
output reg[31:0] IR_out,
input Clk,IR_write_en
);
always@(negedge Clk)
    if(IR_write_en)
        IR_out <= IR_in;
Endmodule
```

说明：指令寄存器模块在第二个周期内接受存储器的输出，将指令寄存，用来在整个指令执行过程中提供指令信息，因为存储器不可能一直处于输出当前指令的状态，如在第二个周期后，PC_out 会被更新，存储器输出会改变。在 lw 以及 sw 类指令中，将涉及数据的读写，也会改变存储器输出。总之，指令寄存器不可缺少。

2.13 地址寄存器模块

```
module AddrReg(
input [31:0] AddrReg_in,
input AddrReg_write_en,Clk,
output reg [31:0] AddrReg_out
);
always @(negedge Clk)
    if(AddrReg_write_en)
        AddrReg_out <= AddrReg_in;
Endmodule
```

说明：地址寄存器模块用于暂存 ALU 的输出，存储的内容可能是地址，也可能是数据。下降沿写入，读电路为纯组合电路。一旦更新，立即输出。

3. 书写测试代码

指令序列：

Add $9 $10 $11

Sub $9 $12 $4

Sub $31 $4 $5

Add $5 $1 $5

Subu $4 $1 $5

SRAV $5 $2 $5

ROTR $9 Shamt $5

Sltu $10 $9 $5

Addi $5 Immediate $5

Addi $31 Immediate $5

Addiu $5 Immediate $5

Addiu $31 Immediate $5

Lui Immediate $11

Xori $31 Immediate $5

Clo $11 $5

Clz $10 $5

Slti $9 $10 $11

Seb Immediate $5

Begz $1 $0

Begz $30 $0

J target

操作	操作数 A	操作数 B	结果
Add	54678932(9#)	34768906（10#）	88de1238（11#）溢出
Sub	54678932（9#）	67893954（12#）	ecde4fde（4#）
Sub	7fffffff（31#）	ecde4fde（4#）	溢出，5#不写
Add	55556789（5#）	11112345(1#)	66668ace（3#）
Subu	ecde4fde（4#）	11112345（1#）	dbcd2c99（5#）
SRAV	dbcd2c99（5#）	算术右移两位 2（2#）	F6f34b26 (5#)
ROTR	54678932（9#）	循环右移 2 位（Sa）	9519e24c（5#）
Sltu	54678932(10#)	34768906（9#）	ffffffff（5#）
Addi	ffffffff（5#）	789a（I）	7899（5#）
Addi	7ffffff（31#）	7abc（I）	溢出，不写入（5#）
Addiu	7ffffff（31#）	7abc（I）	80007abb（5#）
Addiu	80007abb（5#）	789a（I）	8000f355（5#）
Lui		789aH	789a0000（5#）

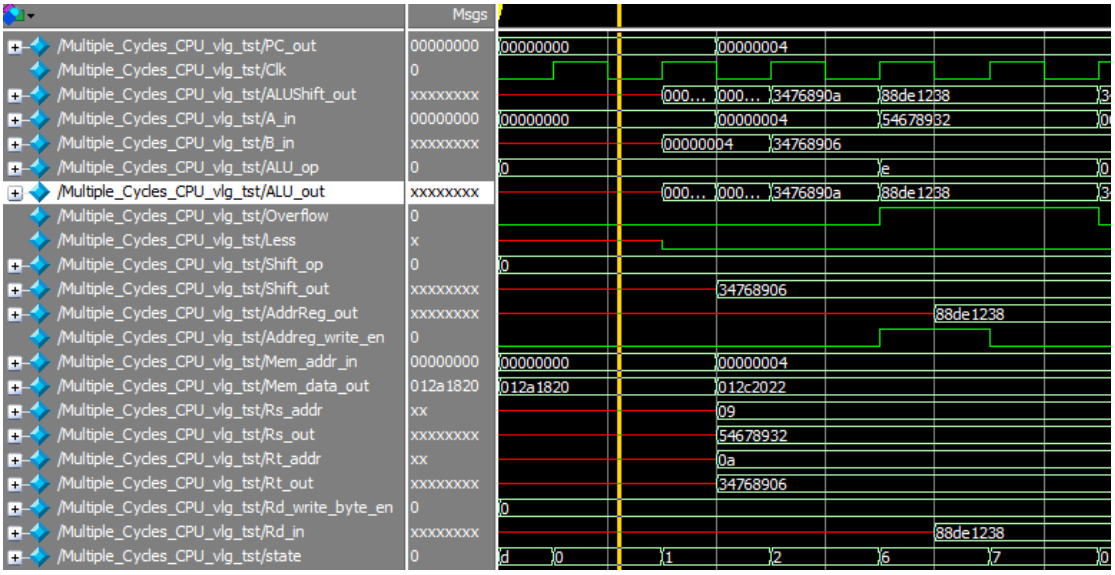
Xori	7ffffff(31#)	abcdH	7fff5423(5#)
Clo	fffabcde(11#)		13（5#）
Clz	54678932(10#)		2（5#）
Slti	2（5#）	（Ffff）abcd（I）	00000000(5#)
Seb	X	00000088	ffffff88
Begz	1	(0)	跳转到下下条
Begz	ffffffff	(0)	不跳转，顺序。
J			跳转回第一条指令

4. 仿真验证

5.0 寄存器的初始化如下：

```
initial
begin
    Data[1] = 32'h11112345;
    Data[2] = 32'h2;
    Data[3] = 32'h3;
    Data[4] = 32'h4;
    Data[5] = 32'h55556789;
    Data[8] = 32'h88;
    Data[9] = 32'h5467_8932;
    Data[10] = 32'h3476_8906;
    Data[11] = 32'hfffa_bcde;
    Data[12] = 32'h6789_3954;
    Data[30] = 32'hffff_ffff;
    Data[31] = 32'h7fff_ffff;
end
```

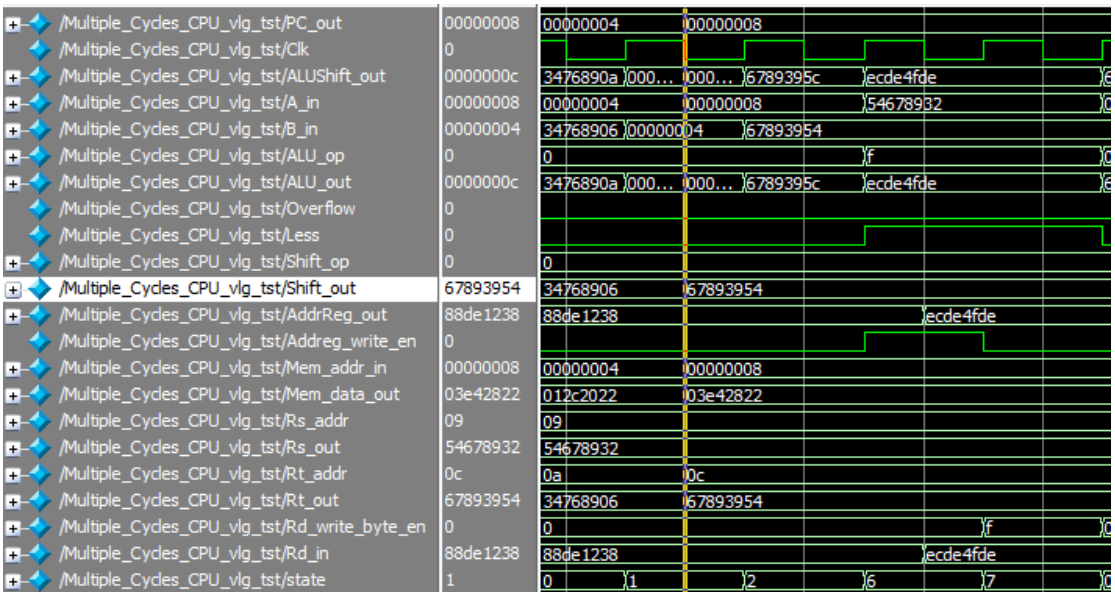
5.1 Add $9 $10 $11



指令为带溢出加操作，前二个周期为公共周期，分别为从内存内取出指令，第 3 个周期，

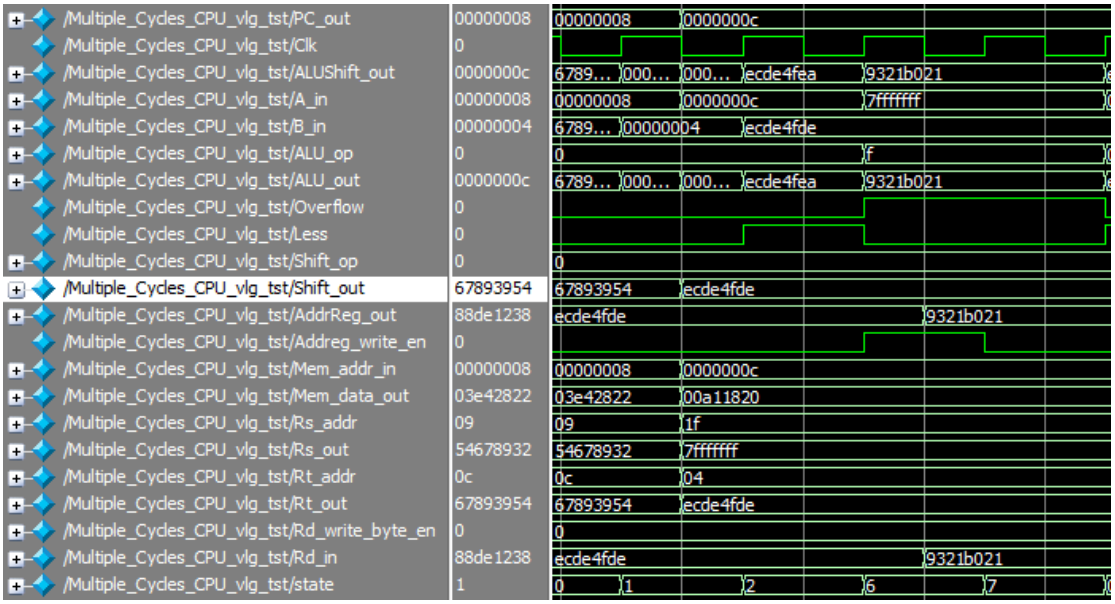
进行指令译码和寄存器取，投机进行 I 型指令结果运算或者 sw 的地址计算，从指令中获得源操作数寄存器号为$9 和$10，目的寄存器为$11，初始 PC 为 0，顺序执行，经下地址计算模块，下地址为 4;B_in 通过选择器选择 Rt_out 作为输入，Rt_out = 34768906,Rs_out = 54678932,经 ALU 计算后，ALUShiftSel 输出选择 ALU_out，ALU_out 输出并存入到地址寄存器中，在最后一个周期，根据 MemtoReg 信号控制，并将结果送到 Rd_in ,Rd_in = AddrReg_out =88de1238，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，结果溢出，写使能有无效（Rd_write_byte.. = 0h），结果不写入。

5.2 Sub $9 $12 $4



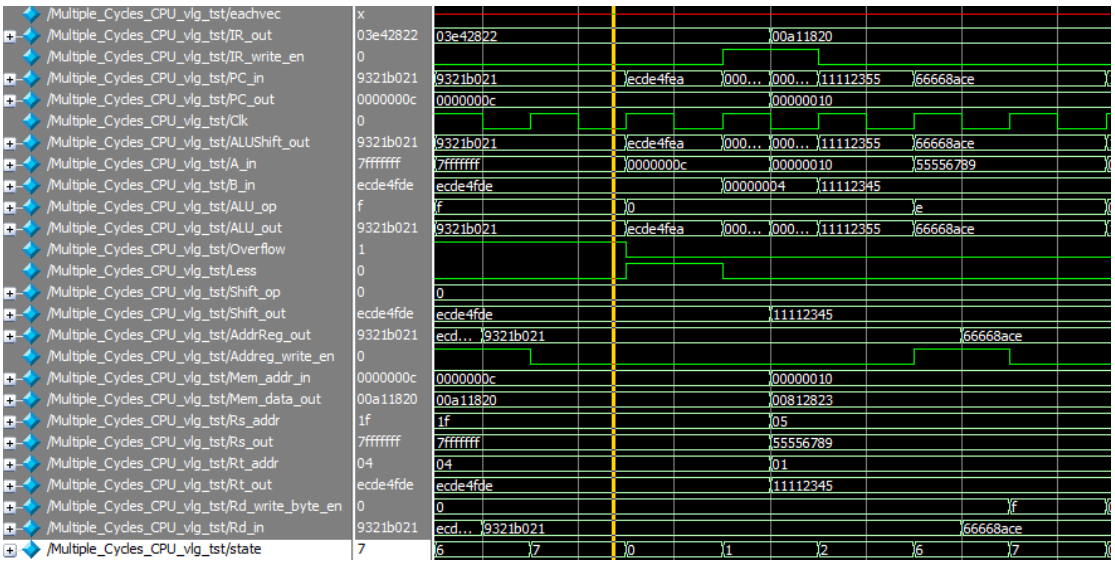
指令为带溢出减操作，从指令中获得源操作数寄存器号为$9 和$10，目的寄存器为$11，初始 PC 为 4，第一周期内从内存取指，第二个周期，将取出的指令送指令寄存器，默认顺序执行，在第二个周期内利用 ALU 进行下地址计算，下地址为 8;第三个周期进行译码和寄存器取，B_in 通过选择器选择 Rt_out 作为输入，Rt_out = 67893954,Rs_out = 54678932，第四个周期，由 func 字段判断为带溢出减法，经 ALU 计算后，ALU_out 输出， ALUShiftSel 输出选择 ALU_out，并将结果暂存在地址寄存器，第五个周期，通过选择器选择，将结果送到 Rd_in,Rd_in = AddrReg_out = ecde4fde，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，结果不溢出，写使能有有效（Rd_write_byte.. = fh），结果写入。

5.3 Sub $31 $4 $5



指令为带溢出减操作，从指令中获得源操作数寄存器号为$31 和$12，目的寄存器为$4，PC 为 0x8，顺序执行，第一周期内从内存取指，第二个周期，将取出的指令送指令寄存器，默认顺序执行，在第二个周期内利用 ALU 进行下地址计算，下地址为 12;第三个周期进行译码和寄存器取，B_in 通过选择器选择 Rt_out 作为输入，Rt_out = ecde4fde,Rs_out = 7fffffff, ，第四个周期，由 func 字段判断为带溢出减法，经 ALU 计算后，ALU_out 输出， ALUShiftSel 输出选择 ALU_out，并将结果暂存在地址寄存器，第五个周期，通过选择器选择，将结果送到 Rd_in，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，结果溢出，写使能无效（Rd_write_byte.. = 0h），结果不写入。并在下一周期内测试。

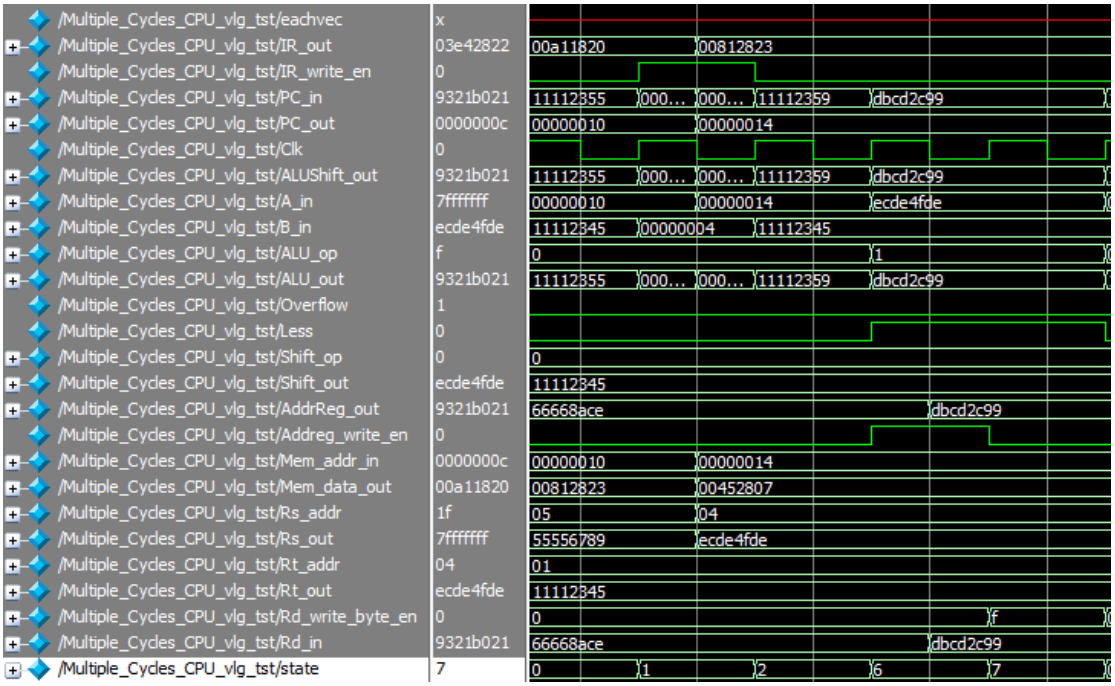
5.4 Add $5 $1 $3



指令为带溢出加操作，从指令中获得源操作数寄存器号为$5 和$1，目的寄存器为$5，PC 为 0xc，顺序执行，第一周期内从内存取指，第二个周期，将取出的指令送指令寄存器，默认顺序执行，在第二个周期内利用 ALU 进行下地址计算，下地址为 0x10;第三个周期进

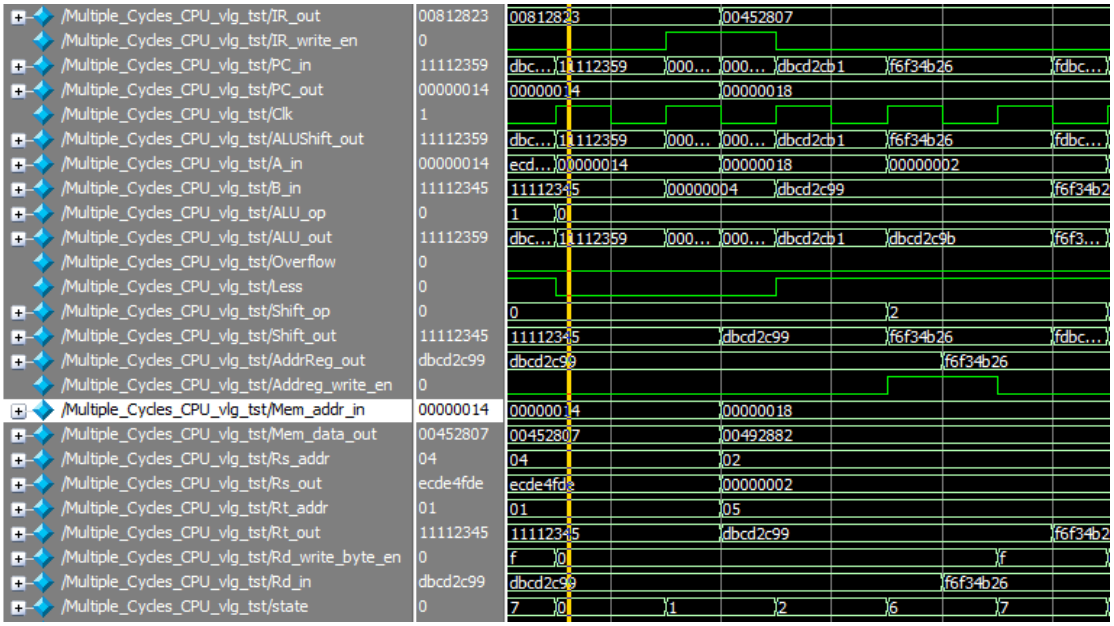
行译码和寄存器取，B_in 通过选择器选择 Rt_out 作为输入，Rt_out = 55556789,Rs_out = 11112345，可见上一周期因溢出结果没有写入$5 寄存器，寄存器内值未发生变化，第四个周期，由 func 字段判断为带溢出加法，经 ALU 计算后，ALU_out 输出， ALUShiftSel 输出选择 ALU_out，并将结果暂存在地址寄存器，第五个周期，通过选择器选择，将结果送到 Rd_in,Rd_in = AddrReg_out = 66668ace，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，结果不溢出，写使能有有效（Rd_write_byte.. = fh），结果写入$3 寄存器。

5.5 Subu $4 $1 $5



指令为无符号减操作，从指令中获得源操作数寄存器号为$4 和$1，目的寄存器为$5，初始 PC 为 0x10，第一周期内从内存取指，第二个周期，将取出的指令送指令寄存器，默认顺序执行，在第二个周期内利用 ALU 进行下地址计算，下地址为 12;第三个周期进行译码和寄存器取，B_in 通过选择器选择 Rt_out 作为输入，Rt_out = 11112345,Rs_out = ecde4fde,由 func 字段判断为不带溢出减法，经 ALU 计算后，ALU_out 输出， ALUShiftSel 输出选择 ALU_out，并将结果暂存在地址寄存器，第五个周期，通过选择器选择，将结果送到 Rd_in,Rd_in = AddrReg_out = dbcd2c99，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，结果不溢出，写使能有有效（Rd_write_byte.. = fh），结果写入。

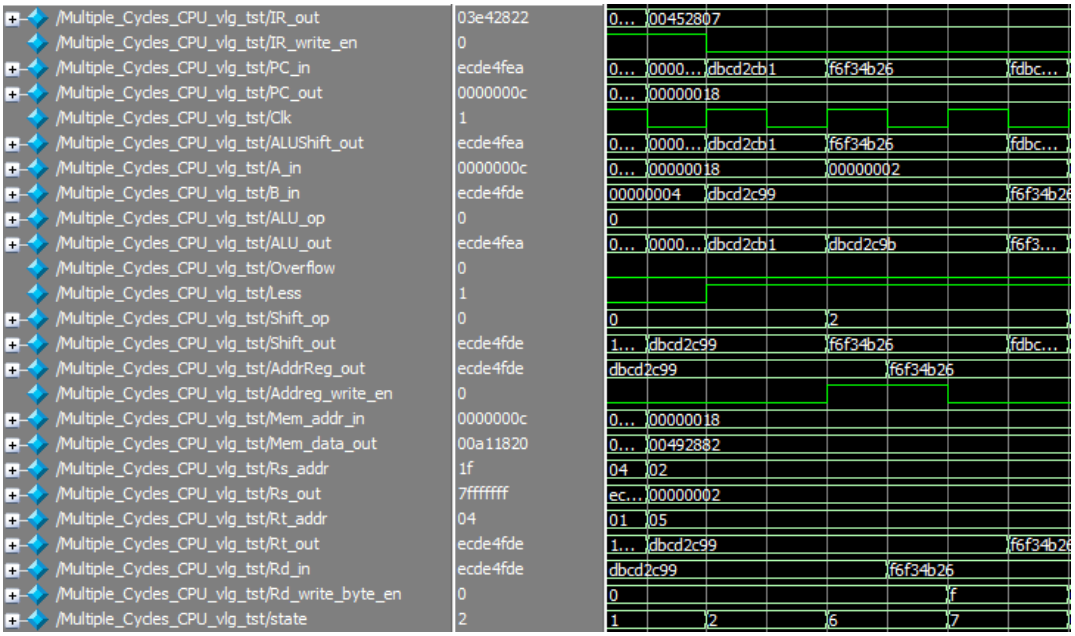
5.6 SRAV $5 $2 $5



指令为算数右移操作，从指令中获得源操作数寄存器号为$5，目的寄存器为$5，初始 PC 为 0x14，顺序执行，第一周期内从内存取指，第二个周期，将取出的指令送指令寄存器，默认顺序执行，在第二个周期内利用 ALU 进行下地址计算，下地址为 0x18;第三个周期进行译码和寄存器取，B_in 通过选择器选择 Rt_out 作为输入，Rt = 2,Rs_out = dbcd2c99，第四个周期，移位方式由控制器指定 shift_op = 2'b01，经桶形移位器计算后，Shift_out 输出，ALUShiftSel 输出选择 Shift_out，并将结果暂存到地址寄存器，第五个周期，将地址寄存器值送到 Rd_in,Rd_in = f6f34b26，根据 Rd_addr 判断写入的位置，写使能有有效（Rd_write_byte.. = fh），结果写入。

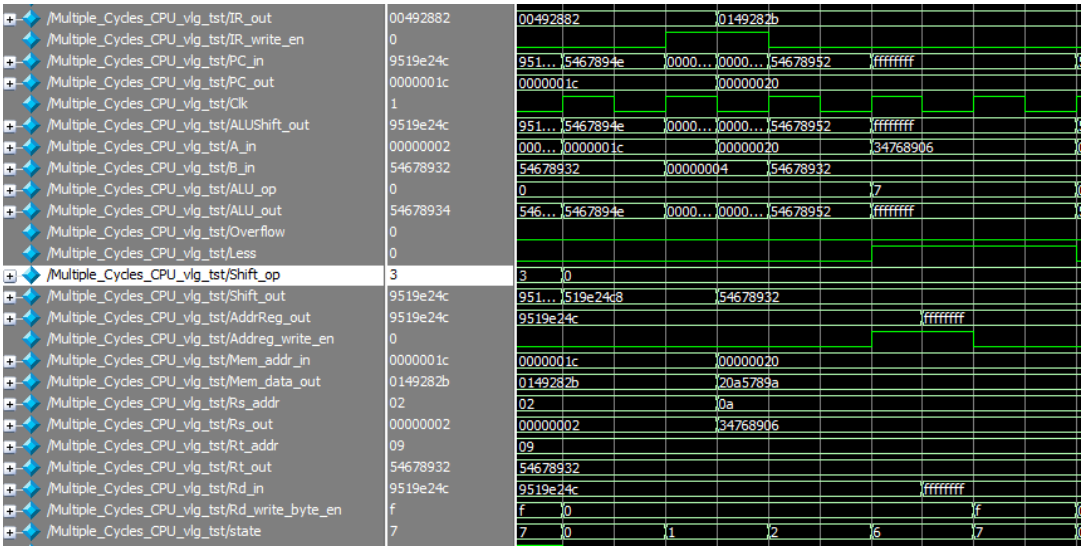
很容易的看出，在下降沿到来时结果写入到寄存器内，因为读和写源自同一个寄存器，所以在下降沿时会有变化，因为 CPU 中数据通路为纯组合电路，所以各线输出随之改变，但是没有下降沿到达，所以并不会写入到寄存器内。

5.7 ROTR $9 Shamt $5



指令为算数右移操作，从指令中获得源操作数寄存器号为$9，目的寄存器为$5，初始 PC 为 0x18，顺序执行，第一周期内从内存取指，第二个周期，将取出的指令送指令寄存器，默认顺序执行，在第二个周期内利用 ALU 进行下地址计算，下地址为 0x1c;第三个周期进行译码和寄存器取，B_in 通过选择器选择 shamt 作为移位位数输入，shamt = 2,Rs_out = dbcd2c99，移位方式由控制器指定 shift_op = 2'b01，经桶形移位器计算后，Shift_out 输出，ALUShiftSel 输出选择 Shift_out，并将结果暂存到地址寄存器，第四个周期，将地址寄存器值送到 Rd_in,Rd_in = f6f34b26，根据 Rd_addr 判断写入的位置，写使能有有效（Rd_write_byte.. = fh），结果写入。

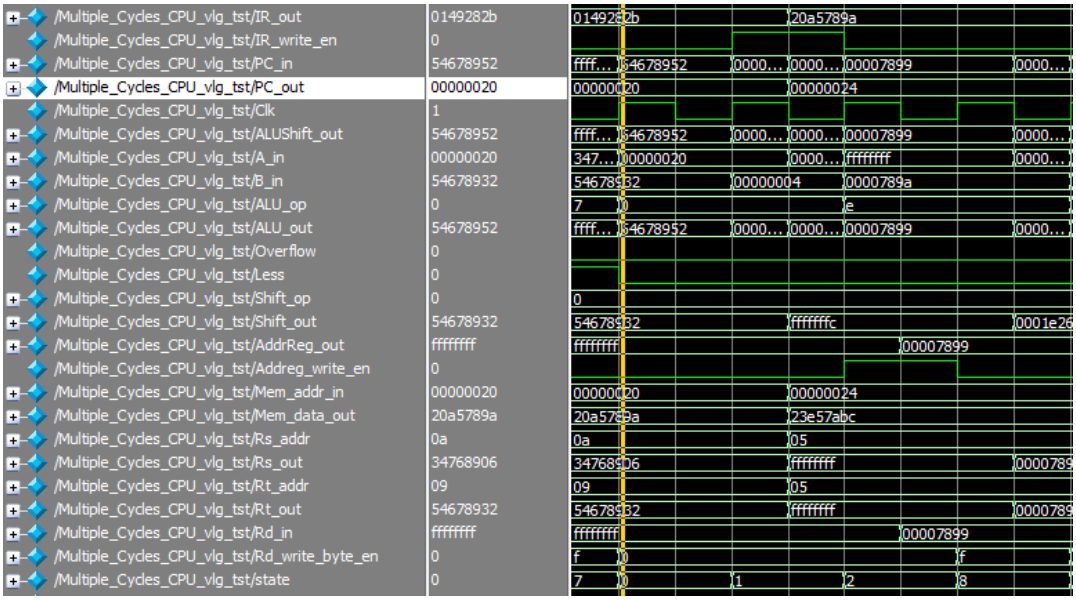
5.8 Sltu $10 $9 $5



指令为无符号比较操作，从指令中获得源操作数寄存器号为$10 和$9，目的寄存器为$5，初始 PC 为 0x1c，第一周期内从内存取指，第二个周期，将取出的指令送指令寄存器，默

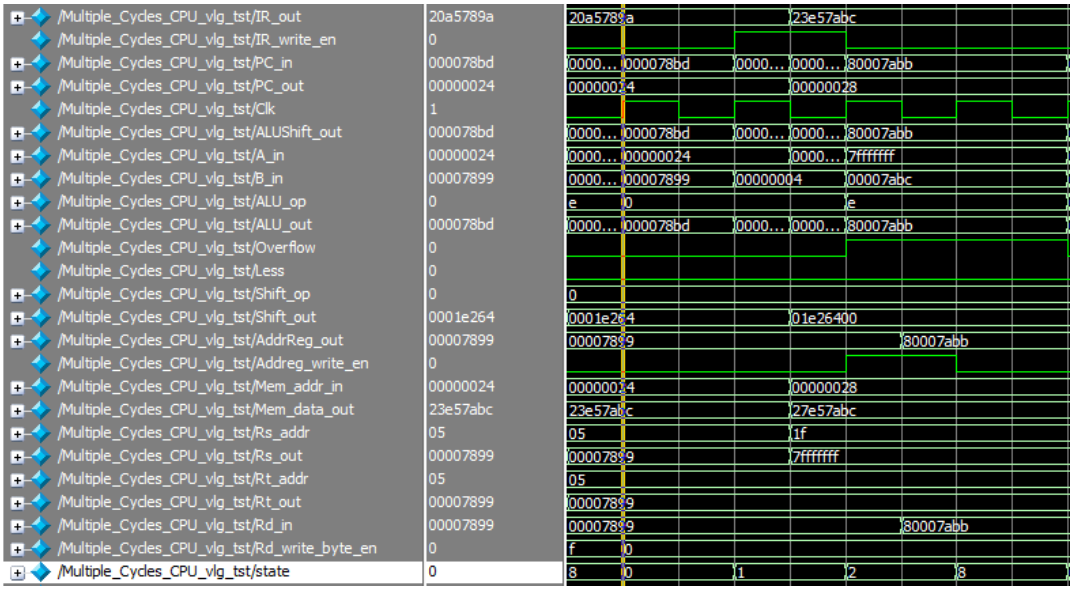
认顺序执行，在第二个周期内利用 ALU 进行下地址计算，下地址为 20;第三个周期进行译码和寄存器取，B_in 通过选择器选择 Rt_out 作为输入，Rt_out = 11112345,Rs_out = ecde4fde,第四个周期，由 func 字段判断为无符号比较，经 ALU 计算后，ALU_out 输出， ALUShiftSel 输出选择 ALU_out，并将结果暂存在地址寄存器，第五个周期，通过选择器选择，将结果送到 Rd_in,Rd_in = AddrReg_out = ffffffff，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，写使能有有效（Rd_write_byte.. = fh），结果写入。

5.9 Addi $5 Immediate $5



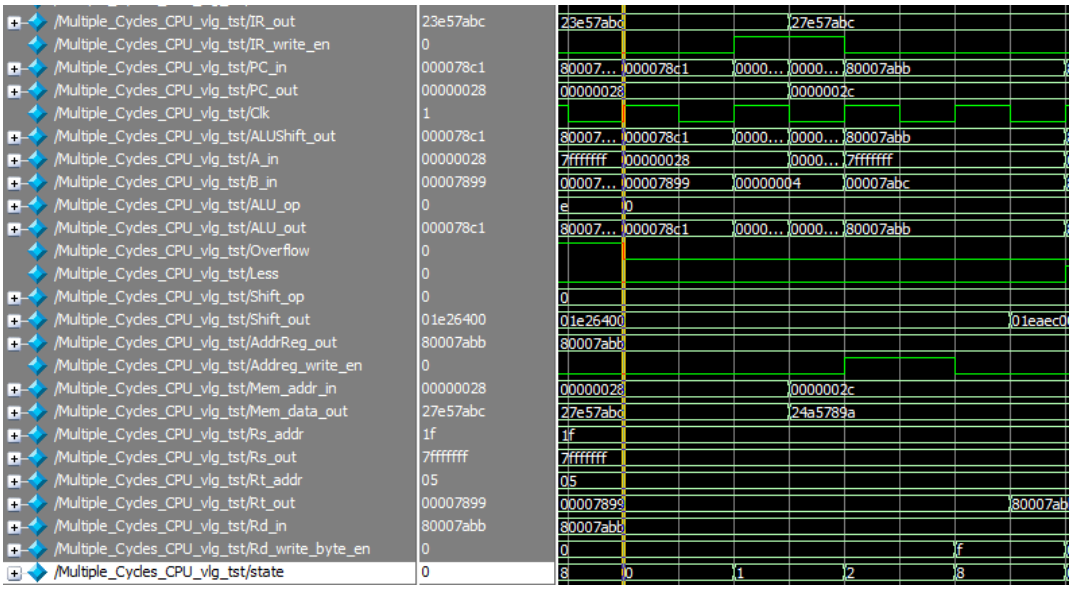
指令为立即数加操作，从指令中获得源操作数寄存器号为$5，目的寄存器为$5，初始 PC 为 0x20，顺序执行，第一周期内从内存取指，在第二个周期内利用 ALU 进行下地址计算，下地址为 0x24;第三个周期，B_in 通过选择器选择 Rt_out 作为输入，Rt_out = 789a,Rs_out = ffffffff, ，由 func 字段判断为带溢出加，经 ALU 计算后，ALU_out 输出，ALUShiftSel 输出选择 ALU_out，并将结果暂存在地址寄存器，第四个周期，通过选择器选择，将结果送到 Rd_in,Rd_in = AddrReg_out = 7899，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，结果没有溢出，写使能有有效（Rd_write_byte.. = fh），结果写入。
从 Rt 字段显示的寄存器情况可以看到，结果在下降沿时，正确的写入到寄存器里。

5.10 Addi $31 Immediate $5



指令为立即数加操作，从指令中获得源操作数寄存器号为$5，目的寄存器为$5，初始 PC 为 0x24，顺序执行，第一周期内从内存取指，在第二个周期内利用 ALU 进行下地址计算，下地址为 0x28; 第三个周期，通过选择器选择符号拓展后的立即数作为输入 B_in = 789a,Rs_out = 7fffffff，，由 func 字段判断为带溢出加，经 ALU 计算后，ALU_out 输出，ALUShiftSel 输出选择 ALU_out，并将结果暂存在地址寄存器，第四个周期，通过选择器选择，将结果送到 Rd_in,Rd_in = AddrReg_out，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，结果有溢出，写使能无效（Rd_write_byte.. = fh），结果未写入。

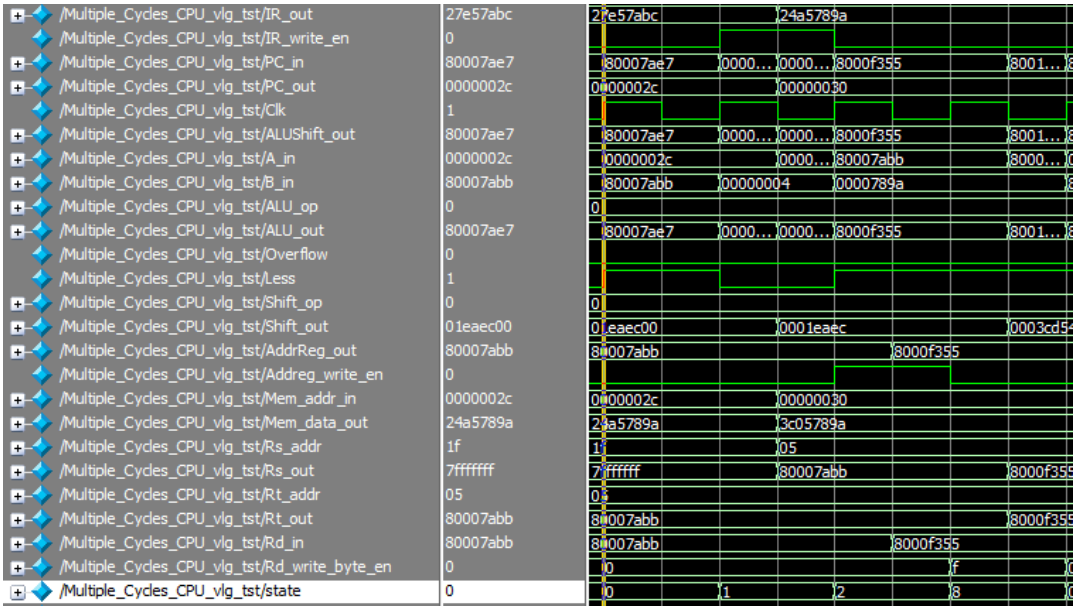
5.11 Addiu $31 Immediate $5



指令为无符号立即数加操作，从指令中获得源操作数寄存器号为$31，目的寄存器为$5，初始 PC 为 0x28，顺序执行，第一周期内从内存取指，在第二个周期内利用 ALU 进行下地址计算，下地址为 0x2c; 第三个周期，通过选择器选择符号拓展后的立即数作为输入 B_in = 789a,Rs_out = 7fffffff，，由 func 字段判断为不带溢出加，经 ALU 计算后，ALU_out 输出，ALUShiftSel 输出选择 ALU_out，并将结果暂存在地址寄存器，第四个周期，通过选择器选

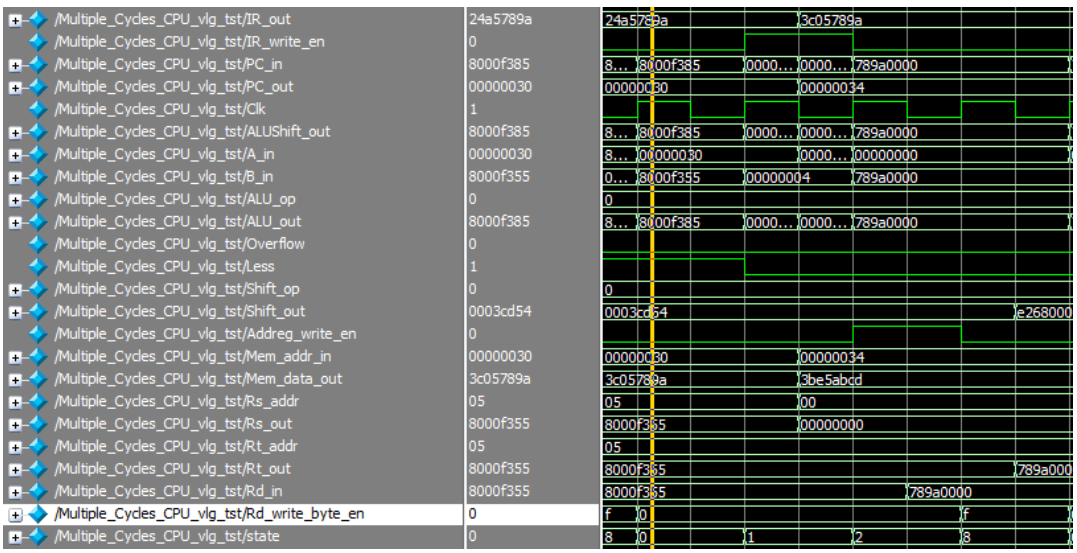
择，将结果送到 Rd_in,Rd_in = AddrReg_out = 80007abb，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，结果无溢出，写使能有效（Rd_write_byte.. = fh），结果写入。

5.12 Addiu $31 Immediate $5



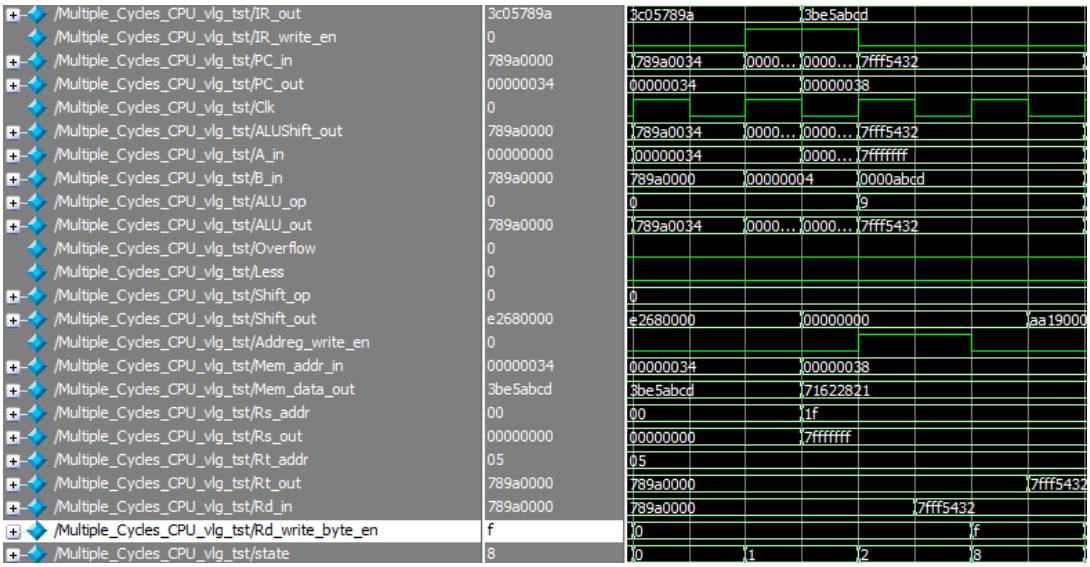
指令为无符号立即数加操作，从指令中获得源操作数寄存器号为$5，目的寄存器为$5，初始 PC 为 0x2c，顺序执行，在第二个周期内利用 ALU 进行下地址计算，下地址为 0x30; 第三个周期，通过选择器选择符号拓展后的立即数作为输入 B_in = 789a,Rs_out = 80007abb, ，由 func 字段判断为不带溢出加，经 ALU 计算后，ALU_out 输出，ALUShiftSel 输出选择 ALU_out，并将结果暂存在地址寄存器，第四个周期，通过选择器选择，将结果送到 Rd_in,Rd_in = AddrReg_out = 8000f355，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，结果无溢出，写使能有效（Rd_write_byte.. = fh），结果写入。

5.13 Lui Immediate $11 $5



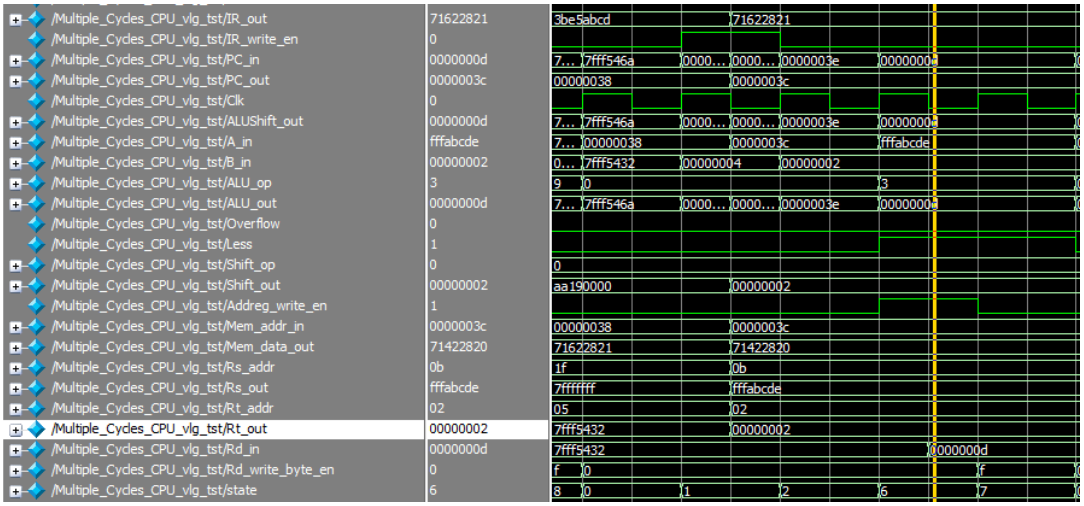
指令为立即数放至高位操作，从指令中目的寄存器为$5，初始 PC 为 0x30，顺序执行，第一周期内从内存取指，在第二个周期内利用 ALU 进行下地址计算，下地址为 0x34; 第三个周期，通过选择器选择拓展后的立即数作为输入 B_in = 7abc0000, ，由 Alu_op 字段判断为不带溢出加，经 ALU 计算后，ALU_out 输出，ALUShiftSel 输出选择 ALU_out，并将结果暂存在地址寄存器，第四个周期，通过选择器选择，将结果送到 Rd_in,Rd_in = AddrReg_out = 789a0000，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，写使能有效（Rd_write_byte.. = fh），结果写入。

5.14 Xori $31 Immediate $5



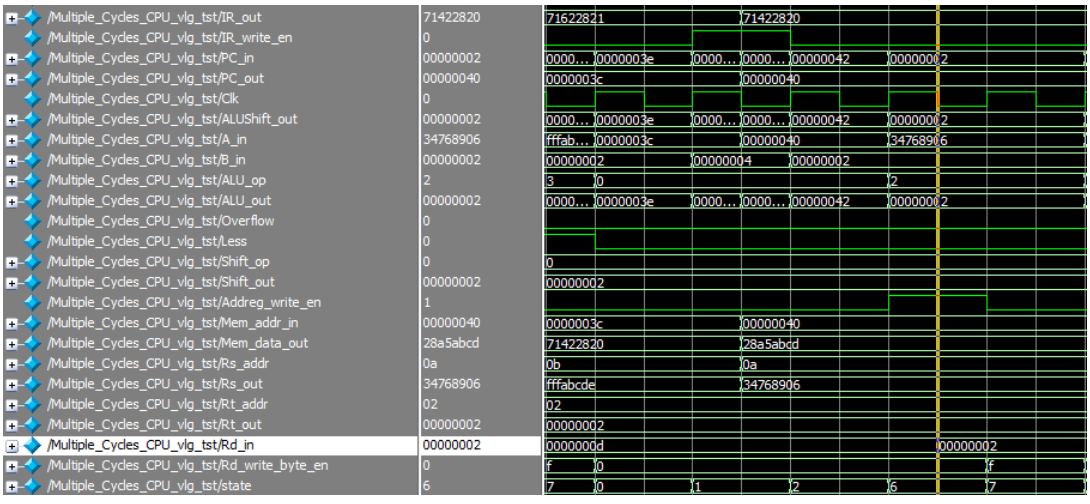
指令为立即数加操作，从指令中获得源操作数寄存器号为$5，目的寄存器为$5，初始 PC 为 0x34，顺序执行，第一周期内从内存取指，在第二个周期内利用 ALU 进行下地址计算，下地址为 0x38; 第三个周期，通过选择器选择符号拓展后的立即数作为输入 B_in = abcd,Rs_out = 7fffffff ，由 func 字段判断为带溢出加，经 ALU 计算后，ALU_out 输出，ALUShiftSel 输出选择 ALU_out，并将结果暂存在地址寄存器，第四个周期，通过选择器选择，将结果送到 Rd_in,Rd_in = AddrReg_out = 7fff5432，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，写使能有效（Rd_write_byte.. = fh），结果写入。

5.15 Clo $11 $5



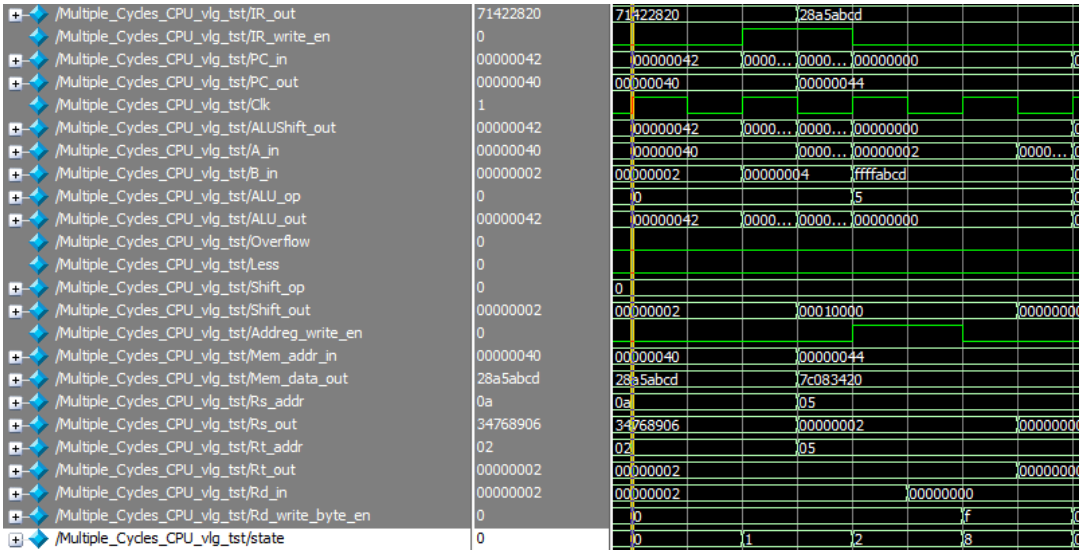
指令为前导 1 操作，从指令中获得源操作数寄存器号为$11，目的寄存器为$5，初始 PC 为 0x38，顺序执行，在第二个周期内利用 ALU 进行下地址计算，下地址为 0x3c;第三个周期译码并通过选择器选择 Rs 输入 B_in = fffabcde，第四个周期，经 ALU 计算后，ALU_out 输出， ALUShiftSel 输出选择 ALU_out，并将结果暂存在地址寄存器，第五个周期，通过选择器选择，将结果送到 Rd_in,Rd_in = AddrReg_out = 13，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，写使能有效（Rd_write_byte.. = fh），结果写入。

5.16 Clz $10 $5



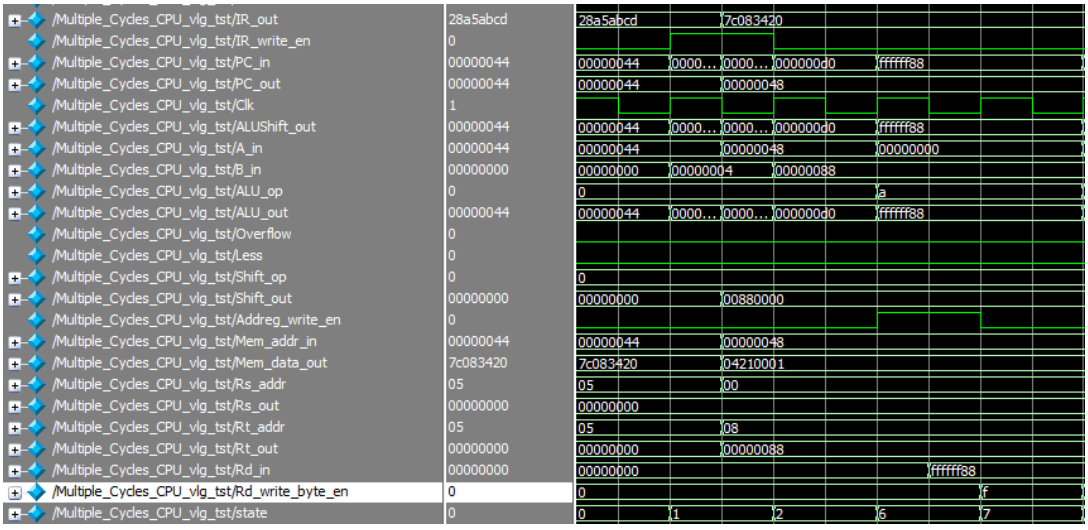
指令为前导 0 操作，从指令中获得源操作数寄存器号为$11，目的寄存器为$5，初始 PC 为 0x3c，第一周期内从内存取指，第二个周期，将取出的指令送指令寄存器，默认顺序执行，在第二个周期内利用 ALU 进行下地址计算，下地址为 40;第三个周期进行译码和寄存器取，通过选择器选择 Rs 输入 B_in =34678932，第四个周期，经 ALU 计算后，ALU_out 输出，ALUShiftSel 输出选择 ALU_out，并将结果暂存在地址寄存器，第五个周期，通过选择器选择，将结果送到 Rd_in,Rd_in = AddrReg_out = 2，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，写使能有效（Rd_write_byte.. = fh），结果写入。

5.17 Slti $9 $10 $11



指令为带符号立即数比较操作，从指令中获得源操作数寄存器号为$5，目的寄存器为$5，初始 PC 为 0x40，顺序执行，第一周期内从内存取指，在第二个周期内利用 ALU 进行下地址计算，下地址为 0x44; 第三个周期，B_in 通过选择器选择经符号拓展的立即数作为输入 B_in = （Ffff）abcd， Rs_out = 2, ，由 func 字段判断为 Slti，经 ALU 计算后，ALU_out 输出，ALUShiftSel 输出选择 ALU_out，并将结果暂存在地址寄存器，第四个周期，通过选择器选择，将结果送到 Rd_in,Rd_in = AddrReg_out = 0000000，因为 2 > ffffabcd，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，写使能有有效（Rd_write_byte.. = fh），结果写入。

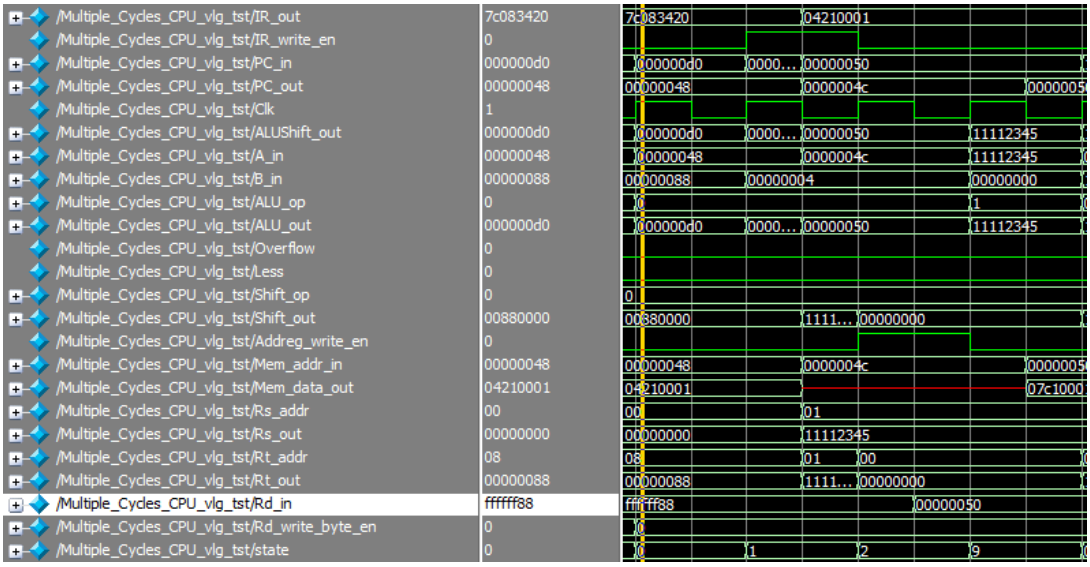
5.18 Seb Immediate $5



指令为低字节位拓展操作，从指令中获得源操作数寄存器号为$8，目的寄存器为$6，初始 PC 为 0x44，第一周期内从内存取指，第二个周期，将取出的指令送指令寄存器，默认顺序执行，在第二个周期内利用 ALU 进行下地址计算，下地址为 12;第三个周期进行译码和寄存器取，B_in 通过选择器选择 Rt_out 作为输入,B_in = Rs_out = 0x88, ，由 func 字段判断为 Seb，经 ALU 计算后，ALU_out 输出，ALUShiftSel 输出选择 ALU_out，并将结果送到

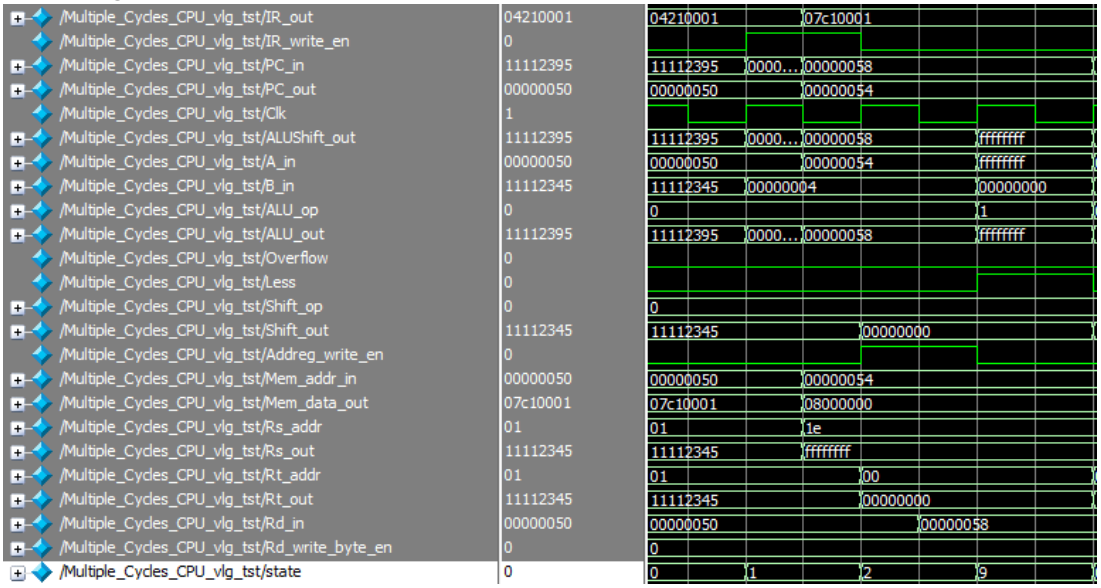
Rd_in,Rd_in = ALU_out = ffffff88，根据使能信号，判断写入与否，根据 Rd_addr 判断写入的位置，写使能有有效（Rd_write_byte.. = fh），结果写入。

5.19 Begz $1 $0



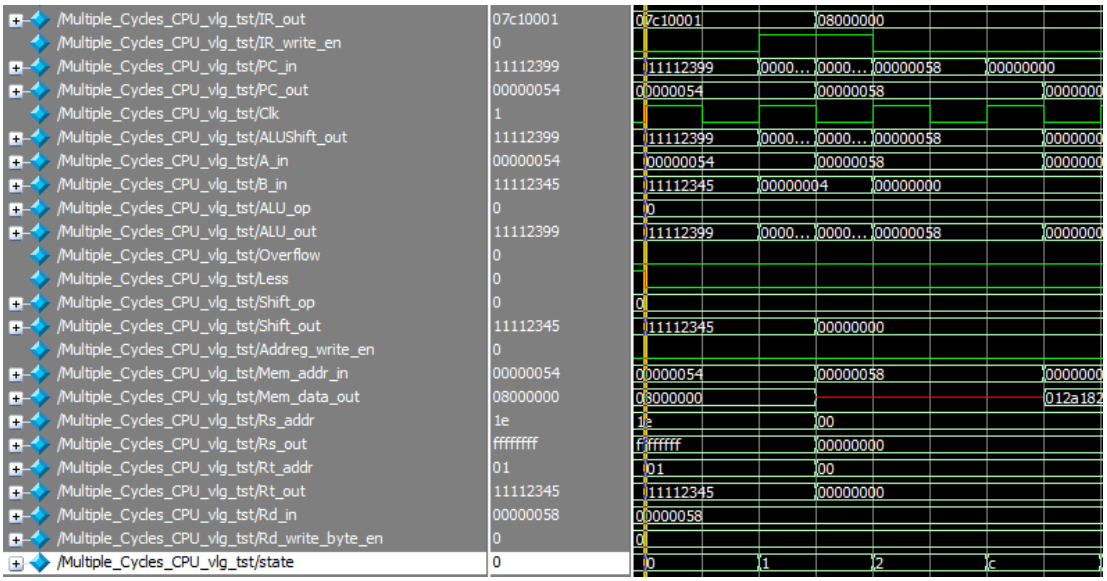
指令为分支指令，从指令中获得源操作数寄存器号为$1，初始 PC 为 0x48，跳转执行，在第二个周期内利用 ALU 进行下地址计算，下地址为 0x50;恰好跳过一条指令，该指令没有初始化，条件判断有 condition 决定判断类型，由 ALU 计算出的信号 Zero 与 Less 来决定条件是否成立。因为 1>=0,条件成立，成功转移。

5.20 Begz $30 $0



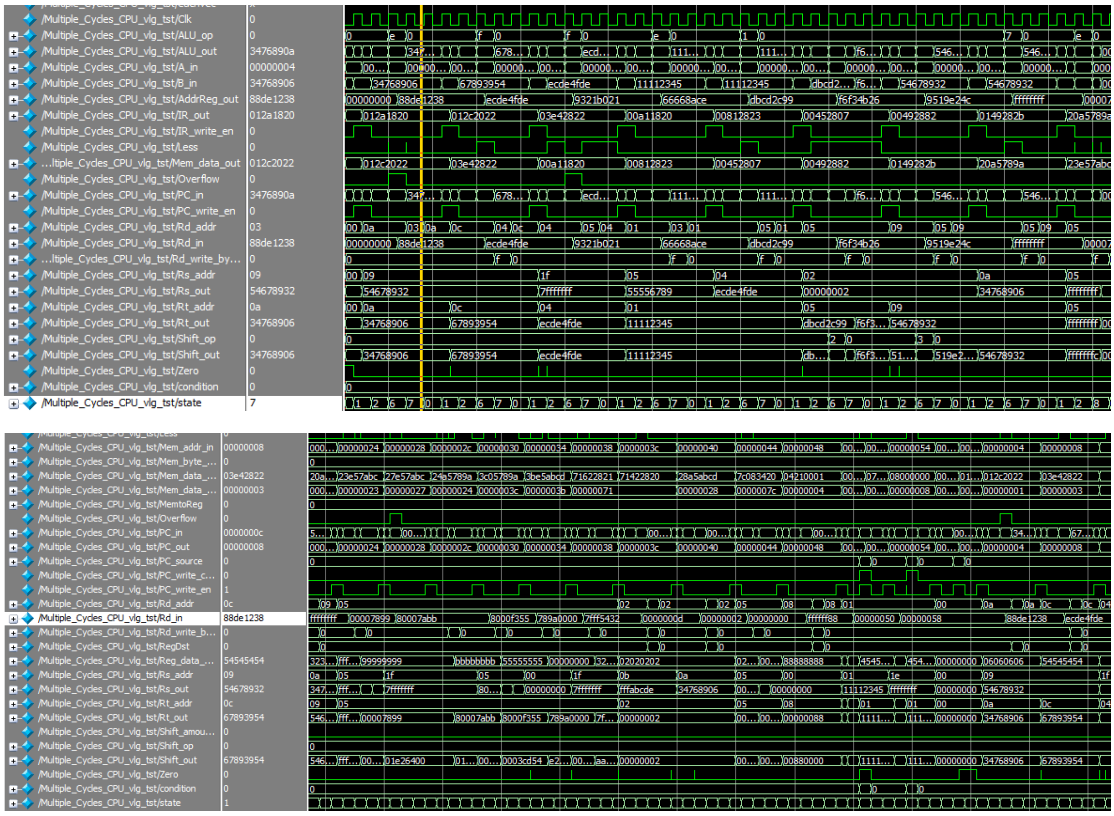
指令为分支指令，从指令中获得源操作数寄存器号为$1，初始 PC 为 0x50，跳转执行，在第二个周期内利用 ALU 进行下地址计算，下地址为 0x54;条件判断有 condition 决定判断类型，由 ALU 计算出的信号 Zero 与 Less 来决定条件是否成立。因为-1 < 0,条件成立，没成功转移。 所以相当于走下地址 0x54。

5.21 J target



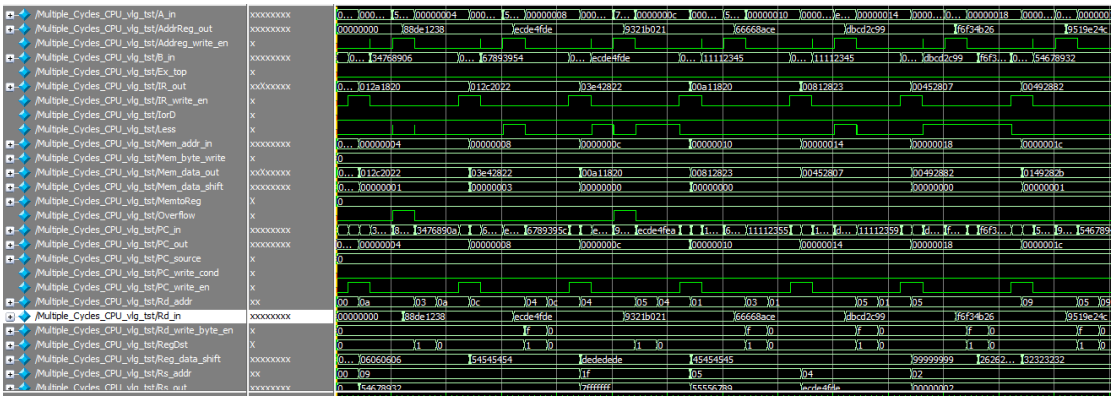
指令为无条件跳转指令，从指令中获得源操作数寄存器号为$1，初始 PC 为 0x54，跳转执行，经 2 选 1 地址选择，下地址为 0x00;转移地址由指令中的 26 位数字决定，地址高 4 位为原来 PC 的高四位，第二位补 0，送到 PC_in,等待时钟信号来更新 PC。 一旦上升沿到来，PC 被更新，然后就实现了无条件跳转指令。

时序仿真截图：

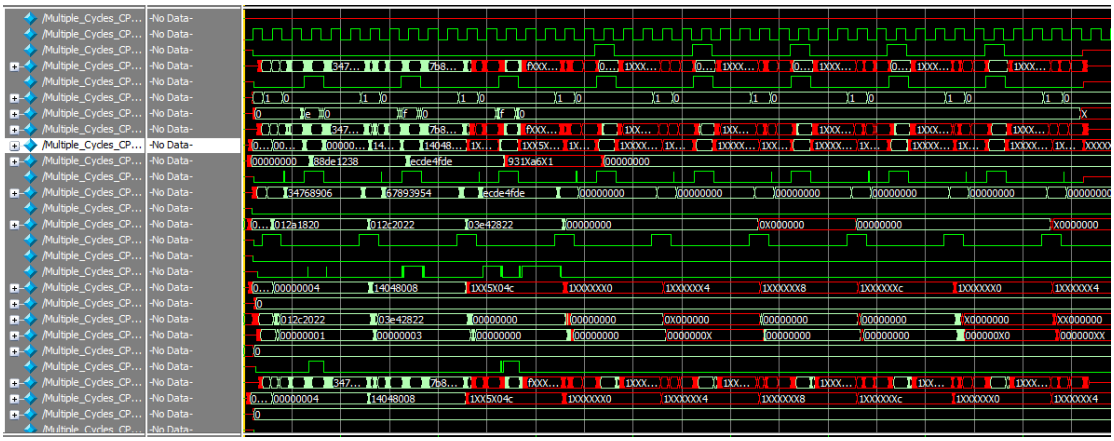


时序仿真将结果正确。

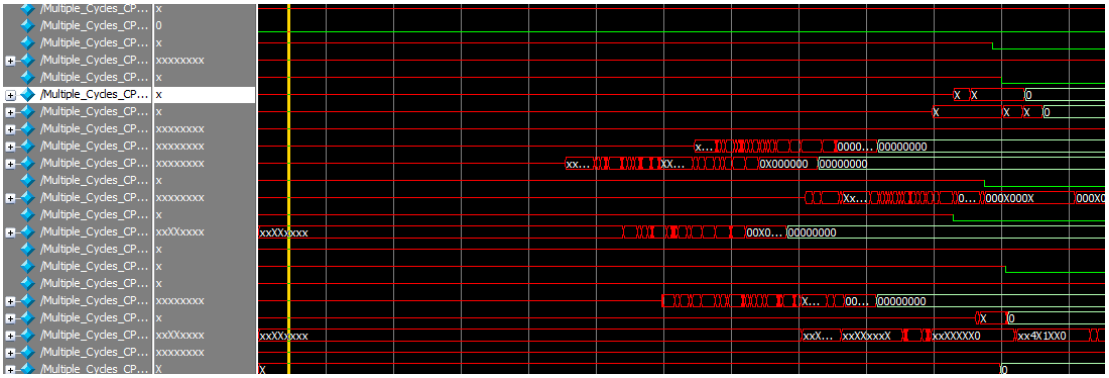
50Mhz 下的仿真结果：可以明显看到毛刺，但结果正确。



500Mhz 下的仿真结果：结果不正确。



毛刺现象：



测试序列二：

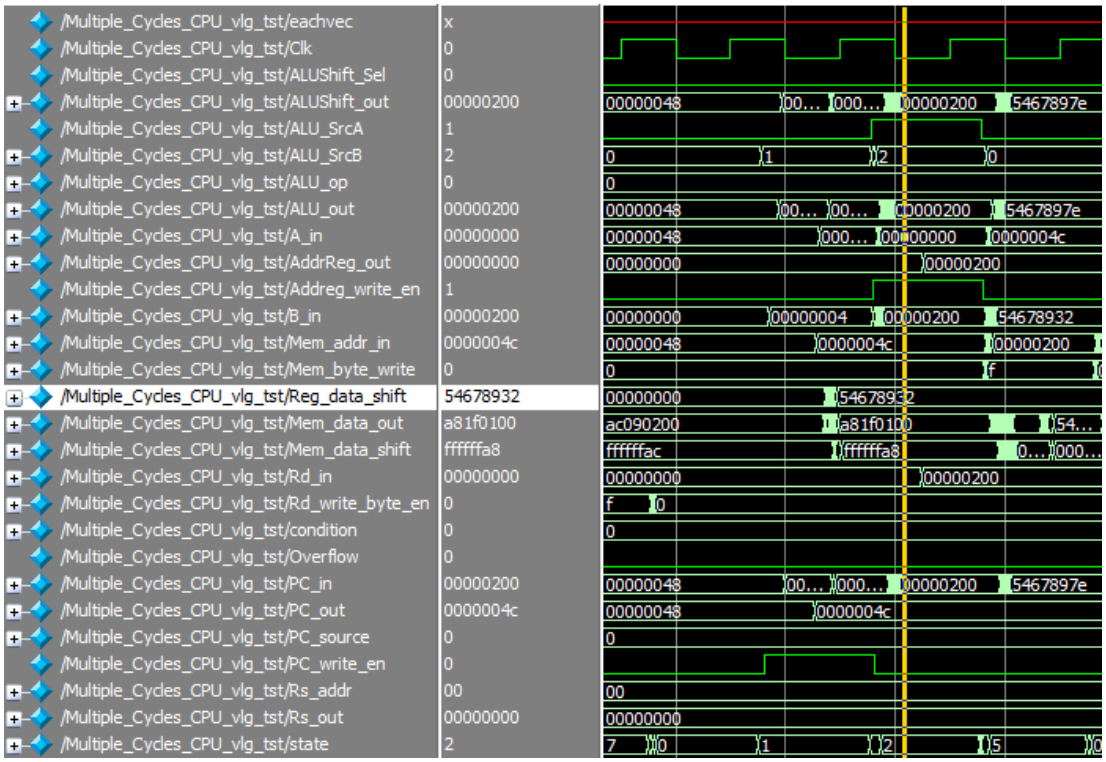
```
sw $9,512($0)
swl $31,256($0)
lw $3,512($0)
lwl $3,513($0)
lwl $3,514($0)
lwl $3,515($0)
lwr $3,513($0)
lwr $3,514($0)
lwr $3,515($0)
swl $31,513($0)
```

```
swl $31,514($0)
swl $31,515($0)
swr $31,513($0)
swr $31,514($0)
swr $31,515($0)
```

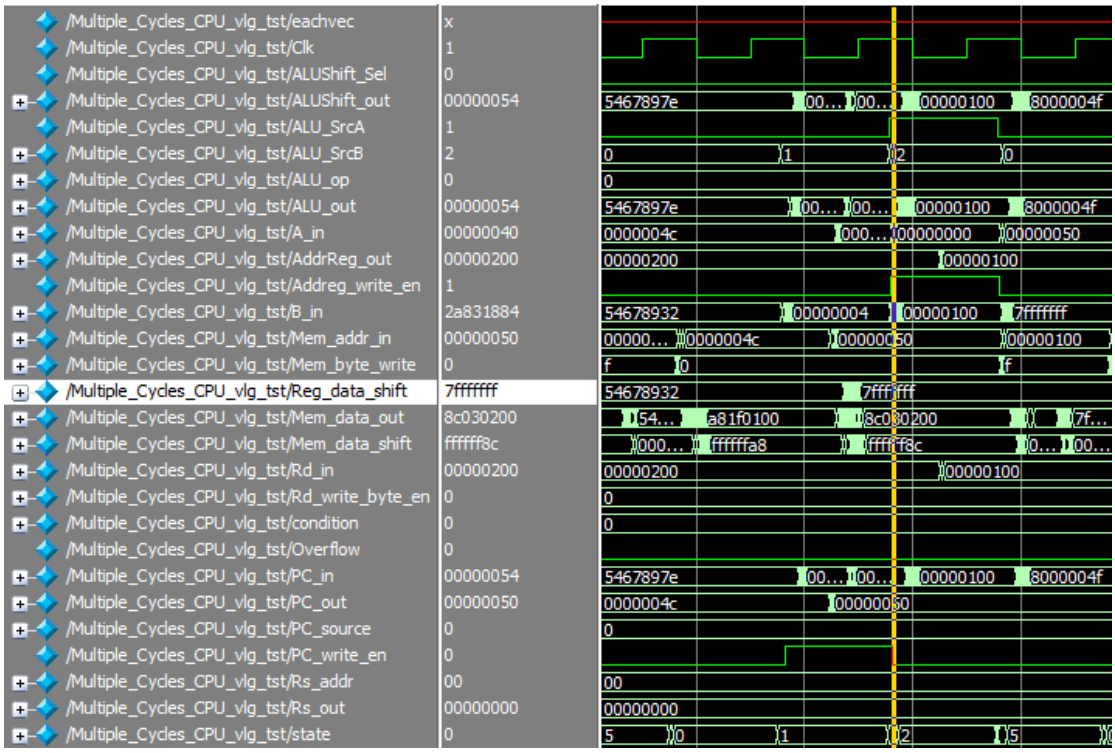
说明：将 9#寄存器内的值存入地址为 512 的内存，存入值为 54678932，将 31#寄存器的值存入地址为 512 的内存地址，存入值为 7fffffff。之后测试 lw 指令，观察正确与否。最后几条指令，测试 sw 系列指令。

功能仿真结果：

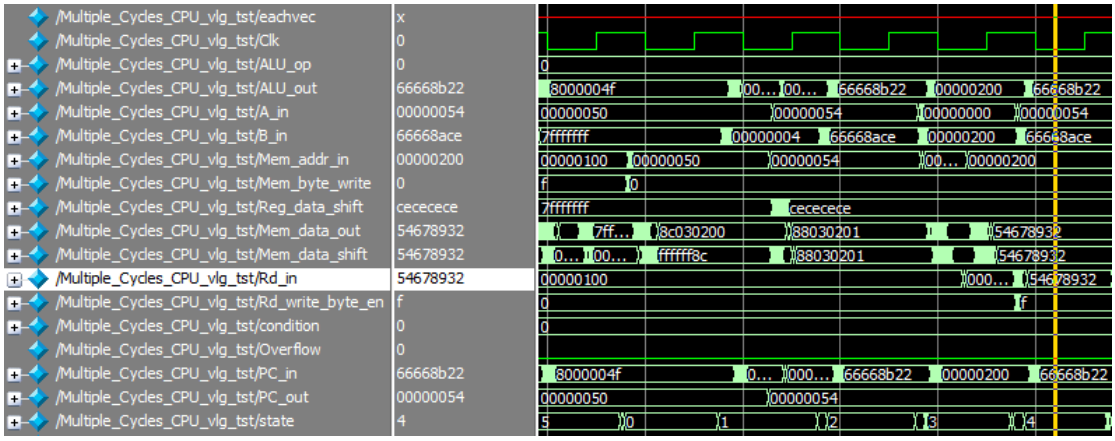
sw $9,512($0)，Reg_Data_Shift 为 Mem_Data_in，根据这一信号值就可以判定存入数据是否正确。同时注意内存写使能是否有效，有效的状态下，才可以写入。



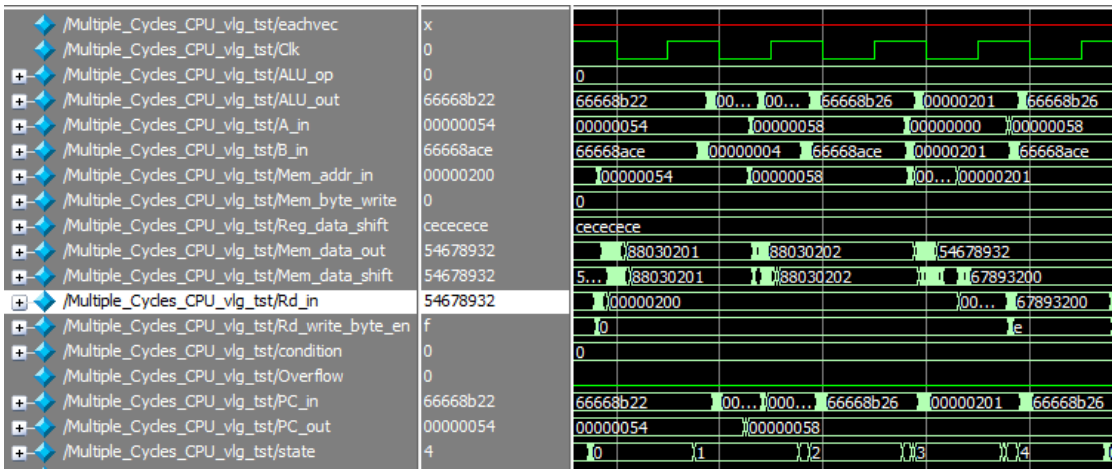
swl $31,256($0)，Reg_Data_Shift 为 Mem_Data_in，根据这一信号值就可以判定存入数据是否正确。同时注意内存写使能是否有效，有效的状态下，才可以写入。



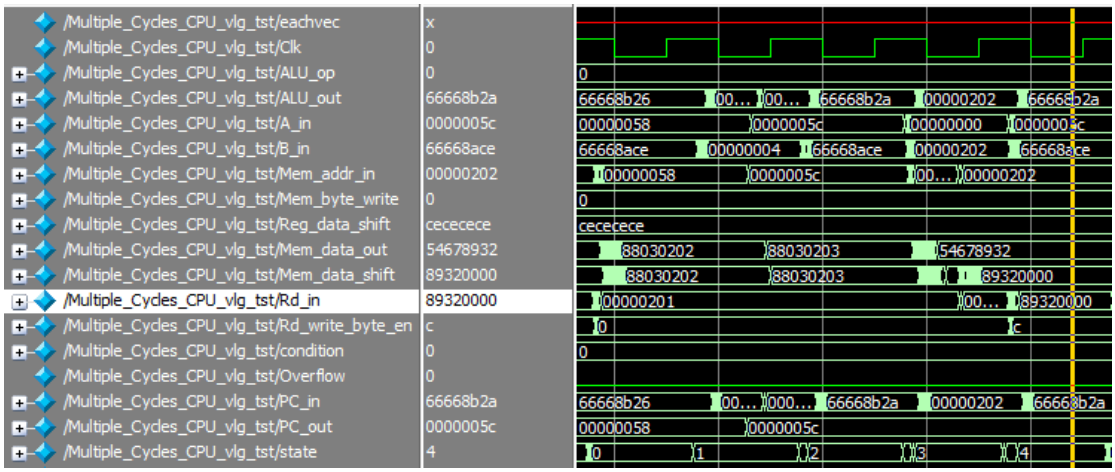
lw $3,512($0)，同时测试是否正确写入，末位为 00，读出为 54678932，正确。



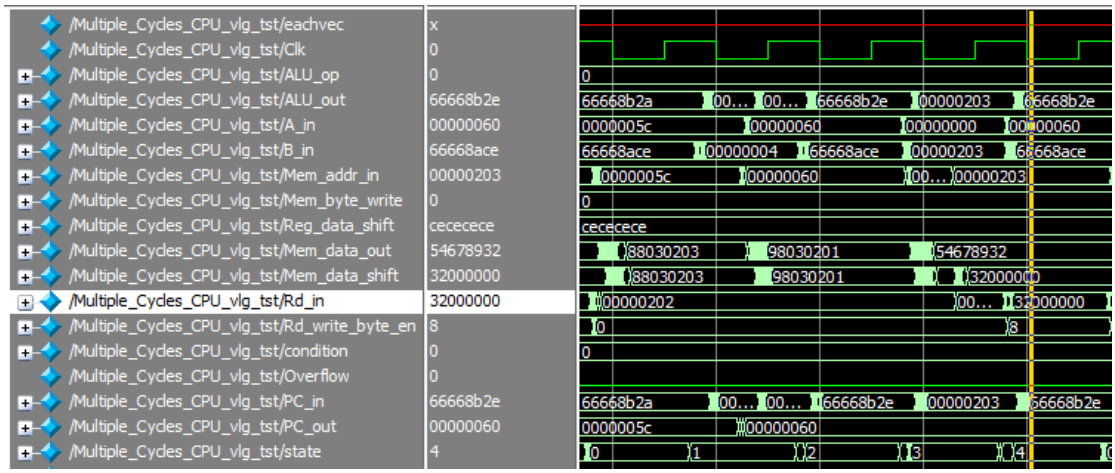
lwl $3,513($0)，末位为 01，读出为 67893200，正确。



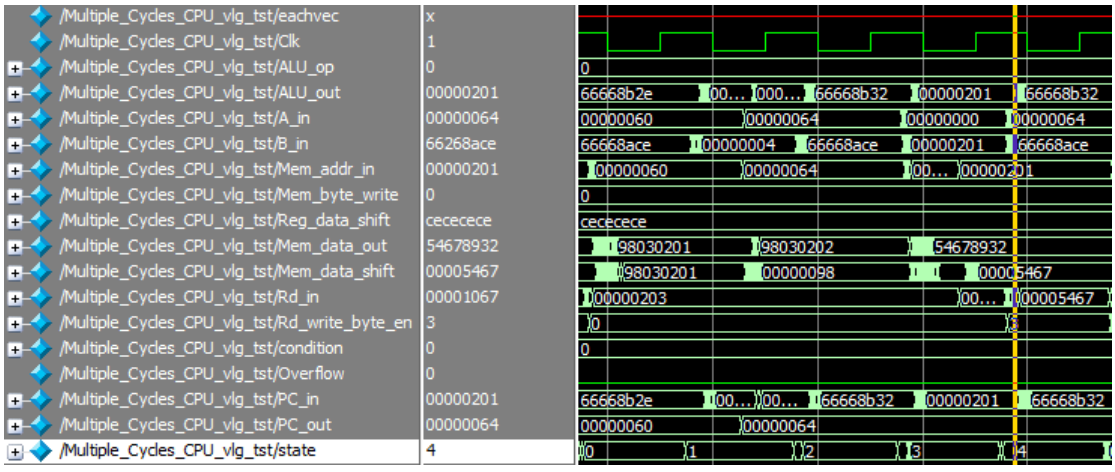
lwl $3,514($0)，末位为 10，读出为 89320000，正确。



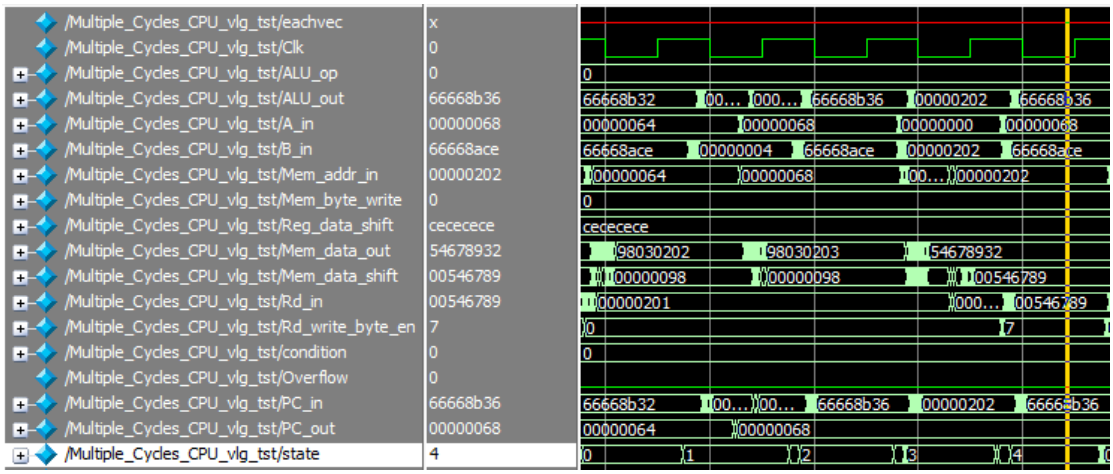
lwl $3,515($0)，末位为 11，读出为 32000000，正确。



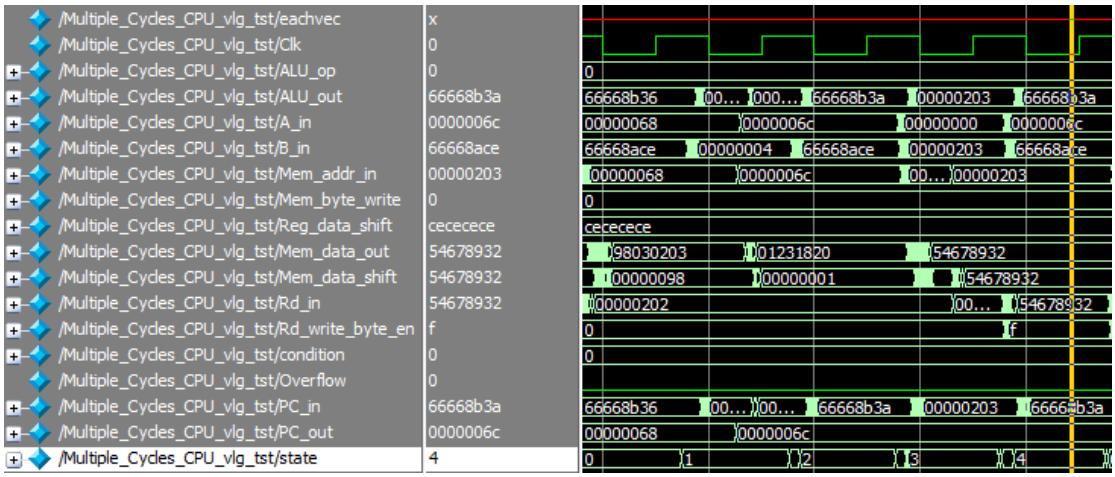
lwr $3,513($0)，末位为 01，读出为 00005467，正确。



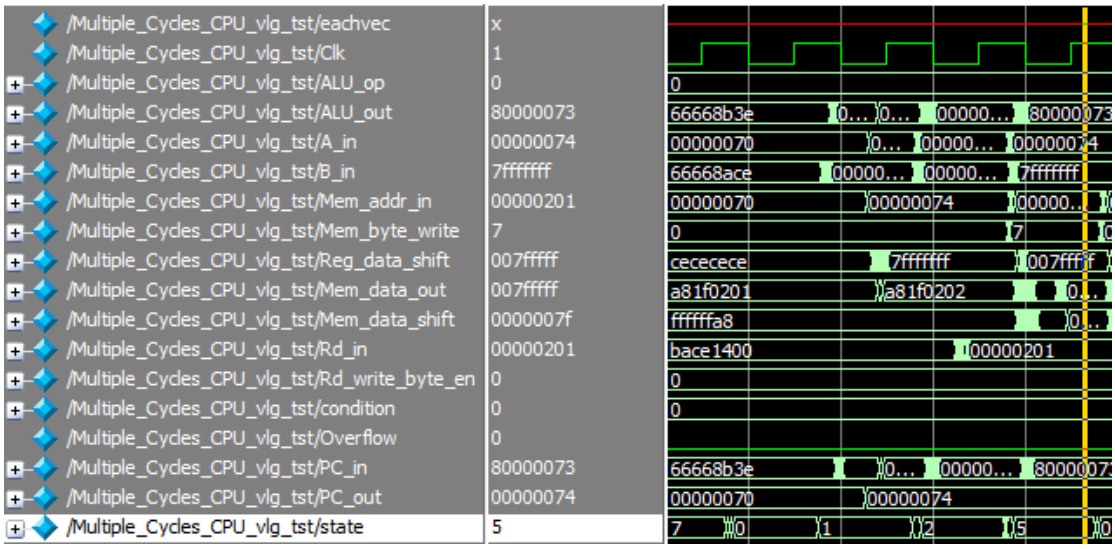
lwr $3,514($0)，末位为 10，读出为 00546789，正确。



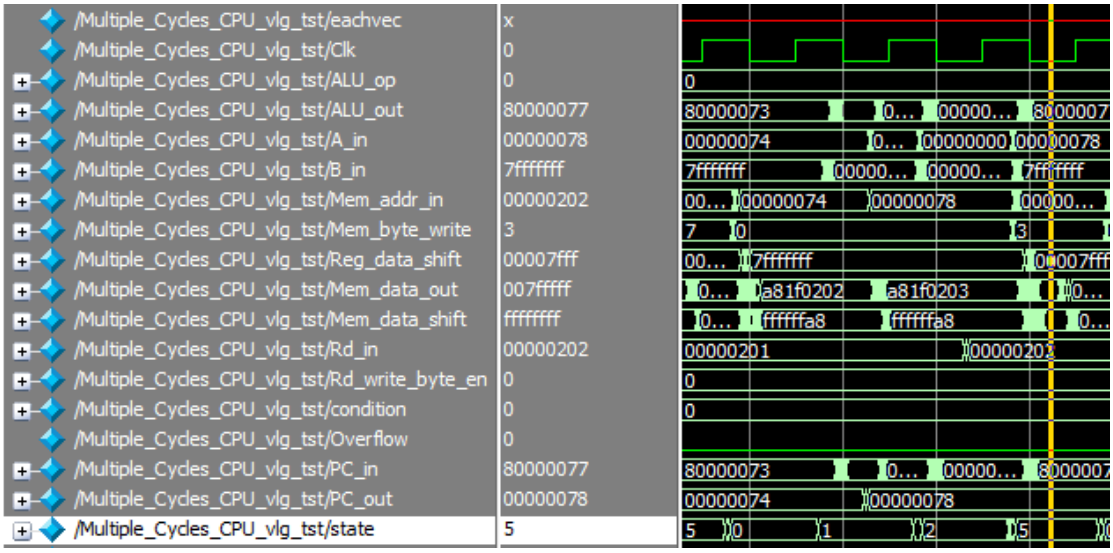
lwr $3,515($0)，末位为 11，读出为 54678932，正确。



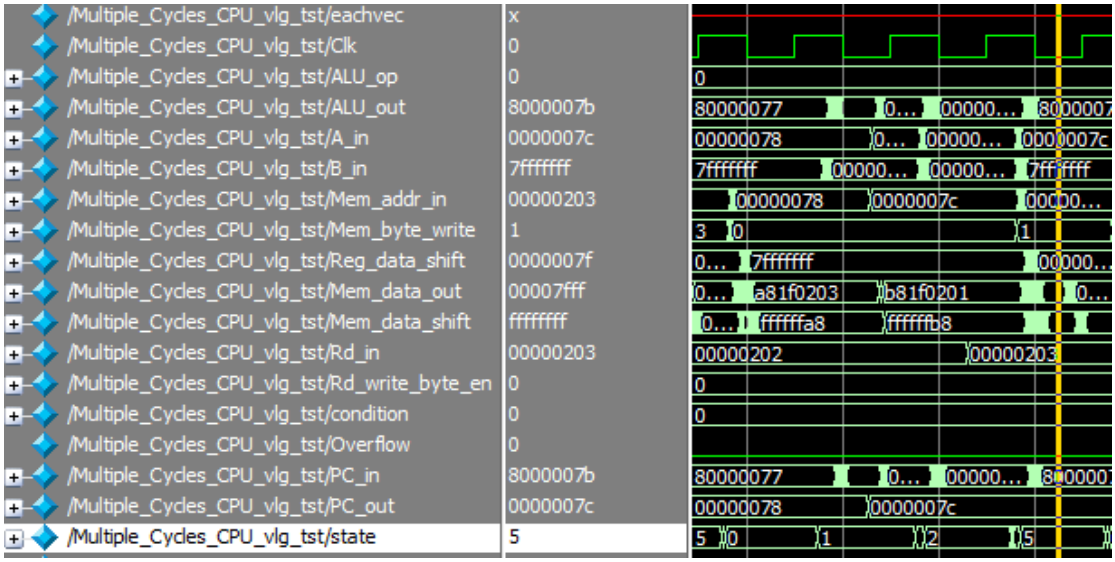
swl $31,513($0)，末位为 01，存入为 007fffff，正确。



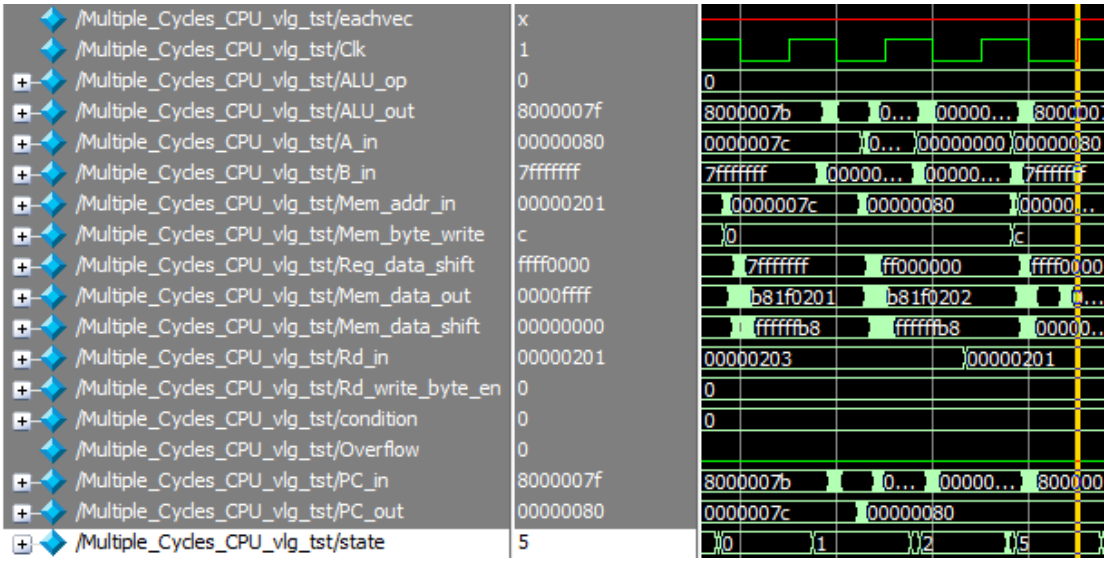
swl $31,514($0) ，末位为 10，存入为 00007fff，正确。



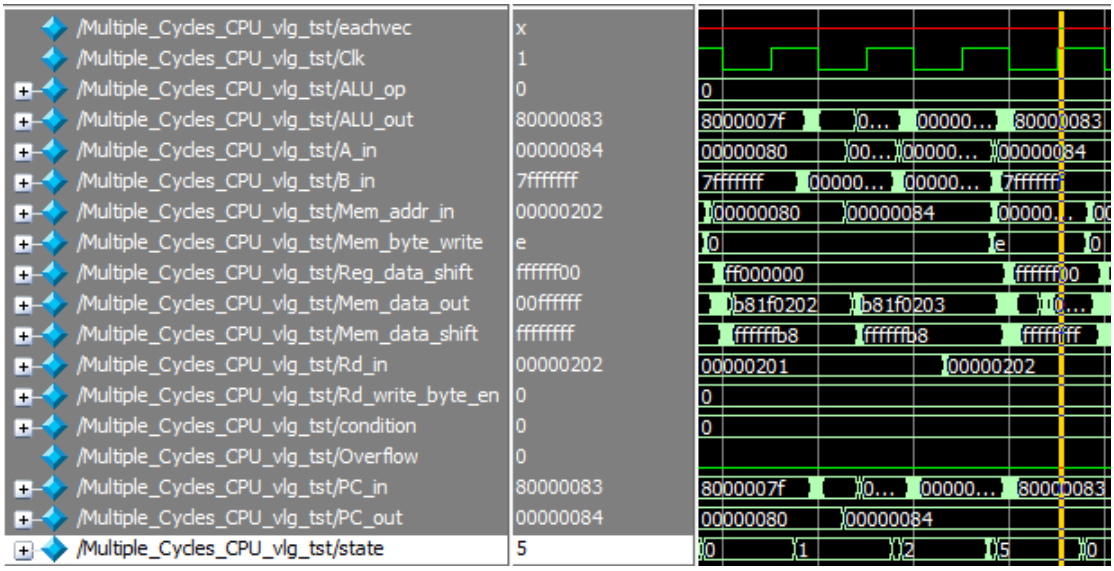
swl $31,515($0)，末位为 11，存入为 0000007f，正确。



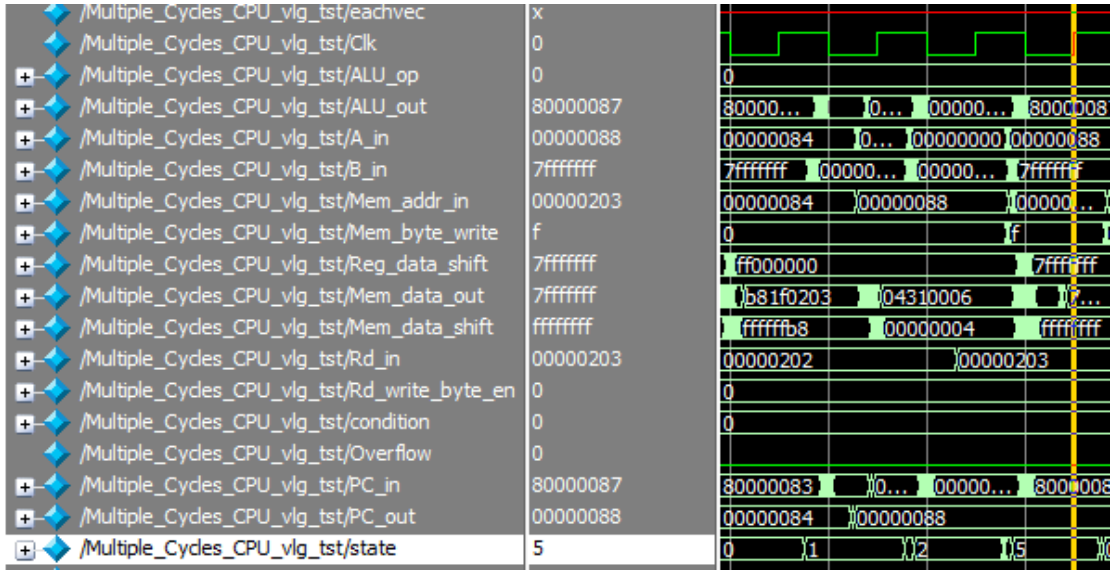
swr $31,513($0)，末位为 01，存入为 ffff0000，正确。



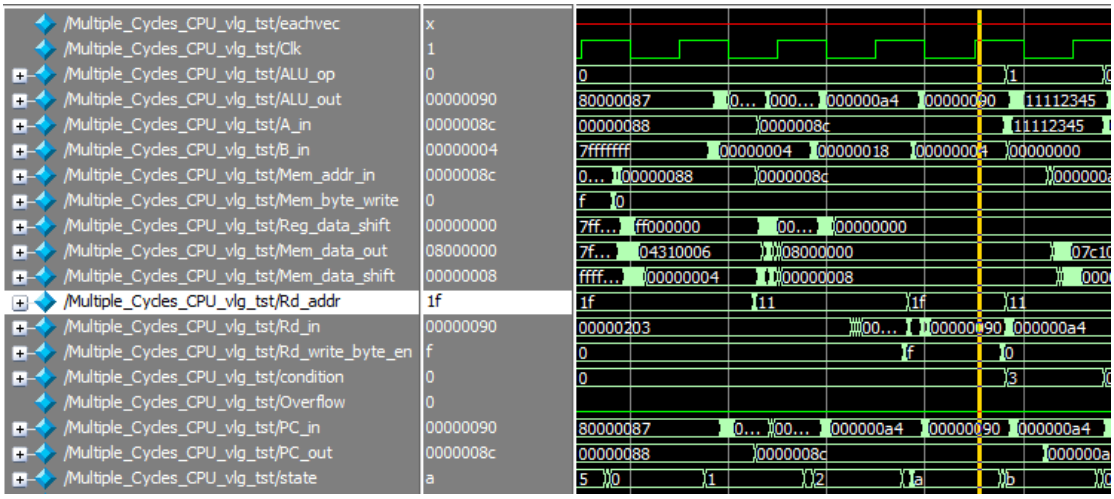
swr $31,514($0) ，末位为 10，存入为 fffff00，正确。



swr $31,515($0) ，末位为 11，存入为 7fffffff，正确。



bgezal $31 1



说明：从结果上看出，下地址 8 被存入到$31 号寄存器，而下条指令地址为 0xc 符合预期要

求。

时序仿真的结果：按照设计进行运转，处理器的最大主频可以达到 50MHz，当调整为 100Mhz 时就会出现不可预期的错误。此多周期处理器有进一步优化的可能。

6. 编写简单程序的测试

6.1 计算 1~100 自然数之和

6.1.1 汇编代码：

add $2 $3 -> $3;

sub $2 $1 -> $2;

bgez $2;

6.1.2 指令代码：

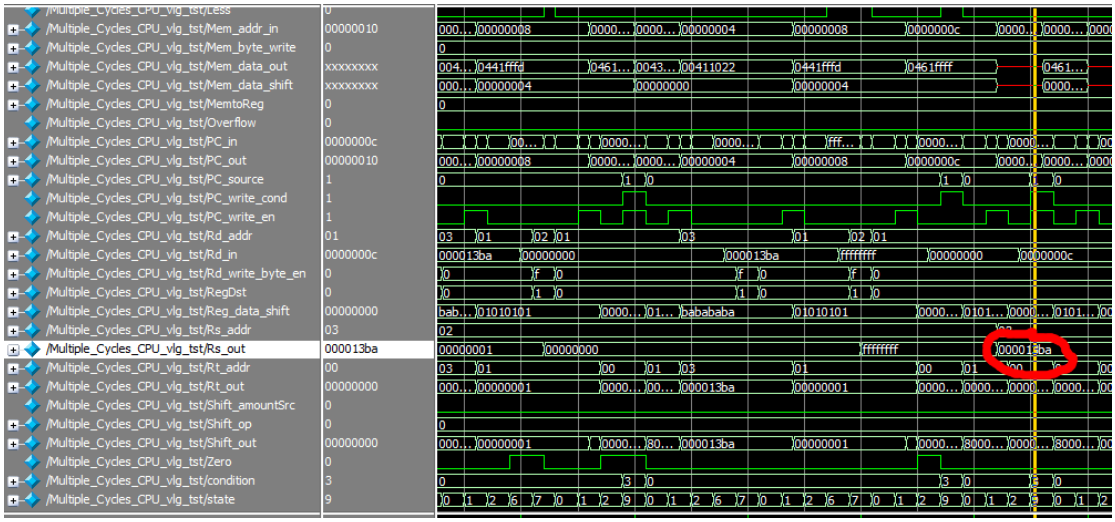
Instruc_Sto[1] = 32'b000000_00010_00011_00011_00000_100000;//2# + 3# ->3#

Instruc_Sto[2] = 32'b000000_00010_00001_00010_00000_100010;//2# - 1#

Instruc_Sto[3] = {16'b000001_00010_00010,16'hfffd};//begz 2# > 0

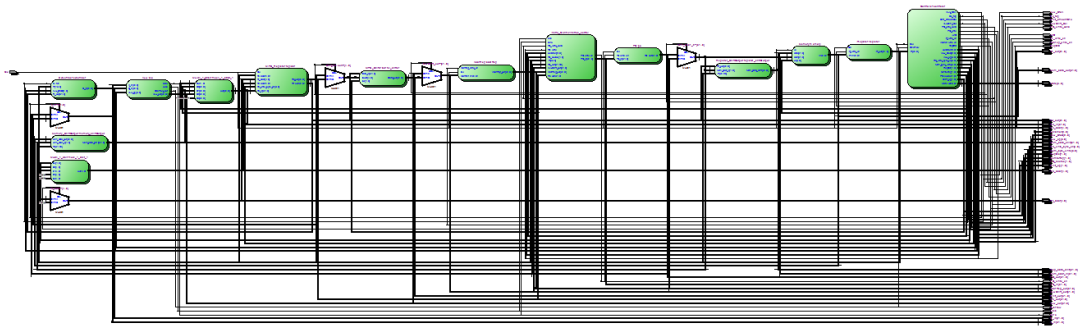
Instruc_Sto[4] = {16'b000001_00011_00011,16'hffff};//begz 3# > 0，无实际意义，只是为了方便查看结果。

6.1.3 结果



加和为 0x13ba = 5050，符合要求，结果正确。

7.RTL Viewer 结果



五、 实验总结

1 问题分析与解决

1.1 2 –> 7，6–>7 周期 I,R 型 Overflow 位的维持

对于功能仿真，只需要维持上一阶段的信号不做修改就可以正常执行，但是在时序仿真时就出现了问题，因为在状态跳转时，如果并不对其赋初值则默认为 0，即 Overflow 不能按照功能仿真时那样保持，从而无法实现根据 Overflow 判写入与否。

采用的方法是，设置一个 Overflow 的寄存器将其存储供下一个状态使用。这样就可以解决这一问题。

1.2 寄存器读写问题

单周期实验报告迟迟后推，主要是时序仿真总是出现不了正确结果，我去翻阅了一下上学期数电大实验的字模初始化模块，数字部分的字模采用的也是 initial 语句来初始化的，这就排除了 initial 语句初始化不能在时序仿真时使用的问题。最后在多周期时序仿真时发现同样的问题，可是发现指令存储器内容还是可以读出指令的，于是比较了一下 Memory 和 Register 模块的区别，发现只是赋初值的位置不同，我将寄存器中的 initial 语句放到了 always 语句块后面，先进行全编译，发现警告中的寄存器输出持续为 GND 的警告被消除，再进行时序仿真，立即得到正确的结果。这似乎与 initial 语句块和 always 语句块并行执行的传统观点有悖。

再次仔细观察，虽然赋值进去，但发现部分赋值不正确，且改变 initial 块的值并不会对错误值有任何影响，且有一部分寄存器写入时出现错误。最后采用 case 语句的方法来进行判断写使能信号，得到的结果是正确的。个人猜测是否因为 4 个 if 语句串行执行造成的错误。

1.3 Lw，Sw 类指令的写使能问题

Lwr，Lwl，Swr，Swl 指令只会修改部分字节，所以要针对不同指令来确定写使能的不同，最初设计时忽略了对写使能的控制，完全定为 4‘b1111，这与 MIPS 手册中所定义的指令功能不符。对于指令，以后要认真研读手册，然后再进行设计，以避免此类错误再次出现。

1.4 CPU 第一条指令的正确执行

只设置 PC == 0 在单周期内可以正确执行，但在多周期设计中感觉并不是很合理，于是加入了一个新的起始状态 13，来表示起始状态。

1.5 本次实验中状态机的优化

本次实验 R 型指令使用了 5 个周期，完全可以优化为 4 个周期来完成，这对于多周期 CPU 的性能提升还是很大的。观察到第 2 状态中 R 型指令的 ALU 是空闲的，可以进行投机计算，故可以把第 6 个状态合并到第 2 个周期，从而就可以使 R 型指令的周期减少为 4 个周期。

1.6 状态机的设计

有限状态机是由寄存器组和组合逻辑构成的硬件时序电路，其状态（即由寄存器组的 1 和 0 的组合状态所构成的有限个状态）只可能在同一时钟跳变沿的情况下才能从一个状态转向另一个状态，究竟转向哪一状态还是留在原状态不但取决于各个输入值，还取决于当前所在状态。

在 Verilog HDL 中可以用许多种方法来描述有限状态机，最常用的方法是用 always 语句和 case 语句。采用 case,casex,或 casez 语句来建立状态机的模型，语句表达清晰明了，可以方便地从当前状态分支转向下一个状态并设置输出。注意写上 case 语句的最后一个分支 default，并将状态变量设为'bx，这就等于告知综合器：case 语句已经指定了所有的状态，这样综合器就可以删除不需要的译码电路，使生成的电路简洁，并与设计要求一致。

1.7 状态机的书写的注意事项

always 块：

1) 每个always块只能有一个事件控制"@(event-expression)",而且要紧跟在always关键字后面。
2) always块可以表示时序逻辑或者组合逻辑，也可以用always块既表示电平敏感的透明锁存器又同时表示组合逻辑。但是**不推荐**使用这种描述方法，因为这容易产生错误和多余的电平敏感的透明锁存器。
3) 带有posedge 或 negedge 关键字的事件表达式表示沿触发的时序逻辑，没有posedge 或negedge 关键字的表示组合逻辑或电平敏感的锁存器，或者两种都表示。在表示时序和组合逻辑的事件控制表达式中如有多个沿和多个电平，其间必须用关键字 “ or ” 连接 。
4) 每个表示时序always块只能由一个时钟跳变沿触发，置位或复位最好也由该时钟跳变沿触发。
5) 每个在 always 块中赋值的信号都必需定义成 reg 型或整型。
6) always 块中应该避免组合反馈回路。每次执行 always 块时，在生成组合逻辑的 always 块中赋值的所有信号必需都有明确的值；否则，需要设计者在设计中加入电平敏感的锁存器来保持赋值前的最后一个值，只有这样综合器才能正常生成电路。

赋值：

1) 对一个寄存器型(reg)和整型(integer)变量给定位的赋值只允许在一个always块内进行，如在另一always块也对其赋值，这是非法的。
2) 把某一信号值赋为'bx，综合器就把它解释成无关状态，因而综合器为其生成的硬件电路最**简洁**。

2 实验感悟

2.1 进行处理器设计的大致步骤

1.分析每条指令的功能，并用 RTL(Register Transfer Language)来表示。

2.根据指令的功能给出所需的元件，并考虑如何将他们互连。

3.确定每个元件所需控制信号的取值。

4.汇总所有指令所涉及到的控制信号，生成一张反映指令与控制信号之间关系的表。

5.根据表得到每个控制信号的逻辑表达式，据此设计控制器电路。

2.2 对于实验的改进

因为大多数 FPGA 内部的触发器数目相当多，又加上独热码状态机（one hot state machine)的译码逻辑最为简单，所以在设计采用 FPGA 实现的状态机时往往采用独热码状态机(即每个状态只有一个寄存器置位的状态机)。

将其改为独热码将减少译码电路的复杂度。

2.3 再次总结一下阻塞与非阻塞赋值的原则：

原则 1：时序电路建模时，用非阻塞赋值。

原则 2：锁存器电路建模时，用非阻塞赋值。

原则 3：用 always 块写组合逻辑时，采用阻塞赋值。

原则 4：在同一个 always 块中同时建立时序和组合逻辑电路时，用非阻塞赋值。

原则 5：在同一个 always 块中不要同时使用非阻塞赋值和阻塞赋值。

原则 6：不要在多个 always 块中为同一个变量赋值。

原则 7：用$strobe 系统任务来显示用非阻塞赋值的变量值

原则 8：在赋值时不要使用 #0 延迟

综合权衡的话，控制器里用非阻塞赋值最优，其实将无关信号量赋值为‘bx 将会更加有助于综合出最简电路。

六、 实验思考题

多时钟周期 CPU 有什么特点？设计多时钟周期 CPU 与单时钟周期 CPU 有什么不同？应注意哪些问题？

1.1 多时钟周期 CPU 有什么特点

a.每条指令分为若干功能阶段一个时钟周期内完成一个阶段，每个时钟到来时，都跳转到下一个状态。其中前三个状态为公共状态，完成功能类似。

b.在第一个周期内，当始终边沿条件被触发，PC 值作为指令内存地址，从内存中读取出指令，在第二周期内默认顺序执行，PC = PC+4，对于跳转指令类，按几种方式分别计算下条指令地址，在 less / zero /PC_Source 的控制下，由 ALU 计算目标地址（条件转移，第三周投机计算），送到 PC 源选择输入端，根据控制信号，选择正确的输入值，并结合 PC 写使能有效，决定将目标地址写入 PC 中。

几种下址方式为：

顺序执行：PC+4

条件转移： PC + 4 + signExt[imm16]*4

过程调用： PC + 8 + signExt[imm16]*4

无条件跳转指令： PC<31:28>拼接 target<25:0>拼接“00”

c.指令取出后被译码，产生指令对应的控制信号。

d.对多周期 CPU，CPI 多少因指令而异，各种指令 CPI 由所需要执行的周期数决定;时钟周期由执行时间最长的阶段来决定。

e.多周期 CPU 主要由数据通路和控制器组成,数据通路用于实现指令集中所有指令的操作功能；控制器用于控制数据通路中各部件进行正确操作。

f. 适合于且对速度要求不高的场景。

g. 阶段的划分要均衡，每个阶段只能完成一个独立、简单的功能，如：一次 ALU 操作，，一次存储器访问，一次寄存器存取。

h.需加临时寄存器存放指令执行的中间结果。

i.同一个功能部件能在不同的时钟中被重复使用。

j.用有限状态机来表示指令执行流程，并以此设计控制器。

1.2 设计多时钟周期 CPU 与单时钟周期 CPU 有什么不同

1. 对多周期 CPU，CPI 多少因指令而异，各种指令 CPI 由所需要执行的周期数决定;时钟周期由执行时间最长的阶段来决定。对单周期 CPU，CPI = 1;时钟周期由执行时间最长的指令来决定。
2. 多周期 CPU 需加临时寄存器存放指令执行的中间结果,单周期 CPU 则不需要。
3. 多周期 CPU 同一个功能部件能在不同的时钟中被重复使用，最明显的就是下地址计算，单周期中仍然使用单独加法器。而多周期则采用 ALU 计算下地址。
4. 用有限状态机来表示指令执行流程，并以此设计控制器，单周期控制器则只是一个组合逻辑电路。

1.3 应注意哪些问题

针对本次实验，因为设计图已经给出，所以第一步是对各个功能模块的连接和功能进行初步了解，并理解控制信号的的具体意义，分析各模块的时序关系。

然后根据每条指令的功能，分析控制信号的取值，并在表中列出。

对整个设计图进行合理的模块划分，归分出位拓展模块以及下地址模块等，分模块逐步设计。

将各个模块在顶层实体进行连接和组合，注意各连接线的位宽大小。