

计算机组成原理实验报告

实验名称：单周期CPU的分析和多周期CPU的实现 班级：张金老师 姓名：王众

指导老师：董前琨 实验地点：实验楼A306 实验时间：5.4

一、实验目的

1. 理解MIPS指令结构，理解MIPS指令集中常用指令的功能和编码，学会对这些指令进行归纳分类。
2. 了解熟悉MIPS体系的处理器结构，如延迟槽，哈佛结构的概念。
3. 熟悉并掌握单周期CPU的原理和设计。
4. 进一步加强运用verilog语言进行电路设计的能力。

二、实验内容

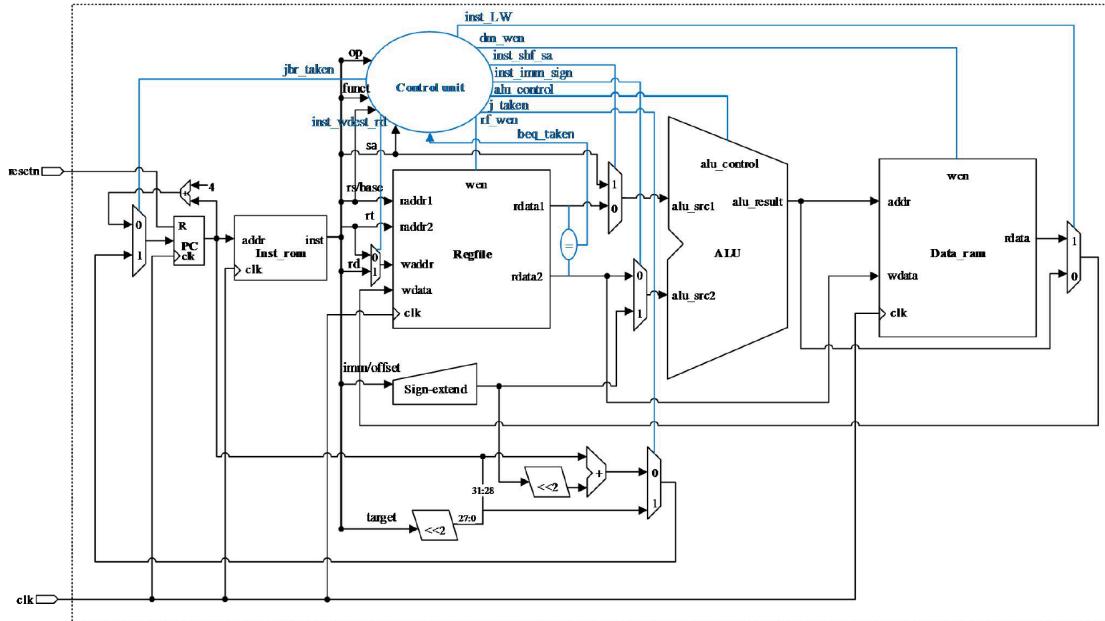
请根据实验指导手册中的单周期CPU和多周期CPU实验内容，完成如下任务并撰写实验报告：

- 1、针对单周期CPU实验，复现并验证功能，同时对三种类型的MIPS指令，挑1~2条具体分析总结其执行过程。
- 2、针对多周期CPU实验，请认真分析指令ROM中的指令执行情况，找到存在的bug并修复，实验报告中总结寻找bug和修复bug的过程。
- 3、将ALU实验中扩展的三种运算，以自定义MIPS指令的形式，添加到多周期CPU实验代码中，并自行编写指令存储到指令ROM，然后验证正确性，波形验证或实验箱上箱验证均可。

注意：单周期CPU使用异步存储器 (.v) 文件格式，多周期CPU使用的是同步存储器 (IP核形式)，二者不要弄混。多周期实验中，每一个IP核请自行创建到项目中，不要使用源码中的dcp文件。

三、实验原理图

单周期CPU实验框架图



四、单周期CPU指令执行

4.1 指令

单周期CPU能够执行MIPS指令集系统的一个子集，共16条指令，包括存储访问指令、运算指令、跳转指令。根据拥有的字段类型不同，我们将指令分为 **R型指令**、**I型指令** 和 **J型指令**。

4.1.1 R型指令

op	rs	rt	rd	shamt	funct
----	----	----	----	-------	-------

- op 段 (6b)：恒为0b000000；
- rs (5b)、rt (5b)：两个源操作数所在的寄存器号；
- rd (5b)：目的操作数所在的寄存器号；
- shamt (5位)：位移量，决定移位指令的移位位数；
- funct (6b)：功能位，决定R型指令的具体功能。

4.1.2 I型指令

op	rs	rt	constant or address
----	----	----	---------------------

- op段 (6b)：决定I型指令类型；
- rs (5b)：是第一个源操作数所在的寄存器号；
- rt (5b)：是第二个源操作数所在的寄存器号 或 目的操作数所在的寄存器编号。
- constant or address (16b)：立即数或地址

4.1.3 J型指令

op	address
----	---------

- op段 (6b) : 决定J型指令类型;
- constant or address (26b) : 转移地址

4.2 不同指令的执行过程

4.2.1 R型指令

- 从指令存储器中取指令，更新PC。
- ALU根据funct字段确定ALU的功能。
- 从寄存器堆中读出寄存器rs和rt。
- ALU根据2中确定的功能，对从寄存器堆读出的数据进行操作。
- 将运算结果写入到rd字段对应的目标寄存器。

4.2.2 I型指令

- 存取指令：
 - 从指令存储器中取指令，更新PC。
 - ALU根据op字段确定ALU的功能。
 - 从寄存器堆中读出寄存器rs的值，并将其与符号扩展后的指令低16位立即数的值相加。
 - 若为存储指令，则将rt寄存器中的值存到上步相加得到的存储器地址；
 - 若为取数指令，则将上步所得存储器地址里所存的数据放到rt目标寄存器中。
- 分支指令：
 - 从指令存储器中取指令，更新PC。
 - 从寄存器堆中读出寄存器rs和rt的值。
 - 将所读寄存器的两值相减。
 - 根据上步的结果是否为0，将PC+4的值或address字段所对应地址存入PC中。

4.2.3 J型指令

- 从指令存储器中取指令，更新PC。
- 取出address字段，作为目标跳转地址。
- 将目标跳转地址存入PC中。

五、单周期CPU的MIPS指令分析

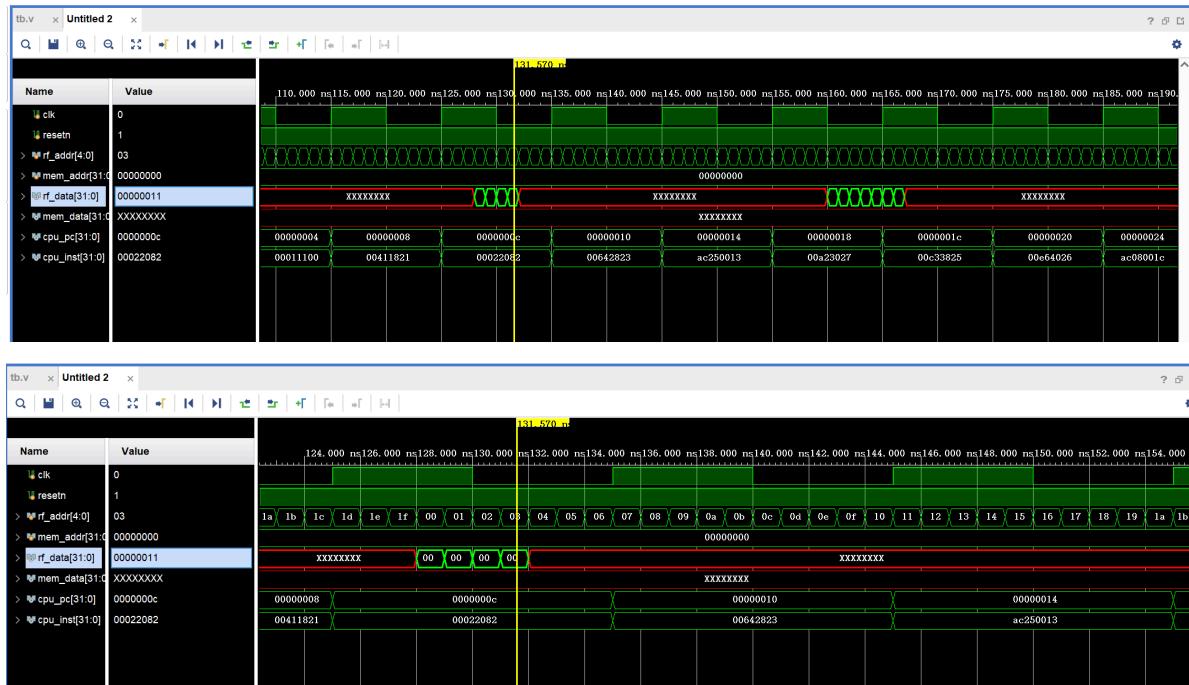
5.1 待处理的模拟任务

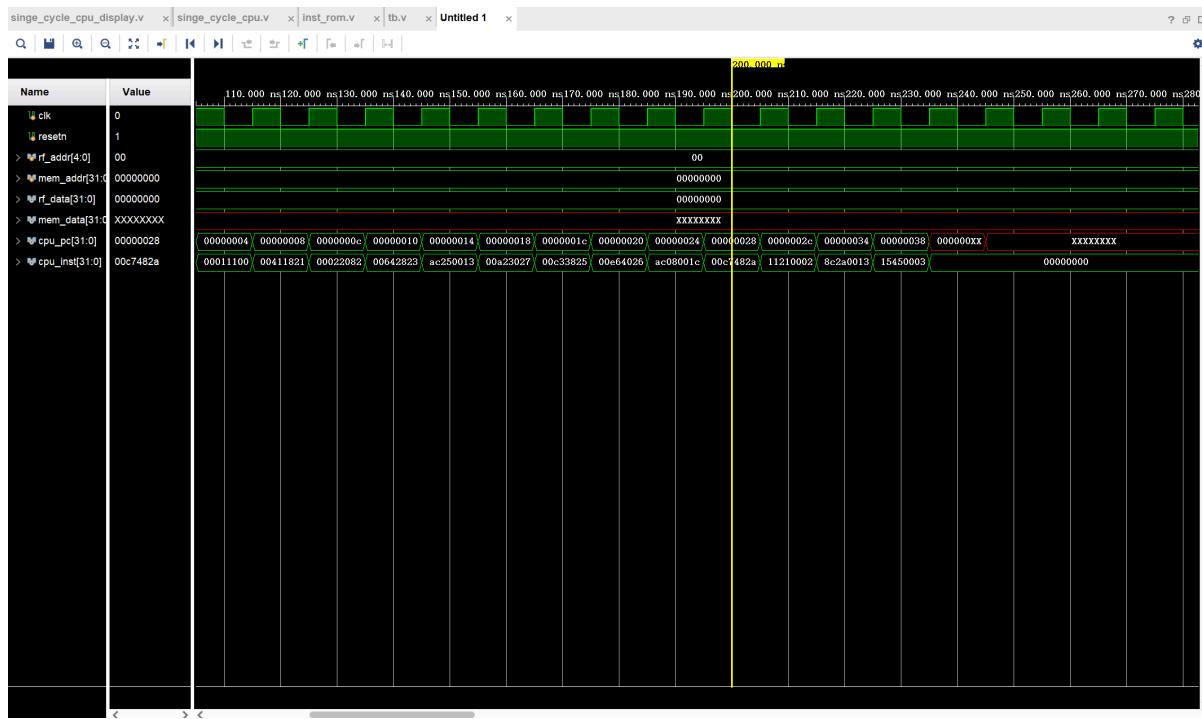
```
initial begin
    // Initialize Inputs
    clk = 0;
    resetn = 0;
    rf_addr = 0;
    mem_addr = 0;

    // Wait 100 ns for global reset to finish
    #100;
    resetn = 1;
    // Add stimulus here
end
always #5 clk=~clk;
```

1. 初始时设置 `resetn` 为 0，复位CPU
2. 100ns 后释放复位信号 (`resetn=1`)，开始执行指令
3. 时钟信号每 10ns 一个周期(5ns 高电平+ 5ns 低电平)
4. 通过输出信号(`rf_data`, `mem_data`, `cpu_pc`, `cpu_inst`)观察执行结果

我们进行仿真实验，并得到了以下图像





随着取值的不断变化，`cpu_inst` 的值也在不断的变化，这意味着我们进行的指令也在不断的变化。

5.2 MIPS指令的执行过程

5.2.1 R型指令

```
assign inst_rom[10] = 32'h00c7482A; // 28H: slt $9,$6,$7 | $9 = 0000_0001H
```

例如在 `cpu_pc=00000004` 的时候，`cpu_inst` 的值为 `00c7482A`，我们在 `inst_rom.v` 的文件中查到这个代表的是指令 `slt $9,$6,$7` 比较 `$6` 和 `$7` 的值，如果 `$6 < $7` 则 `$9=1`，否则 `$9=0`。属于R型指令。在代码中可见 `$6(FFFF_FFE2)` 小于 `$7(FFFF_FFF3)`，所以 `$9=1`。

```
assign inst_rom[ 1] = 32'h00011100; // 04H: sll $2,$1,#4 | $2 = 0000_0010H
```

00011100: 对应 `inst_rom[1]` 中的 `32'h00011100`

指令: `sll $2, $1, #4`

功能: `$2 = $1 左移4位`

类型: R型指令

执行: `$1=1, 左移4位后 $2=16(0x10)`

```
assign inst_rom[ 2] = 32'h00411821; // 08H: addu $3,$2,$1 | $3 = 0000_0011H
```

00411821: 对应 `inst_rom[2]` 中的 `32'h00411821`

指令: `addu $3, $2, $1`

功能: `$3 = $2 + $1`

类型: R型指令

执行: `$3 = $2(16) + $1(1) = 17(0x11)`

5.2.2 I型指令

```
assign inst_rom[ 5] = 32'hAC250013; // 14H: sw    $5 ,#19($1) | Mem[0000_0014H]
= 0000_000DH
```

AC250013: 对应 `inst_rom[5]` 中的 `32'hAC250013`

指令: `sw $5, 19($1)`

功能: 将 `$5` 的值存入内存地址 `$1+19`

类型: I型指令

执行: 将 `$5(13)` 存入内存地址 `$1(1)+19=20(0x14)`

```
assign inst_rom[ 9] = 32'hAC08001C; // 24H: sw    $8 ,#28($0) | Mem[0000_001CH]
= 0000_0011H
```

AC08001C: 对应 `inst_rom[9]` 中的 `32'hAC08001C`

指令: `sw $8, 28($0)`

功能: 将 `$8` 的值存入内存地址 `$0+28`

类型: I型指令

执行: 将 `$8(0x11)` 存入内存地址 `28(0x1C)`

5.2.3 J型指令

```
assign inst_rom[11] = 32'h11210002; // 2CH: beq    $9 ,$1,#2    | 跳转到指令34H
```

11210002: 对应 `inst_rom[11]` 中的 `32'h11210002`

指令: `beq $9, $1, 2`

功能: 如果 `$9=$1`, 则跳转到 `PC+4+offset*4`

类型: j型指令

执行: 因为 `$9=1` 且 `$1=1`, 所以跳转到 `PC+4+(2*4)=PC+12`, 即跳转到 `0x34` 处

```
assign inst_rom[19] = 32'h08000000; // 4CH: j      00H           | 跳转指令00H
```

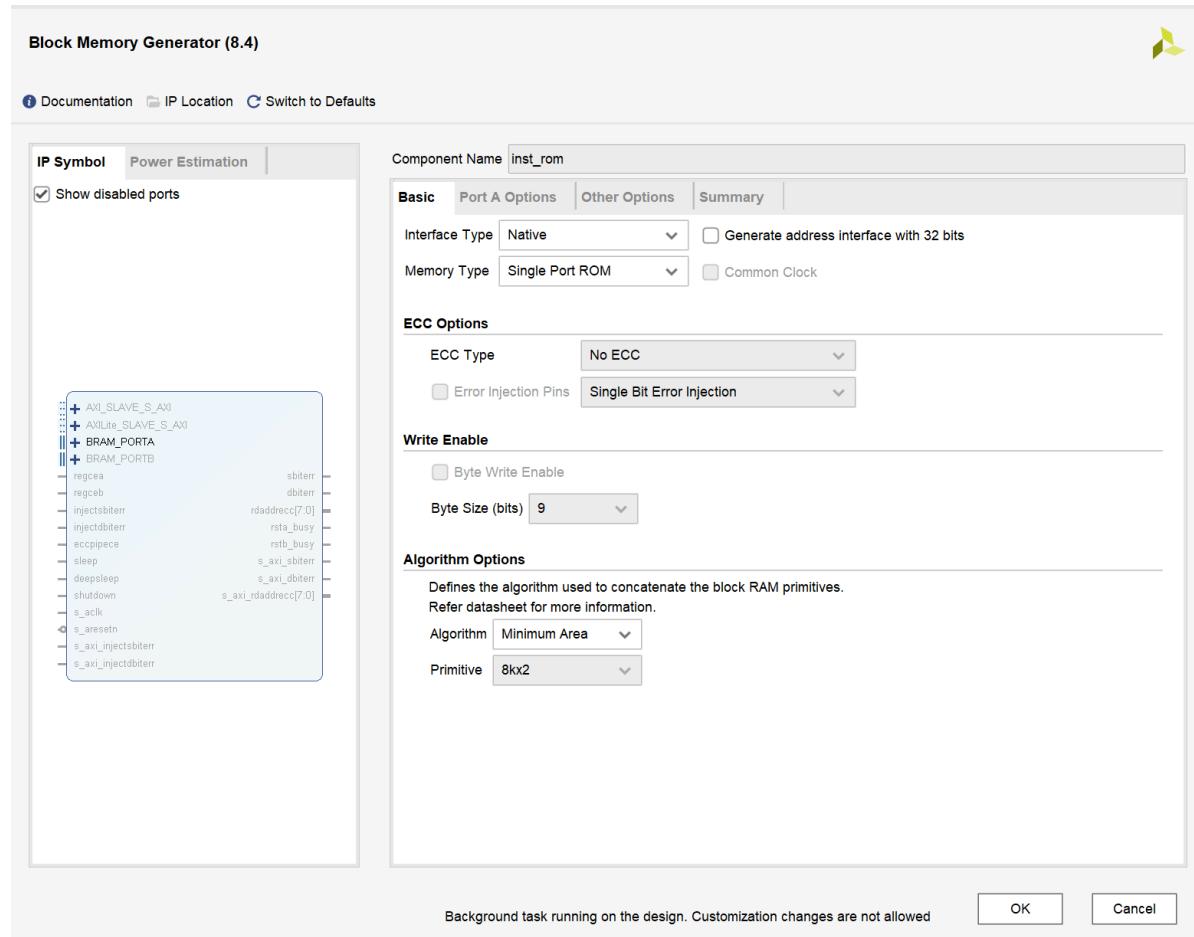
这是一个 `j` (jump) 指令, 操作码为 `000010`。在 MIPS 架构中: `j` 指令: 无条件跳转到指定地址。操作码: `000010`。在 MIPS 中, J型指令的跳转地址计算方式为: `PC = (PC+4)[31:28] | (target << 2)` 即:

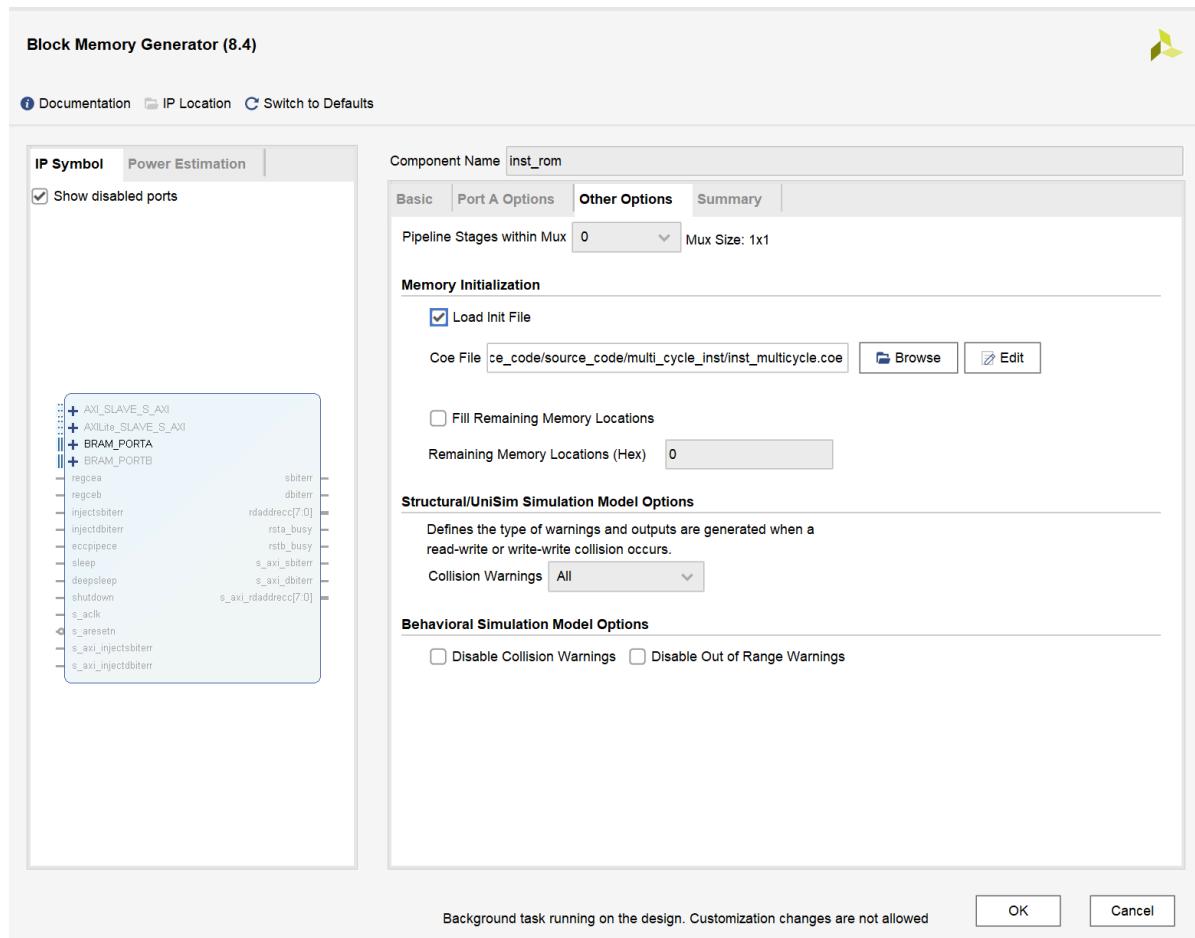
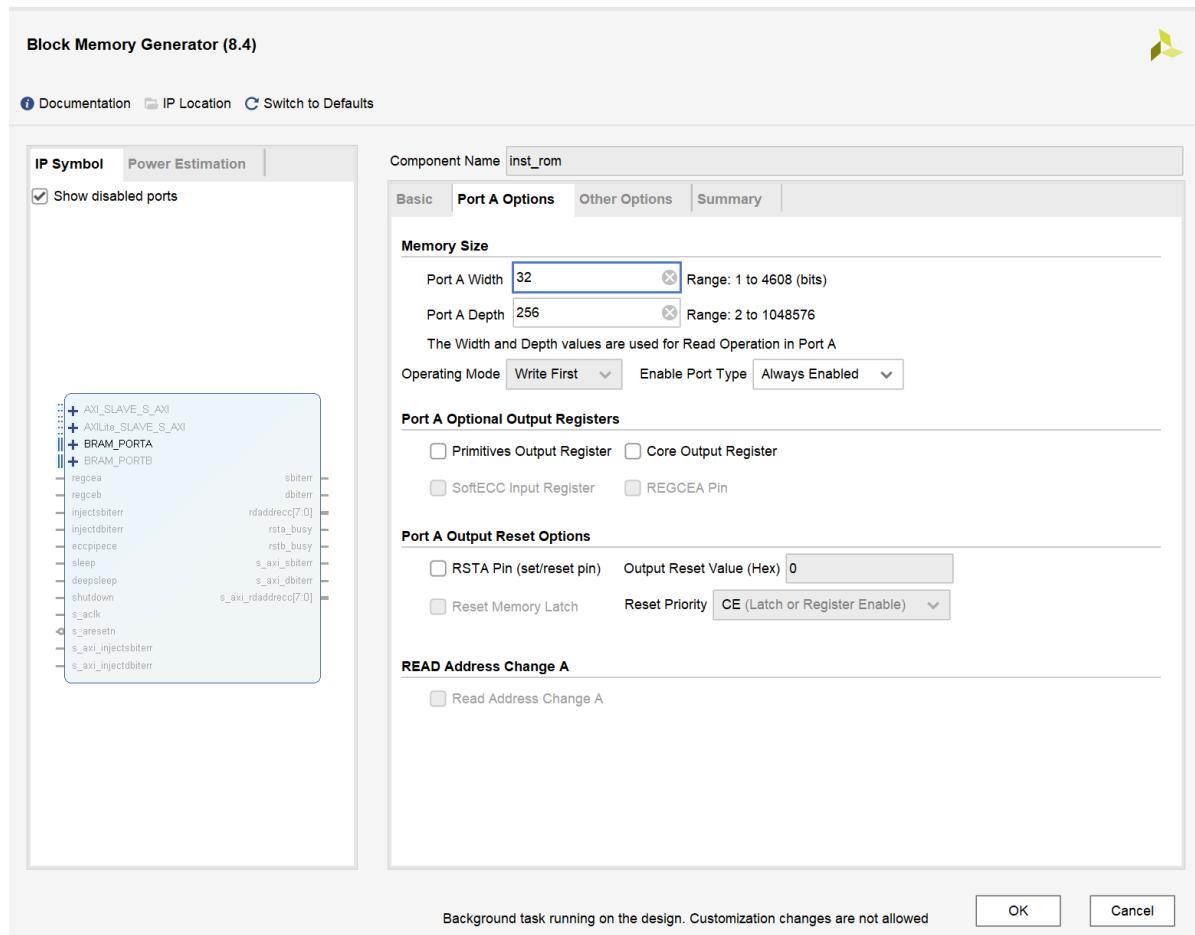
1. 取当前 `PC+4` 的高4位
2. 将指令中的26位 `target` 左移2位(补0)
3. 将上述两部分拼接形成32位地址

六、IP核的配置

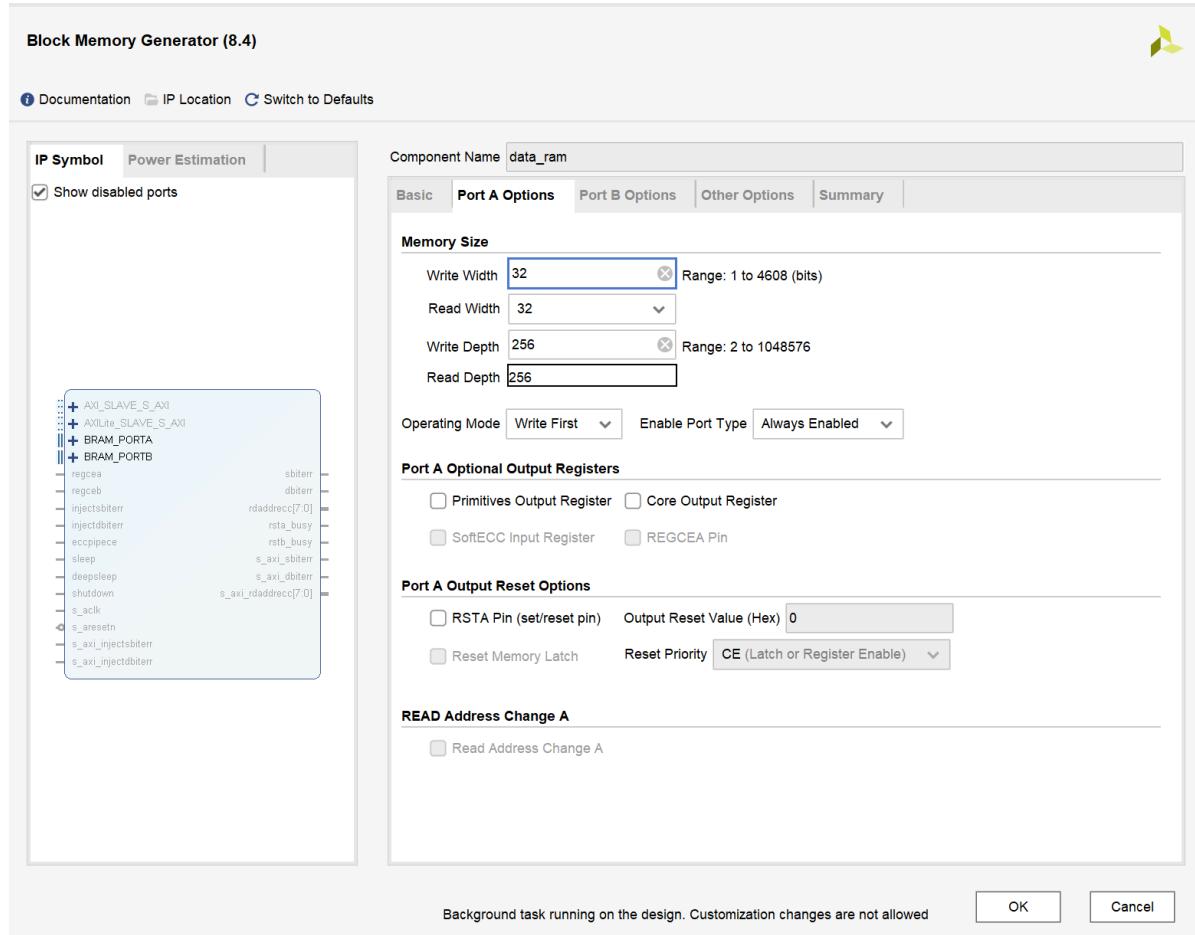
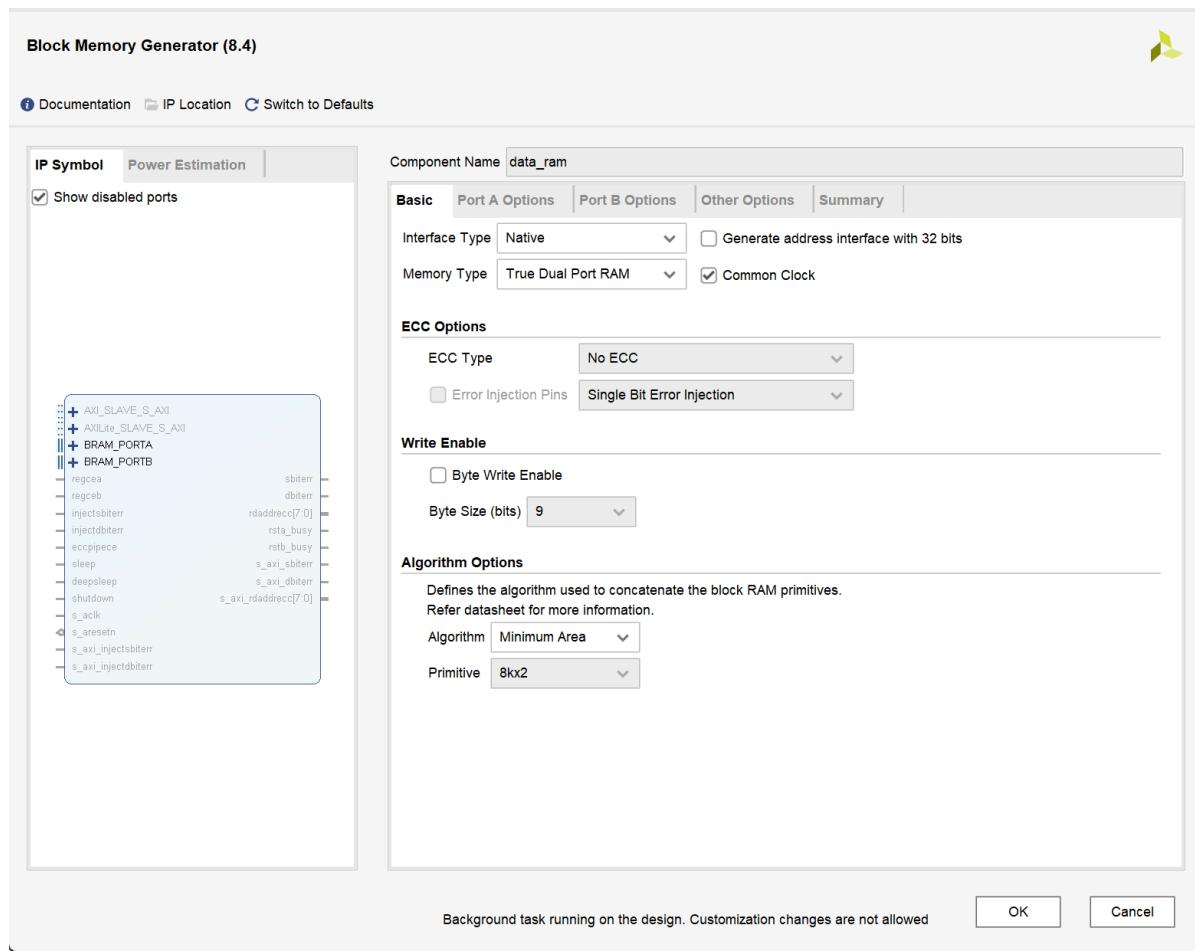
由于多周期CPU并不能用.v文件，所以需要我们配置IP核来进行环境的准备。

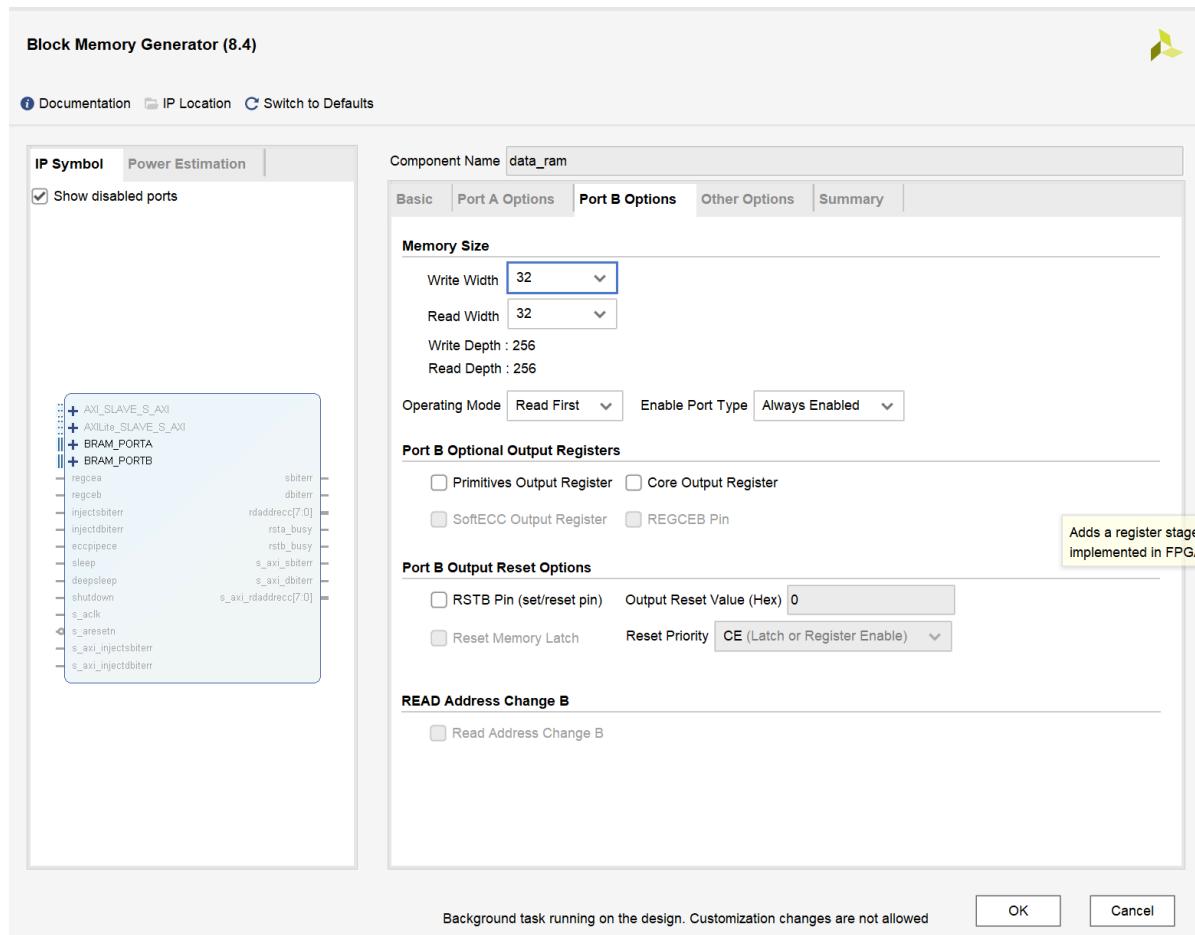
6.1 inst_rom IP核的配置





6.2data_ramIP 核的配置





七、多周期CPU的bug分析

7.1 Bug分析

分析可得是时钟周期的问题，按上述的方式配置IP核即可解决bug。

7.2 指导手册错误

```

initial begin
    // Initialize Inputs
    clk = 0;
    resetn = 0;
    rf_addr = 0;
    mem_addr = 0;

    // wait 100 ns for global reset to finish
    #100;
    resetn = 1;
    // Add stimulus here
end
always #5 clk=~clk;

```

因为 tb.v 将 rf_addr 和 mem_addr 均赋值为0，所以我们为了在每一个指令周期内把所有32个寄存器的状态都观察到，我们加上下面这个语句：

```

always #1 begin
    if(rf_addr == 32'hFFFF) // 使用较小范围循环
        rf_addr <= 0;
    else
        rf_addr <= rf_addr + 1;
end

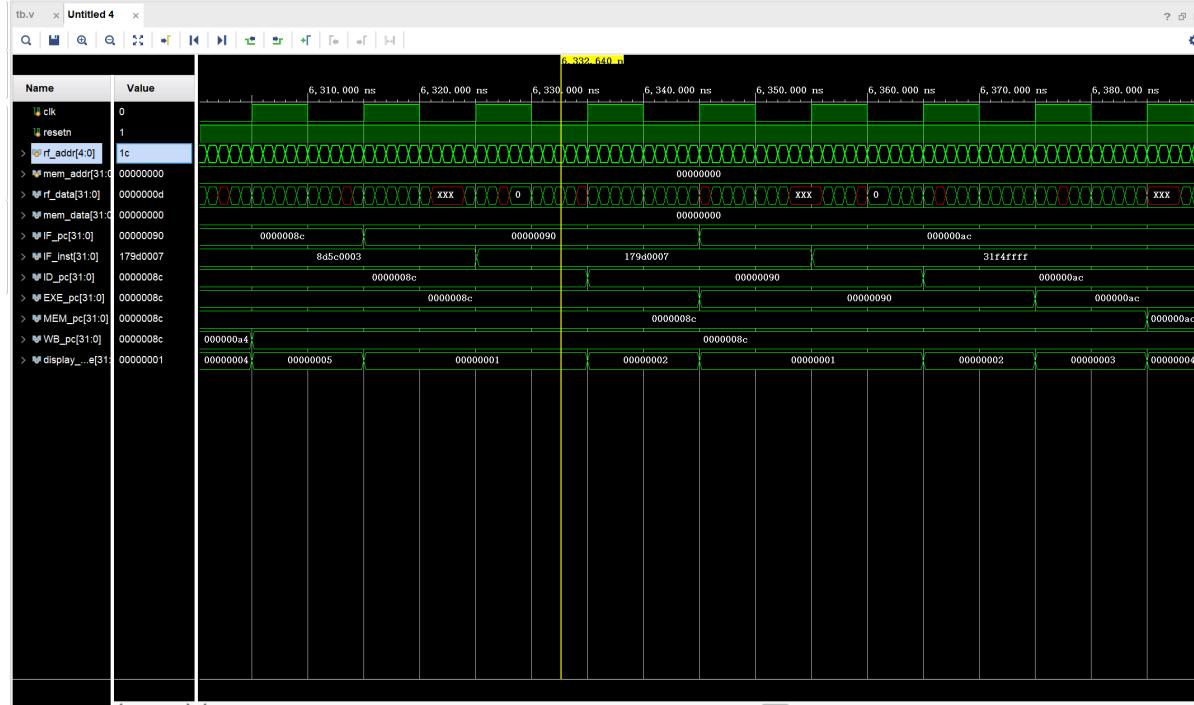
```

使得每 1ns 寄存器的编号都会加1，当加到31时便会变为0。这样我们就可以在每个指令周期内观察到所有寄存器的状态。我们发现在 sourcecode 文件夹内，有一个 多周期CPU测试程序详解。我们将文档中的指令执行情况与我们得到的结果相比较。发现在运行到该指令的时候。

8CH ^③	lw	\$28,#3(\$10) ^④	[\$28] = 000C_000DH /000C_880DH ^⑤	8D5C0003 ^⑥	1000_1101_0101_1100_0000_0000_0000_0011 ^⑦
------------------	----	----------------------------	---	-----------------------	--

应该执行的是将MEM[14H]中的值赋给\$28，在这之前已经执行过将其变为0DH的指令，我们的结果也为0DH，指导书出现了错误。

1CH ^③	sw	\$5,#20(\$0) ^④	Mem[0000_0014H] = 0000_0000DH ^⑤	AC050014 ^⑥	1010_1100_0000_0101_0000_0000_0001_0100 ^⑦
------------------	----	---------------------------	---	-----------------------	--



我们这里将其指向9号寄存器，果然值为0dH。

八、自定义三种MISP指令

8.1 代码更改的实现阶段

在这个阶段中，我们选择了和 ALU 实验中相同的三个操作：大于置位、同或和低位加载。

8.1.1 alu. v(定义运算及其运算过程)

1.首先要将处理的位宽全部加三。并将control位赋给相对应的运算操作。

```
input [14:0] alu_control, // ALU控制信号

assign alu_sgt = alu_control[14];
assign alu_xnor = alu_control[13];
assign alu_hui = alu_control[12];
```

2.对独热码和输出结果进行定义

```
wire alu_xnor;
wire alu_sgt;
wire alu_hui;

wire [31:0] sgt_result;
wire [31:0] xnor_result;
wire [31:0] hui_result;
```

3.对操作的具体运行进行定义

```
assign xnor_result = ~xor_result;
assign hui_result = {16'd0, alu_src2[15:0]};
assign sgt_result[31:1] = 31'd0;
assign sgt_result[0] = (~alu_src1[31] & alu_src2[31]) | // 正数 > 负数
    (~alu_src1[31]^alu_src2[31]) & ~adder_result[31] & (~adder_result); // 符号
相同，差为正且非零
```

4.对相应的结果输出做接线（注意接线的顺序需要与control的定义顺序相一致，不然信号会出现问题）

```
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_hui      ? hui_result :
    alu_xnor     ? xnor_result:
    alu_sgt      ? sgt_result :
    32'd0;
```

8.1.2 decode.v (定义新运算的编码符号)

1.因为alucontrol的位宽在alu.v中进行了扩大，所以之后的所有与alucontrol有关的变量的位宽都需要进行进一步的加宽。

```
output [152:0] ID_EXE_bus, // ID->EXE总线  
wire [14:0] alu_control;
```

2.实现指令列表，并定义运算编码（注意编码不要重复）

```
wire inst_HUI, inst_SGT, inst_XNOR;  
  
assign inst_SGT = op_zero & sa_zero & (funct == 6'b101100);  
assign inst_XNOR = op_zero & sa_zero & (funct == 6'b101101);  
assign inst_HUI = (op == 6'b010000);
```

3.实现将结果写入寄存器

```
assign inst_imm_zero = inst_ANDI | inst_LUI | inst_ORI | inst_XORI| inst_HUI;  
assign inst_wdest_31 = inst_JAL;  
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_XNOR;
```

4.再次实现读热编码的绑定

```
assign alu_control = {inst_sgt,  
                      inst_xnor,  
                      inst_hui,  
                      inst_add,      // ALU操作码，独热编码  
                      inst_sub,  
                      inst_slt,  
                      inst_sltu,  
                      inst_and,  
                      inst_nor,  
                      inst_or,  
                      inst_xor,  
                      inst_sll,  
                      inst_srl,  
                      inst_sra,  
                      inst_lui};
```

8.1.3 exe.v

因为这里也有关于alucontrol的变量更改，所以也需要更改。

```
input [152:0] ID_EXE_bus_r, // ID->EXE总线  
wire [14:0] alu_control;
```

8.1.4 multi_cycle_cpu.v

因为这里也有关于alu_control的变量更改，所以也需要更改。

```
wire [152:0] ID_EXE_bus; // ID->EXE级总线  
  
reg [152:0] ID_EXE_bus_r;
```

8.2 验证结果是否正确

8.2.1 首先我们先验证一下xnor的正确性

我们使用将\$1和\$2同或的结果储存在\$9中这个汇编语句来进行执行。（因为\$9是未被利用过的）

```
xnor $1 $2 $9
```

然后我们根据他进行机器码的翻译：

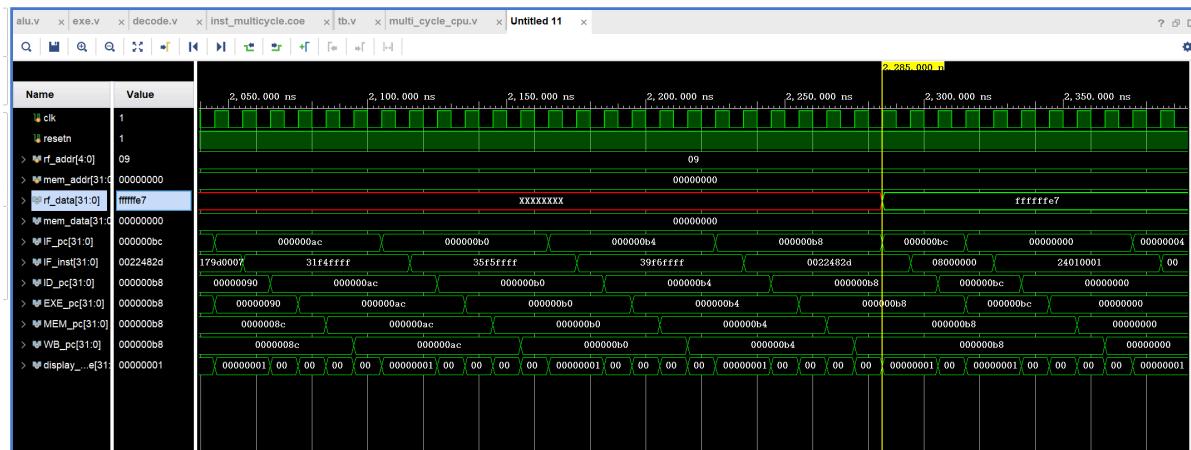
000000 00001 00010 01001 00000 101101

0x0022482D

所以我们将得到的十六进制的机器码插入倒数第二个的位置

```
45 2DF8FFFF  
46 0185E825  
47 01600008  
48 31F4FFFF  
49 35F5FFFF  
50 39F6FFFF  
51 0022482D  
52 08000000  
53
```

我们进行仿真模拟并记录\$9的值的情况



可以看到在执行完之后\$9的值变为了我们预想的 ffffffe7。

8.2.2 sgt大于置位的验证

8.2.2.1 两个均为正数

sgt有三种情况需要分别验证：即大于置位，小于不置位和等于不置位，我们按照这个思路编写汇编语句。同样的，我们选择了三个未使用过的寄存器来出储存我们的值。\$17, \$18, \$19。

```
sgt $1 $2 $17(小于)
sgt $2 $1 $18(大于)
sgt $1 $1 $19(等于)
```

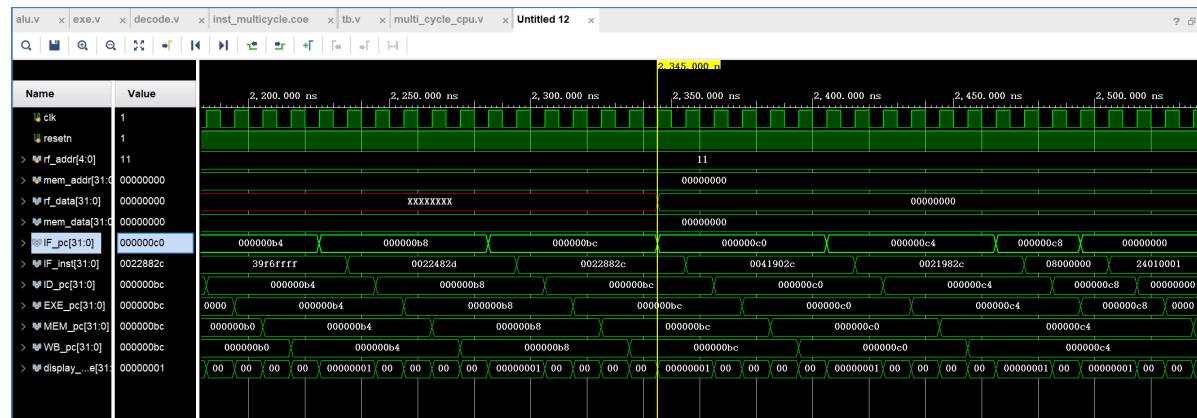
我们接下来根据汇编语句来翻译相对应的机器码（二进制和十六进制）

```
0000000000001000101000100000101100
00000000000010000011001000000101100
0000000000001000011001100000101100

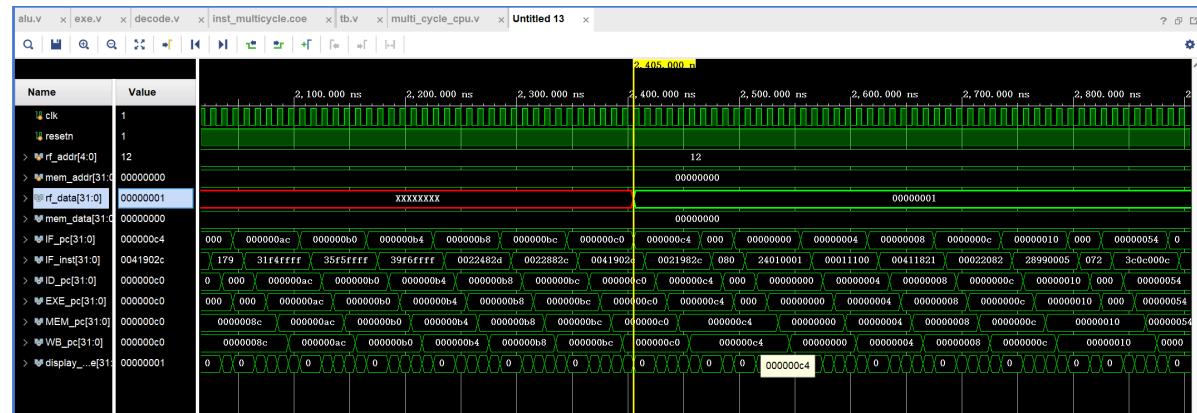
0022882C
0041902C
0021982C
```

得到了机器码之后我们就可以进行仿真验证了：

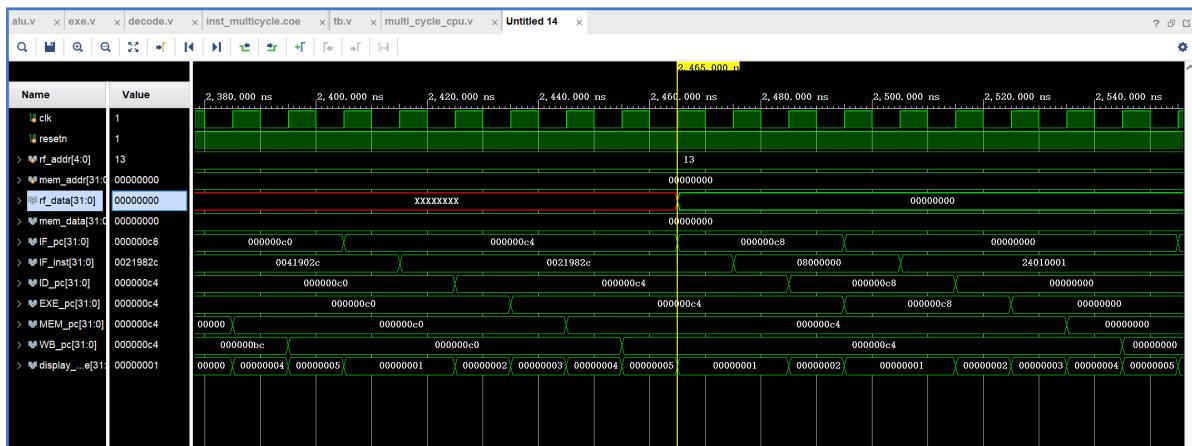
小于的时候不置位



大于的时候置位



等于的时候不置位



8.2.2.2 两个均为负数

对于两个操作数为负数我们从已有的寄存器中选择\$6和\$7作为比较对象,同时我们选择\$29和\$30作为我们的结果存放寄存器:

其中\$6中的值是大于\$7中的值的

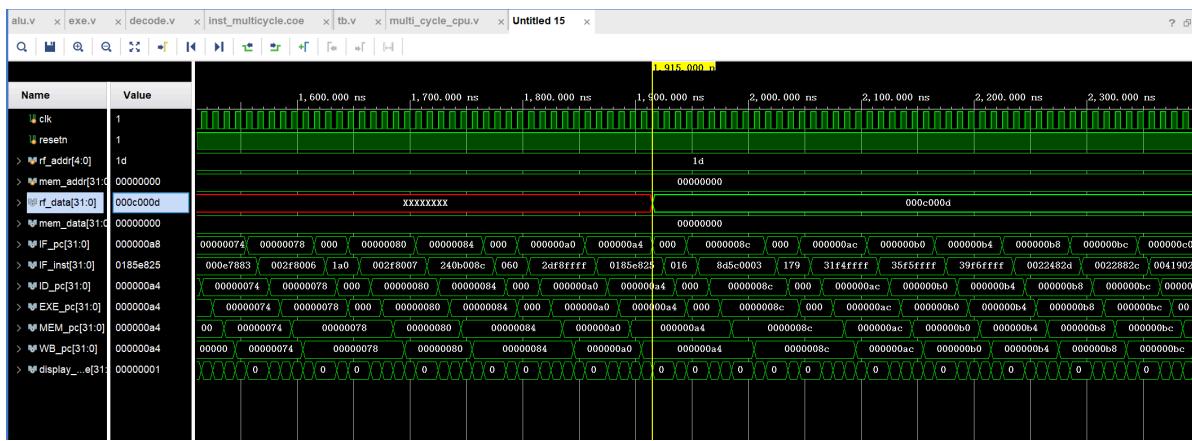
```
sgt $6 $7 $29(大于置位)
sgt $7 $6 $30(小于不置位)
```

我们根据已有的汇编代码进行机器码的编写:

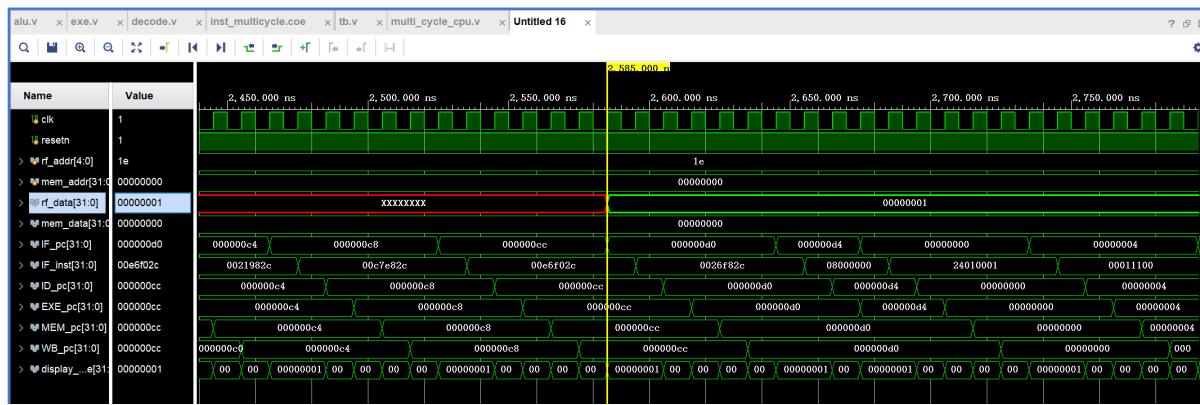
```
000000000110001111110100000101100
000000000111001101111000000101100

00C7E82C
00E6F02C
```

负数小于不置位:



负数大于置位



8.2.2.3 一个正数和一个负数

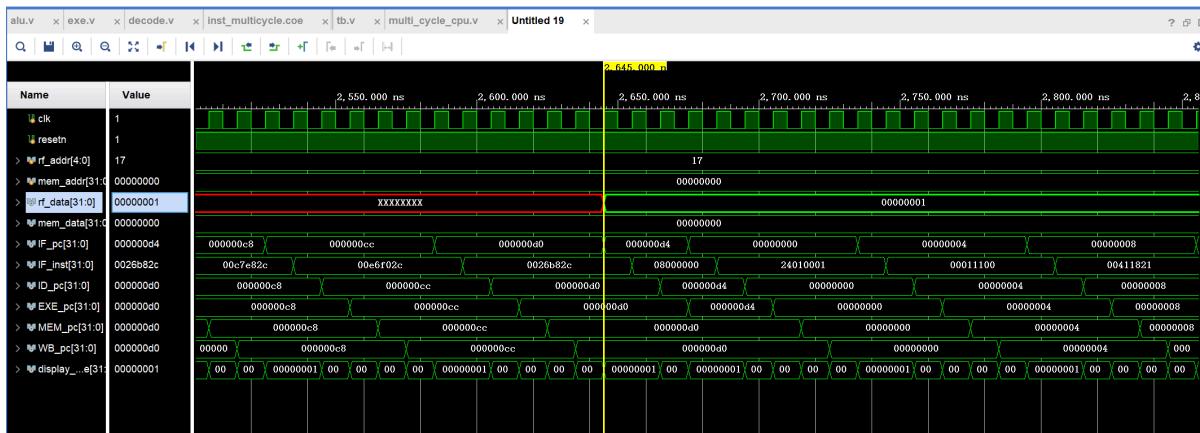
我们使用\$1和\$6作为比较对象，比较的结果储存在\$23中。

```
sgt $1 $6 $23(大于置位)
```

得到机器码：

000000000001001101011100000101100

0026B82C



至此我们大于置位的运算符所有的情况都已经验证完毕。

8.2.3 低位加载 (hui)

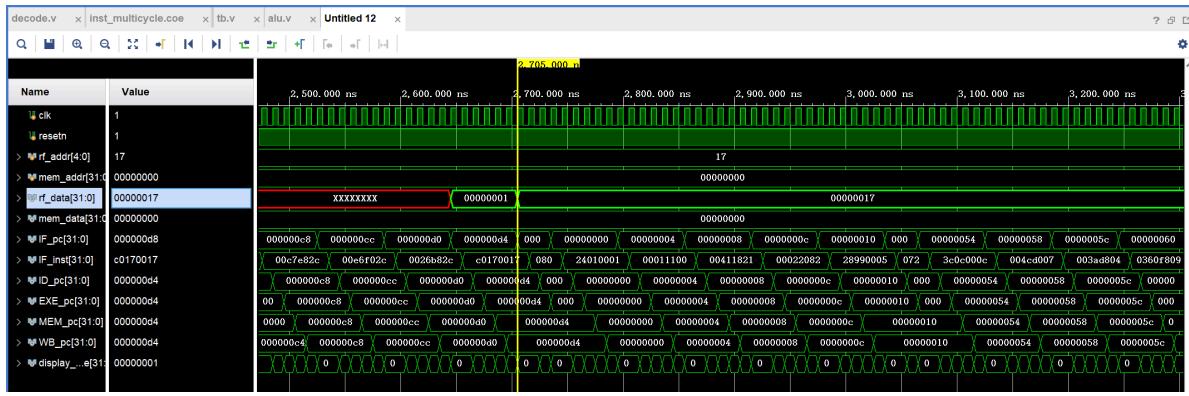
我们选择23进行低位加载，结果放在\$23中。

```
hui $23 #23
```

得到机器码：

11000000000010111000000000000010111

C0170017



果然23被加载进了\$23寄存器中。

至此我们所添加的三种运算全部被添加成功。

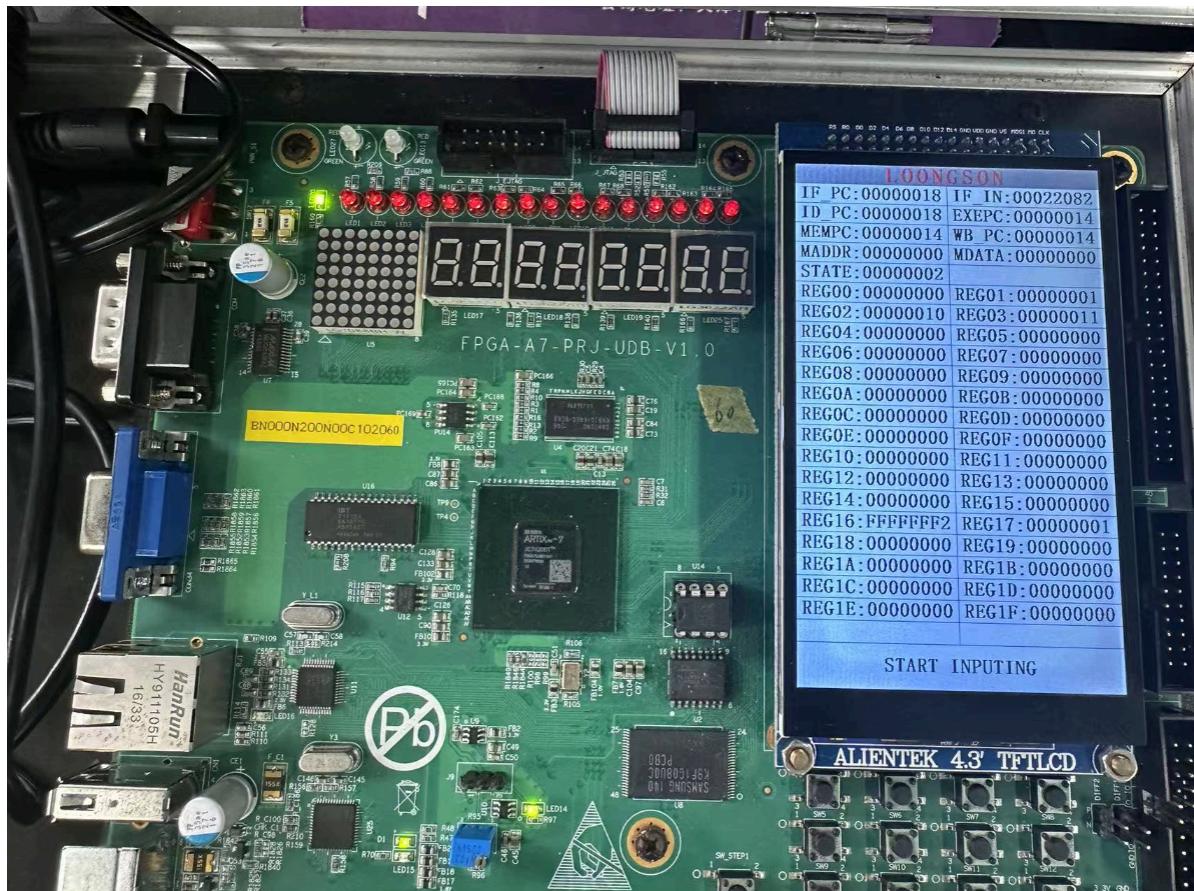
8.3 上箱验证

我们更改一下指令，因为我们之前是把指令加在了最后面，所以我们就新建了几个指令放在最前面以便上箱验证。

C016000C #B8H hui \$22,#12	-----[\$22] = 0x0000000C
02C1B031 #BCH nxor \$22,\$22,\$1	-----[\$22] = 0xFFFFFFFF2
0036B83F #C0H sgt \$23,\$1,\$22	-----[\$23] = 0x00000001

根据汇编代码我们可以得到机器码：

```
01000000001011000000010110
000001011010110000000010111
00000101110000110110010110
```



可以看到22号寄存器变成了我们理想的结果，23号寄存器也是。

九、心得体会

通过本次实验，我深入学习并实践了单周期 CPU 的指令执行过程，掌握了 MIPS 架构中 R 型、I 型和 J 型指令的格式、功能及执行流程。这次实验不仅加深了我对计算机组成原理的理解，还让我在实践中体会到了理论与实际结合的重要性。

尽管实验取得了一定成果，但我也发现了一些不足之处。例如，在初次编写 Verilog 代码时，我对信号的时序关系理解不够深入，导致仿真结果与预期不符。今后，我计划进一步学习时序分析和硬件设计的优化方法，以提高代码的效率和可靠性。此外，我希望能在后续实验中接触更复杂的多周期 CPU 或流水线 CPU 设计，探索更高效的指令执行机制。

总的来说，本次实验让我从理论走向实践，从抽象走向具体，不仅提升了我的技术能力，也激发了我对计算机体系结构设计的浓厚兴趣。我期待在未来的学习中继续深入探索 CPU 设计与优化，掌握更多前沿技术，为计算机体系结构领域的发展贡献自己的力量。