组成原理课程第五次实验报告

实验名称：单-多周期 CPU 实验

学号:2310764 姓名:王亦辉 班次:计科一班

1 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。
5. 为后续设计多周期cpu的实验打下基础。
6. 在单周期 CPU 实验完成的提前下，理解多周期的概念。
7. 熟悉并掌握多周期 CPU 的原理和设计。
8. 进一步提升运用 verilog 语言进行电路设计的能力。
9. 为后续实现流水线cpu的课程设计打下基础。

2 实验内容说明

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。

3　实验原理图

单周期 CPU ：

单周期 CPU 的实现框图如下：

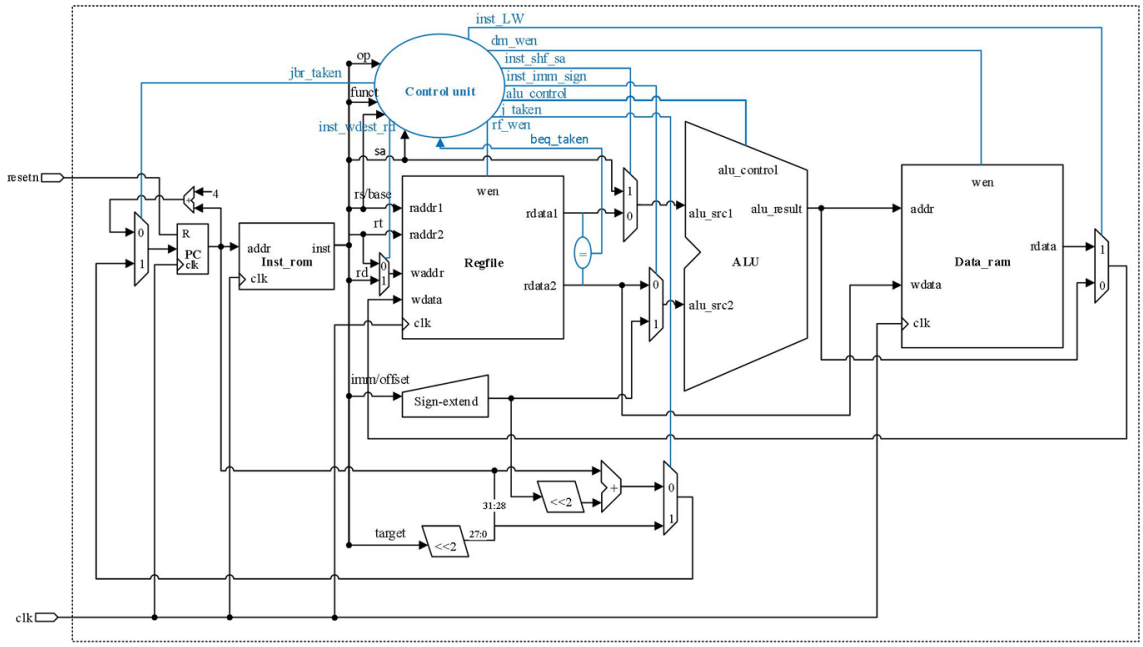


图 7.3 单周期 CPU 的实现框图

多周期 CPU：

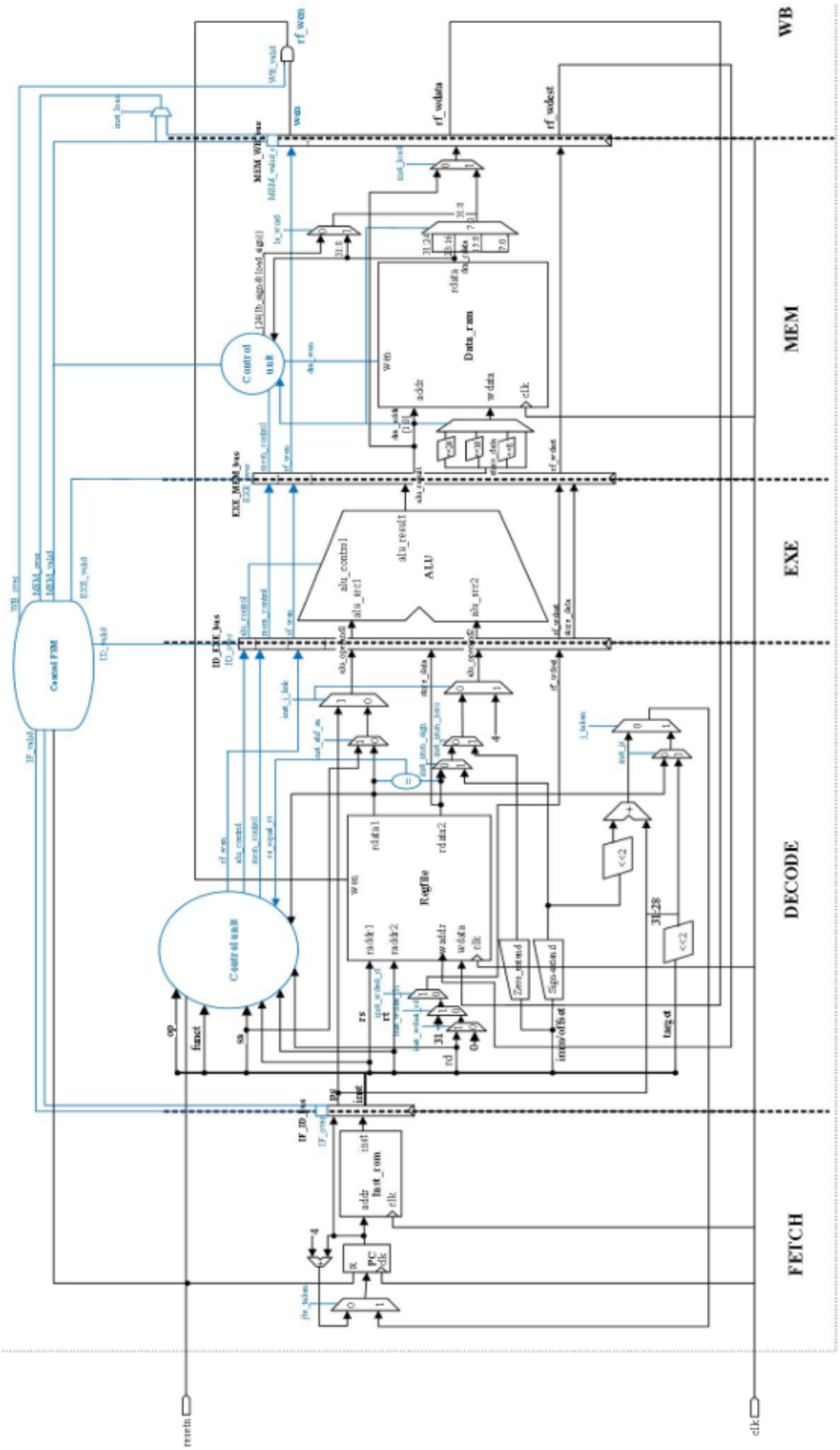


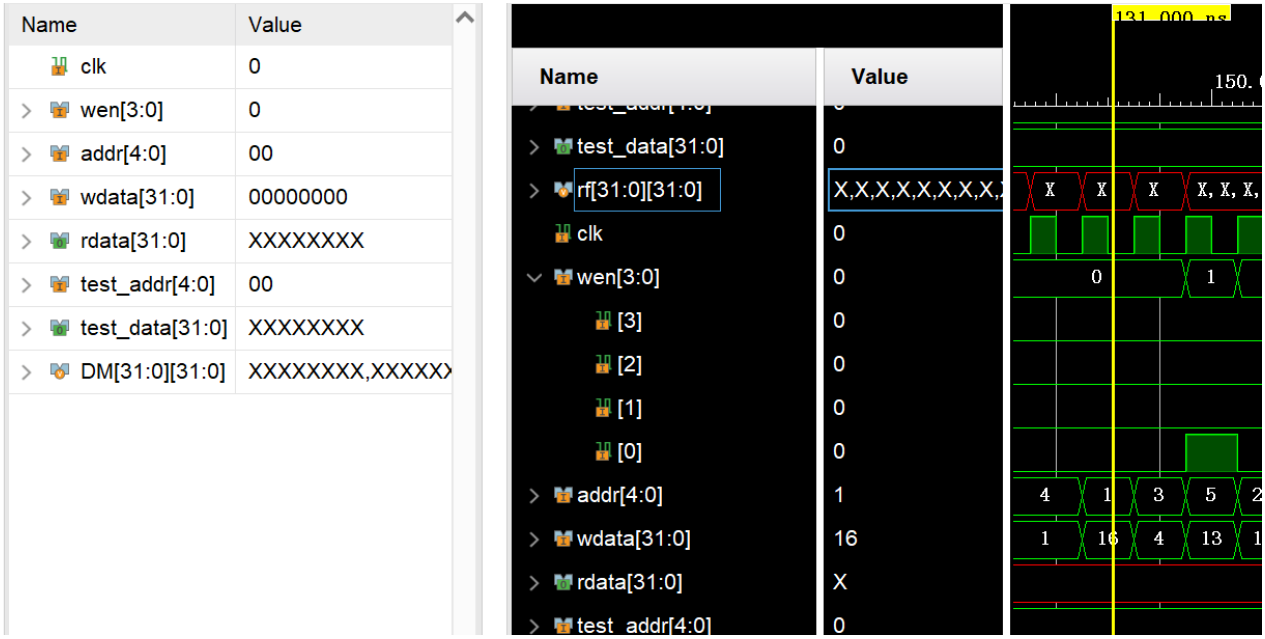
图 8.4 多周期 CPU 的实现框图

4 实验步骤

4.1 单周期 CPU

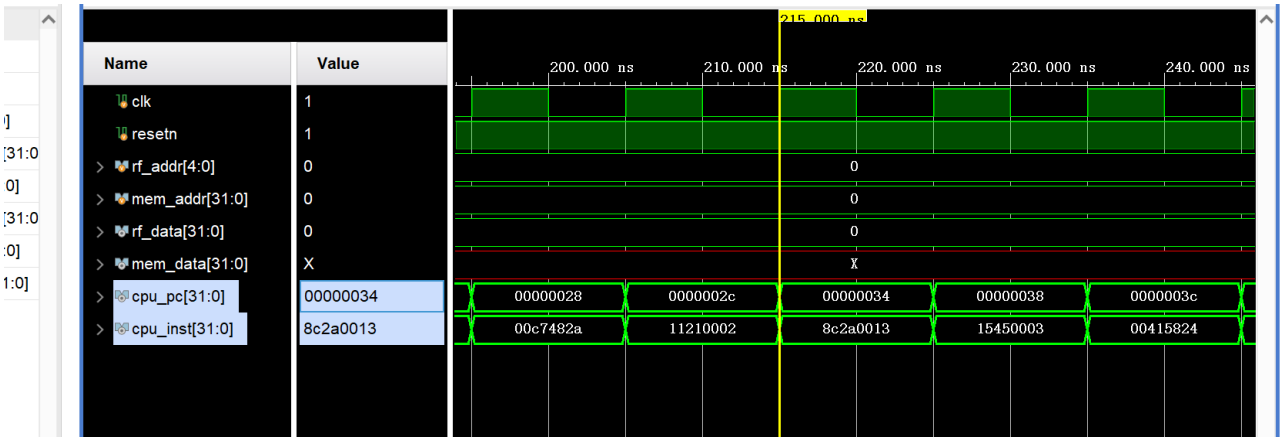
4.1.1 寄存器 BUG

需要将 `assign dm_wen = {4{inst_SW}} & resetn;`
改为 `assign dm_wen = {4{inst_SW}} & {4{resetn}};` 前者会导致将 `resetn` 左侧补 0 成 4 位再与 `{4{inst_SW}}` 进行与操作，得到错误的结果。



4.1.2 复现功能并解析指令

加入相关文件，运行仿真，可以看到 CPU 正常运作，比如这里从 2cH 跳转到 34H

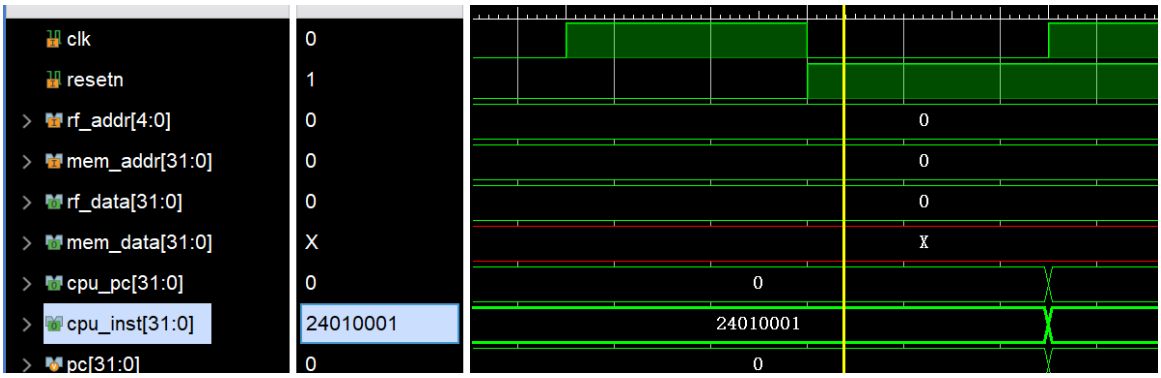


2CH	beq $9,$1,#2	跳转到指令 34H	11210002	0001_0001_0010_0001_0000_0000_0000_0010

1. addiu。立即数加法，将立即数 1 与恒零的 0 号寄存器的和赋值给 1 号寄存器。

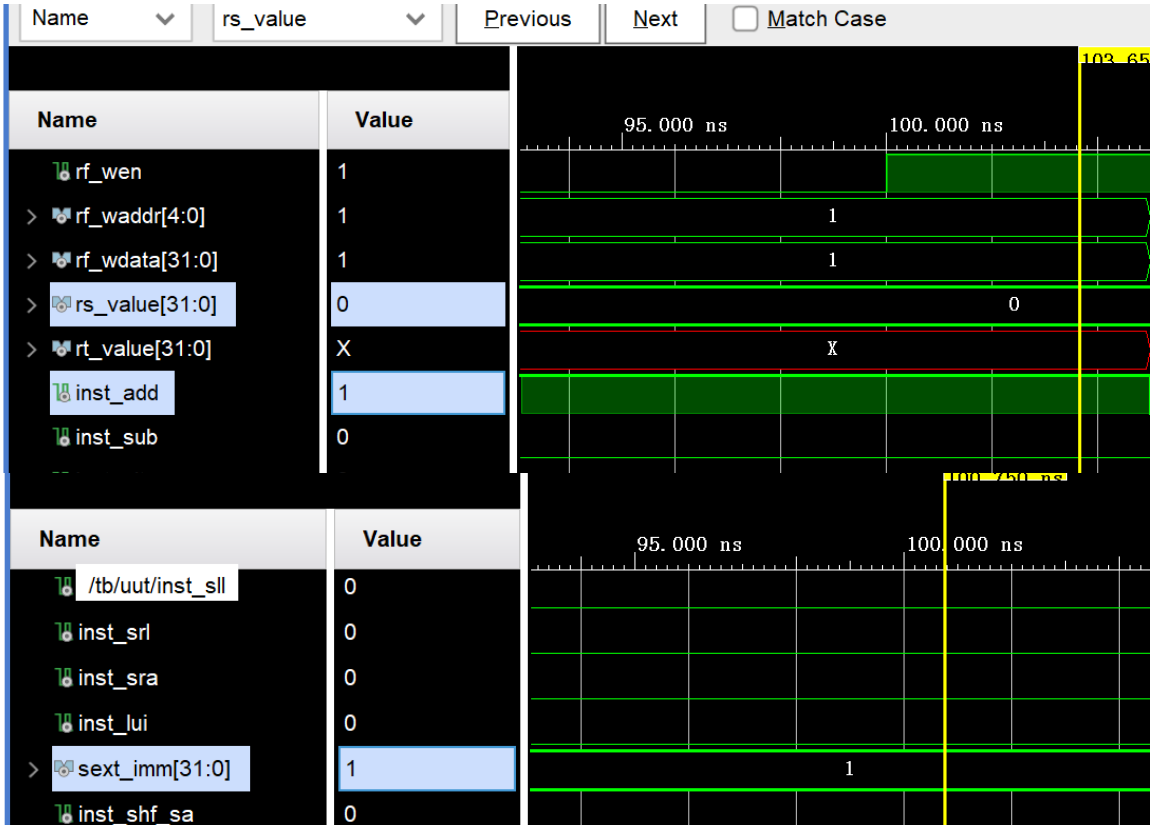
```
assign inst_rom[ 0] = 32'h24010001; // 00H: addiu $1 ,$0,#1    | $1 = 0000_0001H
```

(a) 取指。当 resetn 不为 0 时，从 rom 中得到指令，可以看到 `cpu_inst` 的值与上图 `inst_rom[0]` 的值一致。

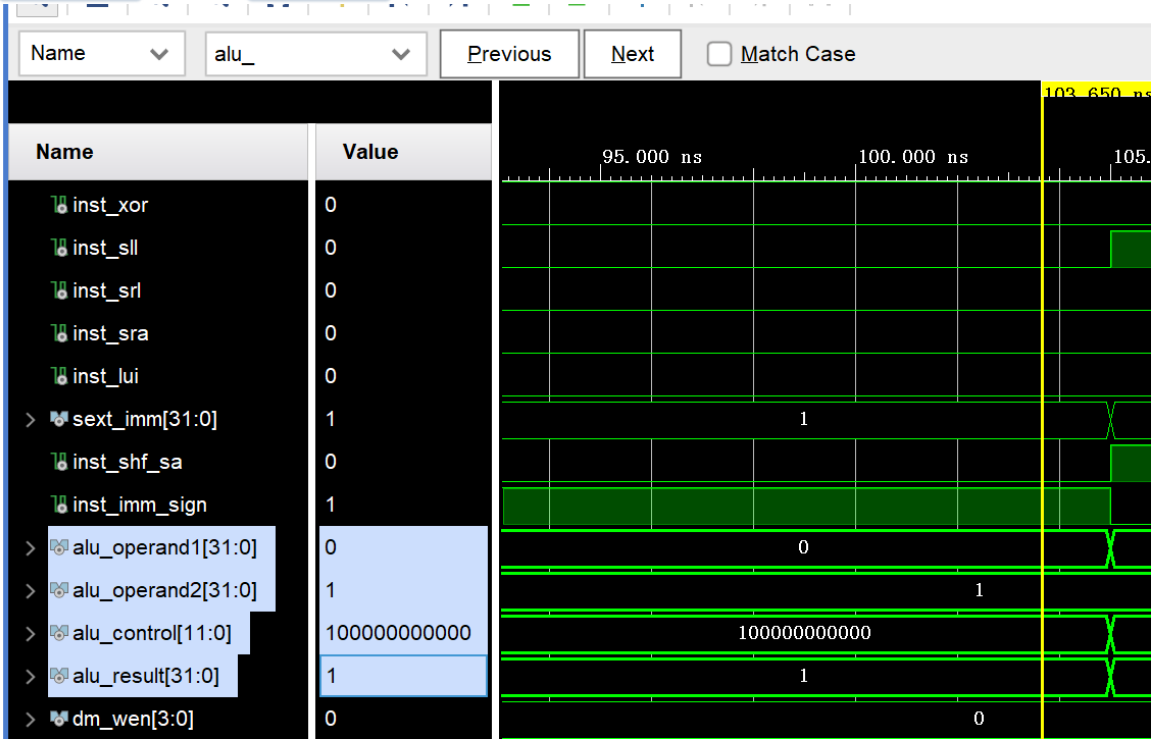



```
assign inst_rom[ 0] = 32'h24010001; // 00H: addiu $1 ,$0,#1    | $1 = 0000_0001H
```

(b) 译码。根据取得的指令，进行判断后，从寄存器取出 0 号寄存器，设置让 alu 做加法操作，以及处理立即数将其扩展成 32 位以便进行加法。

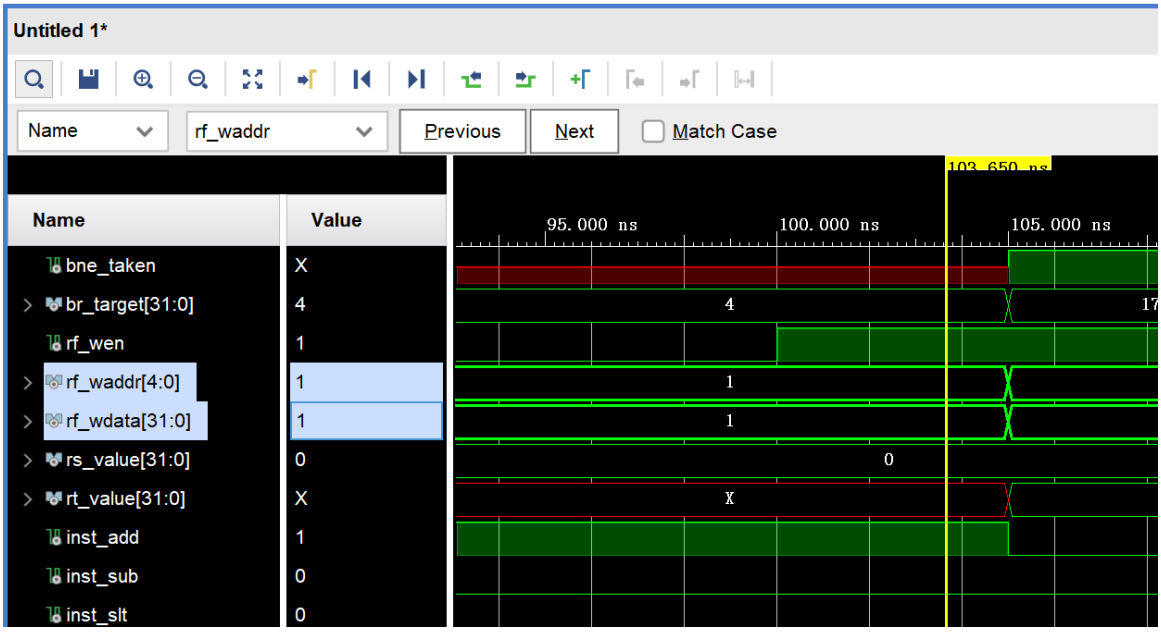


(c) 执行。将取出的 0 和立即数 1 放进 alu 进行运算并得到结果。这里 alu 的 `operand1` 是 0，`operand2` 是 1，控制信号表明进行加法，得到结果为 1。



(d) 访存。addiu 不涉及访存操作。

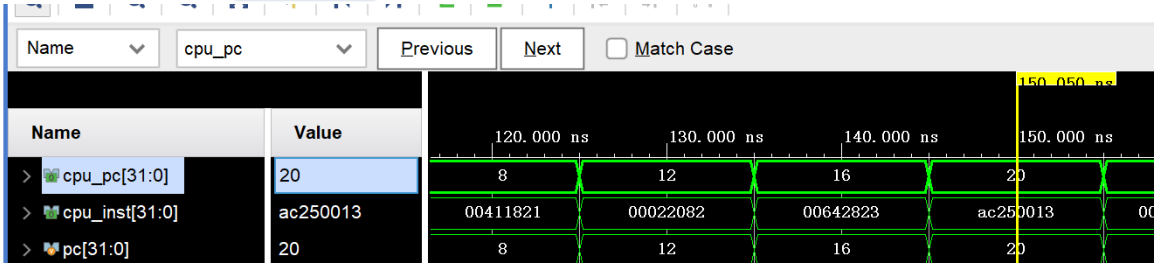
(e) 写回。将 alu 加法的结果写回寄存器 rt，即将 1 写入 1 号寄存器。



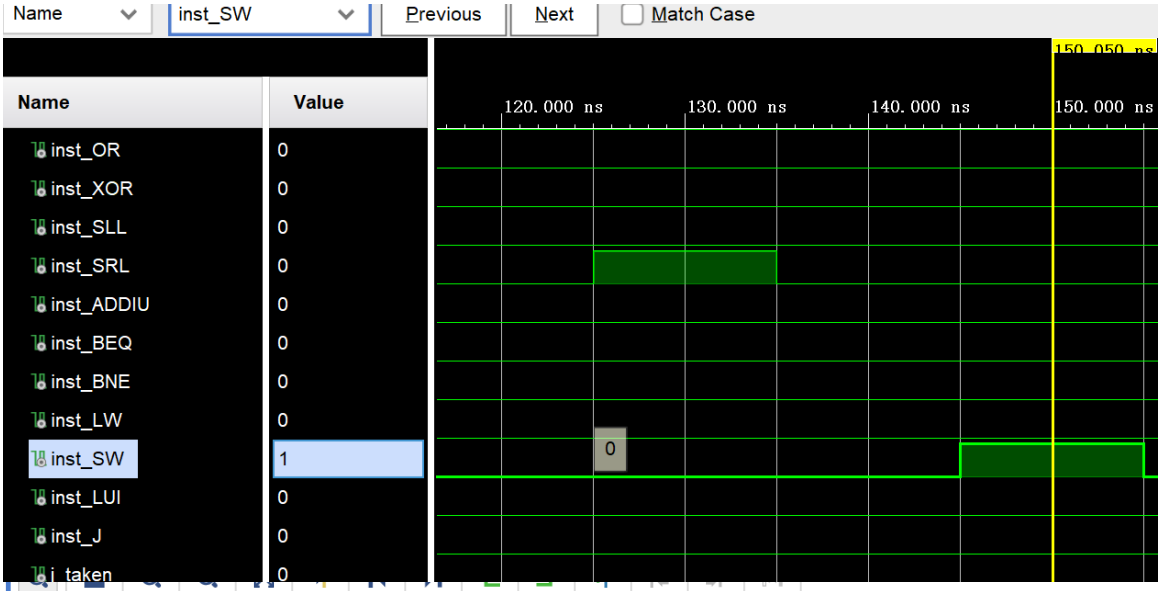
2. sw。向内存中存储字。将 5 号寄存器的值，存储到内存中 `1 号寄存器的值+偏移19` 地址处。

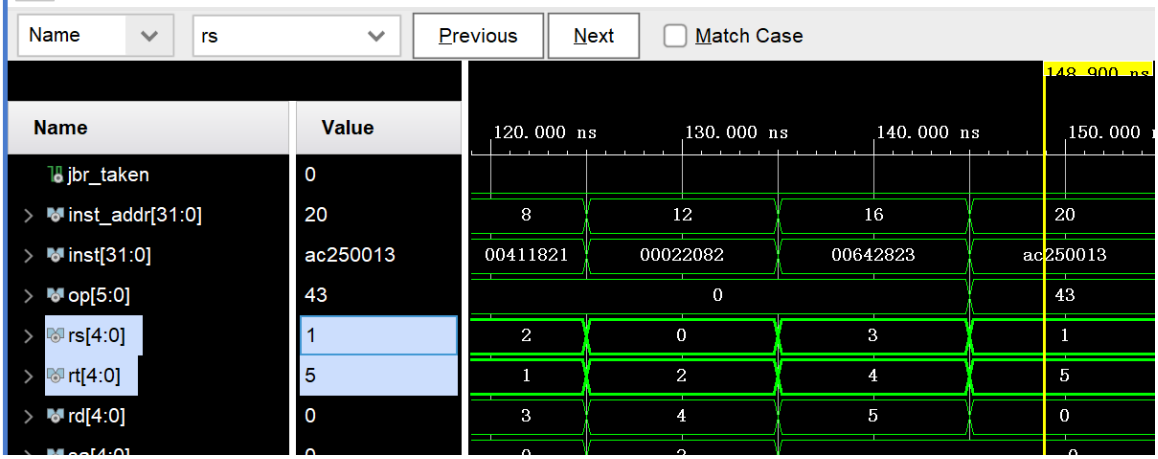
```
21  assign inst_rom[ 5] = 32'hAC250013; // 14H: sw    $5 ,#19($1) | Mem[0000_0014H] = 0000_000DH
```

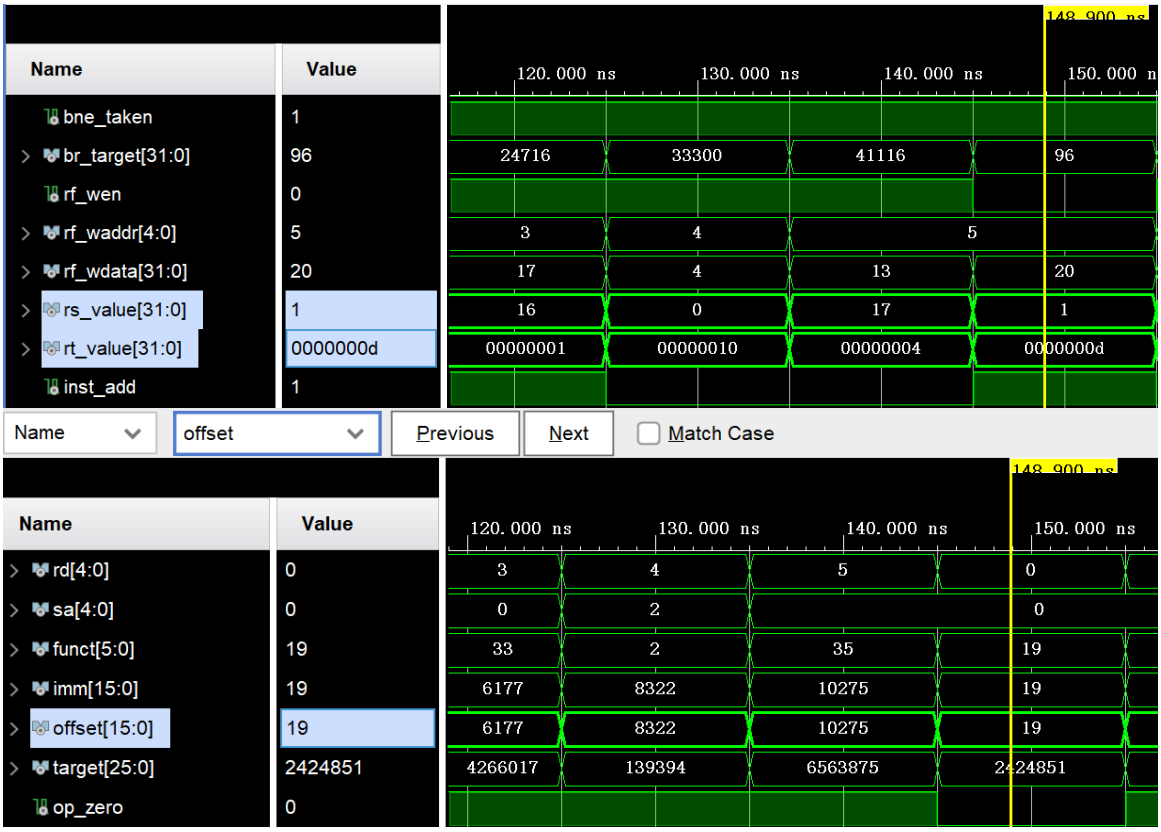
(a) 取指。可以看到 `cpu_isnt` 与上图相同。



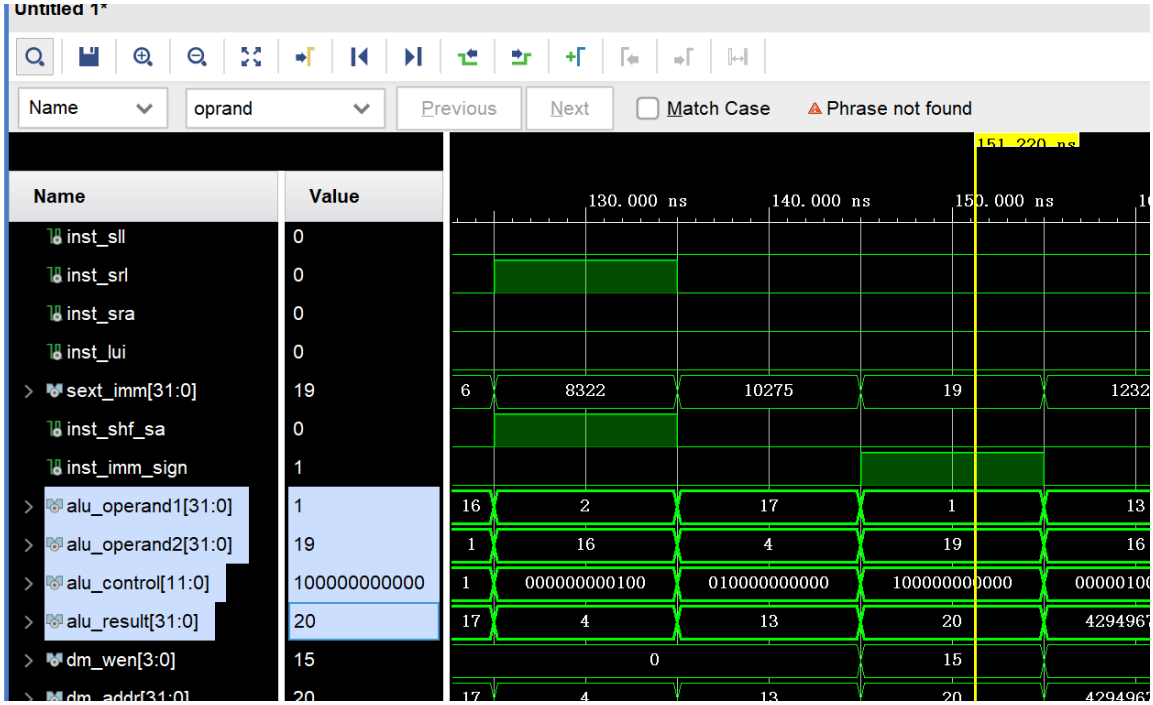
(b) 译码。从指令中解析出所需的各种东西。 `inst_SW` 被置为 1，从寄存器中取得 1、5 号寄存器的值，获取偏移量 19.



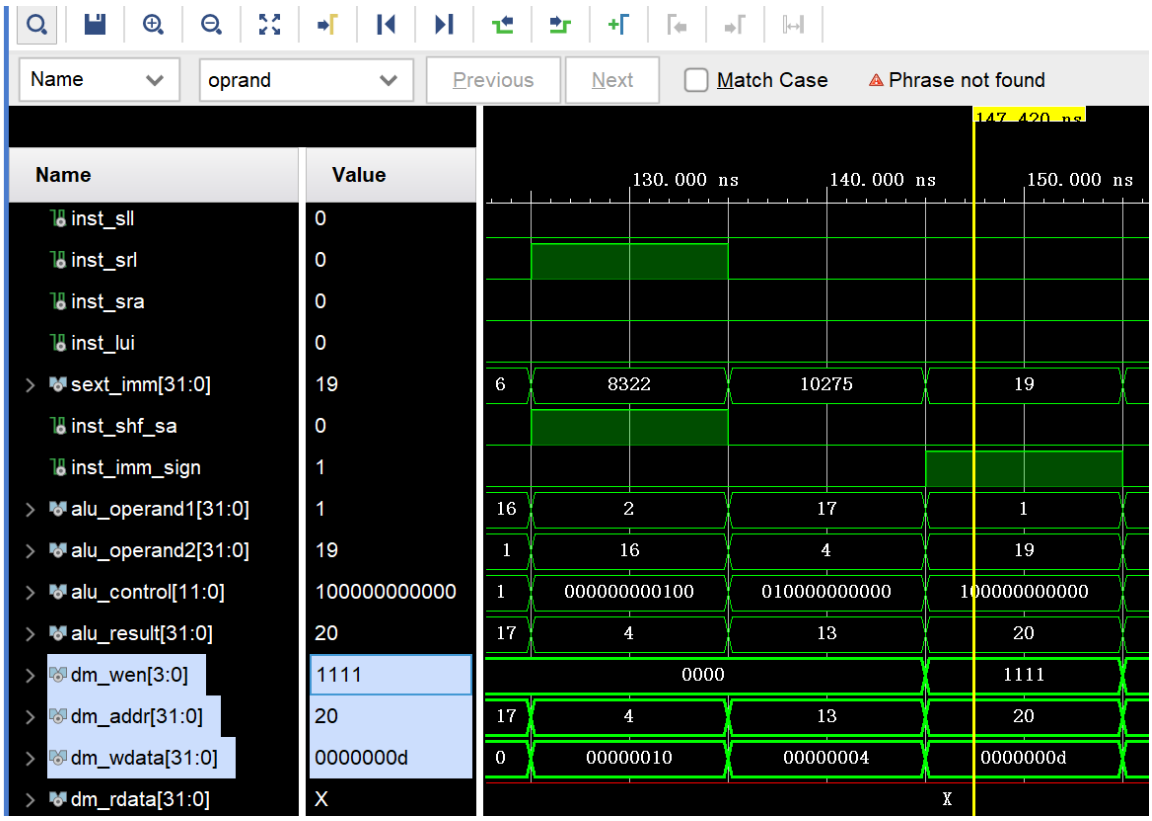




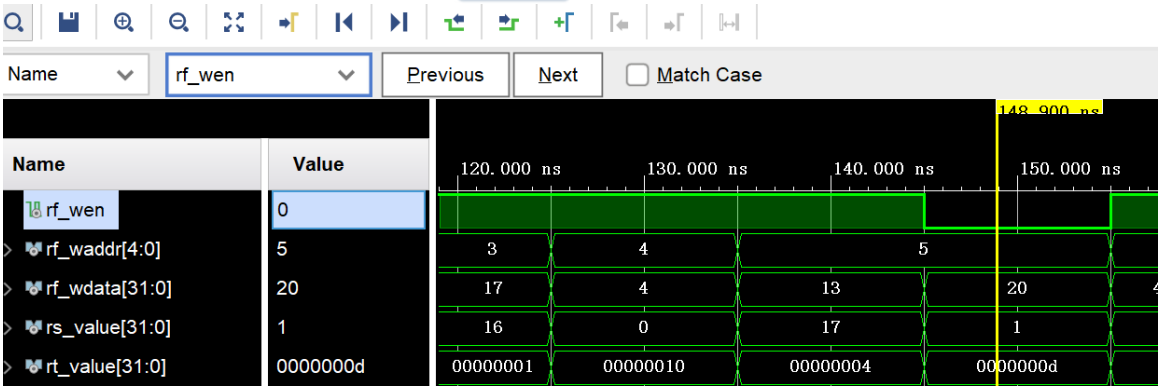
(c) 执行。进行加法得到需要写入的内存地址 20。



(d) 访存。访问内存，将 5 号寄存器中得到的值写入内存的偏移 20 位置处。



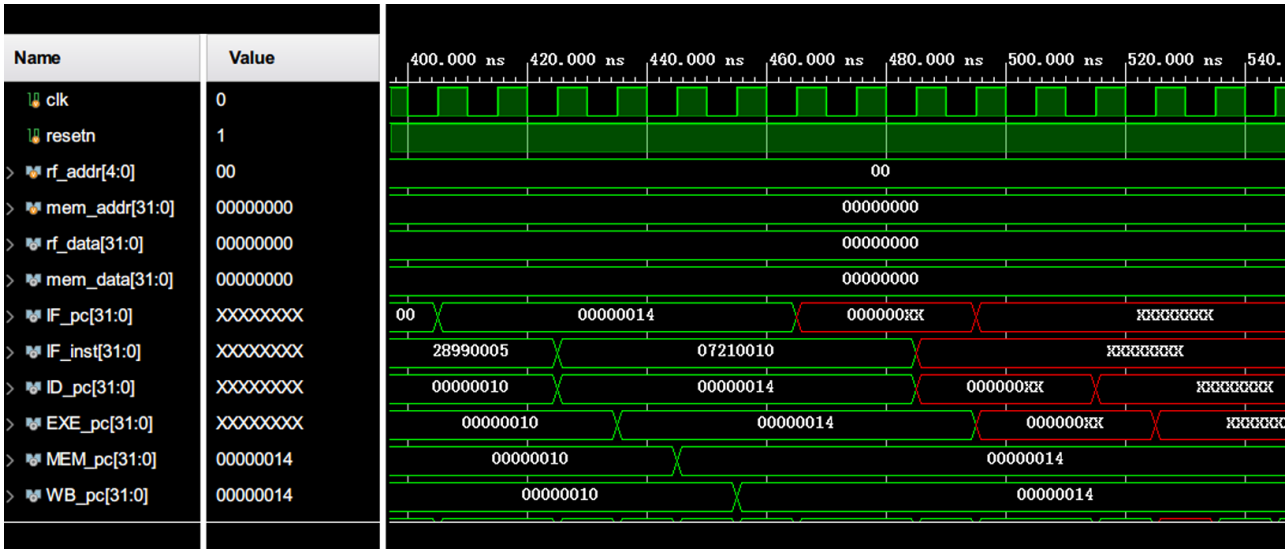
(e) 写回。不需要写回寄存器，因此 `rf_wen` 为 0.



4.2 多周期 CPU

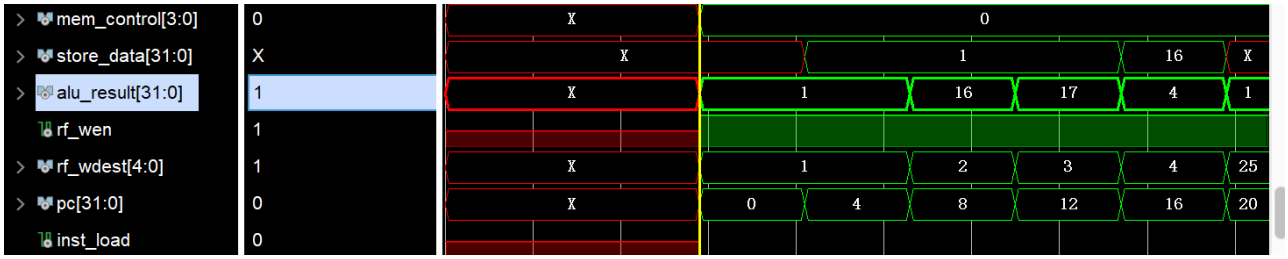
4.2.1 BUG 修复

执行14H `bgez $25,#16` 跳转到54H 07210010 之后，`IF_pc` 显示为 X，显然这里有 bug，跳转指令执行过程中发生了错误，导致后面全部都变成 X 了。

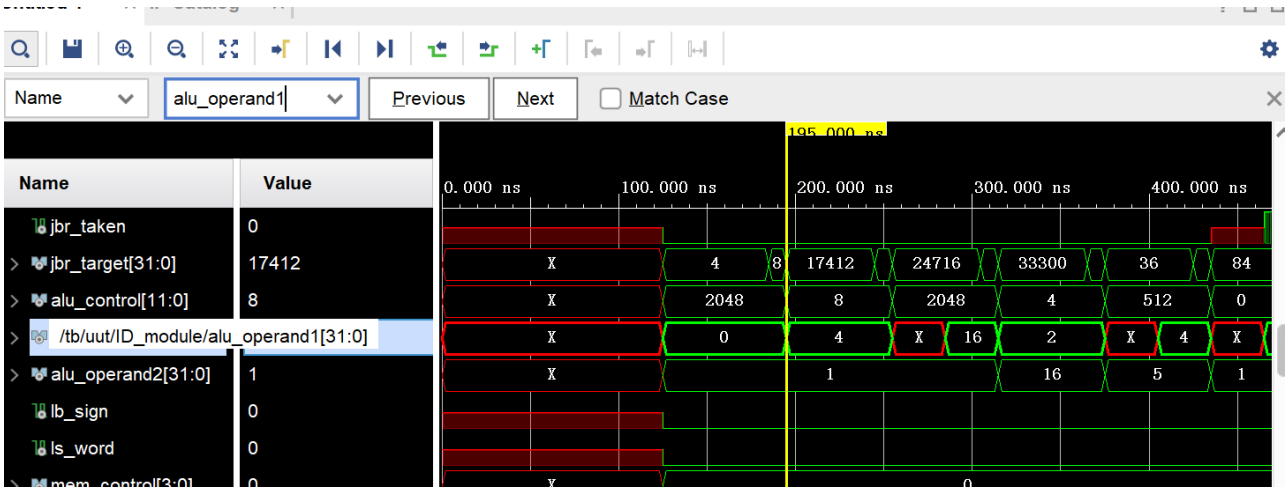


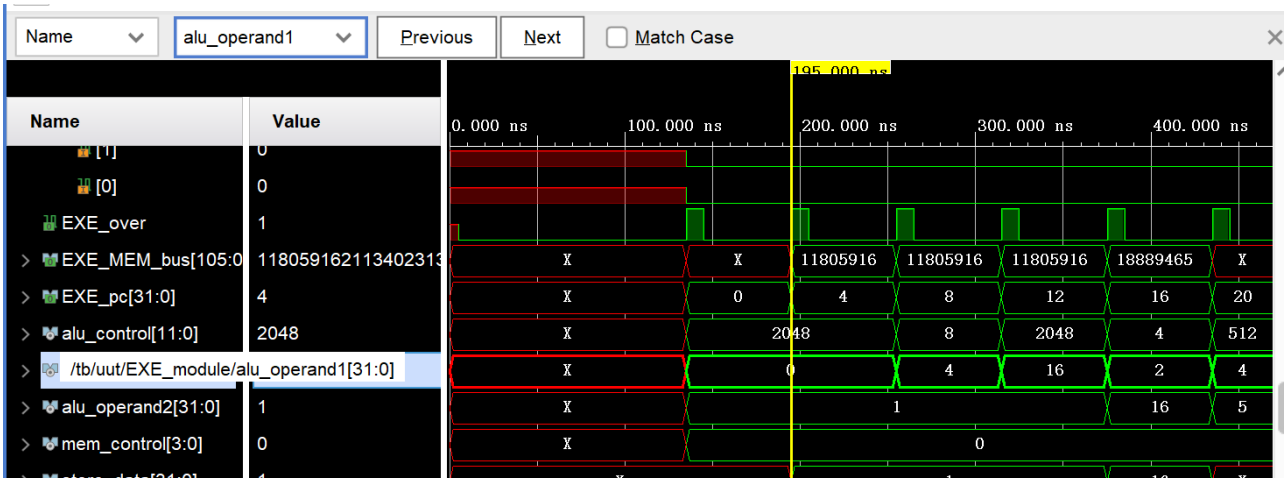
首先找到不对的东西，即 bgez 指令之后，pc 不正确了
然后根据数据的流动，谁赋值给 pc ，这个变量又被谁赋值，逐级往上查找，直到找到源头。

持续了两条指令的时间，显然不对。

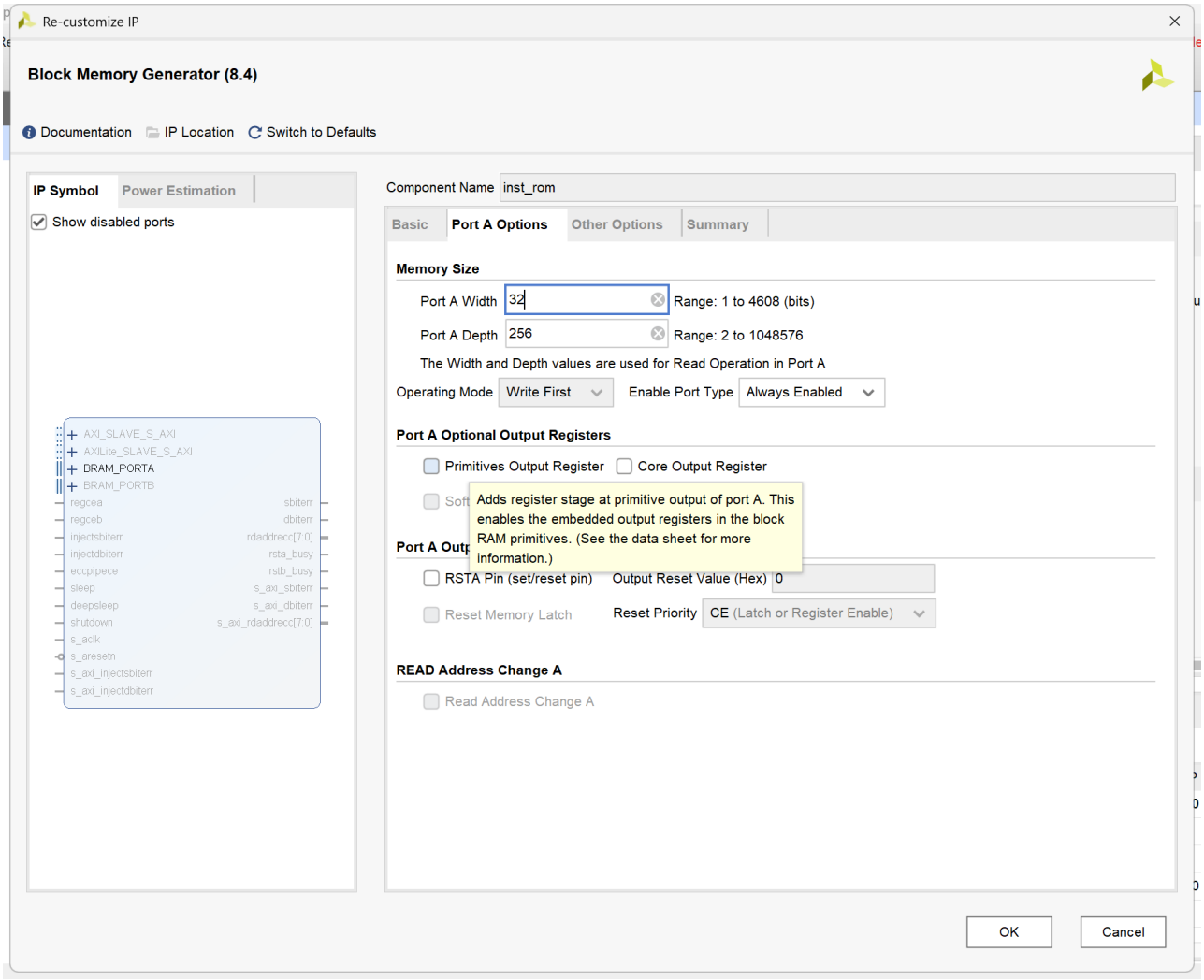


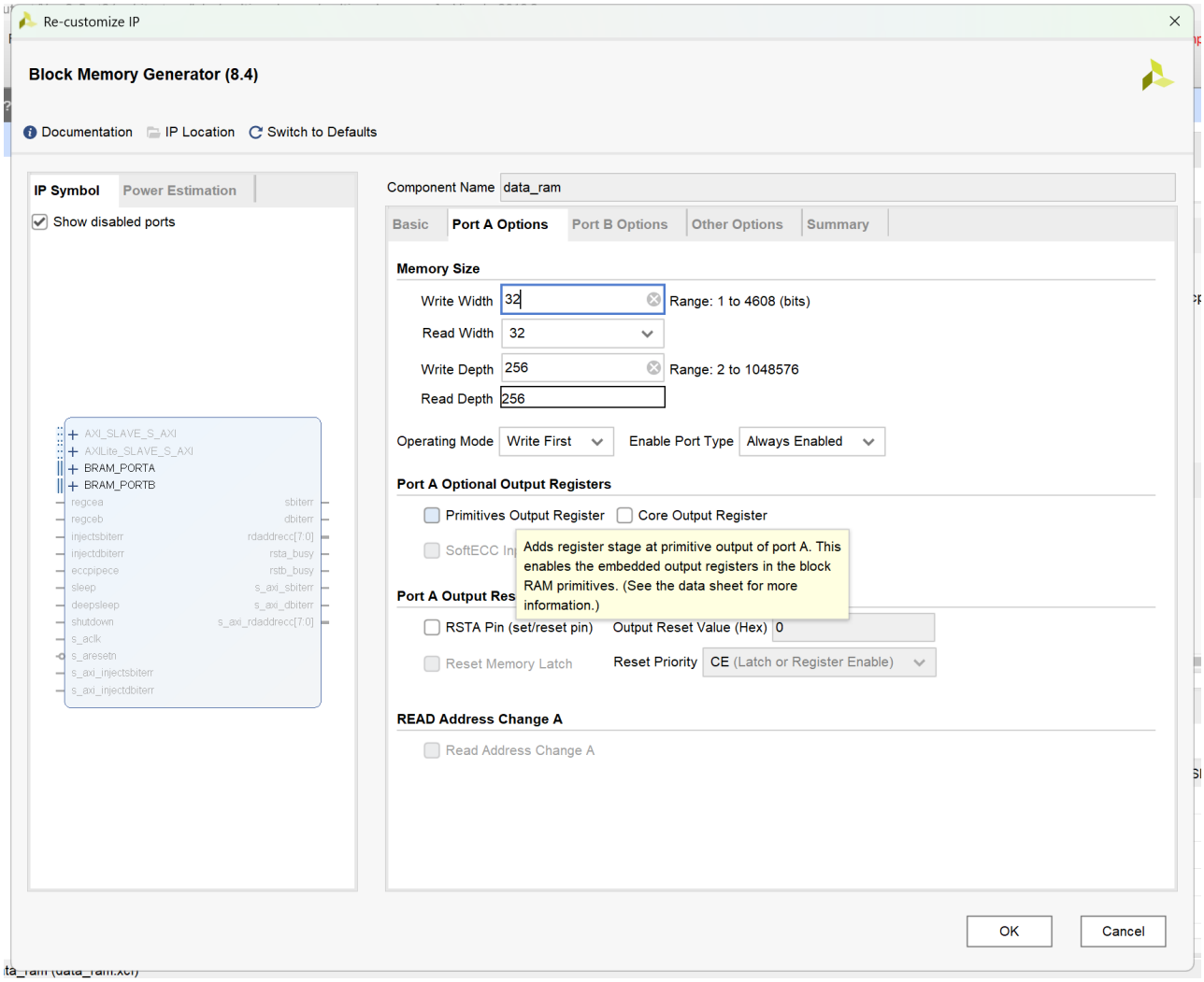
从第二条指令开始，exe 模块中的数据慢一条指令



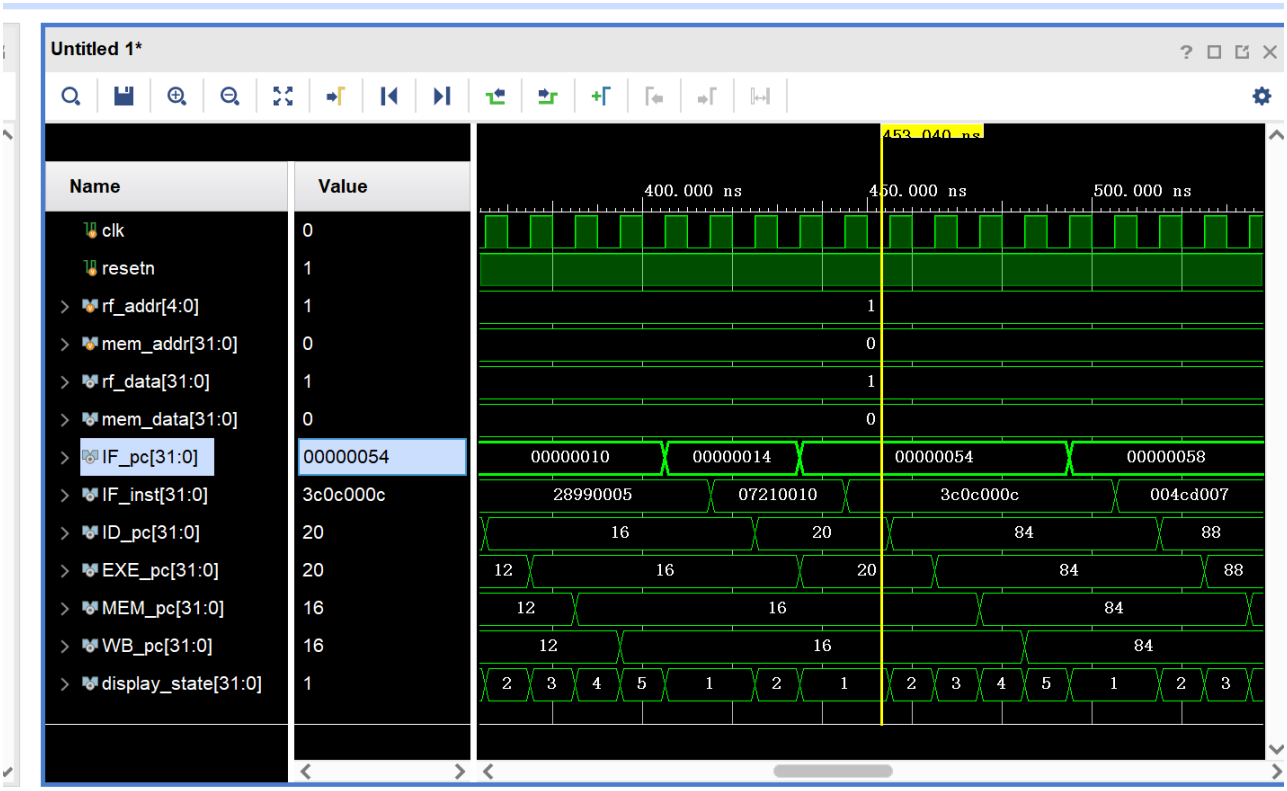


之后就可以发现，在 `IF_ID_bus` 锁存后，数据还有变动，数据的变动慢了一拍。之后发现是因为这里 `Primitives Output Register` 选项，导致读取时，需要 2 个 cycle 而不是 1 个 cycle 才能拿到想要的数据，与我们 `multi_cycle_cpu` 设计的假设不同，故产生错误。取消该选项即可。同理可知，data_ram 也会有相同的情况，也取消。




Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
AXI_SLAVE_S_AXI
AXILite_SLAVE_S_AXI
BRAM_PORTA
BRAM_PORTB
regcea
regceb
injectsbiterr
injectdbiterr
eccpipece
sleep
deepsleep
shutdown
s_aclk
s_aresetn
s_axi_injectsbiterr
s_axi_injectdbiterr
sbiterr
dbiterr
rdaddrecc[7:0]
rsta_busy
rstb_busy
s_axi_sbiterr
s_axi_dbiterr
s_axi_rdaddrecc[7:0]
Component Name
data_ram
Basic
Port A Options
Port B Options
Other Options
Summary
Memory Size
Write Width
32
Range: 1 to 4608 (bits)
Read Width
32
Write Depth
256
Range: 2 to 1048576
Read Depth
256
Operating Mode
Write First
Enable Port Type
Always Enabled
Port A Optional Output Registers
Primitives Output Register
Core Output Register
SoftECC In
Adds register stage at primitive output of port A. This enables the embedded output registers in the block RAM primitives. (See the data sheet for more information.)
Port A Output Res
RSTA Pin (set/reset pin)
Output Reset Value (Hex)
0
Reset Memory Latch
Reset Priority
CE (Latch or Register Enable)
READ Address Change A
Read Address Change A
OK
Cancel

修改之后，CPU 可以正确往下执行指令。



4.2.2 自定义 MIPS 指令

添加三个新运算，分别是**有符号数的大于置位、同或、低位加载**。
修改的代码如下

4.2.2.1 `multi_cycle_cpu.v` :

因为独热编码位宽增加，需要修改总线的位宽。

```
134 //------------------------{5级间的总线}begin--------------------
135     wire [ 63:0] IF_ID_bus;   // IF->ID级总线
136     wire [149:0] ID_EXE_bus;  // ID->EXE级总线
137     wire [105:0] EXE_MEM_bus; // EXE->MEM级总线
138     wire [ 69:0] MEM_WB_bus;  // MEM->WB级总线
139
140     //锁存以上总线信号
141     reg [ 63:0] IF_ID_bus_r;
142     reg [149:0] ID_EXE_bus_r;
143     reg [105:0] EXE_MEM_bus_r;
144     reg [ 69:0] MEM_WB_bus_r;
```

```
134 //------------------------{5级间的总线}begin--------------------
135     wire [ 63:0] IF_ID_bus;   // IF->ID级总线
136     wire [152:0] ID_EXE_bus;  // ID->EXE级总线 //FIXME 多增加3位新
137     wire [105:0] EXE_MEM_bus; // EXE->MEM级总线
138     wire [ 69:0] MEM_WB_bus;  // MEM->WB级总线
139
140     //锁存以上总线信号
141     reg [ 63:0] IF_ID_bus_r;
142     reg [152:0] ID_EXE_bus_r;  //FIXME 新增 3个运算
143     reg [105:0] EXE_MEM_bus_r;
144     reg [ 69:0] MEM_WB_bus_r;
```

4.2.2.2 `decode.v`:

修改总线的位宽，`alu_control` 的位宽，以及为增添的三个运算添加信号以译码。

```
17    output            ID_over,     // ID模块执行完成
18    output    [149:0] ID_EXE_bus,  // ID->EXE总线
19
```

```
17    output            ID_over,     // ID模块执行完成
18    output    [152:0] ID_EXE_bus,  // ID->EXE总线 //FIXME 多增加3
19
```

```
wire inst_BLTZ, inst_LB   , inst_LBU , inst_SB;
wire inst_ANDI, inst_ORI  , inst_XORI, inst_JAL;
wire op_zero;  // 操作码全0
wire sa_zero;  // sa域全0
assign op_zero = ~(|op);
assign sa_zero = ~(|sa);
assign inst_ADDU  = op_zero & sa_zero    & (funct == 6'b10000
```

```
wire inst_BLTZ, inst_LB   , inst_LBU , inst_SB;
wire inst_ANDI, inst_ORI  , inst_XORI, inst_JAL;
wire inst_XNOR, inst_SGT, inst_LLI;  //FIXME 多增加3位新增运算
wire op_zero;  // 操作码全0
wire sa_zero;  // sa域全0
assign op_zero = ~(|op);
assign sa_zero = ~(|sa);
assign inst_XNOR =  op_zero & sa_zero    & (funct == 6'b01010
assign inst_SGT  =  op_zero & sa_zero    & (funct == 6'b10100
assign inst_LLI  =   (op == 6'b011111) & (rs==5'd0);  //FIXME
assign inst_ADDU  = op_zero & sa_zero    & (funct == 6'b10000
```

```
//alu操作分类
wire inst_add, inst_sub, inst_slt,inst_sltu;
```

```
//alu操作分类
wire inst_xnor, inst_sgt, inst_lli; //FIXME 新增 3个运算
wire inst_add, inst_sub, inst_slt,inst_sltu;
```

```
assign inst_lui = inst_LUI;                    // 立
//使用sa域作为偏移量的移位指令
wire inst_shf_sa;
assign inst_shf_sa =  inst_SLL | inst_SRL | inst_SRA;
//依据立即数扩展方式分类
wire inst_imm_zero; //立即数0扩展
wire inst_imm_sign; //立即数符号扩展
assign inst_imm_zero = inst_ANDI  | inst_LUI  | inst_ORI | in
assign inst_imm_sign = inst_ADDIU | inst_SLTI | inst_SLTIU
                     | inst_load  | inst_store;
//依据目的寄存器号分类
wire inst_wdest_rt;  // 寄存器堆写入地址为rt的指令
wire inst_wdest_31;  // 寄存器堆写入地址为31的指令
wire inst_wdest_rd;  // 寄存器堆写入地址为rd的指令
assign inst_wdest_rt = inst_imm_zero | inst_ADDIU | inst_SLTI
                     | inst_SLTIU    | inst_load;
assign inst_wdest_31 = inst_JAL;
assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT  | in
                     | inst_JALR | inst_AND  | inst_NOR  | in
                     | inst_XOR  | inst_SLL  | inst_SLLV | in
                     | inst_SRAV | inst_SRL  | inst_SRLV;
//-----{指令译码}end
//-----{分支指令执行}begin
//无条件跳转
wire        j_taken;
wire [31:0] j_target;
```

```
assign inst_lui = inst_LUI;                    // 立
assign inst_xnor = inst_XNOR; //FIXME 新增 同或
assign inst_sgt = inst_SGT;
assign inst_lli = inst_LLI; //FIXME 新增 低位加载
//使用sa域作为偏移量的移位指令
wire inst_shf_sa;
assign inst_shf_sa =  inst_SLL | inst_SRL | inst_SRA;
//依据立即数扩展方式分类
wire inst_imm_zero; //立即数0扩展
wire inst_imm_sign; //立即数符号扩展
assign inst_imm_zero = inst_ANDI  | inst_LUI  | inst_ORI | in
                     | inst_LLI; //FIXME 新增 的LLI
assign inst_imm_sign = inst_ADDIU | inst_SLTI | inst_SLTIU
                     | inst_load  | inst_store;
//依据目的寄存器号分类
wire inst_wdest_rt;  // 寄存器堆写入地址为rt的指令
wire inst_wdest_31;  // 寄存器堆写入地址为31的指令
wire inst_wdest_rd;  // 寄存器堆写入地址为rd的指令
assign inst_wdest_rt = inst_imm_zero | inst_ADDIU | inst_SLTI
                     | inst_SLTIU    | inst_load;
assign inst_wdest_31 = inst_JAL;
assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT  | in
                     | inst_JALR | inst_AND  | inst_NOR  | in
                     | inst_XOR  | inst_SLL  | inst_SLLV | in
                     | inst_SRAV | inst_SRL  | inst_SRLV
                     | inst_XNOR | inst_SGT;  //FIXME 新增的两
//-----{指令译码}end
//-----{分支指令执行}begin
//无条件跳转
wire        j_taken;
wire [31:0] j_target;
```

```
//EXE需要用到的信息
//ALU两个源操作数和控制信号
wire [11:0] alu_control;
wire [31:0] alu_operand1;
wire [31:0] alu_operand2;
//所谓链接跳转是将跳转返回的PC值存放到31号寄存器里
//在多周期CPU里，不考虑延迟槽，故链接跳转需要计算PC+4，存放到31号
assign alu_operand1 = inst_j_link ? pc :
                      inst_shf_sa ? {27'd0,sa} : rs_value;
assign alu_operand2 = inst_j_link ? 32'd4 :
                      inst_imm_zero ? {16'd0, imm} :
                      inst_imm_sign ?  {{16{imm[15]}}, imm} :
assign alu_control = {inst_add,        // ALU操作码，独热编码
                      inst_sub,
```

```
//EXE需要用到的信息
//ALU两个源操作数和控制信号
wire [14:0] alu_control; //FIXME 多增加3位新增运算 control
wire [31:0] alu_operand1;
wire [31:0] alu_operand2;
//所谓链接跳转是将跳转返回的PC值存放到31号寄存器里
//在多周期CPU里，不考虑延迟槽，故链接跳转需要计算PC+4，存放到31号
assign alu_operand1 = inst_j_link ? pc :
                      inst_shf_sa ? {27'd0,sa} : rs_value;
assign alu_operand2 = inst_j_link ? 32'd4 :
                      inst_imm_zero ? {16'd0, imm} :
                      inst_imm_sign ?  {{16{imm[15]}}, imm} :
assign alu_control = {inst_xnor, //FIXME 新增 3个运算
                      inst_sgt, //FIXME 新增 3个运算
                      inst_lli, //FIXME 新增 3个运算
                      inst_add,        // ALU操作码，独热编码
                      inst_sub,
```

4.2.2.3 `exe.v`:

修改总线和 `alu_control` 的位宽。

```
8  module exe(                          // 执行级
9      input              EXE_valid,    // 执行级有效信号
10     input      [149:0] ID_EXE_bus_r,// ID->EXE总线
11     output             EXE_over,     // EXE模块执行完成
12     output     [105:0] EXE_MEM_bus, // EXE->MEM总线
13
14     //展示PC
15     output     [ 31:0] EXE_pc
16 );
17 //-----{ID->EXE总线}begin
18     //EXE需要用到的信息
19     //ALU两个源操作数和控制信号
20     wire [11:0] alu_control;
21     wire [31:0] alu_operand1;
22     wire [31:0] alu_operand2;
```

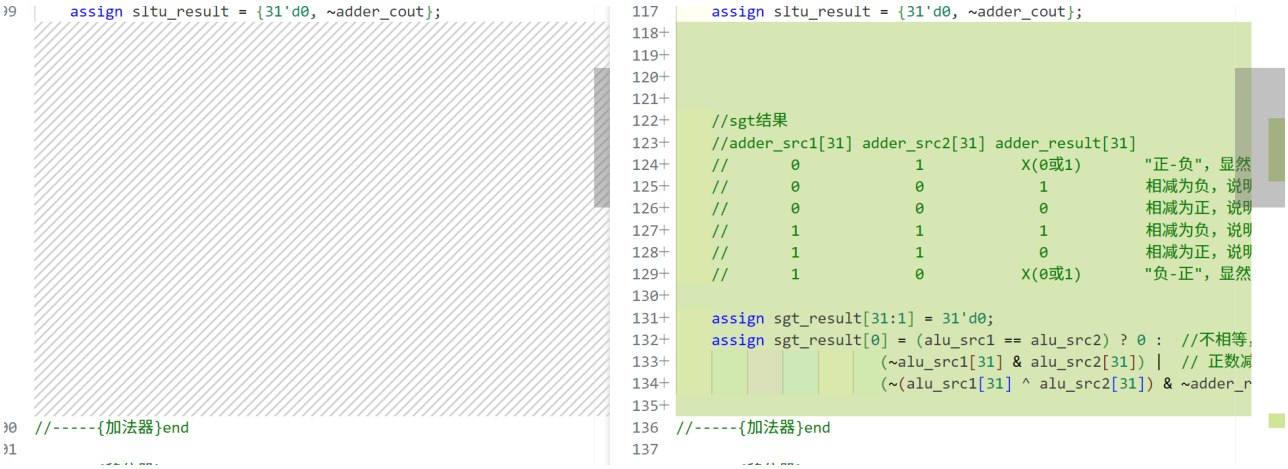
```
8  module exe(                          // 执行级
9      input              EXE_valid,    // 执行级有效信号
10     input      [152:0] ID_EXE_bus_r,// ID->EXE总线 //FIXME 新增 3
11     output             EXE_over,     // EXE模块执行完成
12     output     [105:0] EXE_MEM_bus, // EXE->MEM总线
13
14     //展示PC
15     output     [ 31:0] EXE_pc
16 );
17 //-----{ID->EXE总线}begin
18     //EXE需要用到的信息
19     //ALU两个源操作数和控制信号
20     wire [14:0] alu_control;  //FIXME 新增 3个运算
21     wire [31:0] alu_operand1;
22     wire [31:0] alu_operand2;
```

4.2.2.4 alu.v:

添加相应在 alu 实验中添加过的运算，并增加独热编码位数。

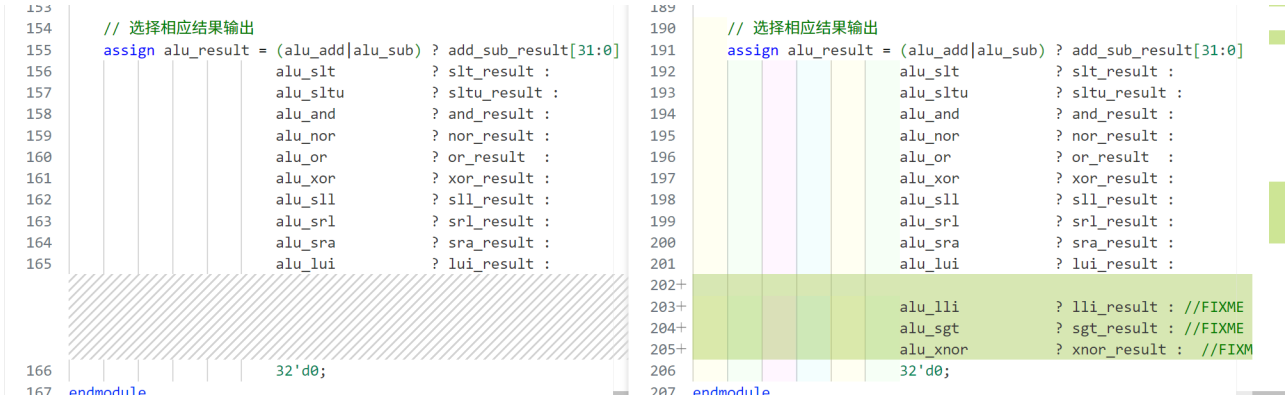
```
//   > 日期  : 2016-04-14
//*************************************************************
module alu(
    input  [11:0] alu_control,  // ALU控制信号
    input  [31:0] alu_src1,     // ALU操作数1,为补码
    input  [31:0] alu_src2,     // ALU操作数2，为补码
    output [31:0] alu_result    // ALU结果
    );

    // ALU控制信号，独热码
    wire alu_add;   //加法操作
    wire alu_sub;   //减法操作
    wire alu_slt;   //有符号比较，小于置位，复用加法器做减法
    wire alu_sltu;  //无符号比较，小于置位，复用加法器做减法
    wire alu_and;   //按位与
    wire alu_nor;   //按位或非
    wire alu_or;    //按位或
    wire alu_xor;   //按位异或
    wire alu_sll;   //逻辑左移
    wire alu_srl;   //逻辑右移
    wire alu_sra;   //算术右移
    wire alu_lui;   //高位加载

    assign alu_add  = alu_control[11];
    assign alu_sub  = alu_control[10];
    assign alu_slt  = alu_control[ 9];
    assign alu_sltu = alu_control[ 8];
    assign alu_and  = alu_control[ 7];
    assign alu_nor  = alu_control[ 6];
    assign alu_or   = alu_control[ 5];
    assign alu_xor  = alu_control[ 4];
    assign alu_sll  = alu_control[ 3];
    assign alu_srl  = alu_control[ 2];
    assign alu_sra  = alu_control[ 1];
    assign alu_lui  = alu_control[ 0];

    wire [31:0] add_sub_result;
    wire [31:0] slt_result;
    wire [31:0] sltu_result;
    wire [31:0] and_result;
    wire [31:0] nor_result;
    wire [31:0] or_result;
    wire [31:0] xor_result;
    wire [31:0] sll_result;
    wire [31:0] srl_result;
    wire [31:0] sra_result;
    wire [31:0] lui_result;

    assign and_result = alu_src1 & alu_src2;      // 与结果为两数按
    assign or_result  = alu_src1 | alu_src2;      // 或结果为两数按
    assign nor_result = ~or_result;               // 或非结果为或结
    assign xor_result = alu_src1 ^ alu_src2;      // 异或结果为两数
    assign lui_result = {alu_src2[15:0], 16'd0};  // 立即数装载结果

//-----{加法器}begin
//add,sub,slt,sltu均使用该模块
    wire [31:0] adder_operand1;
    wire [31:0] adder_operand2;
    wire        adder_cin     ;
    wire [31:0] adder_result  ;
    wire        adder_cout    ;
    assign adder_operand1 = alu_src1;
    assign adder_operand2 = alu_add ? alu_src2 : ~alu_src2;
    assign adder_cin      = ~alu_add; //减法需要cin
    adder adder_module(
```

```
//   > 日期  : 2016-04-14
//*************************************************************
module alu(
    input  [14:0] alu_control,  // ALU控制信号 //FIXME 新增三个运算
    input  [31:0] alu_src1,     // ALU操作数1,为补码
    input  [31:0] alu_src2,     // ALU操作数2，为补码
    output [31:0] alu_result    // ALU结果
    );

    // ALU控制信号，独热码
    wire alu_add;   //加法操作
    wire alu_sub;   //减法操作
    wire alu_slt;   //有符号比较，小于置位，复用加法器做减法
    wire alu_sltu;  //无符号比较，小于置位，复用加法器做减法
    wire alu_and;   //按位与
    wire alu_nor;   //按位或非
    wire alu_or;    //按位或
    wire alu_xor;   //按位异或
    wire alu_sll;   //逻辑左移
    wire alu_srl;   //逻辑右移
    wire alu_sra;   //算术右移
    wire alu_lui;   //高位加载

    wire alu_xnor; //FIXME 新增 同或
    wire alu_sgt;  //FIXME 有符号数大于置位
    wire alu_lli;   //FIXME 低位加载

    assign alu_xnor = alu_control[14];//FIXME 新增 同或
    assign alu_sgt = alu_control[13];//FIXME 有符号数大于置位
    assign alu_lli = alu_control[12]; //FIXME 低位加载

    assign alu_add  = alu_control[11];
    assign alu_sub  = alu_control[10];
    assign alu_slt  = alu_control[ 9];
    assign alu_sltu = alu_control[ 8];
    assign alu_and  = alu_control[ 7];
    assign alu_nor  = alu_control[ 6];
    assign alu_or   = alu_control[ 5];
    assign alu_xor  = alu_control[ 4];
    assign alu_sll  = alu_control[ 3];
    assign alu_srl  = alu_control[ 2];
    assign alu_sra  = alu_control[ 1];
    assign alu_lui  = alu_control[ 0];

    wire [31:0] add_sub_result;
    wire [31:0] slt_result;
    wire [31:0] sltu_result;
    wire [31:0] and_result;
    wire [31:0] nor_result;
    wire [31:0] or_result;
    wire [31:0] xor_result;
    wire [31:0] sll_result;
    wire [31:0] srl_result;
    wire [31:0] sra_result;
    wire [31:0] lui_result;

    wire [31:0] lli_result;//FIXME 新增
    wire [31:0] sgt_result;//FIXME 新增
    wire [31:0] xnor_result;//FIXME 新增

    assign and_result = alu_src1 & alu_src2;      // 与结果为两数按
    assign or_result  = alu_src1 | alu_src2;      // 或结果为两数按
    assign nor_result = ~or_result;               // 或非结果为或结
    assign xor_result = alu_src1 ^ alu_src2;      // 异或结果为两数
    assign lui_result = {alu_src2[15:0], 16'd0};  // 立即数装载结果

    assign xnor_result = ~(alu_src1 ^ alu_src2); //FIXME 修改 同或
    assign lli_result = {16'd0, alu_src2[15:0]}; //FIXME 修改 装在

//-----{加法器}begin
//add,sub,slt,sltu均使用该模块
    wire [31:0] adder_operand1;
    wire [31:0] adder_operand2;
    wire        adder_cin     ;
    wire [31:0] adder_result  ;
    wire        adder_cout    ;
    assign adder_operand1 = alu_src1;
    assign adder_operand2 = alu_add ? alu_src2 : ~alu_src2;
    assign adder_cin      = ~alu_add; //减法需要cin
    adder adder_module(
```



```
assign sltu_result = {31'd0, ~adder_cout};
//-----{加法器}end
```

```
assign sltu_result = {31'd0, ~adder_cout};

//sgt结果
//adder_src1[31] adder_src2[31] adder_result[31]
//       0            1           X(0或1)       "正-负", 显然
//       0            0             1           相减为负，说明
//       0            0             0           相减为正，说明
//       1            1             1           相减为负，说明
//       1            1             0           相减为正，说明
//       1            0           X(0或1)       "负-正", 显然

assign sgt_result[31:1] = 31'd0;
assign sgt_result[0] = (alu_src1 == alu_src2) ? 0 :  //不相等,
                       (~alu_src1[31] & alu_src2[31]) |  // 正数减
                       (~(alu_src1[31] ^ alu_src2[31]) & ~adder_r

//-----{加法器}end
```



```
// 选择相应结果输出
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 :
                    32'd0;
endmodule
```

```
// 选择相应结果输出
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_lli           ? lli_result : //FIXME
                    alu_sgt           ? sgt_result : //FIXME
                    alu_xnor          ? xnor_result :  //FIXM
                    32'd0;
endmodule
```

4.2.2.5 添加指令

向 rom 中写入三条新指令以验证

```
在0CH之后插入 这三条指令
sgt $11 $3 $2  // $3 = 17 > $2 = 16 $11置为1
xnor $12 $3 $2 // $3 = 0000_0011H $2 = 000_0010H , $12置为FFFF_FFFEH
lli $13,#12    // $13 置为0000_000CH
```

指令对应的 2进制编码和 16 进制编码为

```
sgt $11 $3 $2
0000_00 | 00_011 | 0_0010 | 0101_1 | 000_00 | 10_1000
00625828

xnor $12 $3 $2
0000_00 | 00_011 | 0_0010 | 0110_0 | 000_00 | 01_0101
00626015

lli $13,#12
0111_11 | 00_000 | 0_1101 | 0000_0000_0000_1100
7C0D000C
```

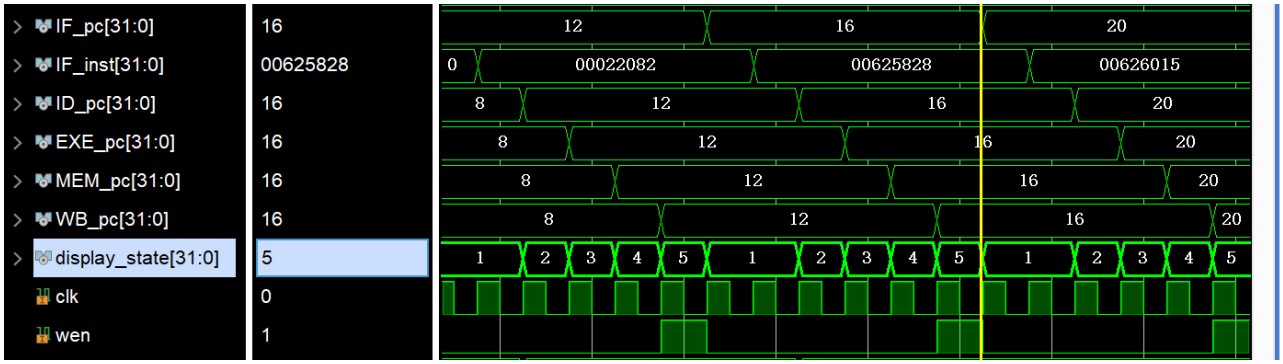
将 16 进制写入 coe 文件，在 rom 中载入，运行。

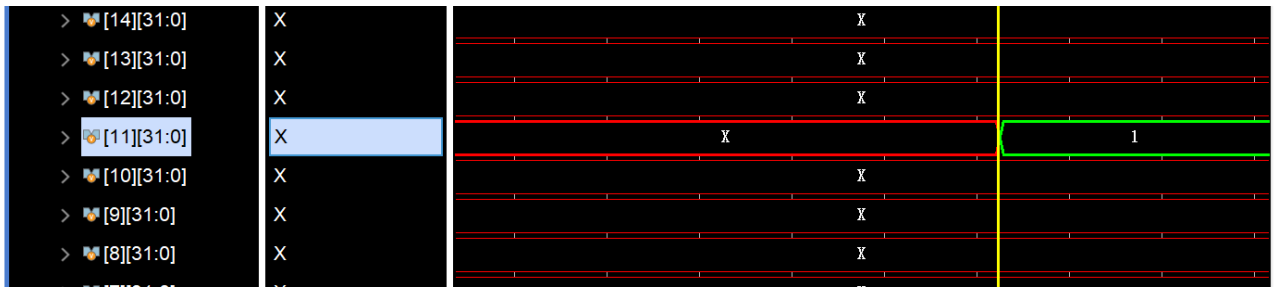
5 实验结果分析

5.1 多周期 CPU 添加 3 条新指令

5.1.1 有符号数大于置位

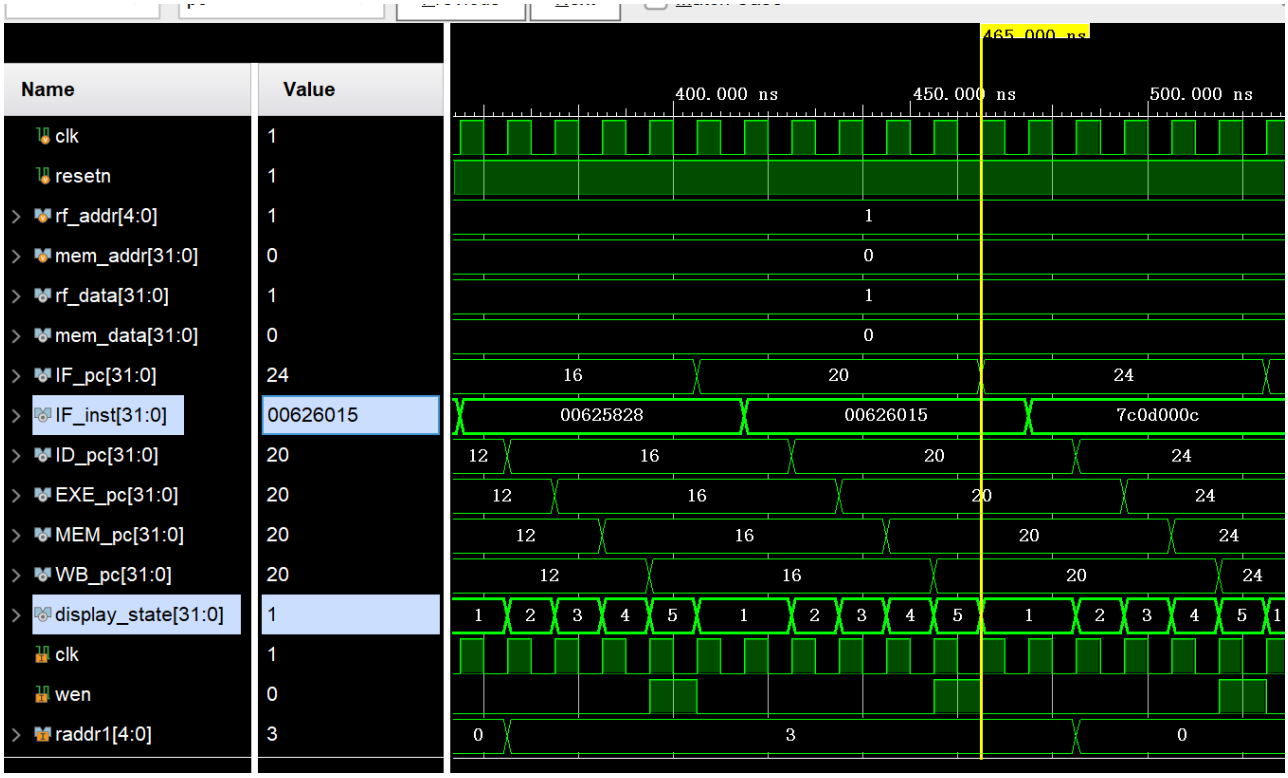
`sgt $11 $3 $2` 指令写回后，`rf[11] == 1`，结果与预期相符，正确。

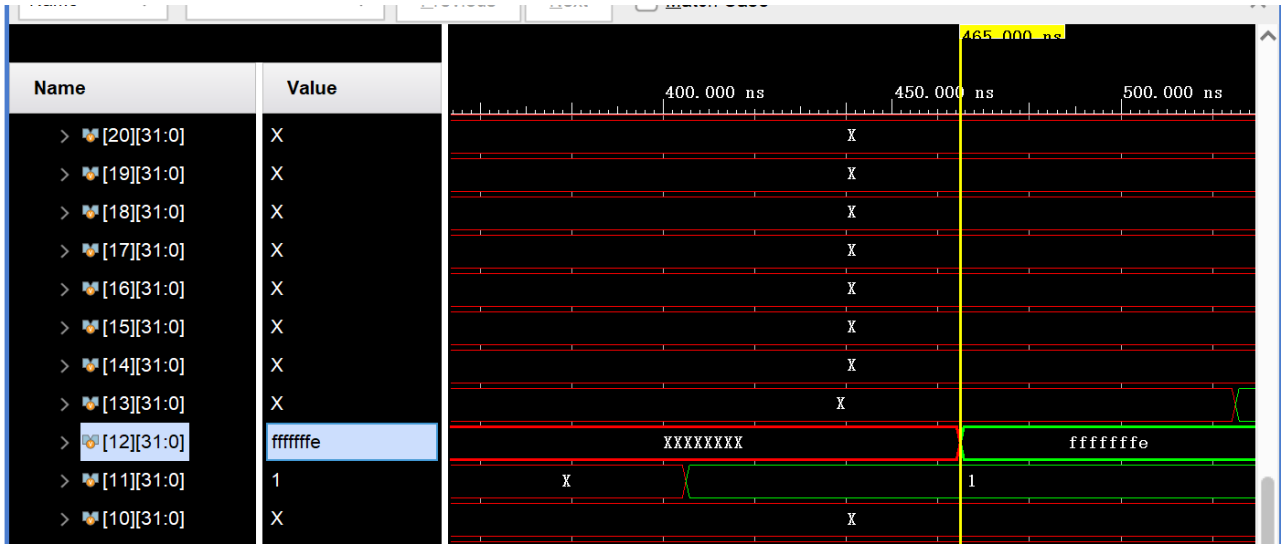




5.1.2 同或

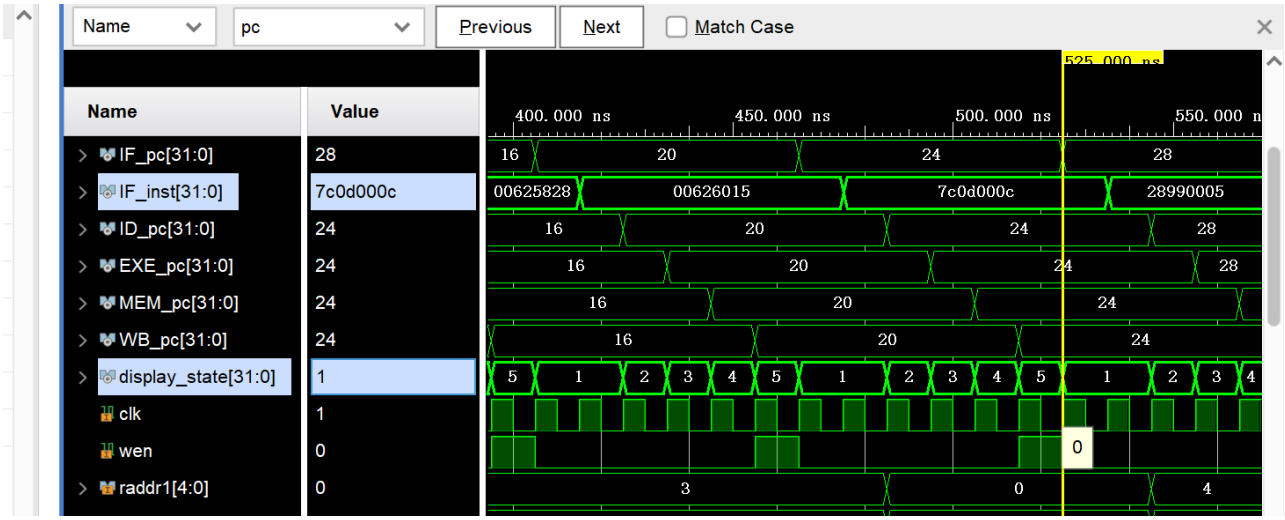
`xnor $12 $3 $2` 指令写回后，`rf[12] == 0xFFFF_FFFE`，符合预期，正确。

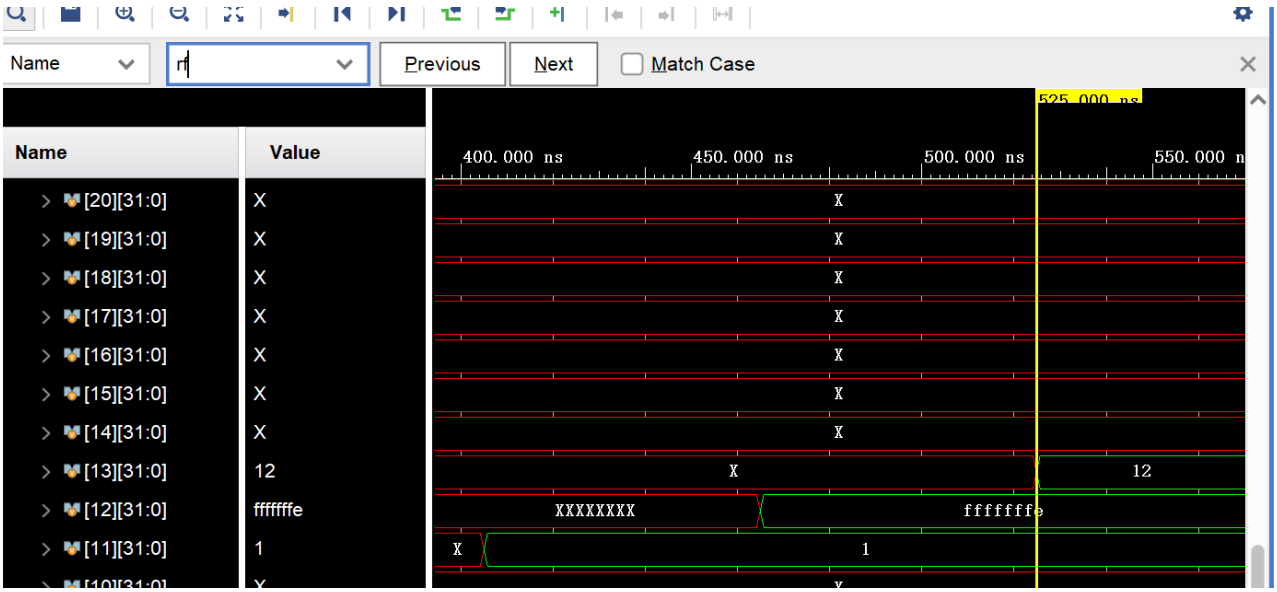




5.1.3 低位加载

`lli $13, #12` 指令写回后，`rf[13] == 12`，正确。





6 总结感想

单周期 CPU 实验让我熟悉了 CPU 运行的五个阶段：取指、译码、执行、访存、写回，为多周期 CPU 的实验打下了基础。

多周期 CPU 实验让我体会到多周期 CPU 是如何在时钟下一步一步地运作，各个模块协同过程也给我建立了一幅 CPU 运作的图景。添加指令让我更好地理解了 mips 的指令格式，以及译码等过程。