# 组成原理课程第 六 次实验报告

# 实验名称：多周期 CPU 实验

学号： 2312141 姓名： 张德民 班次： 李涛老师

## 一、 实验目的

在单周期 CPU 实验完成的提前下，理解多周期的概念。
熟悉并掌握多周期 CPU 的原理和设计。
进一步提升运用 verilog 语言进行电路设计的能力。
为后续实现流水线 cpu 的课程设计打下基础。

## 二、 实验内容说明

请根据实验指导手册完成多周期 CPU 实验，并撰写实验总结，要求：
1、多周期 CPU 实验使用同步 IP 核构造 data_ram 和 inst_rom，原始 source_code 中的同名.v 文件和 ngc 文件不要导入到项目。
2、多周期 CPU 运行的指令在 inst_rom 中，这里面的指令须导入 coe 文件。
3、请把 ALU 实验中添加的三个运算，自行定义类似 MIPS 指令格式的指令，把对应的指令和功能增加到多周期 CPU 中，并自行在 coe 文件中添加指令，然后进行运行验证（仿真波形验证或实验箱验证即可）。
4、实验报告中可以不放原理图，关于验证结果的图片（仿真图片或实验箱图片）需要仔细介绍图中的信息和对指令验证的情况。

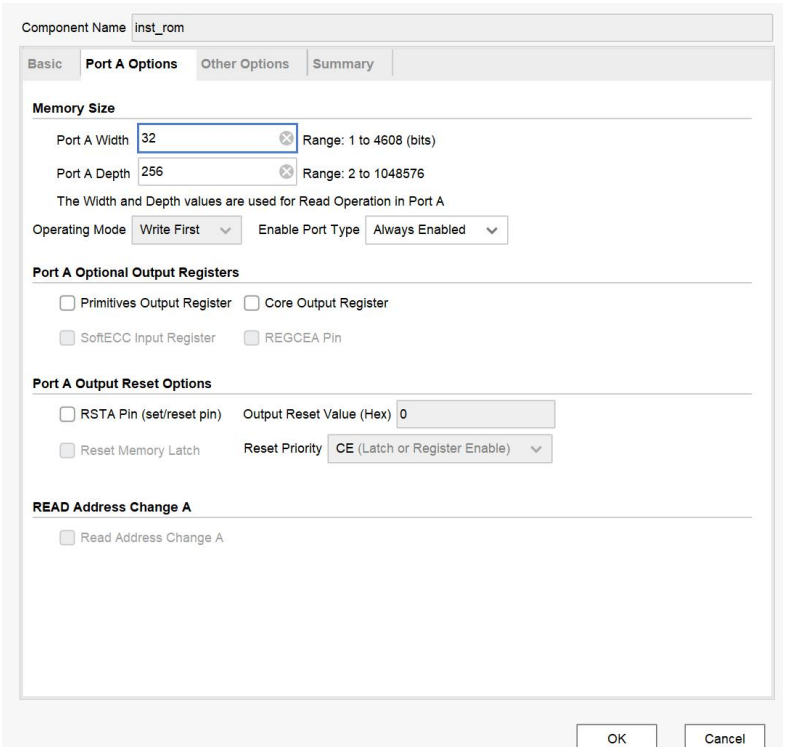
## 三、 实验步骤

### 1. 修改原代码 BUG

在加入老师要求的三个功能之前，我首先需要把原先的代码跑通，我选择和单周期一样的指令，在我配置好环境和代码之后，我先进行了仿真实验，结果发现，代码一到关于跳转的部分就会出错。


如上图，PC=2cH 的时候，是一个 beq 语句，结果应该跳转到 34h，但是我看了 pc 和 next_pc 的值，发现里边多出一段 30h 和一段 3Xh，我又看了一下寄存器相关的值，发现了问题，例如当第二条指令运行完以后，它的结果没有即时写回，而是在第三条指令运行完以后才写回，后边的指令也是如此。

为了改正这个问题，我查了很多资料，最后问了老师，知道了解决的办法。如下图，把两个 IP 核设置里的 primitives output register 的小框取消就可以了。我查阅后得知，这个选项用于在 IP 核的输出端插入一个寄存器，以提高输出信号的时序性能。它会导致增加一个时钟周期的延迟（因为寄存器会导致输出延后一个周期），取消这个选项之后，仿真就正常了。



## 2. 增加三种新运算

当仿真正常跑通以后，我就开始增加三条我自己的指令了。我增加的三条指令的功能是有符号数比较和无符号数比较的大于置位运算、还有按位与非三种运算。

我按照 IF,ID,EX,MEM,WB 的顺序依次去看哪里需要改。

首先是 IF 取址阶段，没有什么需要改动的。

然后是 ID 译码和取值阶段，这里需要改动的地方不少。

```
// 实现指令列表
wire inst_ADDU, inst_SUBU , inst_SLT , inst_AND;
wire inst_NOR , inst_OR   , inst_XOR , inst_SLL;
wire inst_SRL , inst_ADDIU, inst_BEQ , inst_BNE;
wire inst_LW  , inst_SW   , inst_LUI , inst_J;
wire inst_SLTU, inst_JALR , inst_JR  , inst_SLLV;
wire inst_SRA , inst_SRAV , inst_SRLV, inst_SLTIU;
wire inst_SLTI, inst_BGEZ , inst_BGTZ, inst_BLEZ;
wire inst_BLTZ, inst_LB   , inst_LBU , inst_SB;
wire inst_ANDI, inst_ORI  , inst_XORI, inst_JAL;
wire inst_SGT,inst_SGTU,inst_ANDOR;
```

如上图，首先要在实现指令列表里加入我新加的三种运算：有符号数比较和无符号数比较的大于置位运算、还有按位与非运算。

```
assign inst_SGT   = op_zero & sa_zero    & (funct == 6'b110000);//有符号大于置位
assign inst_SGTU   = op_zero & sa_zero    & (funct == 6'b110001);//无符号大于置位
assign inst_ANDOR = op_zero & sa_zero    & (funct == 6'b101111);//按位与非操作
```

然后编写这三种运算的判断逻辑，这三种运算都属于 R 型指令，功能码是我随机挑选的没有用过的。

```
assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT  | inst_SLTU
                     | inst_JALR | inst_AND  | inst_NOR  | inst_OR
                     | inst_XOR  | inst_SLL  | inst_SLLV | inst_SRA
                     | inst_SRAV | inst_SRL  | inst_SRLV|inst_SGT|inst_SGTU|inst_ANDOR;
```

然后还要确定这三个操作需要写回到 rd 寄存器里。

```
//alu操作分类
wire inst_add, inst_sub, inst_slt,inst_sltu;
wire inst_and, inst_nor, inst_or, inst_xor;
wire inst_sll, inst_srl, inst_sra,inst_lui;
wire inst_sgt,inst_sgtu,inst_andor;
assign inst_sgt=inst_SGT;
assign inst_sgtu=inst_SGTU;
assign inst_andor=inst_ANDOR;
assign inst_add = inst_ADDU  | inst_ADDIU | inst_load
                | inst_store | inst_j_link;          // 做加法
assign inst_sub = inst_SUBU;                        // 减法
assign inst_slt = inst_SLT | inst_SLTI;             // 有符号小于置位
assign inst_sltu= inst_SLTIU | inst_SLTU;           // 无符号小于置位
assign inst_and = inst_AND | inst_ANDI;             // 逻辑与
assign inst_nor = inst_NOR;                         // 逻辑或非
```

然后就是根据得到的指令判断码取确定 alu 要进行什么操作了，这里我新增加了三种相应的操作，后边会在 alu 里实现。

```
assign alu_control =
inst_add   ? 4'b0001 :
inst_sub   ? 4'b0010 :
inst_slt   ? 4'b0011 :
inst_sltu  ? 4'b0100 :
inst_and   ? 4'b0101 :
inst_nor   ? 4'b0110 :
inst_or    ? 4'b0111:
inst_xor   ? 4'b1000:
inst_sll   ? 4'b1001 :
inst_srl   ? 4'b1010 :
inst_sra   ? 4'b1011 :
inst_lui   ? 4'b1100 :
inst_sgt   ? 4'b1101 :
inst_sgtu  ? 4'b1110 :
inst_andor ? 4'b1111 :
             4'b0000; // default case
```

接下来这里要改一下 alu_control 的赋值逻辑，原来使用 16 位的独热编码，我把压缩成 4 位的，支持 16 种操作。不过要注意的是需要把总线的宽度对齐，压缩为 142 位，节省了空间。

```
output     [141:0]  ID_EXE_bus,   // ID->EXE总线
```

如下图，接下来在 EXE 模块里把 ID_EXE_bus_r 和 alu_control 的位宽都改一下。

```
    input      [141:0] ID_EXE_bus_r,// ID->EXE总线
    output             EXE_over,    // EXE模块执行完成
    output     [105:0] EXE_MEM_bus, // EXE->MEM总线

    //展示PC
    output     [ 31:0] EXE_pc
);
//-----{ID->EXE总线}begin
    //EXE需要用到的信息
    //ALU两个源操作数和控制信号
    wire [3:0] alu_control;
```

然后在 alu 模块里完成对操作码的赋值。

```
assign alu_add = (alu_control == 4'b0001) ? 1'b1 : 1'b0;
assign alu_sub  =(alu_control == 4'b0010) ? 1'b1 : 1'b0;
assign alu_slt  =(alu_control == 4'b0011) ? 1'b1 : 1'b0;
assign alu_sltu = (alu_control == 4'b0100) ? 1'b1 : 1'b0;
assign alu_and  = (alu_control == 4'b0101) ? 1'b1 : 1'b0;
assign alu_nor  = (alu_control == 4'b0111) ? 1'b1 : 1'b0;
assign alu_or   = (alu_control == 4'b0110) ? 1'b1 : 1'b0;
assign alu_xor  =(alu_control == 4'b1000) ? 1'b1 : 1'b0;
assign alu_sll  = (alu_control == 4'b1001) ? 1'b1 : 1'b0;
assign alu_srl  = (alu_control == 4'b1010) ? 1'b1 : 1'b0;
assign alu_sra  = (alu_control == 4'b1011) ? 1'b1 : 1'b0;
assign alu_lui  =(alu_control == 4'b1100) ? 1'b1 : 1'b0;
assign alu_sgt  =(alu_control == 4'b1101) ? 1'b1 : 1'b0;
assign alu_sgtu  =(alu_control == 4'b1110) ? 1'b1 : 1'b0;
assign alu_andor  =(alu_control == 4'b1111) ? 1'b1 : 1'b0;
```

然后与非运算直接用系统运算符实现。

```
assign andor_result=~(alu_src1 & alu_src2);
```

大于比较置位的两个运算需要一些其他逻辑来实现。

```
assign sgt_result[31:1] = 31'd0;
assign sgt_result[0]    = (~alu_src1[31] & alu_src2[31]) | (~(alu_src1[31]^alu_src2[31]) & ~adder_result[31]&(|adder_result[30:0]));
assign sgtu_result = {31'd0, adder_cout&(|adder_result[31:0])};
```

然后在最后结果里加入新的三种运算的结果。

```
assign alu_result = (alu_add|alu_sub) ? add_sub_result[31:0] :
                    alu_slt           ? slt_result :
                    alu_sltu          ? sltu_result :
                    alu_and           ? and_result :
                    alu_nor           ? nor_result :
                    alu_or            ? or_result  :
                    alu_xor           ? xor_result :
                    alu_sll           ? sll_result :
                    alu_srl           ? srl_result :
                    alu_sra           ? sra_result :
                    alu_lui           ? lui_result :
                    alu_sgt           ? sgt_result :
                    alu_sgtu          ? sgtu_result :
                    alu_andor         ? andor_result:
                    32'd0;
```

这样就改完了代码部分。

## 3.增加三条新的指令

首先加有符号大于置位的指令，op 为 000000，我设定操作数 1 是寄存器 00010，操作数 2 是寄存器 00001，目的寄存器是 00101，sa 是 00000，funct 码是 110000。

合起来就是 00000000010000010010100000110000，转为 16 进制就是 00412830。其作用是比较寄存器 2 和 1 的值，如果 2 大于 1 的值，那么就令寄存器 5 为 1，否则为 0。

接着进行无符号大于置位的指令，op 为 000000，我设定操作数 1 是寄存器 00011，操作数 2 是寄存器 00101，目的寄存器是 00110，sa 是 00000，funct 码是 110001。

合起来就是 00000000011001010011000000110001，转为 16 进制就是 00653031。其作用是比较寄存器 2 和 1 的值，如果 2 大于 1 的值，那么就令寄存器 6 为 1，否则为 0。

最后进行按位与非的指令，op 为 000000，我设定操作数 1 是寄存器 00001，操作数 2 是寄存器 00100，目的寄存器是 00111，sa 是 00000，funct 码是 101111。

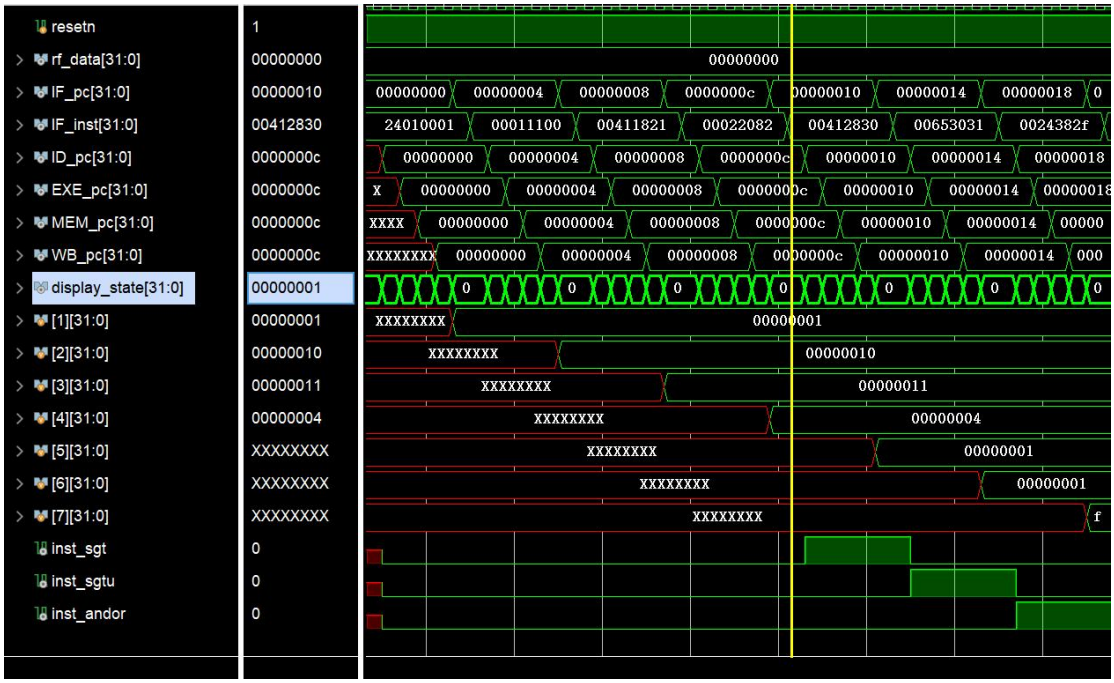
合起来就是 00000000001001000011100000101111，转为 16 进制就是 0024382f。其作用是将寄存器 1 和 4 的值进行按位与非运算，结果存到 7 号寄存器里。

把它们写入 coe 文件即可，我会在实验结果分析部分仔细讲解它们的计算过程。

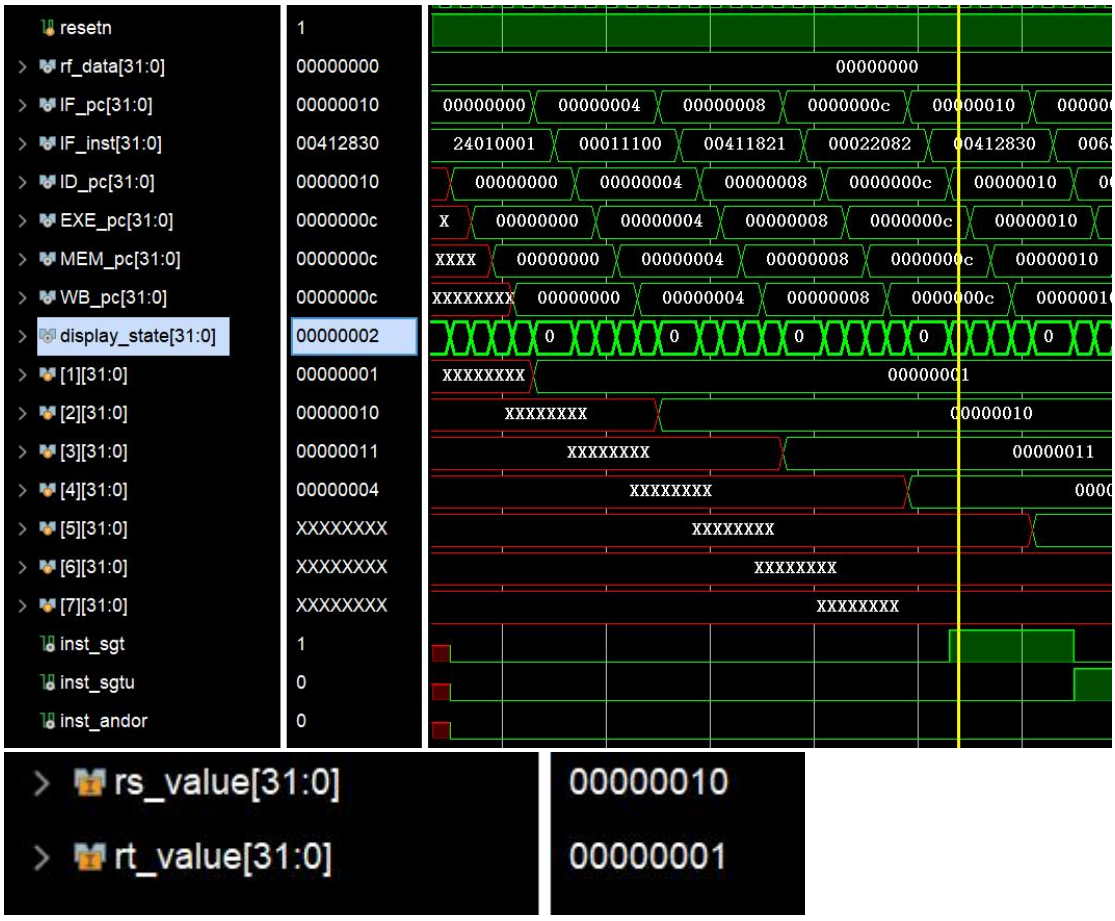
# 四、 实验结果分析

本部分主要介绍新添加的三条指令的运行情况，原先已经实现的指令不多介绍，其分析方法和之前的单周期实验差不多。

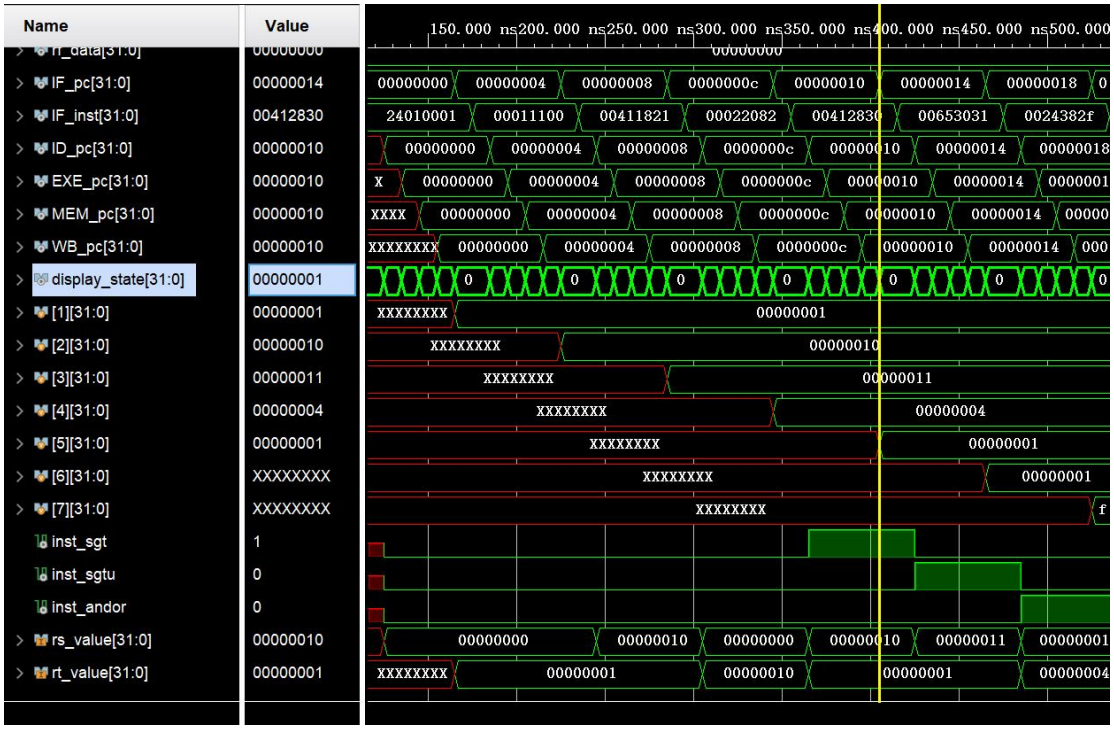
首先是 00412830，有符号大于置位的指令。其作用是比较寄存器 2 和 1 的值，如果 2 大于 1 的值，那么就令寄存器 5 为 1，否则为 0。




如上图，[1],[2],[3]等等代表的是 1，2，3 号寄存器。display_state 是 1，进行取址操作，这时候可以看见 IF_inst 是 00412830，正确。

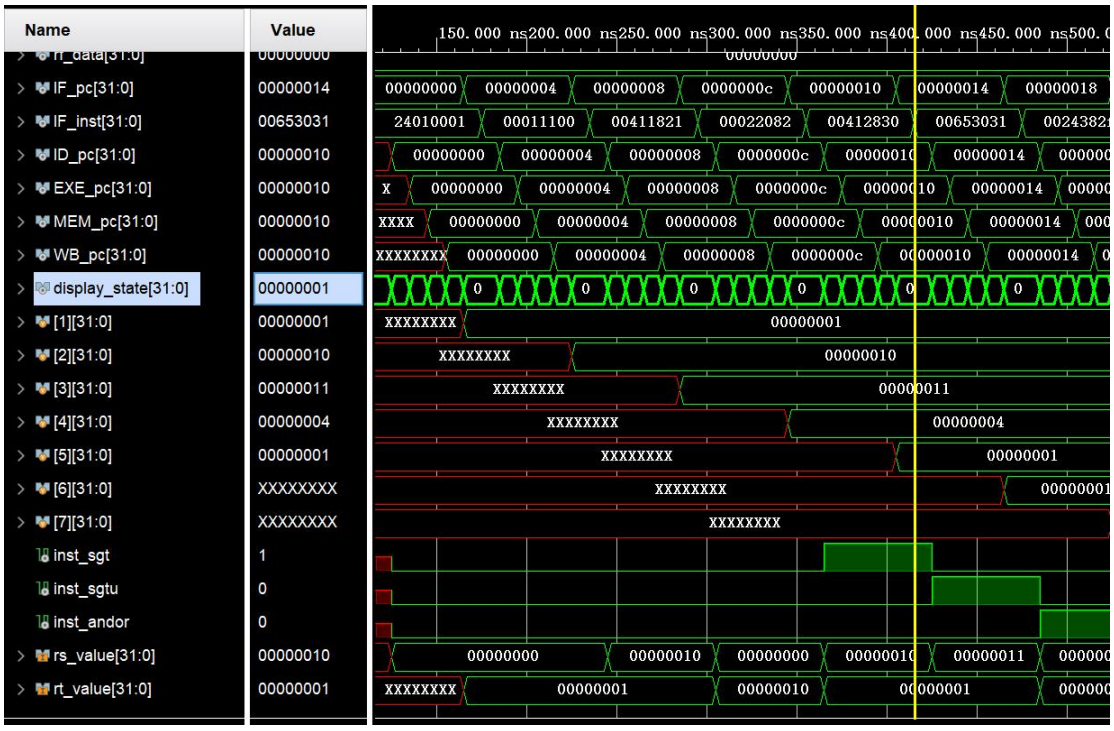



如上图，display_state 是 2，进行译码操作和读寄存器，这时候可以看见 inst_sgt 是 1，说明要进行的运算是有符号大于比较。读取的操作数 1 rs 是 2 号寄存器的值 10，操作数 2 rt 是 1 号寄存器的值 1，正确，没有问题。

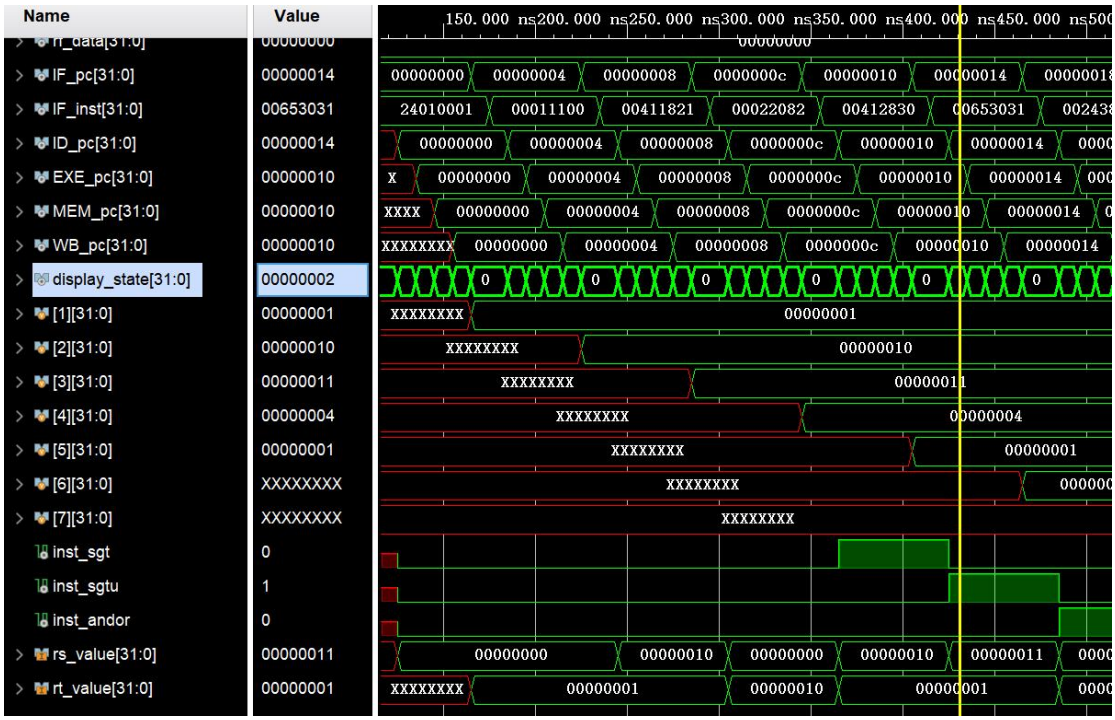



然后依次进行 ex，mem，wb 阶段，都完成后，可以看见 5 号目的寄存器的值置为了 1，因为操作数 1 的值 10 大于操作数 2 的值 1，因此要置位，答案正确。

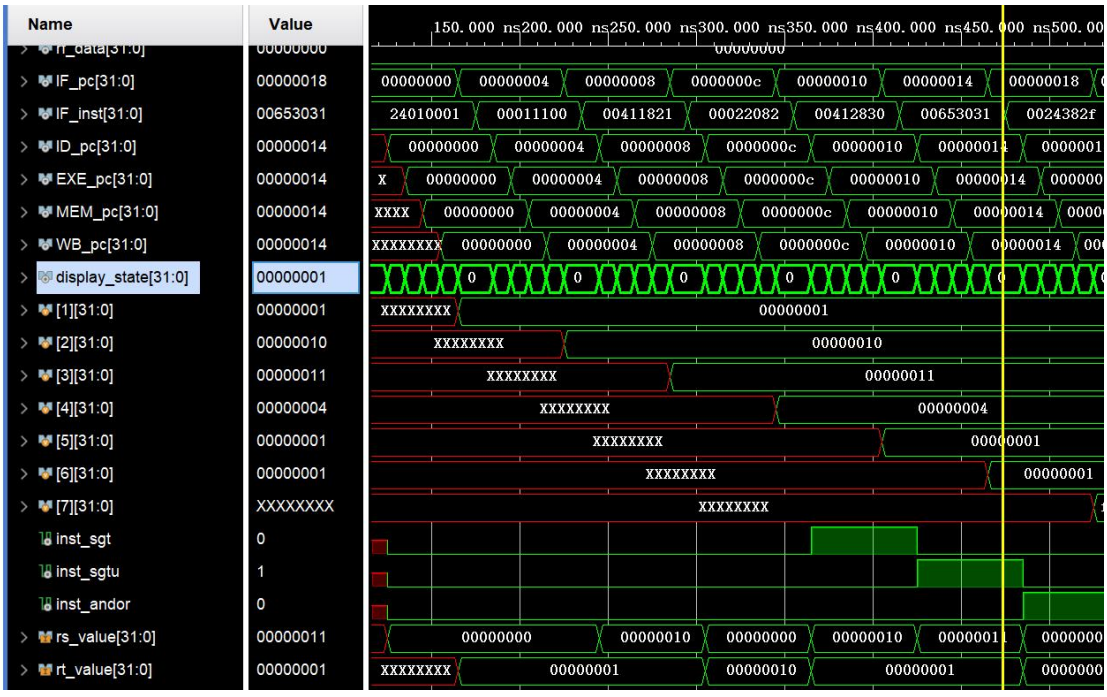
然后执行 00653031，即无符号大于置位的运算。其作用是比较寄存器 3 和 5 的值，如果 3 大于 5 的值，那么就令寄存器 6 为 1，否则为 0。




如上图，display_state 是 1，进行取址操作，这时候可以看见 IF_inst 是 00653031，正确。

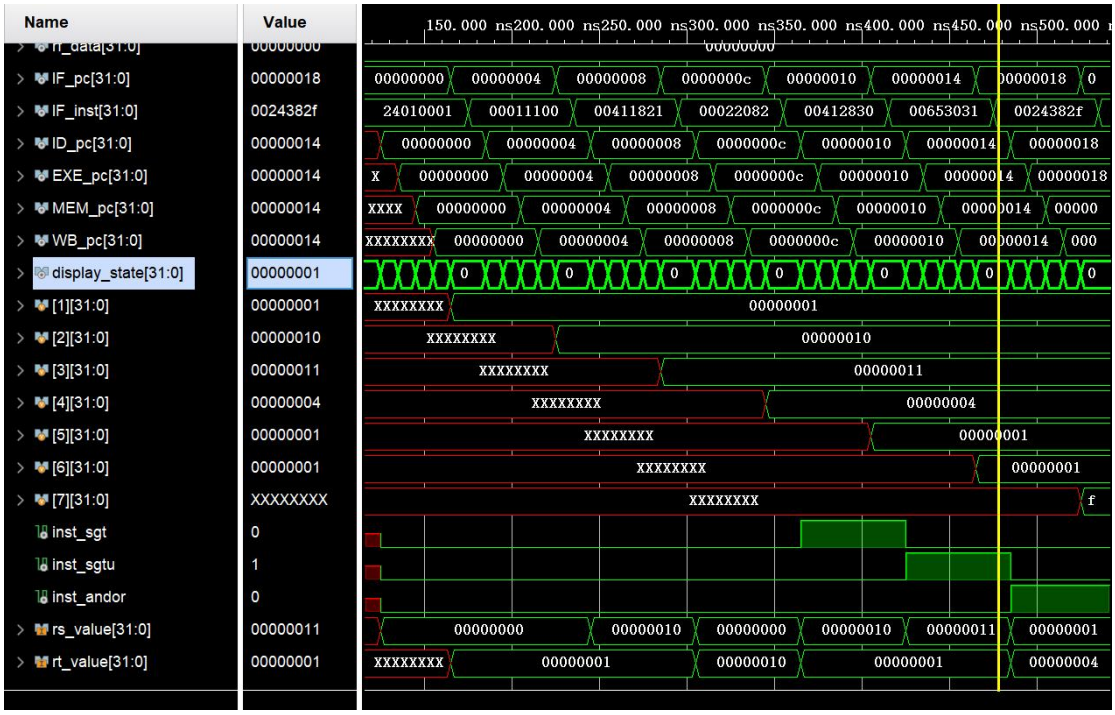



如上图，display_state 是 2，进行译码操作和读寄存器，这时候可以看见 inst_sgtu 是 1，说明要进行的运算是无符号大于比较。读取的操作数 1 rs 是 3 号寄存器的值 11，操作数 2 rt 是 5 号寄存器的值 1，正确，没有问题。

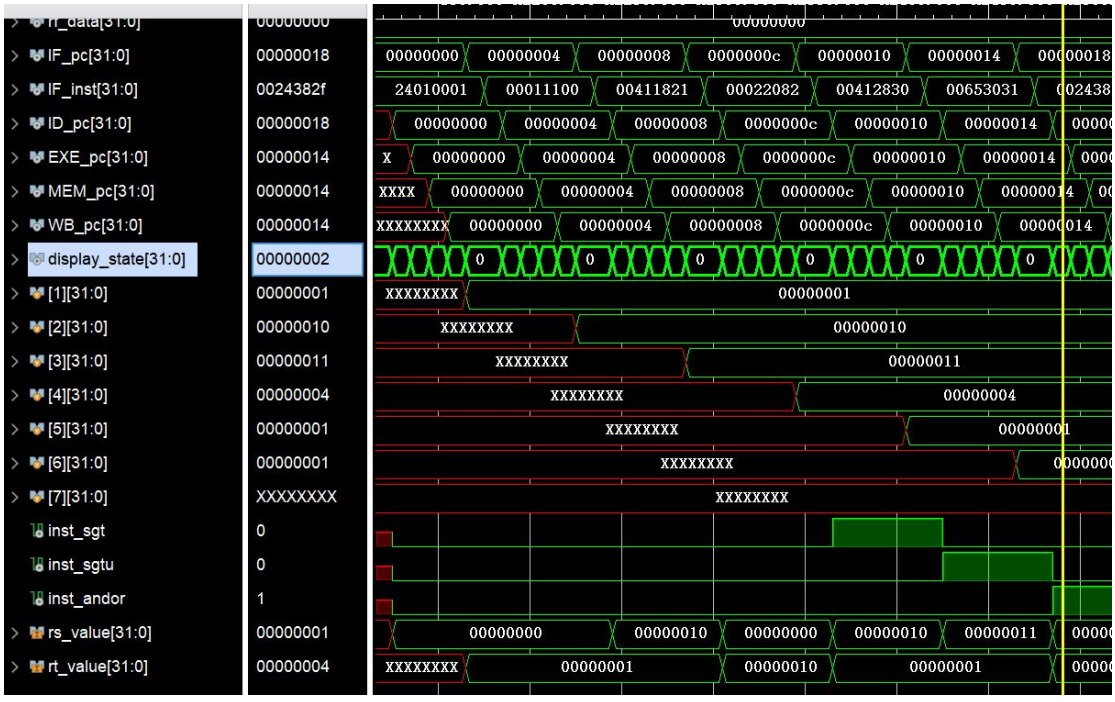



然后依次进行 ex，mem，wb 阶段，都完成后，可以看见 6 号目的寄存器的值置为了 1，因为操作数 1 的值 11 大于操作数 2 的值 1，因此要置位，答案正确。

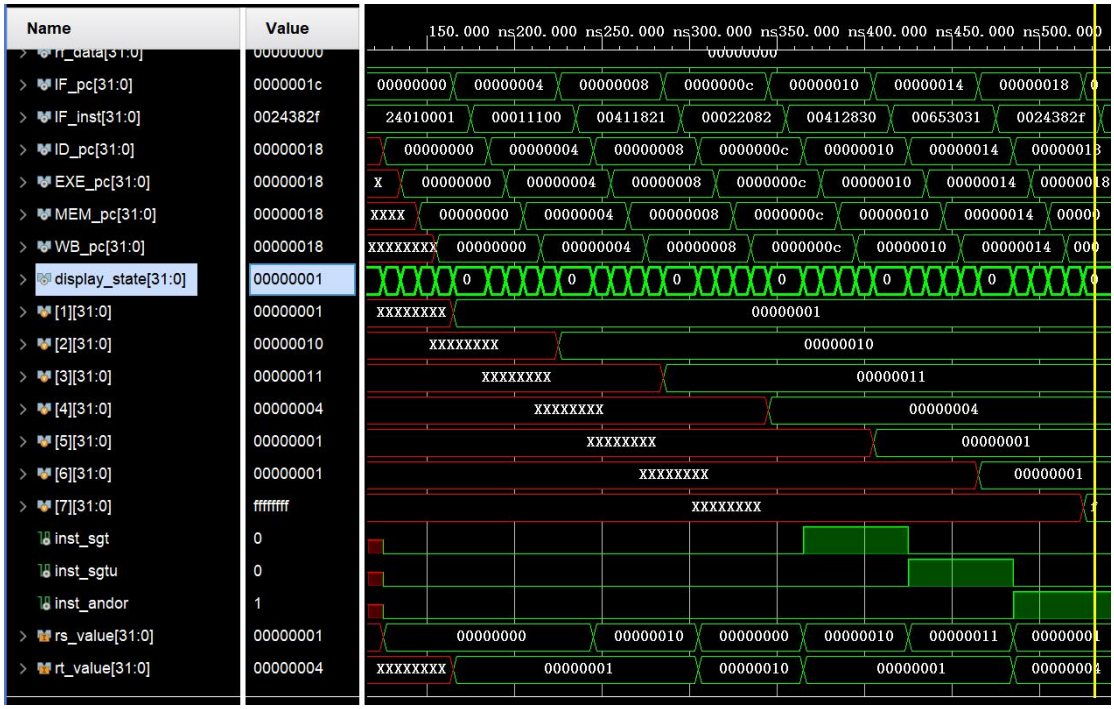
最后进行 0024382f。按位与非的运算。其作用是将寄存器 1 和 4 的值进行按位与非运算，结果存到 7 号寄存器里。




如上图，display_state 是 1，进行取址操作，这时候可以看见 IF_inst 是 0024382f，正确。




如上图，display_state 是 2，进行译码操作和读寄存器，这时候可以看见 inst_andor 是 1，说明要进行的运算按位与非运算。读取的操作数 1 是 1 号寄存器的值 1，操作数 2 是 4 号寄存器的值 4，正确，没有问题。




然后依次进行 ex，mem，wb 阶段，都完成后，可以看见 7 号目的寄存器的值置为了 ffffffff，因为操作数 1 的值 1 先与操作数 2 的值 4 进行与运算，得到的结果是 00000000，然后进行一个非操作，得到的结果就是 ffffffff，答案正确。

# 五、 总结感想

本次实验我进行了多周期 CPU 的测试，在完成单周期 CPU 的基础上，我进行了本次多周期的实验。

多周期 CPU 的实验后，我逐渐理解了其核心思想：将一条指令的执行过程分解为多个步骤（本次实验是 IF,ID,EX,MEM,WB 五个阶段），每一步用独立的时钟周期完成。这种设计方式提高了 CPU 的整体性能和资源利用率。

在实验过程中，我学习了如何将指令划分为多个阶段（如取指、译码、执行、访存和写回），并根据每个阶段构建相应的数据通路与控制信号。并且也帮助我理解李涛老师在理论课上讲的知识。

并且由于原始的代码存在一些问题，我在寻找问题的过程中也深入了解了指令具体执行的步骤，对于多周期 CPU 有了更加深入的了解，对 verilog 语言也更加熟悉了。

下一次实验我将要进行五级流水线的设计，多周期 CPU 是实现流水线结构的重要过渡阶段。本次实验帮助我理解了指令的阶段划分和控制流调度，这些概念将在后续流水线 CPU 的设计中继续发挥关键作用。

通过本次多周期 CPU 的设计与实现，我不仅巩固了对处理器结构的理解，也提升了 Verilog 硬件描述语言的实际编程能力。这个实验是一个重要的过渡节点，为我后续的五级流水线实验奠定了坚实的基础。