计算机组成原理与接口技术（实验）——基于MIPS架构

Apr, 2021

实验2&3 类MIPS单周期微处理器设计（第8~9周）

杨明
华中科技大学电子信息与通信学院
myang@hust.edu.cn

Agenda

- 实验内容
  - 目的
  - 任务及时间安排
  - 报告要求
- 原理回顾
  - 计算机工作原理
  - 简单微处理器的基本构成
- Vivado软件使用
  - 模块设计
  - 功能仿真

实验目的

- 了解微处理器的基本结构
- 掌握哈佛结构的计算机工作原理
- 学会设计简单的微处理器
- 了解软件控制硬件工作的基本原理

实验任务及时间安排

- 任务（实验教材第6章内容）
  - 基本要求：利用Verilog HDL语言，基于Xilinx FPGA nexys4实验平台，设计一个能够执行以下MIPS指令集的单周期类MIPS处理器，要求完成所有支持指令的功能仿真，验证指令执行的正确性，要求编写汇编程序将本人学号的ASCII码存入RAM的连续内存区域
    - 支持基本的算术逻辑运算如add，sub，and，or，slt，andi指令
    - 支持基本的内存操作如lw，sw指令
    - 支持基本的程序控制如beq，j指令
  - 提高要求：为该处理器提供简单的输入、输出功能，下载到Nexys4实验板进行验证(完成加分)。

[提醒1] andi 指令功能需要自己完成。
[提醒2] 在理解原理的基础上，自己想怎么编程就怎么编程。
[提醒3] 课本及课件步骤/代码不全，仅供参考。

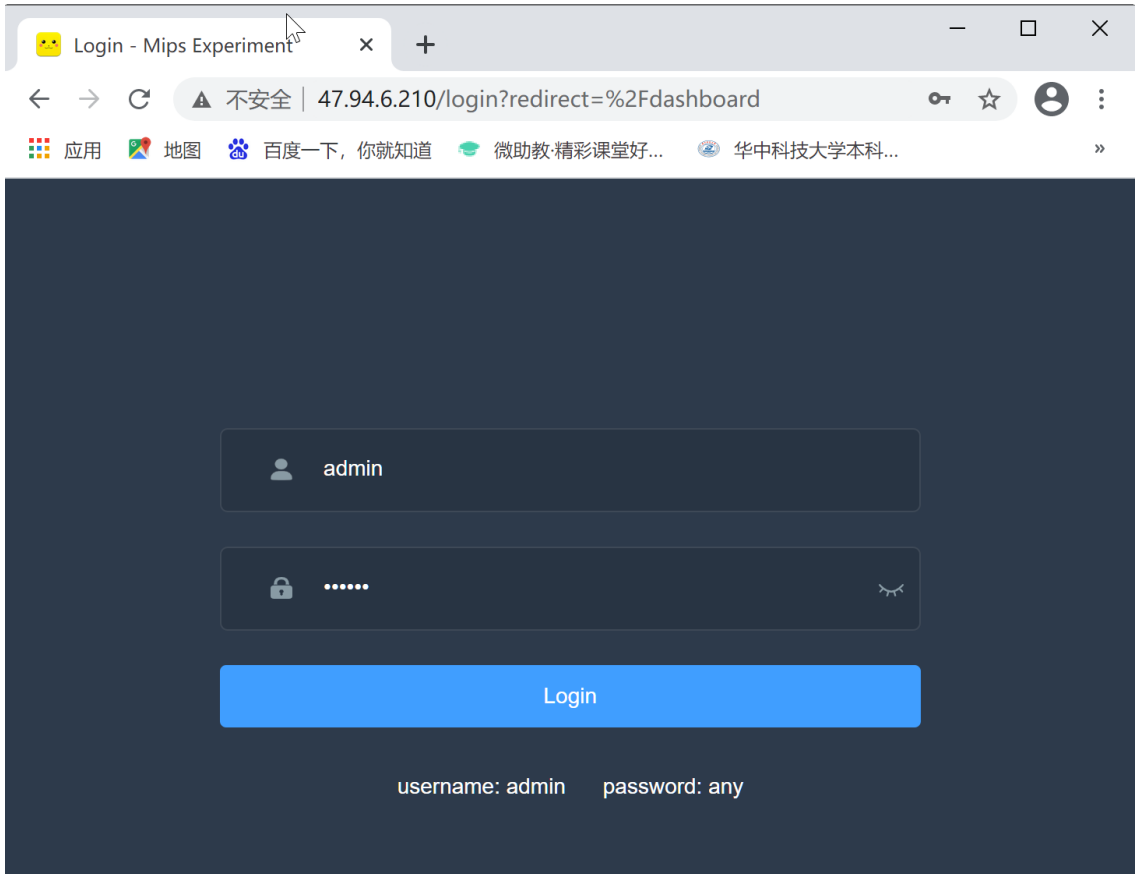
- 时间安排（第8~9周）
  - 课内：两次课
  - 课外：两周时间内自行安排

实验报告要求

- 1. 实验任务、目标
- 2. 微处理器各个模块硬件设计原理、verilog代码
- 3. Rom汇编程序设计、代码
- 4. 各个模块的仿真激励代码、仿真结果截图以及文字说明如何验证其正确性
- 5. 心得体会

Verilog代码提交

- 在网站提交
  - http://47.94.6.210



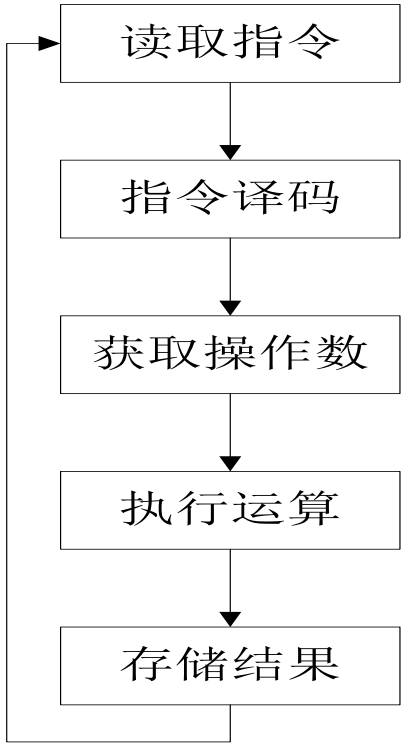
Agenda

- 实验内容
  - 目的
  - 任务及时间安排
  - 报告要求
- 原理回顾
  - 计算机工作原理
  - 简单微处理器的基本构成
- Vivado软件使用
  - 模块设计
  - 功能仿真

原理回顾

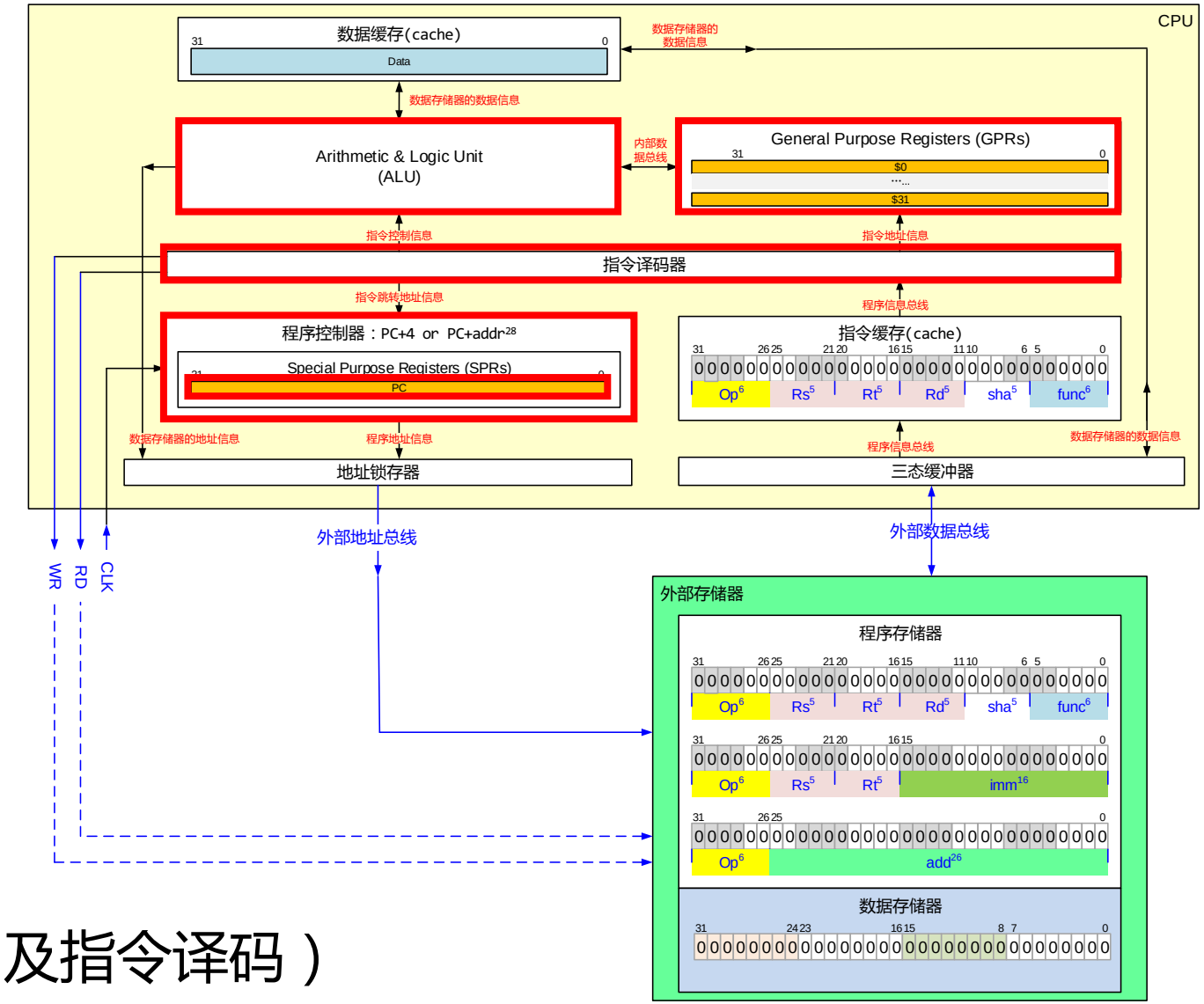
▶ 计算机工作原理

- 存储程序和程序控制”



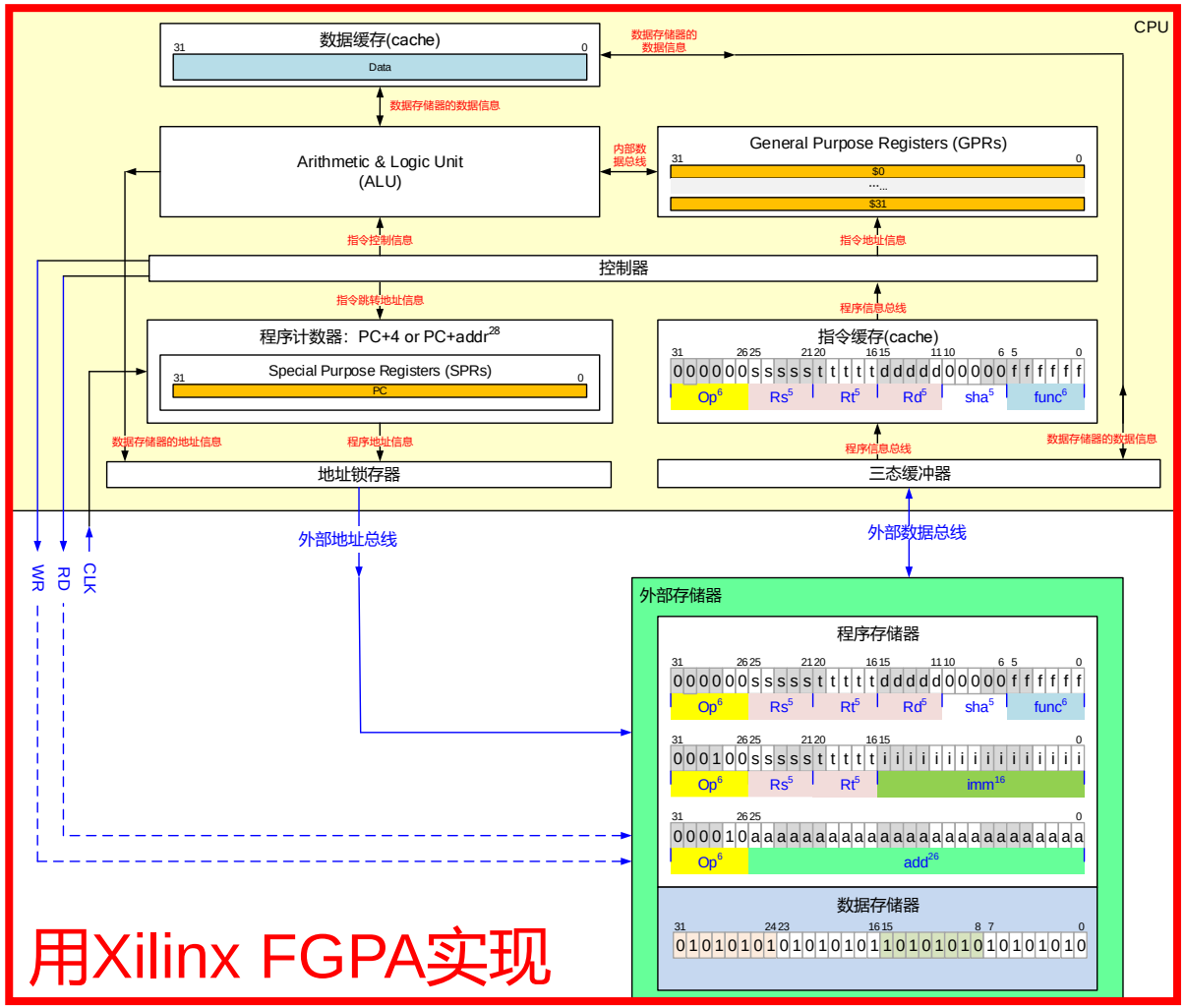
▶ 简单微处理器的基本构成

- 运算器、Regs、控制器（程序控制及指令译码）

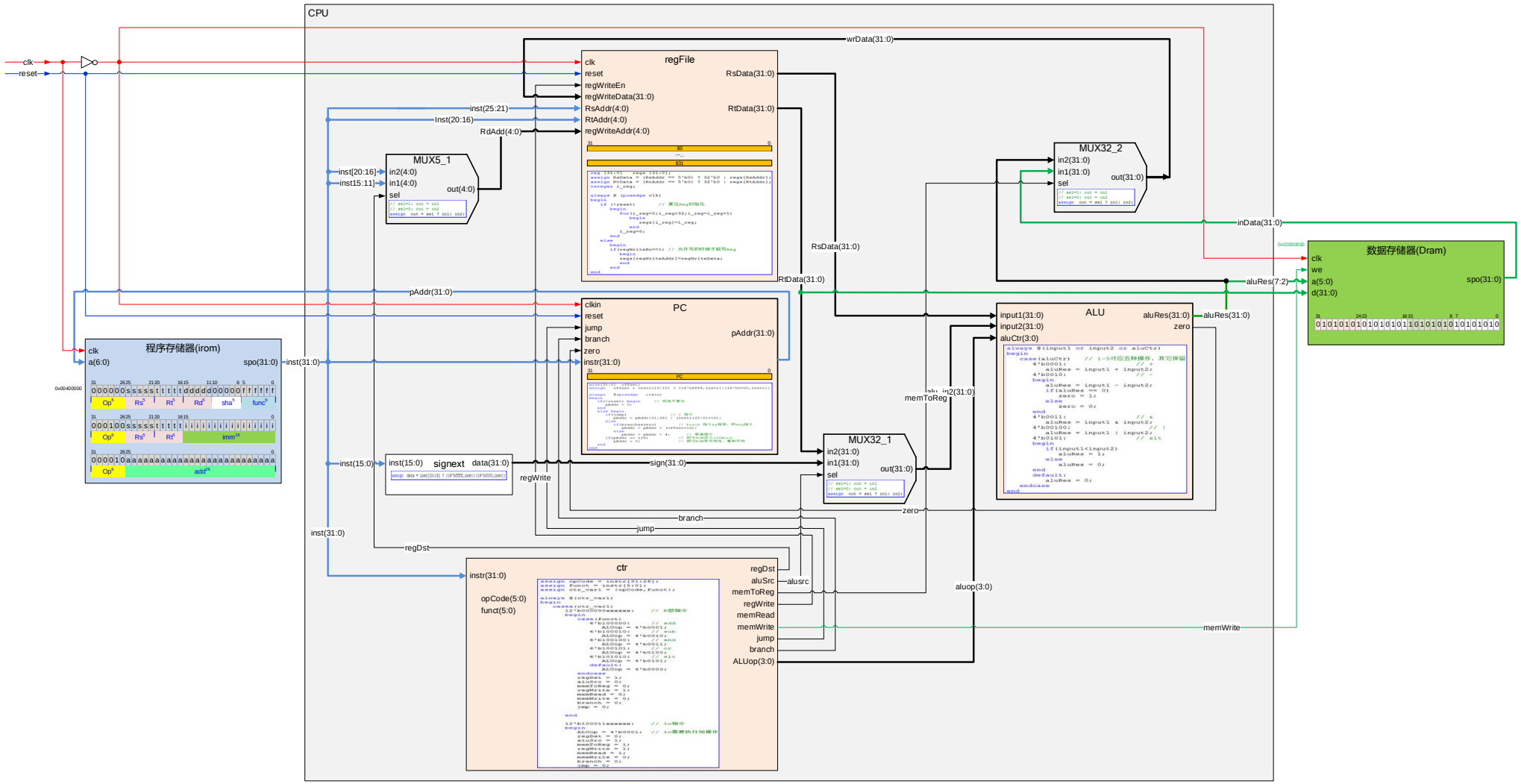


简单微处理器的基本构成

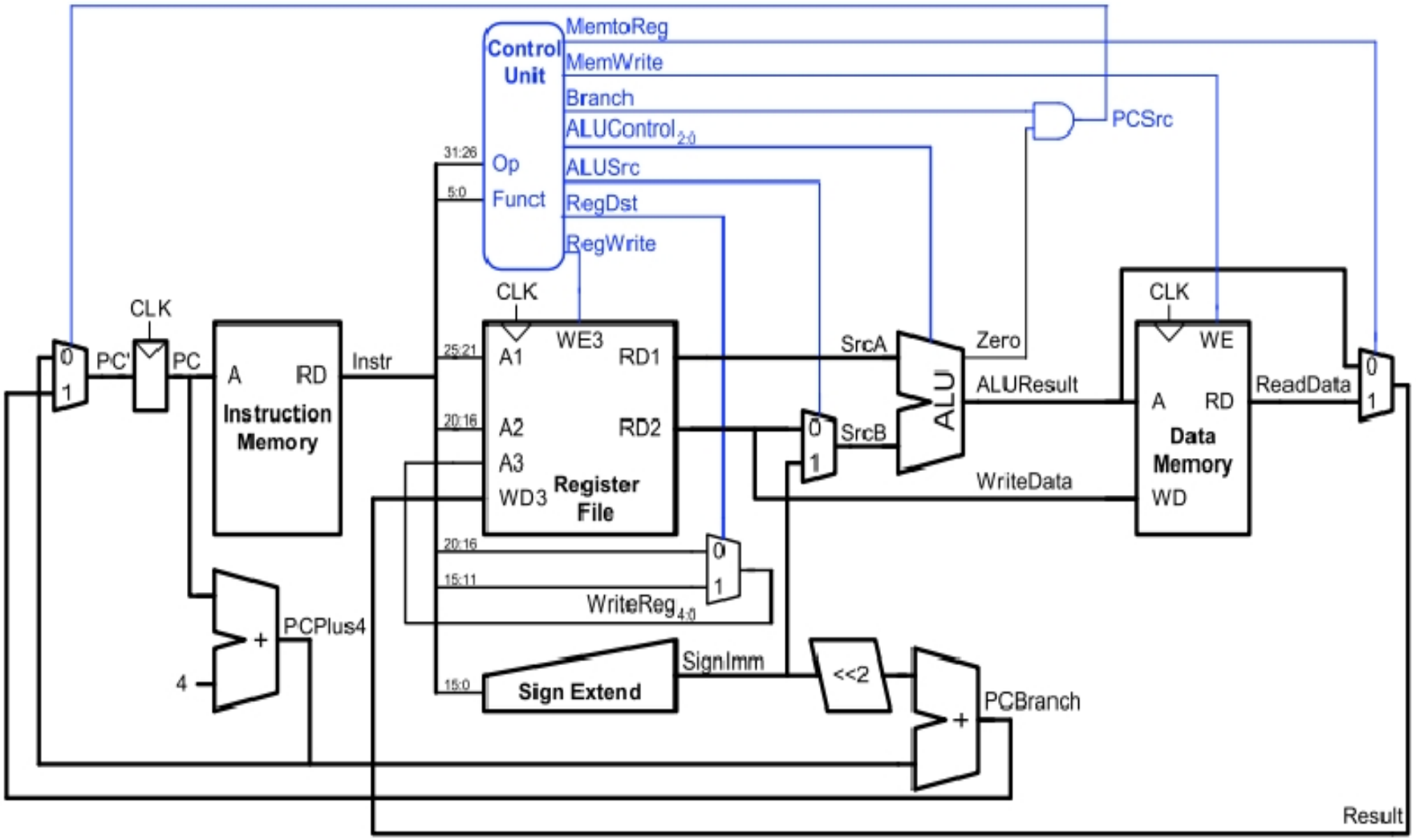
- 假设MIPS微处理器支持以下指令
  - 算术逻辑运算如add，sub，and，or，slt指令
  - 存储器读写：lw，sw指令
  - 程序控制如beq，j指令



▶ 简单微处理器的基本构成（哈佛结构）—— 框图1



▶ 简单微处理器的基本构成（哈佛结构）—— 框图2



▶ 指令类型及机器码

- 支持add，sub，and，or，slt，andi，lw，sw，beq，j 等指令
  - 加、减、与、或、slt的op-code均为000000，通过function code来区分（2级译码）：
    - Func = 100000：add
    - Func = 100010：sub
    - Func = 100100：and
    - Func = 100101：or
    - Func = 101010：slt

R 型指令

Add:
add Rd, Rs, Rt # RF[Rd] = RF[Rs] + RF[Rt]

Op-Code	Rs	Rt	Rd	Function Code
000000	sssss	ttttt	ddddd	00000100000

Add contents of Reg.File[Rs] to Reg.File[Rt] and store result in Reg.File[Rd]. If overflow occurs in the two's complement number system, an exception is generated.

Subtract:
sub Rd, Rs, Rt # RF[Rd] = RF[Rs] - RF[Rt]

Op-Code	Rs	Rt	Rd	Function Code
000000	sssss	ttttt	ddddd	00000100010

Subtract contents of Reg.File[Rt] from Reg.File[Rs] and store result in Reg.File[Rd]. If overflow occurs in the two's complement number system, an exception is generated.

And:
and Rd, Rs, Rt # RF[Rd] = RF[Rs] AND RF[Rt]

Op-Code	Rs	Rt	Rd	Function Code
000000	sssss	ttttt	ddddd	00000100100

Bitwise logically AND contents of Register File[Rs] with Reg.File[Rt] and store result in Reg.File[Rd].

OR:
or Rd, Rs, Rt # RF[Rd] = RF[Rs] OR RF[Rt]

Op-Code	Rs	Rt	Rd	Function Code
000000	sssss	ttttt	ddddd	00000100101

Bit wise logically OR contents of Register File[Rs] with Reg.File[Rt] and store result in Reg.File[Rd].

Set on Less Than: (Used in branch macro instructions)
slt Rd, Rs, Rt # if (RF[Rs] < RF[Rt]) then RF[Rd] =1 else RF[Rd] = 0

Op-Code	Rs	Rt	Rd	Function Code
000000	sssss	ttttt	ddddd	00000101010

If the contents of Reg.File[Rs] are less than the contents of Reg.File[Rt], then Reg.File[Rd] is set to one, otherwise Reg.File[Rd] is set to zero; assuming the two's complement number system representation.

I 型指令

Add Immediate:
addi Rt, Rs, Imm # RF[Rt] = RF[Rs] + Imm

Op-Code	Rs	Rt	Imm
001000	sssss	ttttt	iiiiiiiiiiiiiiii

Add contents of Reg.File[Rs] to sign extended Imm value, store result in Reg.File [Rt]. If overflow occurs in the two's complement number system, an exception is generated.

Load Word:
lw Rt, offset(Rs) # RF[Rt] = Mem[RF[Rs] + Offset]

Op-Code	Rs	Rt	Offset
100011	sssss	ttttt	iiiiiiiiiiiiiiii

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. A 32-bit word is read from memory at the effective address and loaded into Reg.File[Rt]. If the least two significant bits of the effective address are not zero, an address error exception occurs. There are four bytes in a word, so word addresses must be binary numbers that are a multiple of four, otherwise an address error exception occurs.

Store Word:
sw Rt, offset(Rs) # Mem[RF[Rs] + Offset] = RF[Rt]

Op-Code	Rs	Rt	Offset
101011	sssss	ttttt	iiiiiiiiiiiiiiii

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. The contents of Reg.File[Rt] are stored in memory at the effective address. If the least two significant bits of the effective address are not zero, an address error exception occurs. There are four bytes in a word, so word addresses must be binary numbers that are a multiple of four, otherwise an address error exception occurs.

Branch if Equal:
Beq Rs, Rt, Label # If (RF[Rs] == RF[Rt])then PC = PC + Imm<< 2

Op-Code	Rs	Rt	Imm
000100	sssss	ttttt	iiiiiiiiiiiiiiii

If Reg.File[Rs] is equal to Reg.File[Rt] then branch to label.

J 型指令

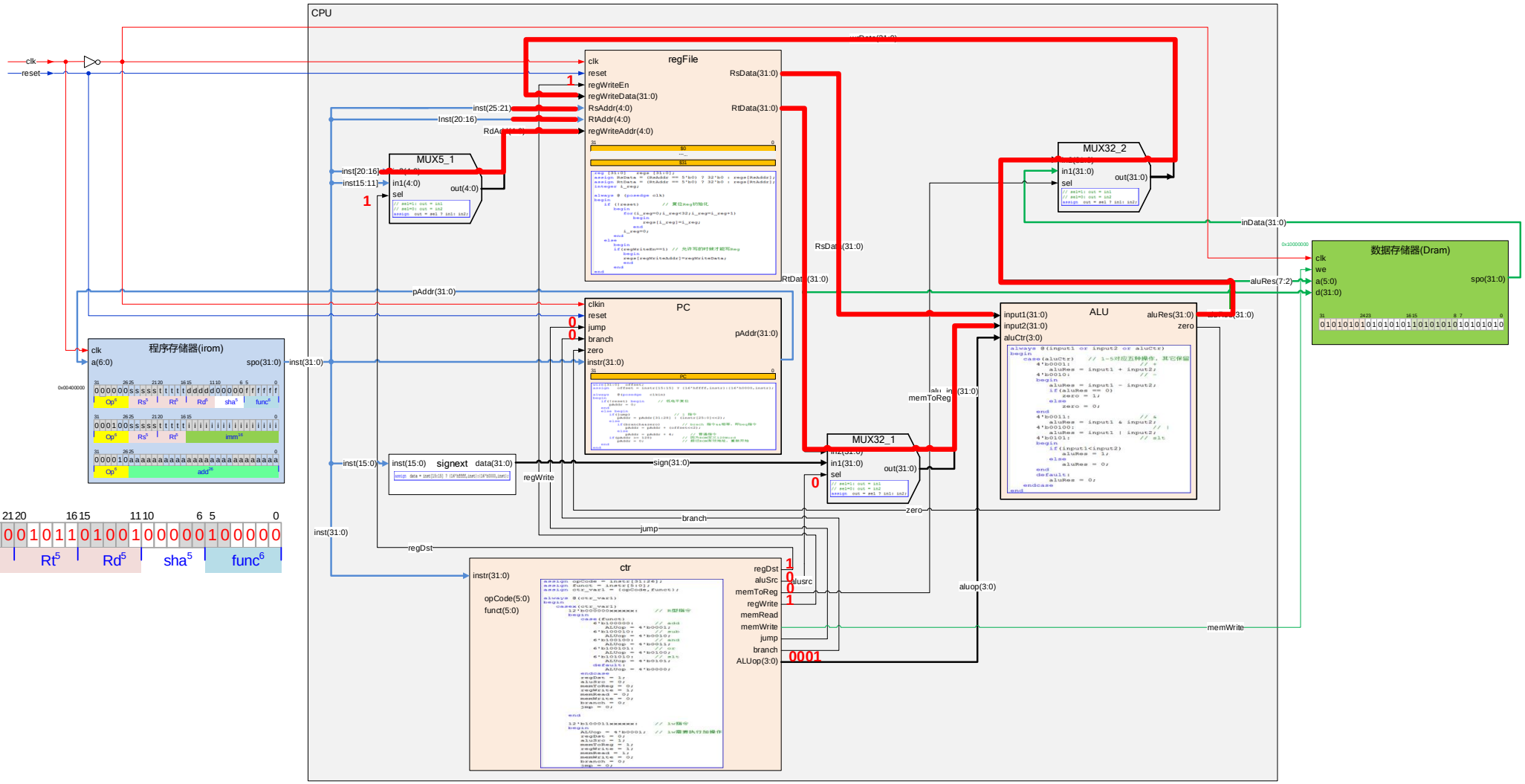
Jump:
j Label # PC = PC(31:28) | Imm<< 2

Op-Code	Imm
000010	iiiiiiiiiiiiiiiiiiiiiiiiii

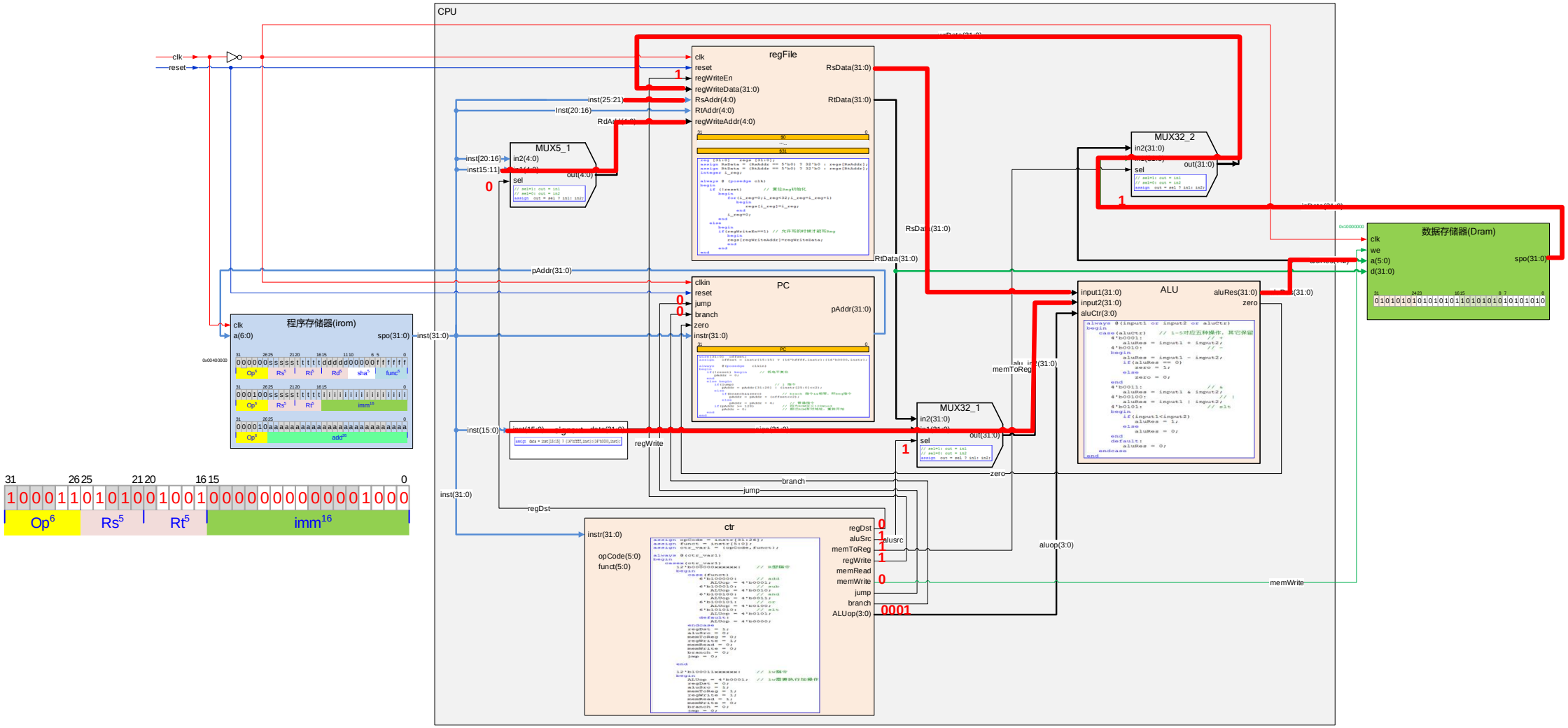
Load the PC with an address formed by concatenating the first 4-bits of the current PC with the value in the 26-bit immediate field shifted left 2-bits.

原理回顾

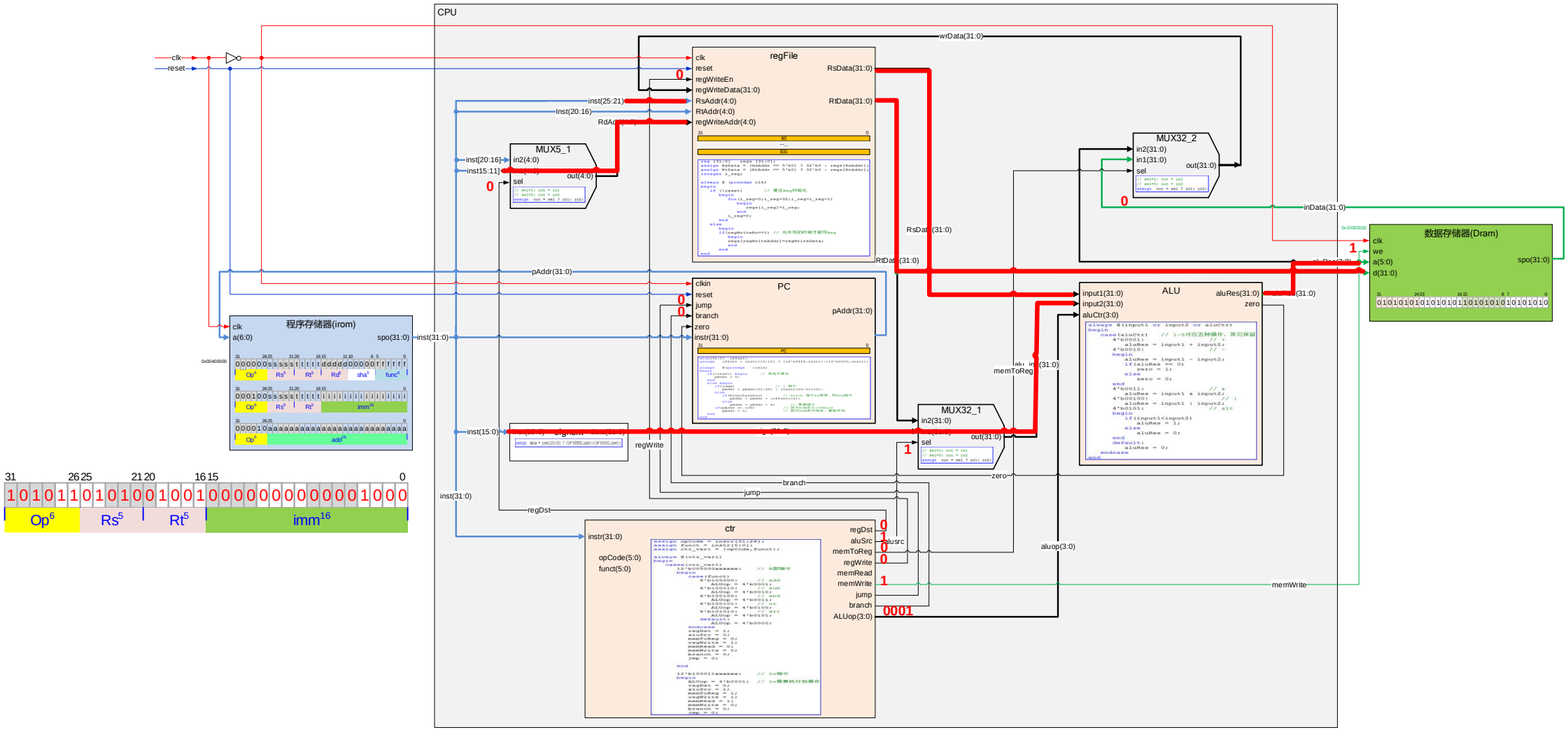
- 简单微处理器的基本构成（数据流：add $t1, $t2, $t3 指令）



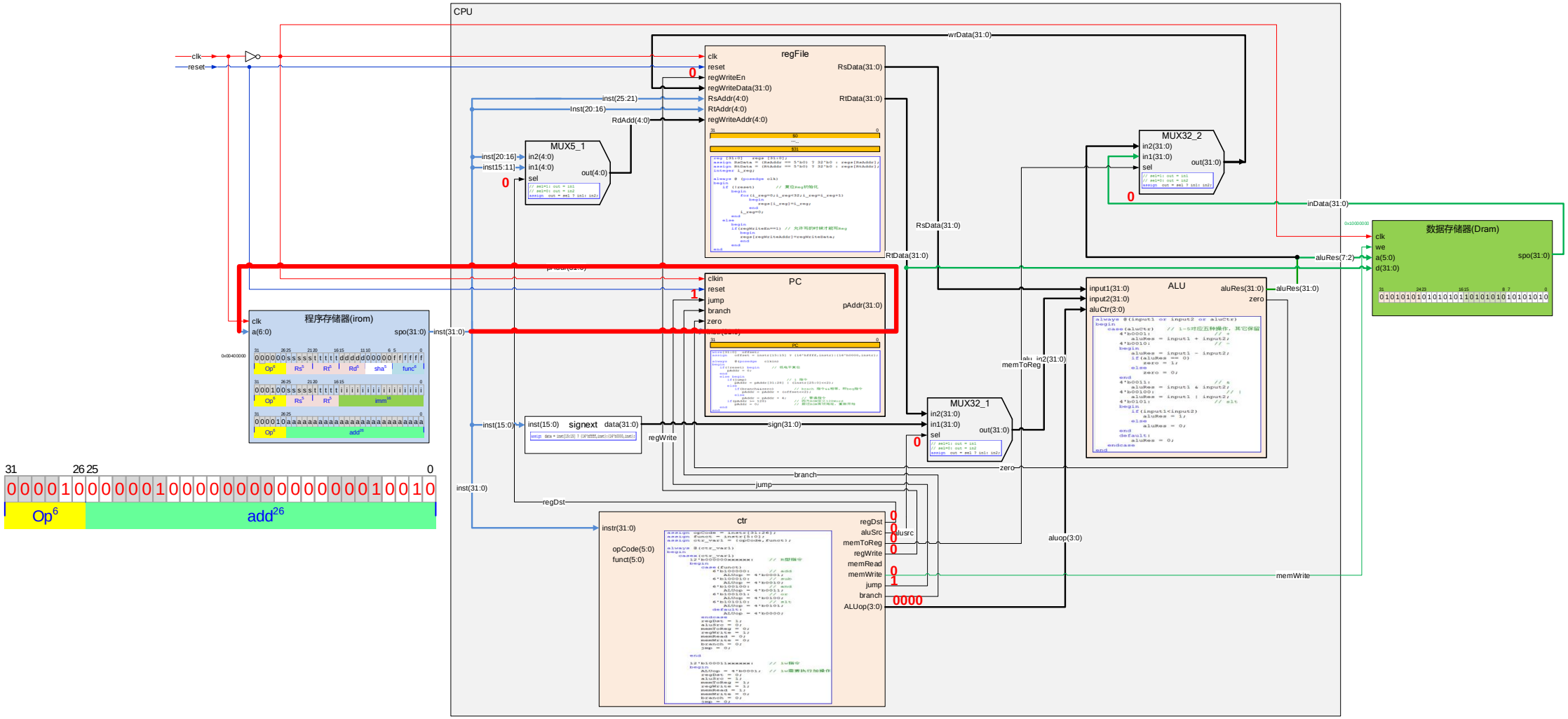
▶ 简单微处理器的基本构成（数据流：lw $t1,8($t2)指令）



▶ 简单微处理器的基本构成（数据流：sw $t1,8($t2)指令）



▶ 简单微处理器的基本构成（数据流：j xxxx指令）

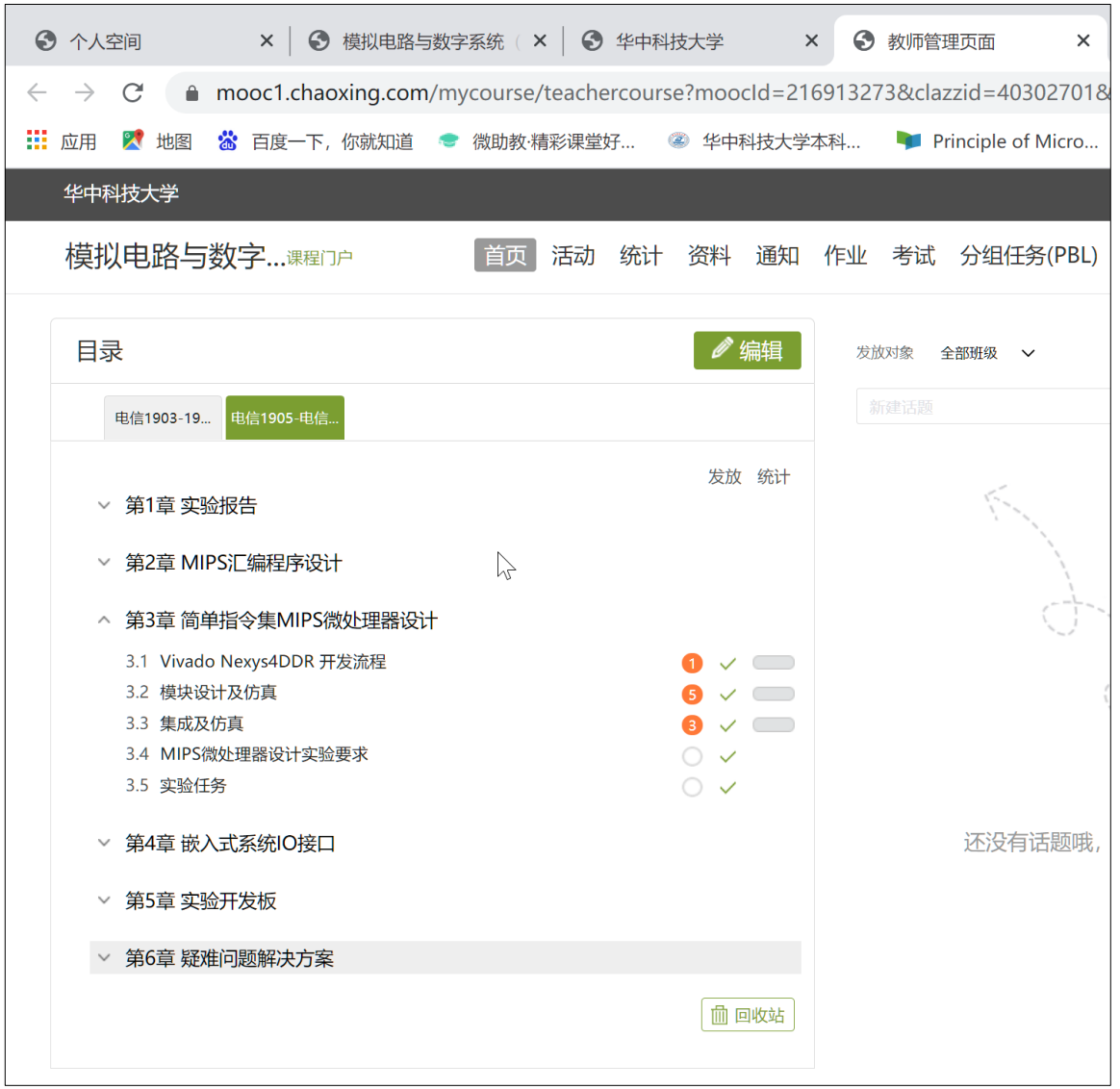


Agenda

- 实验内容
  - 目的
  - 任务及时间安排
  - 报告要求
- 原理回顾
  - 计算机工作原理
  - 简单微处理器的基本构成
- Vivado软件使用
  - 模块设计
  - 功能仿真

Vivado软件使用

- 详见超星资源视频(9个)
  - Vivado开发流程 ：1
  - 模块设计及仿真 ：5
    - 寄存器模块设计及仿真
    - ALU模块设计及仿真
    - 指令、数据存储器
    - Mars导出汇编指令机器码
    - 主控制器实现
  - 集成及仿真 ：3
    - 顶层模块实现
    - 处理器仿真一
    - 处理器仿真二

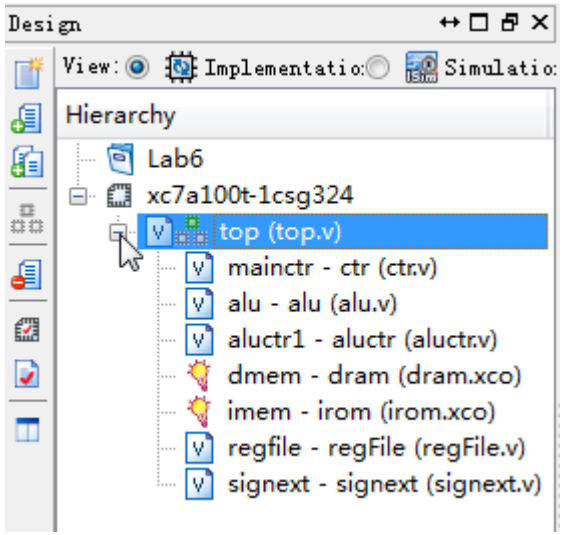


- 顶层模块
  - 采用Verilog实现：top.v
    - 定义信号线
    - 模块实例化，连接模块
    - 时钟信号线

```
21  module top(
22      input clkin,
23      input reset
24      );
25
26  //wire for ROM
27  wire[31:0]  inst;
28
29  //wire for controller
30  wire        regdst,jump,branch,memread,memwrite,memtoreg;
31  wire[3:0]   aluop;
32  wire        alusrc,regwrite;
33
34  //wire for aluunit
35  wire        zero;
36  wire[31:0]  aluRes;
37  wire[31:0]  alu_in2;        // ALU数据输入之前的多路器输出
38
39  //wire for memory
40  wire[31:0]  inData;
41
42  //wire for register
43  wire[31:0]  RsData,RtData;
44  wire[4:0]   RdAddr;
45  wire[31:0]  wrData;
46
47  //wire for PC
48  wire[31:0] pAddr;
49
50  //wire for ext
51  wire[31:0]  sign;
52
```

Vivado软件使用

- 顶层模块
  - 采用Verilog实现：top.v
    - 定义信号线
    - 模块实例化，连接模块

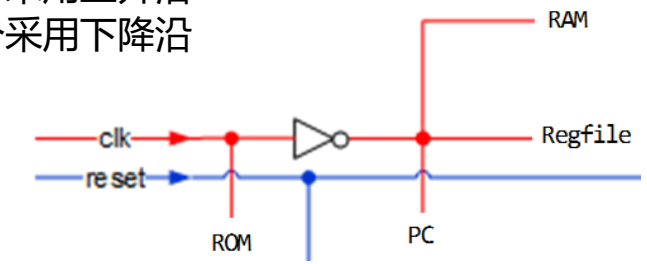


    - 时钟信号线

```
52
53  irom imem(
54      .a(pAddr[8:2]),
55      .clk(clkin),          // 上升沿读指令
56      .spo(inst));
57
58  PC PCnt(
59      .clkin(!clkin),       // 下降沿改变PC
60      .reset(reset),
61      .instr(inst),
62      .jump(jump),
63      .branch(branch),
64      .zero(zero),
65      .pAddr(pAddr));
66
67  ctr mainctr(
68      .instr(inst),
69      .ALUop(aluop),
70      .regDst(regdst),
71      .aluSrc(alusrc),
72      .memToReg(memtoreg),
73      .regWrite(regwrite),
74      .memRead(memread),
75      .memWrite(memwrite),
76      .branch(branch),
77      .jmp(jump));
78
79  alu alu(
80      .input1(RsData),
81      .input2(alu_in2),
82      .aluCtr(aluop),
83      .aluRes(aluRes),
84      .zero(zero));
85
86  // ALU input2 数据多路器
87  mux32 max32_1(
88      .in1(sign),
89      .in2(RtData),
90      .out(alu_in2),
91      .sel(alusrc));
```

```
92
93  dram dmem(
94      .a(aluRes[7:2]),
95      .d(RtData),
96      .clk(!clkin),     // 下降沿写数据
97      .we(memwrite),
98      .spo(inData));
99
100 // Reg 写数据多路器
101 mux32 max32_2(
102     .in1(inData),
103     .in2(aluRes),
104     .out(wrData),
105     .sel(memtoreg));
106
107 regFile regfile(
108     .RsAddr(inst[25:21]),
109     .RtAddr(inst[20:16]),
110     .clk(!clkin),         // 下降沿写Reg
111     .reset(reset),
112     .regWriteAddr(RdAddr),
113     .regWriteData(wrData),
114     .regWriteEn(regwrite),
115     .RsData(RsData),
116     .RtData(RtData));
117
118 // Reg 地址选择多路器
119 mux5 max5_1(
120     .in1(inst[15:11]),
121     .in2(inst[20:16]),
122     .out(RdAddr),
123     .sel(regdst));
124
125 // 符号数扩展
126 signext signext(
127     .inst(inst[15:0]),
128     .data(sign));
129
130 endmodule
```

- 顶层模块
  - 采用Verilog实现：top.v
    - 定义信号线
    - 模块实例化，连接模块
    - 时钟信号线
      - 单周期
        - 执行指令时不能取指令，取指令由ROM输出决定，执行指令包括改变PC的值、写Regs、写RAM等
        - 因此改变PC的值、 写Regs、写RAM等与从ROM读取指令不能同时进行，这里采用时钟的两个不同边沿实现
          - 一个采用上升沿
          - 一个采用下降沿




```
52
53  irom imem(
54      .a(pAddr[8:2]),
55      .clk(clkin),          // 上升沿读指令
56      .spo(inst));
57
58  PC PCnt(
59      .clkin(!clkin),       // 下降沿改变PC
60      .reset(reset),
61      .instr(inst),
62      .jump(jump),
63      .branch(branch),
64      .zero(zero),
65      .pAddr(pAddr));
66
67  ctr mainctr(
68      .instr(inst),
69      .ALUop(aluop),
70      .regDst(regdst),
71      .aluSrc(alusrc),
72      .memToReg(memtoreg),
73      .regWrite(regwrite),
74      .memRead(memread),
75      .memWrite(memwrite),
76      .branch(branch),
77      .jmp(jump));
78
79  alu alu(
80      .input1(RsData),
81      .input2(alu_in2),
82      .aluCtr(aluop),
83      .aluRes(aluRes),
84      .zero(zero));
85
86  // ALU input2 数据多路器
87  mux32 max32_1(
88      .in1(sign),
89      .in2(RtData),
90      .out(alu_in2),
91      .sel(alusrc));
```

```
92
93  dram dmem(
94      .a(aluRes[7:2]),
95      .d(RtData),
96      .clk(!clkin),     // 下降沿写数据
97      .we(memwrite),
98      .spo(inData));
99
100 // Reg 写数据多路器
101 mux32 max32_2(
102     .in1(inData),
103     .in2(aluRes),
104     .out(wrData),
105     .sel(memtoreg));
106
107 regFile regfile(
108     .RsAddr(inst[25:21]),
109     .RtAddr(inst[20:16]),
110     .clk(!clkin),          // 下降沿写Reg
111     .reset(reset),
112     .regWriteAddr(RdAddr),
113     .regWriteData(wrData),
114     .regWriteEn(regwrite),
115     .RsData(RsData),
116     .RtData(RtData));
117
118 // Reg 地址选择多路器
119 mux5 max5_1(
120     .in1(inst[15:11]),
121     .in2(inst[20:16]),
122     .out(RdAddr),
123     .sel(regdst));
124
125 // 符号数扩展
126 signext signext(
127     .inst(inst[15:0]),
128     .data(sign));
129
130 endmodule
```

Agenda

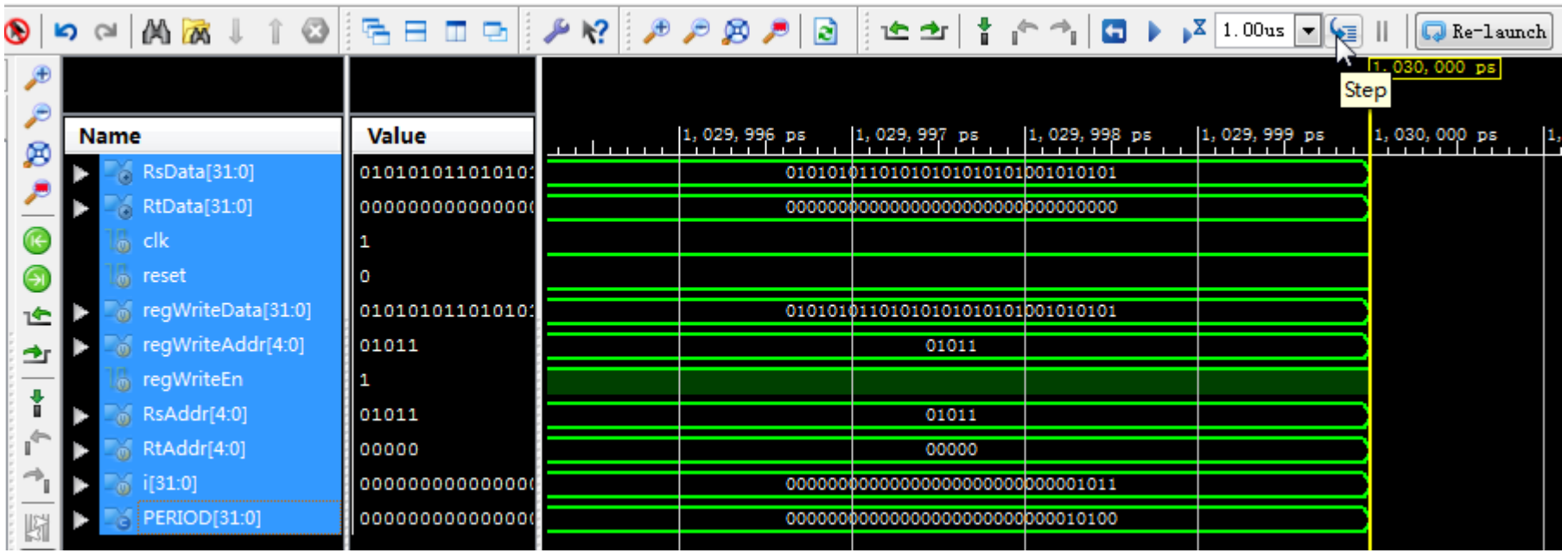
- 实验内容
  - 目的
  - 任务及时间安排
  - 报告要求
- 原理回顾
  - 计算机工作原理
  - 简单微处理器的基本构成
- Vivado软件使用
  - 模块设计
  - 功能仿真

Vivado软件使用

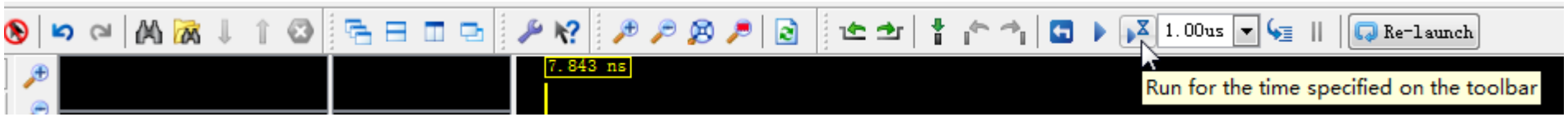
- 仿真
  - 子模块仿真
    - 寄存器组仿真（学会仿真步骤）
    - …
  - 顶层仿真（掌握仿真结果的查看）

寄存器组仿真

- 观察结果
  - Step

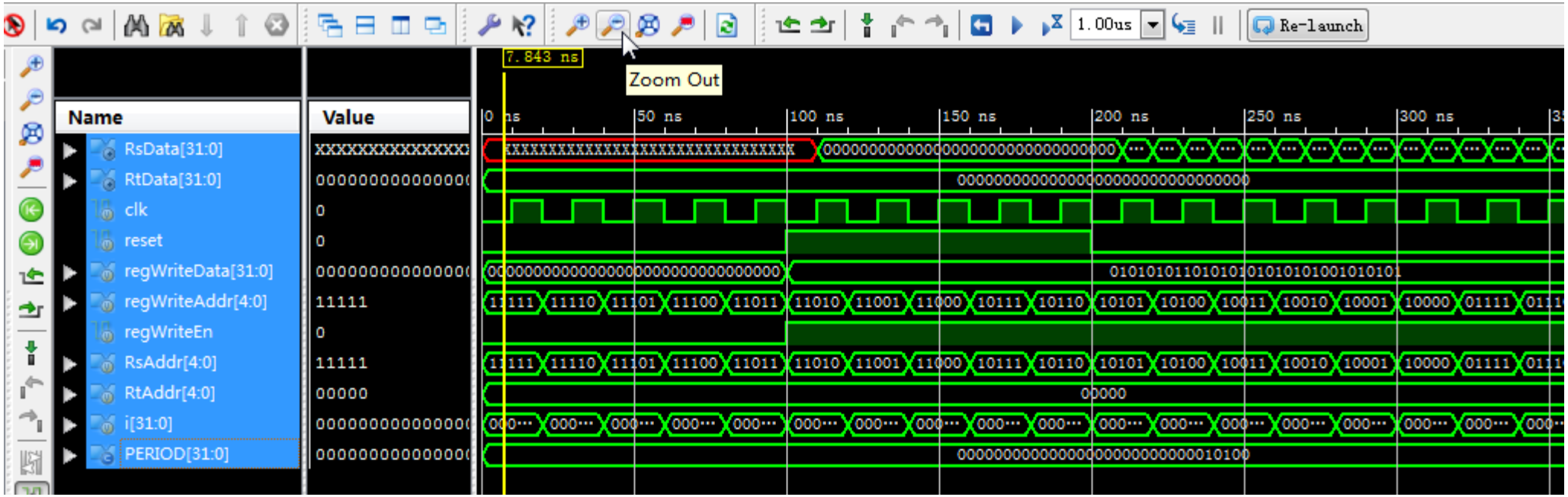


  - Run for the time specified on the toolbar



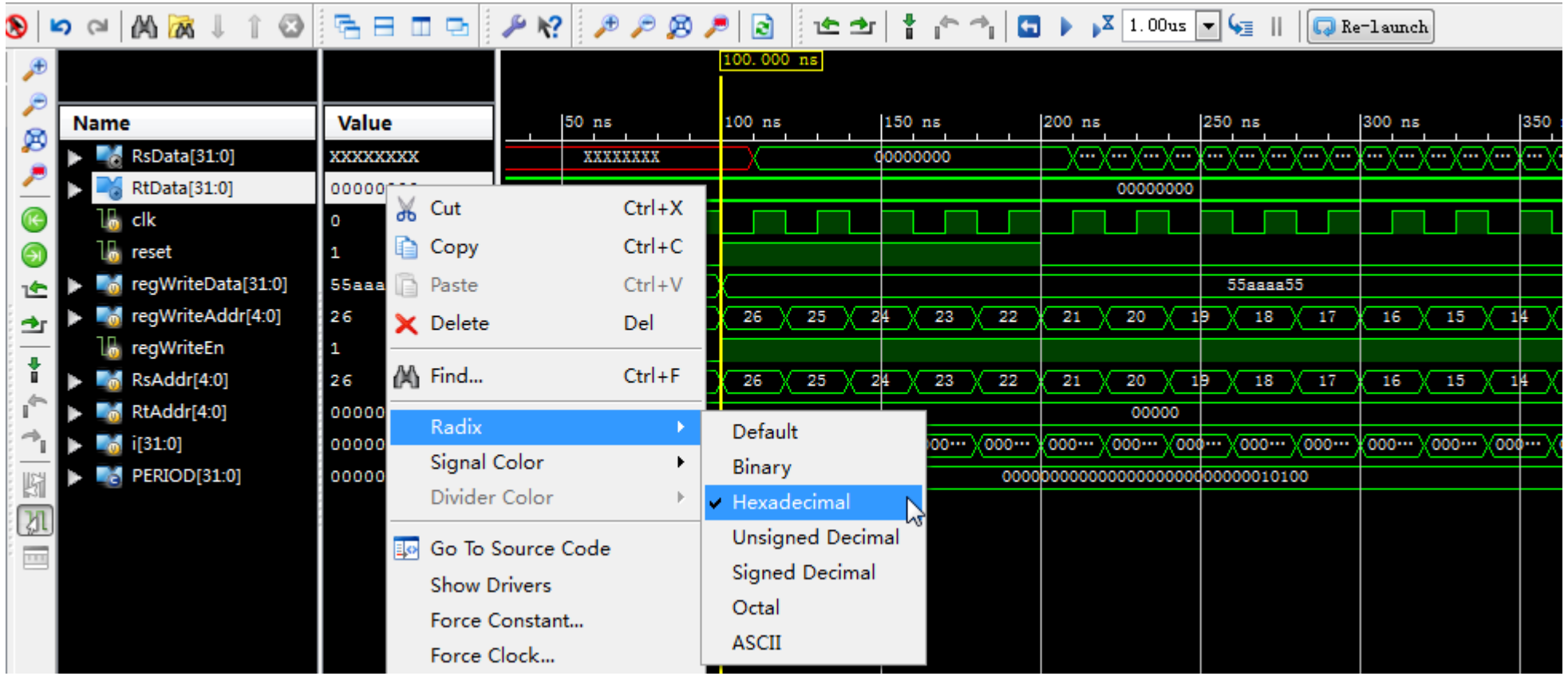
▶ 寄存器组仿真

- 观察结果
  - Run for the time specified on the toolbar
  - Zoom Out / Zoom In

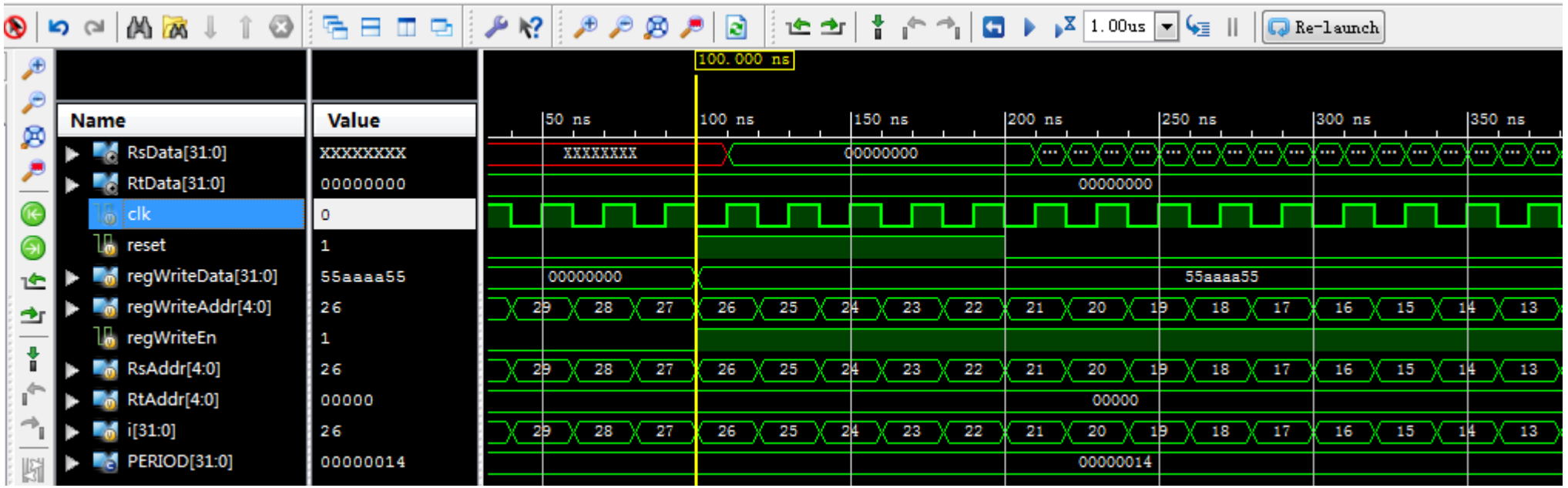


▶ 寄存器组仿真

- 观察结果
  - 显示格式调整



- 寄存器组仿真
  - 观察结果
    - 结果分析

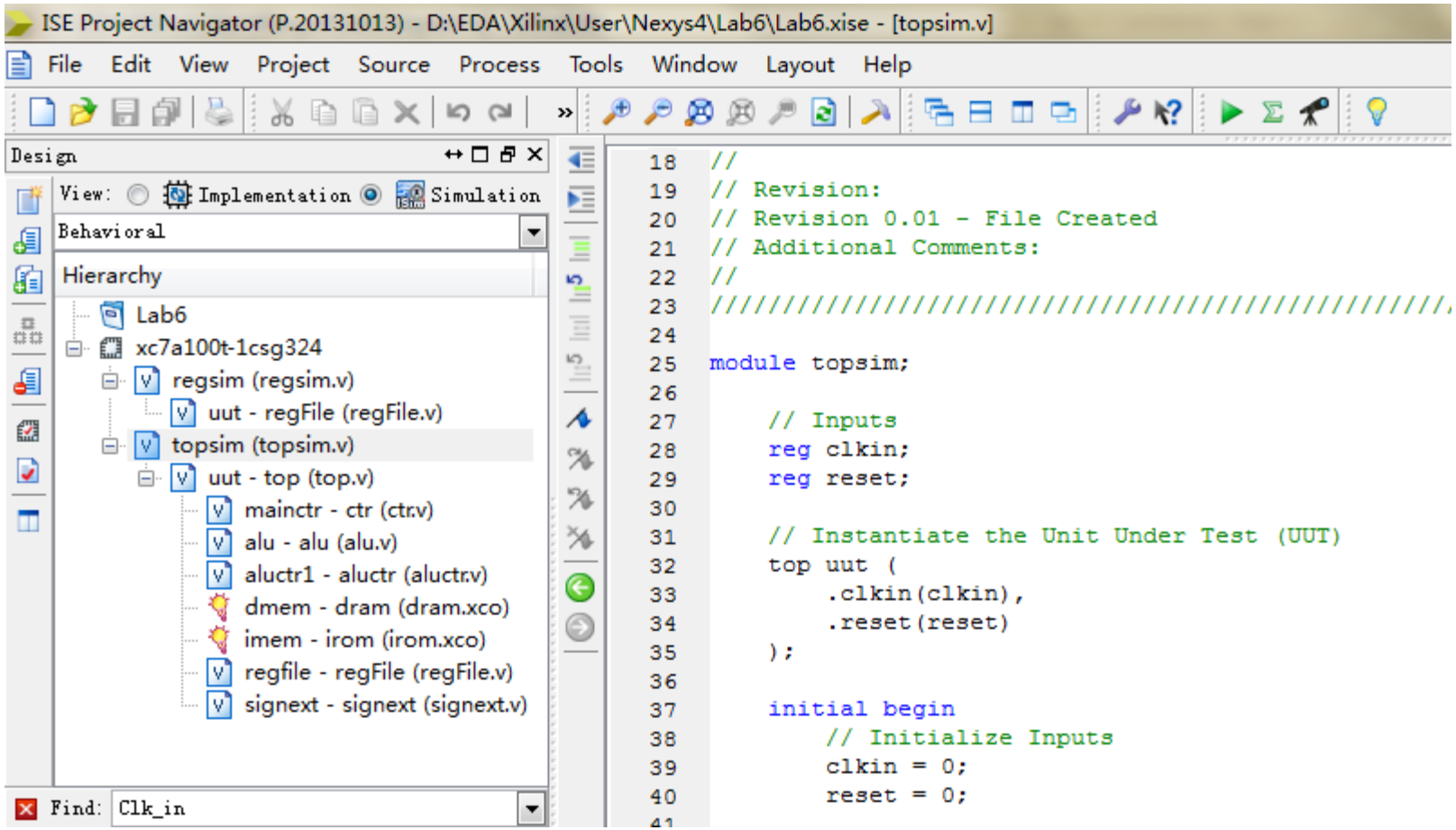


- 其它模块功能仿真
  - 控制器模块
  - ALU模块
  - ROM模块
  - RAM模块
  - …

自行完成这些模块的功能仿真，保证模块功能是正确的；之后可以进行Top的功能仿真；或者如发现Top功能仿真不对，可以返回来仿真各个模块功能，找出错误所在。

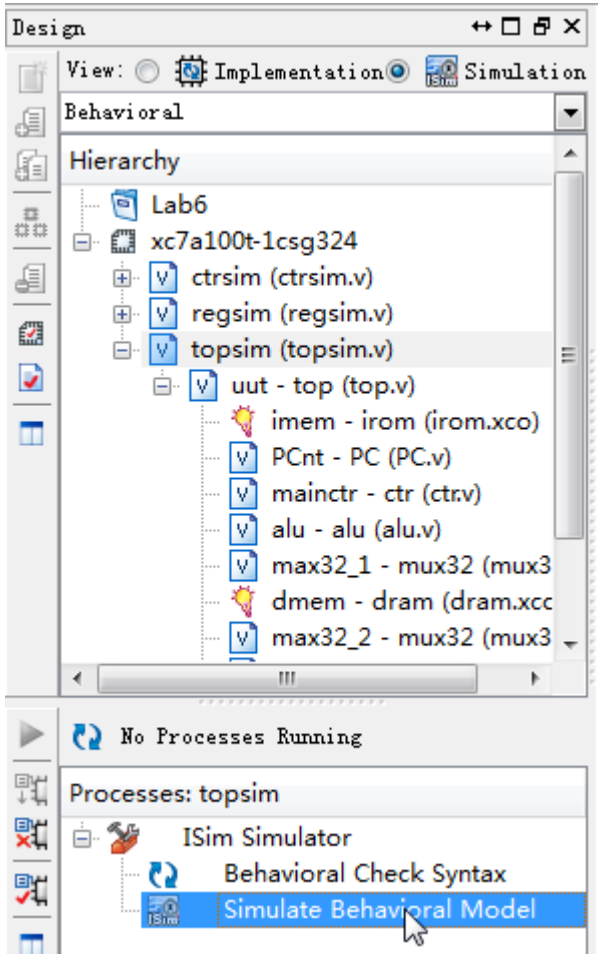
▶ 顶层仿真

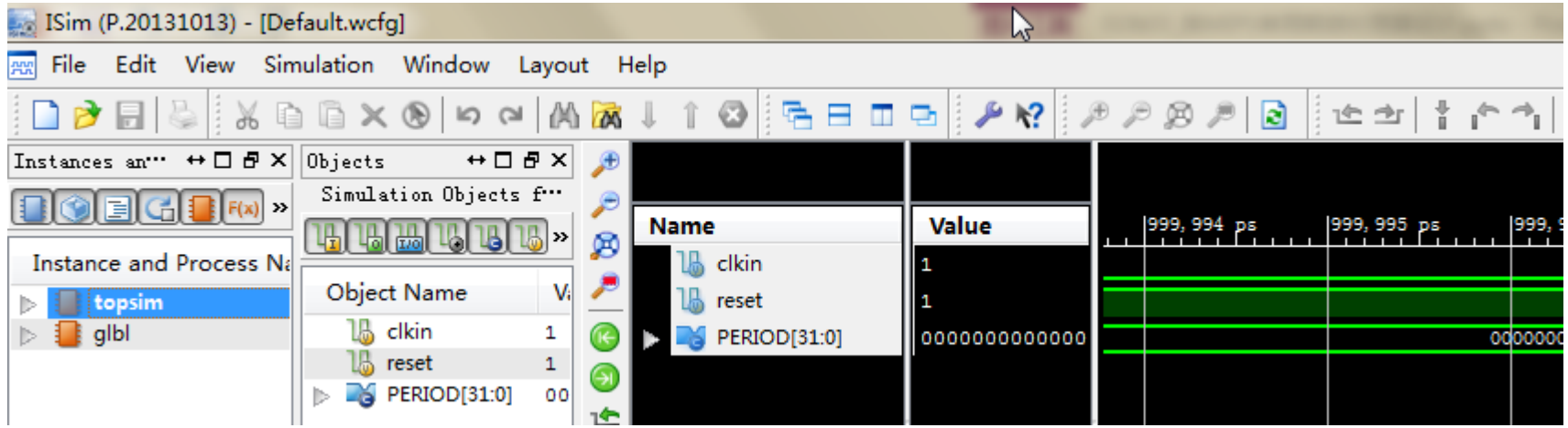
- 自动生成激励代码topsim.v



▶ 顶层仿真

- 运行仿真&查看结果

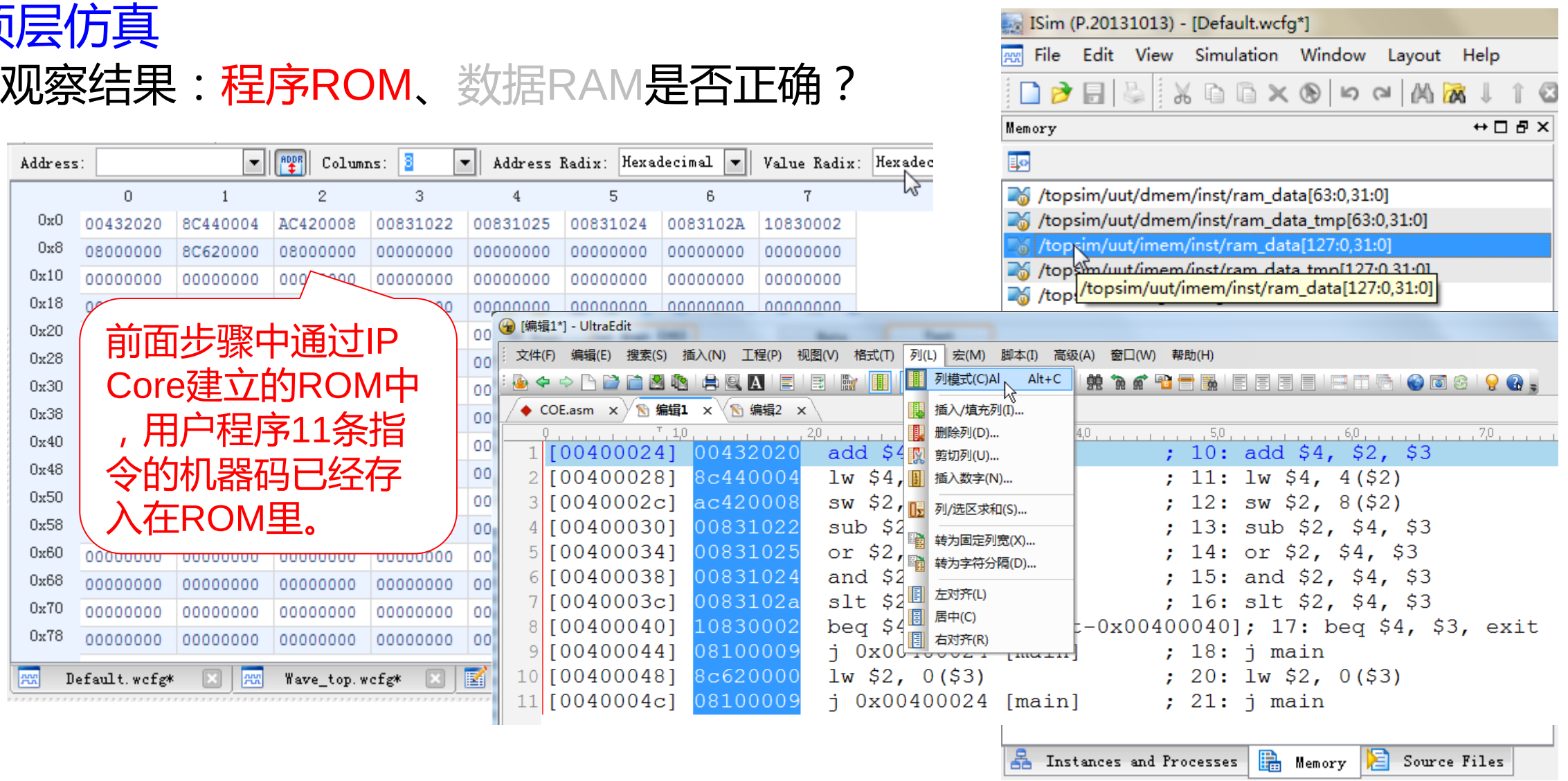




Vivado软件使用

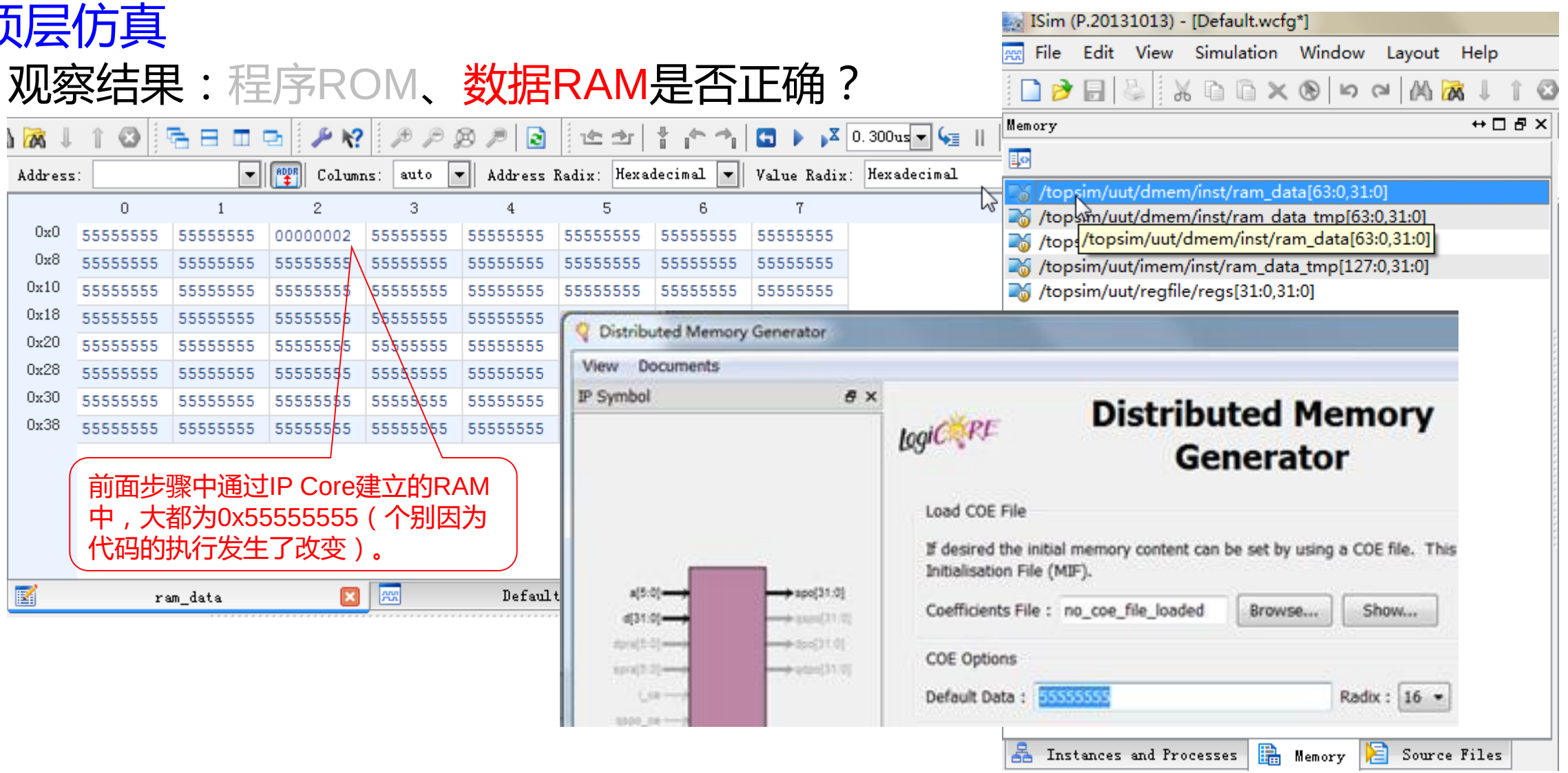
▶ 顶层仿真

- 观察结果：程序ROM、数据RAM是否正确？



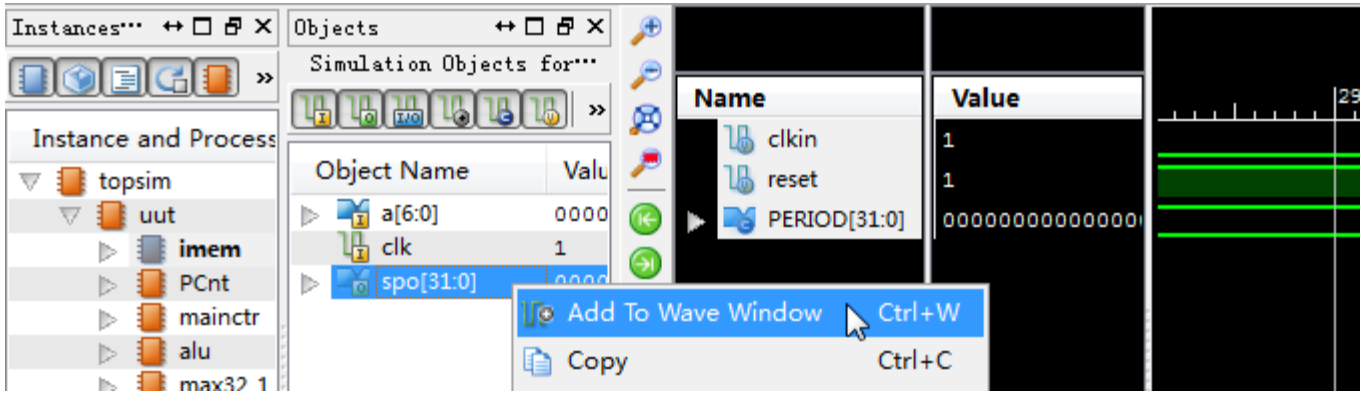
▶ 顶层仿真

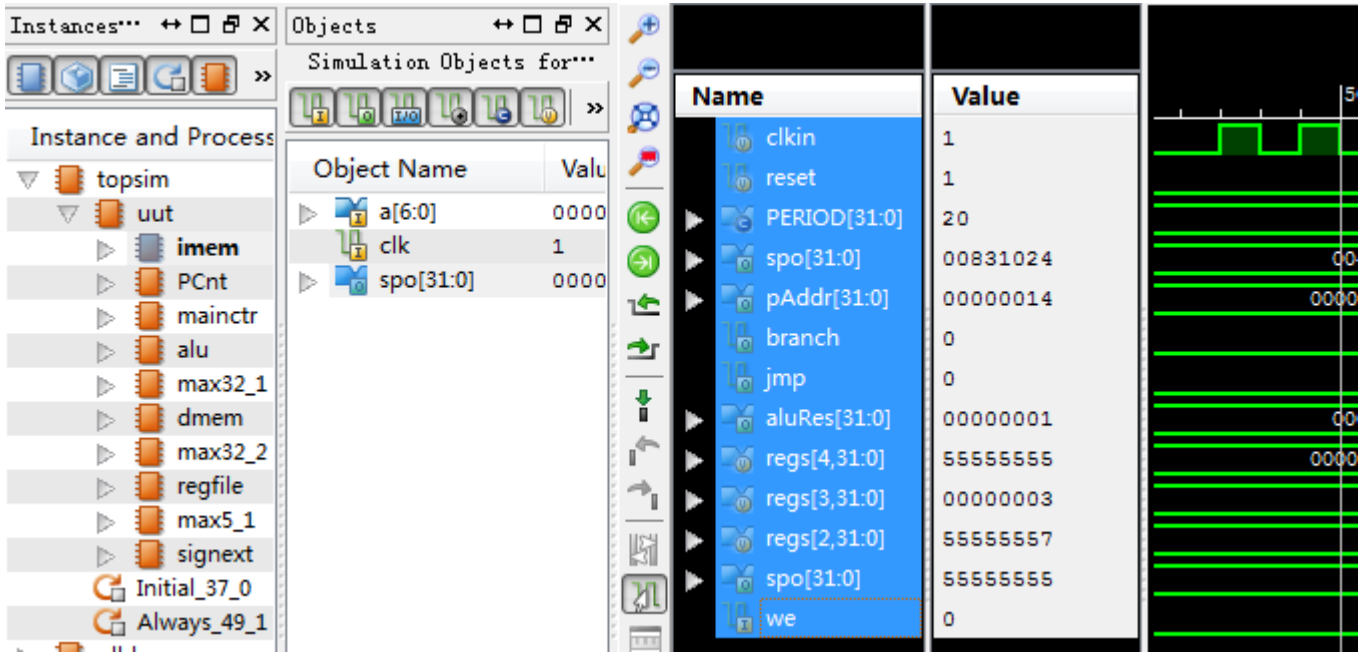
- 观察结果：程序ROM、数据RAM是否正确？



Vivado软件使用

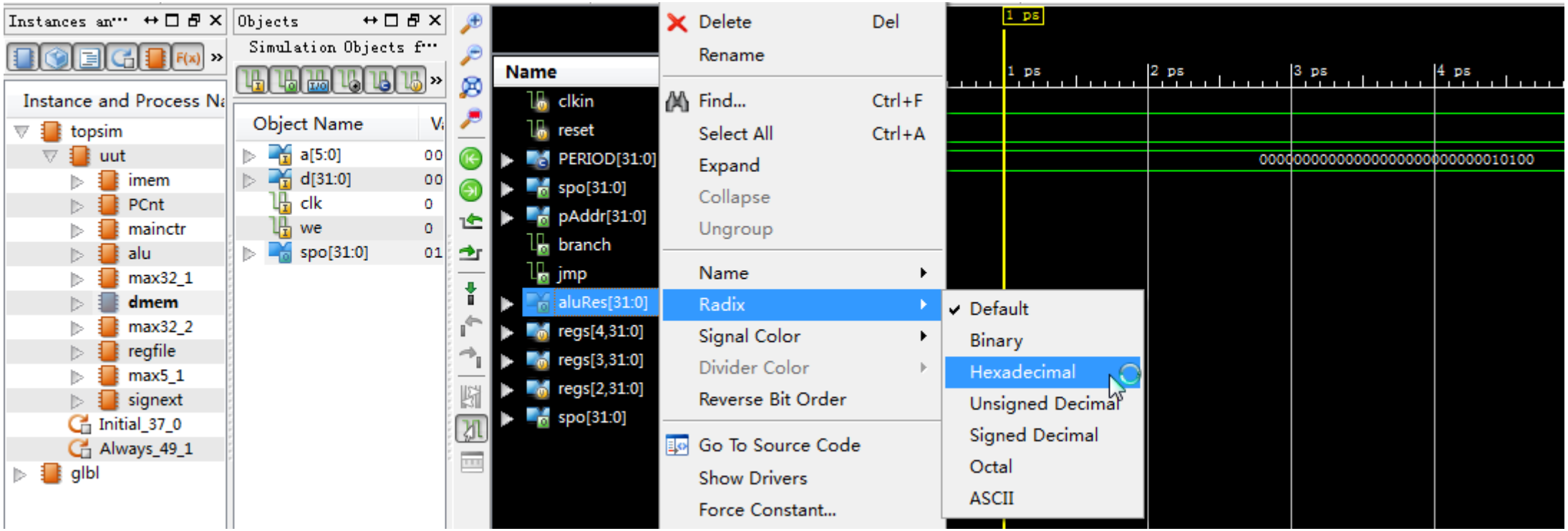
- 顶层仿真
  - 观察结果：加入需要观测的信号
    - 程序信息总线：imem/spo[31:0]；
    - 程序地址总线：PCnt/pAddr[31:0]；
    - 控制信息：mainctr/branch、jmp；
    - Alu结果：alu/aluRes[31:0]；
    - $4/$3/$2：regfile/regs[4,31:0]、regfile/regs[3,31:0]、regfile/regs[2,31:0]；
    - RAM数据总线：dmem/spo[31:0]
    - RAM写控制线：dmem/we





▶ 顶层仿真

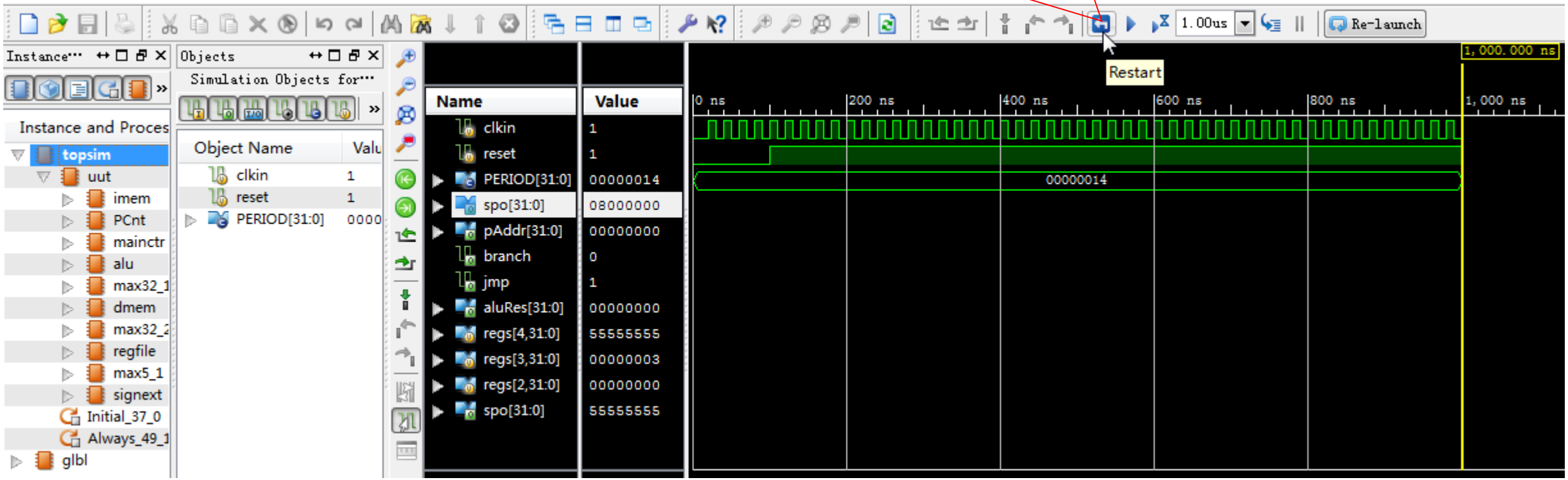
- 观察结果：
  - 调整观测信号的显示格式（总线均按照16进制显示）
  - Zoom in / Zoom out，使得整个窗口能够显示多个时钟周期信号



Vivado软件使用

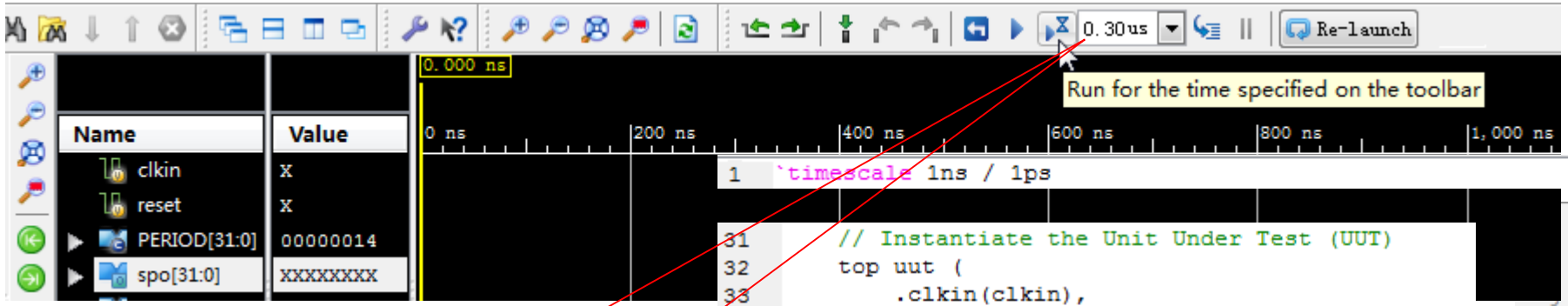
- 顶层仿真
  - 观察结果：Restart

1）一进入仿真界面，就自动执行了1.00us时间，仅仅显示了top模块的信号变化；后加入进来的信号变化过程没有显示。所以先Restart，重新开始仿真。



▶ 顶层仿真

- 观察结果：修改仿真时间，然后 Run for the time specified on the toolbar



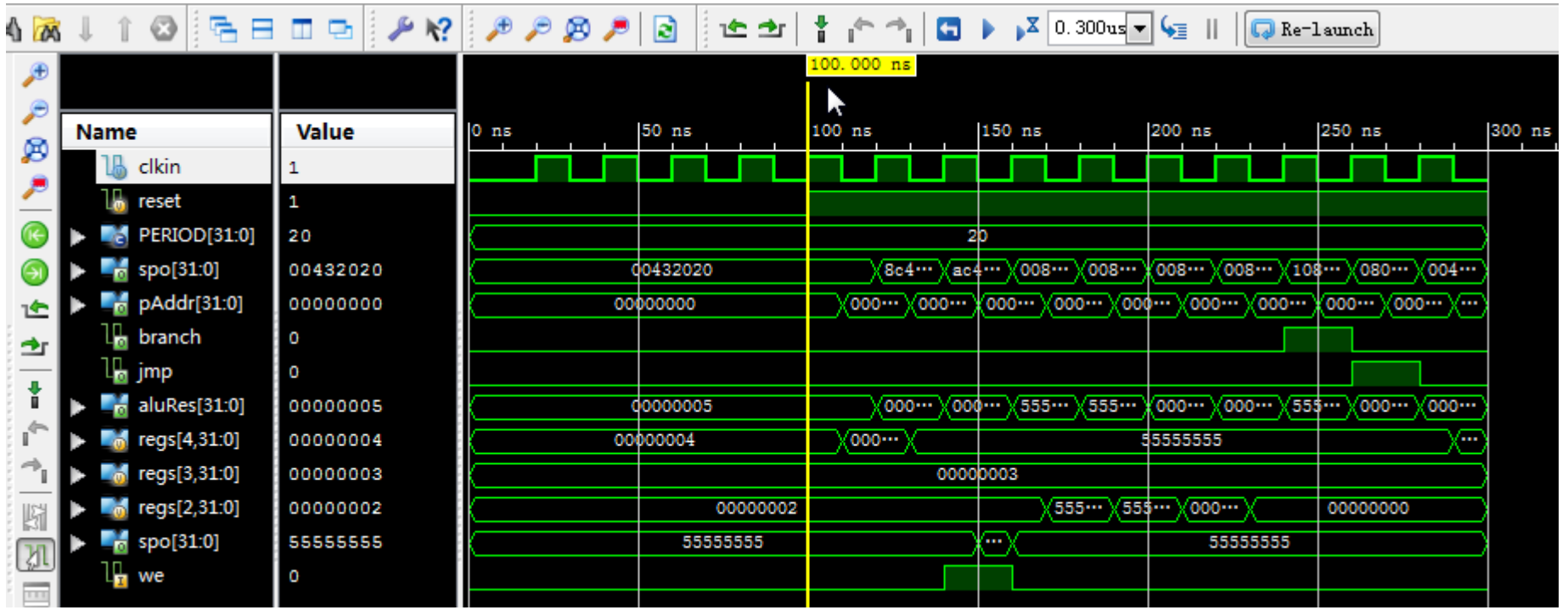
1）仿真信号clkin周期20ns；
2）9/10条指令执行完，最多需要200ns；
3）加上复位100ns；
4）仿真时间共计300ns；
5）0.3us仿真时间里可以跑完全部指令。

```
1  `timescale 1ns / 1ps
31     // Instantiate the Unit Under Test (UUT)
32     top uut (
33         .clkin(clkin),
34         .reset(reset)
35     );
36
37     initial begin
38         // Initialize Inputs
39         clkin = 0;
40         reset = 0;
41
42         // Wait 100 ns for global reset to finish
43         #100;
44         reset = 1;
45         // Add stimulus here
46     end
47
48     parameter PERIOD = 20;
49     always  begin
50         #(PERIOD/2) clkin = 1'b0;
51         #(PERIOD/2) clkin = 1'b1;
52     end
```

```
.globl main
.text
main:
  add $4, $2, $3
  lw  $4, 4($2)
  sw  $2, 8($2)
  sub $2, $4, $3
  or  $2, $4, $3
  and $2, $4, $3
  slt $2, $4, $3
  beq $4, $3, exit
  j   main
exit:
  lw  $2, 0($3)
  j main
```

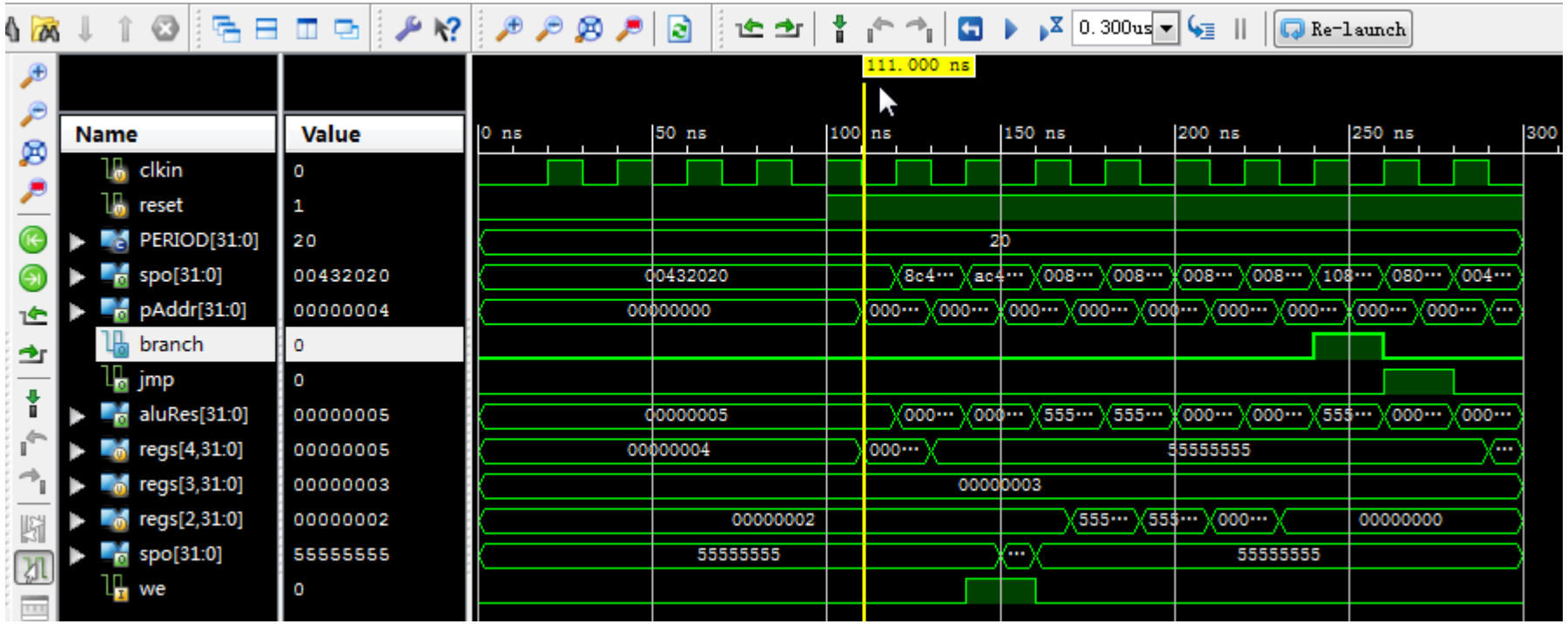
▶ 顶层仿真

- 观察结果：复位后第1个时钟周期，应该执行 add $4, $2, $3 （机器码:00432020）
  - 周期时钟上升沿（100ns处）从ROM读指令，PC指针pAddr = 0x0000 0000，控制器根据机器码产生对应的控制信号。
  - 此时$4, $3, $2寄存器的值为复位初始化值：4,3,2。



▶ 顶层仿真

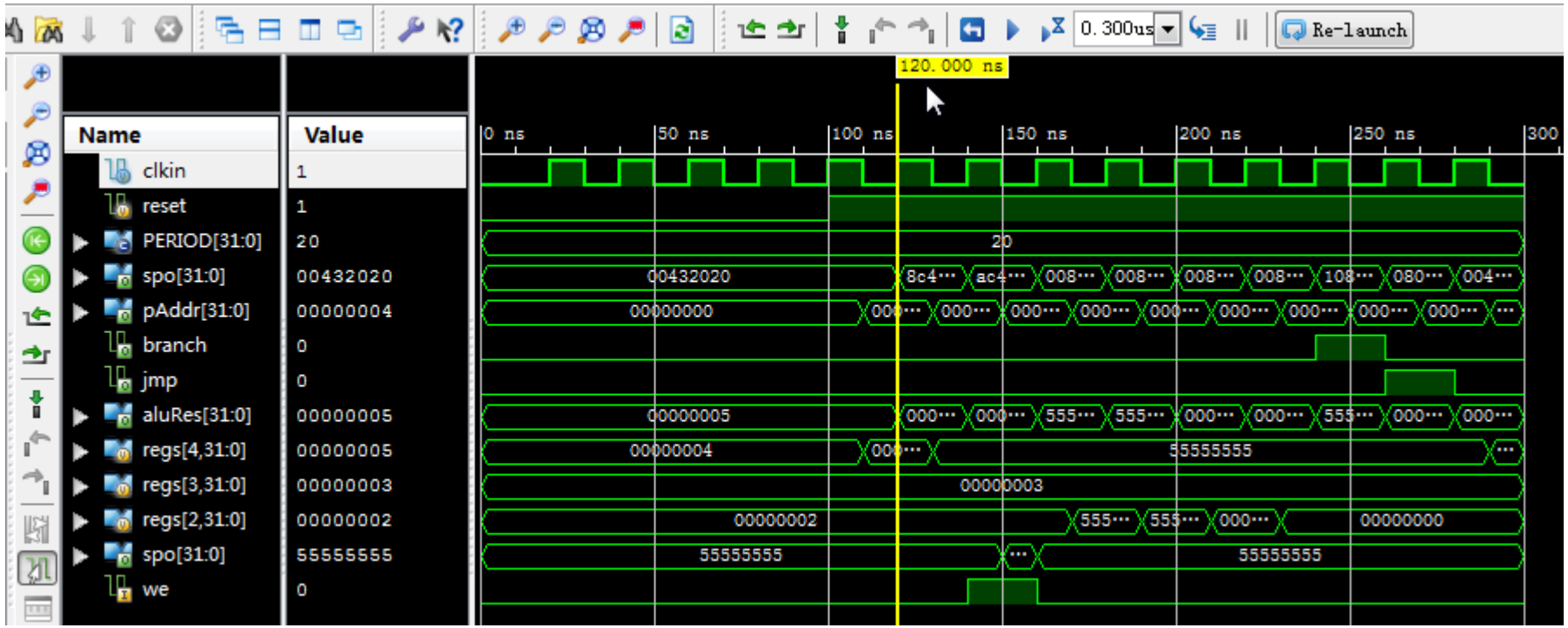
- 观察结果：复位后第1个时钟周期，应该执行 add $4, $2, $3 （机器码:00432020）
  - 周期时钟下降沿（110ns处）执行指令，写Reg、改变PC指针，执行后，如111ns处：
    - 机器码spo = 00432020
    - $2 + $3 → $4，2+3=5，$4 = 5（之前 $4, $3, $2的初始化值：4,3,2），从仿真可知：Alu的计算结果aluRes为5， $4的值也为5；PC指针pAddr = 0x0000 0004，指向了下一个地址。



由此可见，add 指令被正确执行。

▶ 顶层仿真

- 观察结果：复位后第2个时钟周期，应该执行 lw $4, 4($2) （机器码:8c440004）
  - 周期时钟上升沿（120ns处）从ROM读指令，PC指针pAddr = 0x0000 0004，控制器根据机器码产生对应的控制信号。
  - 此时$4, $2寄存器的值为：5, 2。

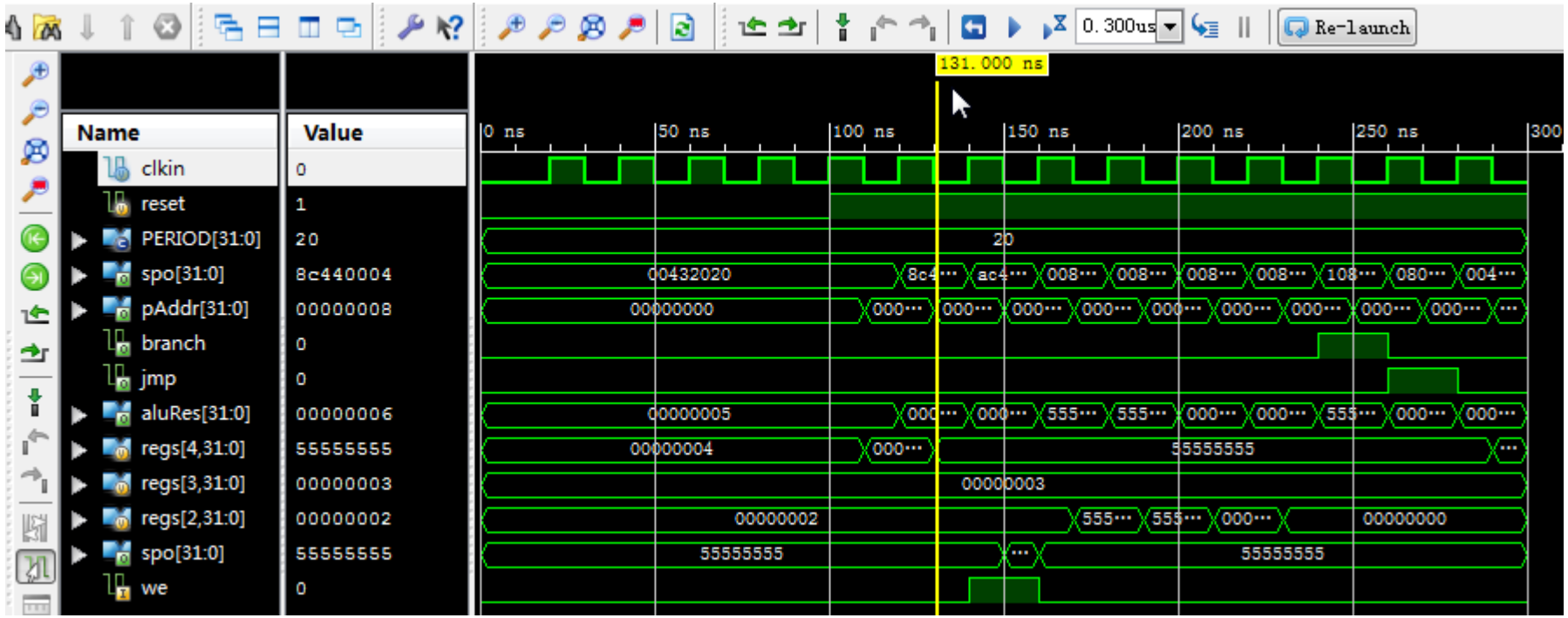


Vivado软件使用

▶ 顶层仿真

- 观察结果：复位后第2个时钟周期，应该执行 lw $4, 4($2) （机器码:8c440004）
  - 周期时钟下降沿（130ns处）执行指令，写Reg、改变PC指针，执行后，如131n处：
    - 机器码spo = 8c440004
    - ($2 + 4) → $4，（之前 $4, $2的值：5,2），从仿真可知：$4的值变为55555555（即地址为6的字单元中的值）；PC指针pAddr = 0x0000 0008，指向了下一个地址。

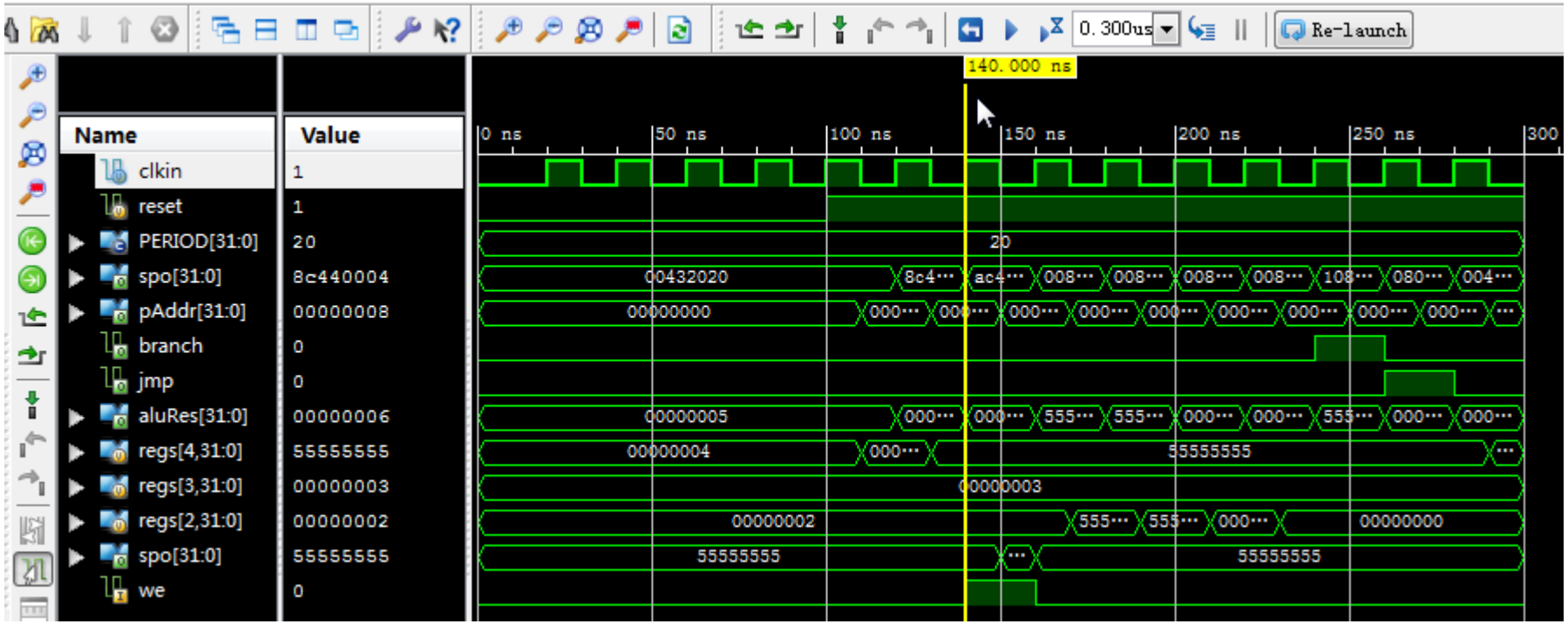
实际为地址4的字单元，因为ram硬件按照字读写：.a(aluRes[7:2]),



由此可见，lw 指令被正确执行。

▶ 顶层仿真

- 观察结果：复位后第3个时钟周期，应该执行 sw $2, 8($2) （机器码:ac420008）
  - 周期时钟上升沿（140ns处）从ROM读指令，PC指针pAddr = 0x0000 0008，控制器根据机器码产生对应的控制信号。
  - 此时$2寄存器的值为：2。

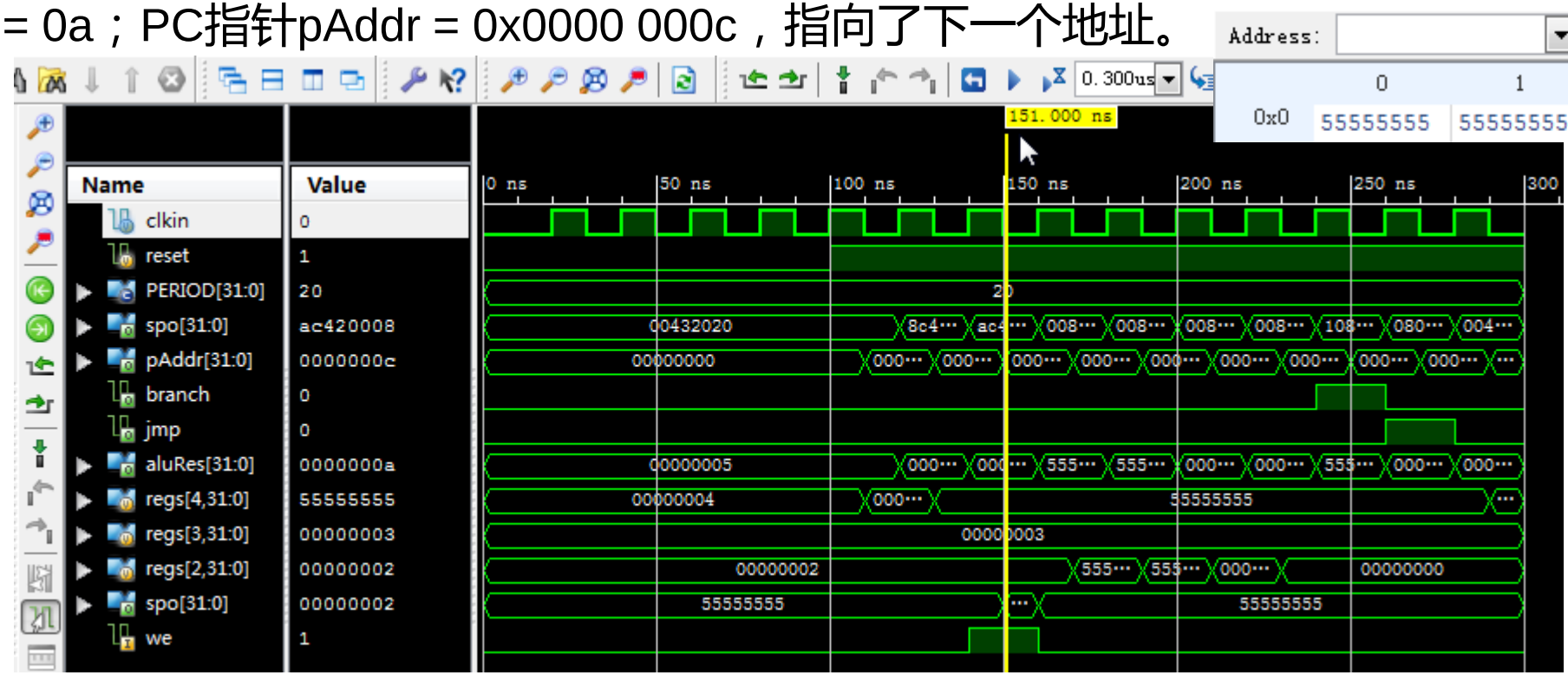


Vivado软件使用

▶ 顶层仿真

- 观察结果：复位后第3个时钟周期，应该执行 sw $2, 8($2) （机器码:ac420008）
  - 周期时钟下降沿（150ns处）执行指令，写RAM、改变PC指针，执行后，如151ns处：
    - 机器码spo = ac420008
    - $2 → ($2 + 8) ，（之前 $2的值：2），从仿真可知：Ram的数据线spo=2，地址线为aluRes = 0a；PC指针pAddr = 0x0000 000c，指向了下一个地址。

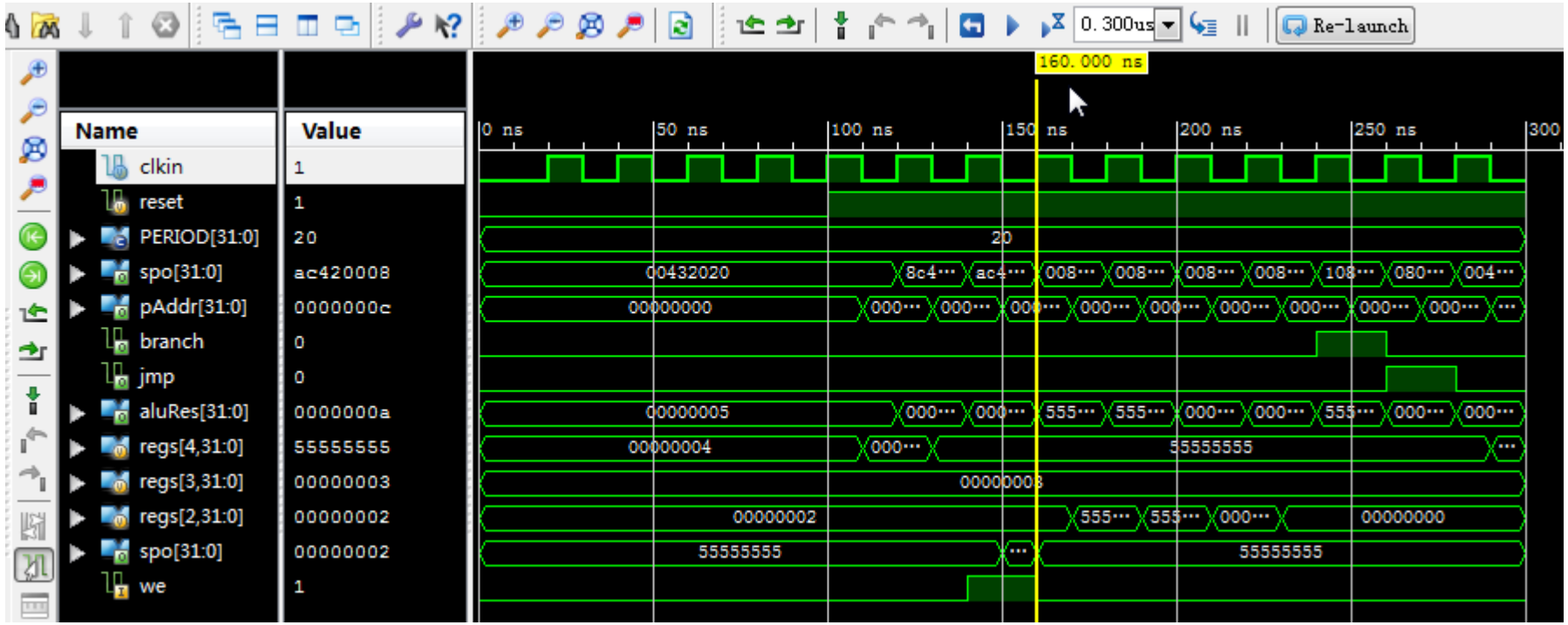
实际为地址8的字单元，因为ram硬件按照字读写：.a(aluRes[7:2]),



RAM区08地址存储单元已经变为$2中的内容2（需要保证仿真时没有后继指令修改过此存储单元），由此可见，lw 指令被正确执行。

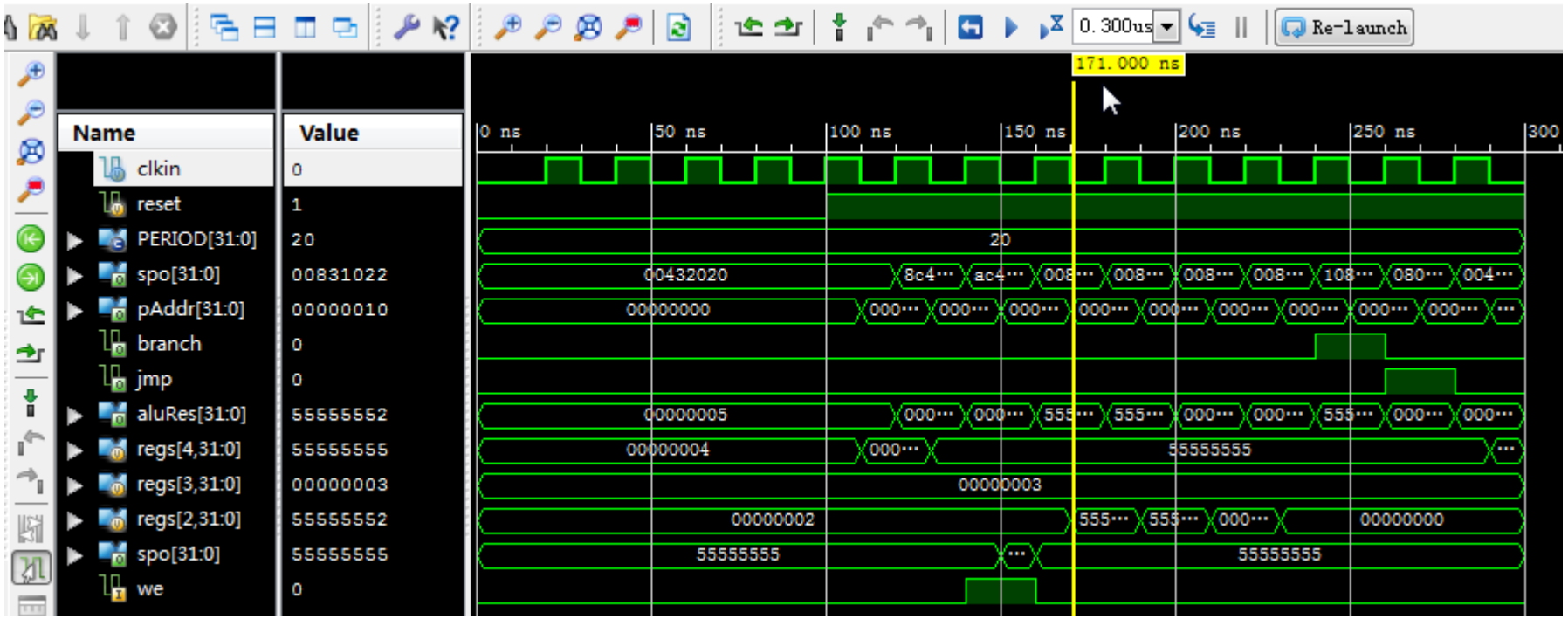
▶ 顶层仿真

- 观察结果：复位后第4个时钟周期，应该执行 sub $2, $4, $3 （机器码:00831022）
  - 周期时钟上升沿（160ns处）从ROM读指令，PC指针pAddr = 0x0000 000c，控制器根据机器码产生对应的控制信号。
  - 此时$2，$3，$4寄存器的值为：2，3，0x55555555。



▶ 顶层仿真

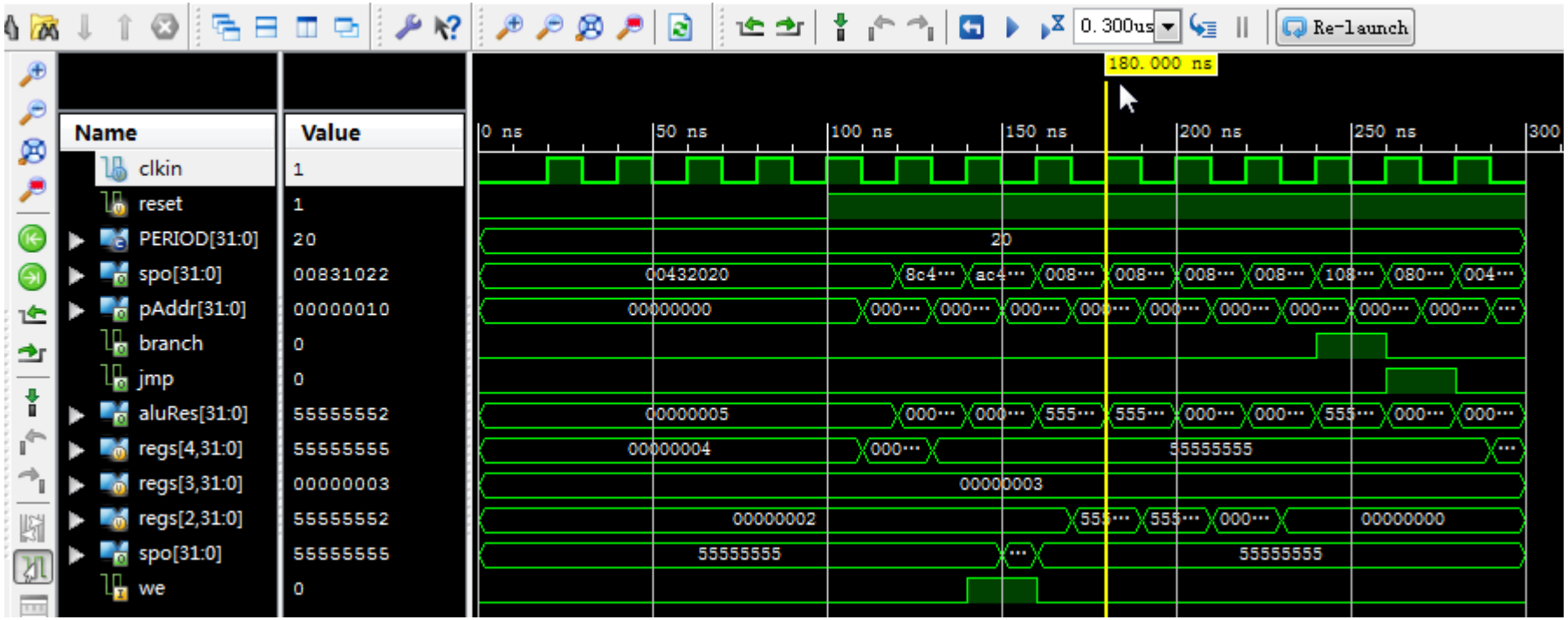
- 观察结果：复位后第4个时钟周期，应该执行 sub $2, $4, $3 （机器码:00831022）
  - 周期时钟下降沿（170ns处）执行指令，写Reg、改变PC指针，执行后，如171ns处：
    - 机器码spo = 00831022
    - $4-S3 → $2=0x55555552，（之前$2，$3，$4寄存器的值为：2，3,0x55555555）；从仿真可知：reg[2,31:0]=0x55555552，PC指针pAddr = 0x0000 0010，指向了下一个地址。



由此可见，sub 指令被正确执行。

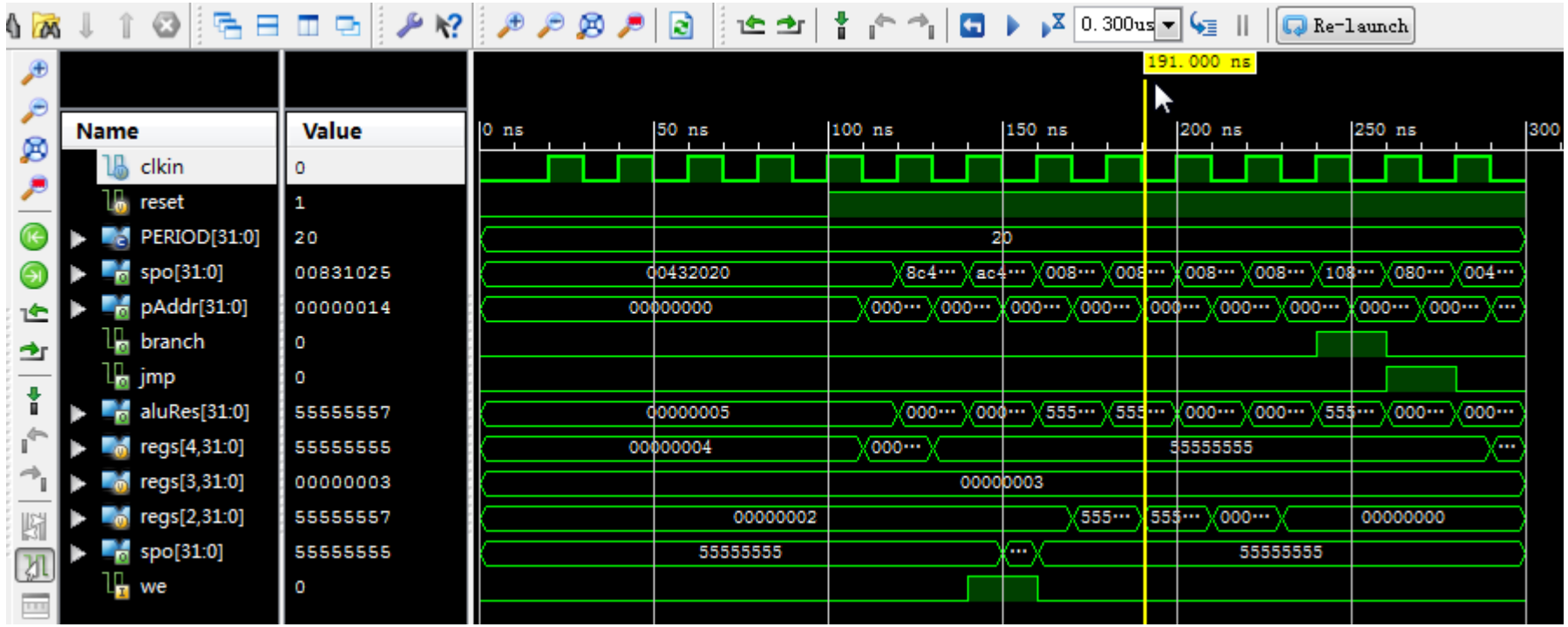
▶ 顶层仿真

- 观察结果：复位后第5个时钟周期，应该执行 or $2, $4, $3 （机器码:00831025）
  - 周期时钟上升沿（180ns处）从ROM读指令，PC指针pAddr = 0x0000 0010，控制器根据机器码产生对应的控制信号。
  - 此时$2，$3，$4寄存器的值为：0x55555552，3，0x55555555。



▶ 顶层仿真

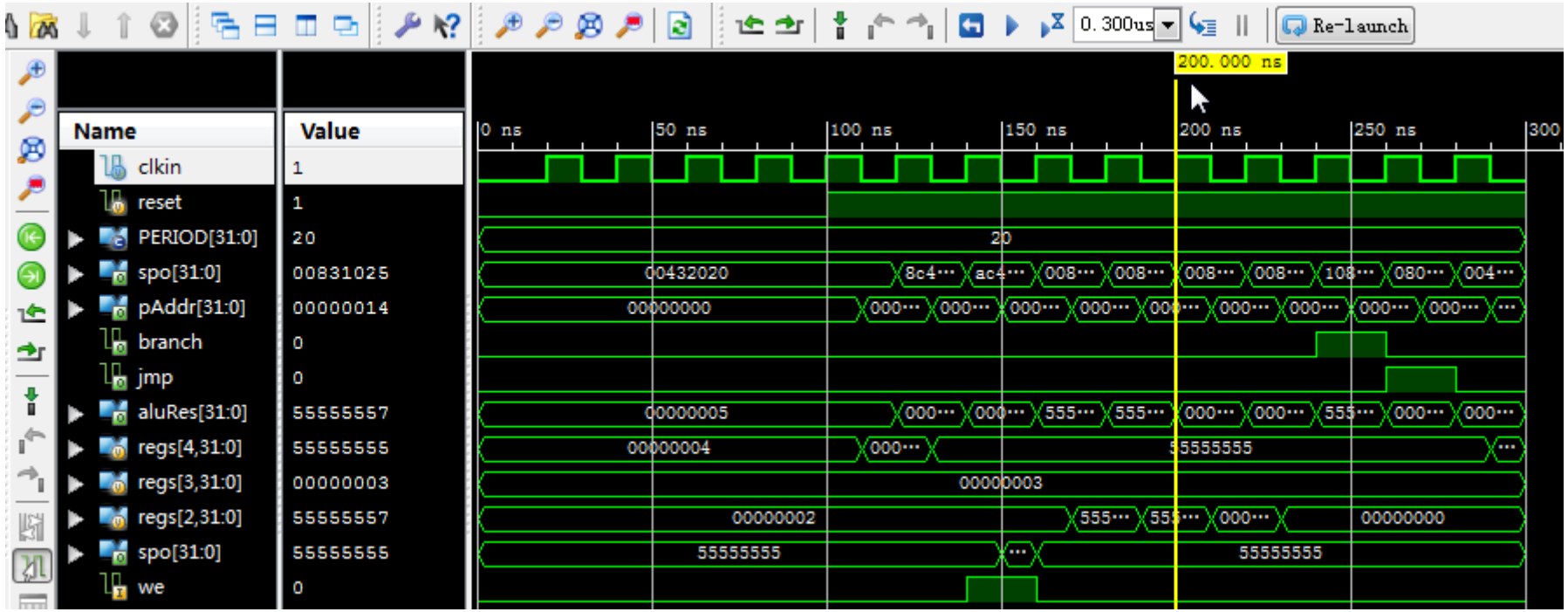
- 观察结果：复位后第5个时钟周期，应该执行 or $2, $4, $3 （机器码:00831025）
  - 周期时钟下降沿（190ns处）执行指令，写Reg、改变PC指针，执行后，如191ns处：
    - 机器码spo = 00831025；$4|S3 → $2=0x55555557，（之前$2，$3，$4寄存器的值为：0x55555552, 3, 0x55555555）；从仿真可知：reg[2,31:0]=0x55555557，PC指针pAddr = 0x0000 0014，指向了下一个地址。



由此可见，or 指令被正确执行。

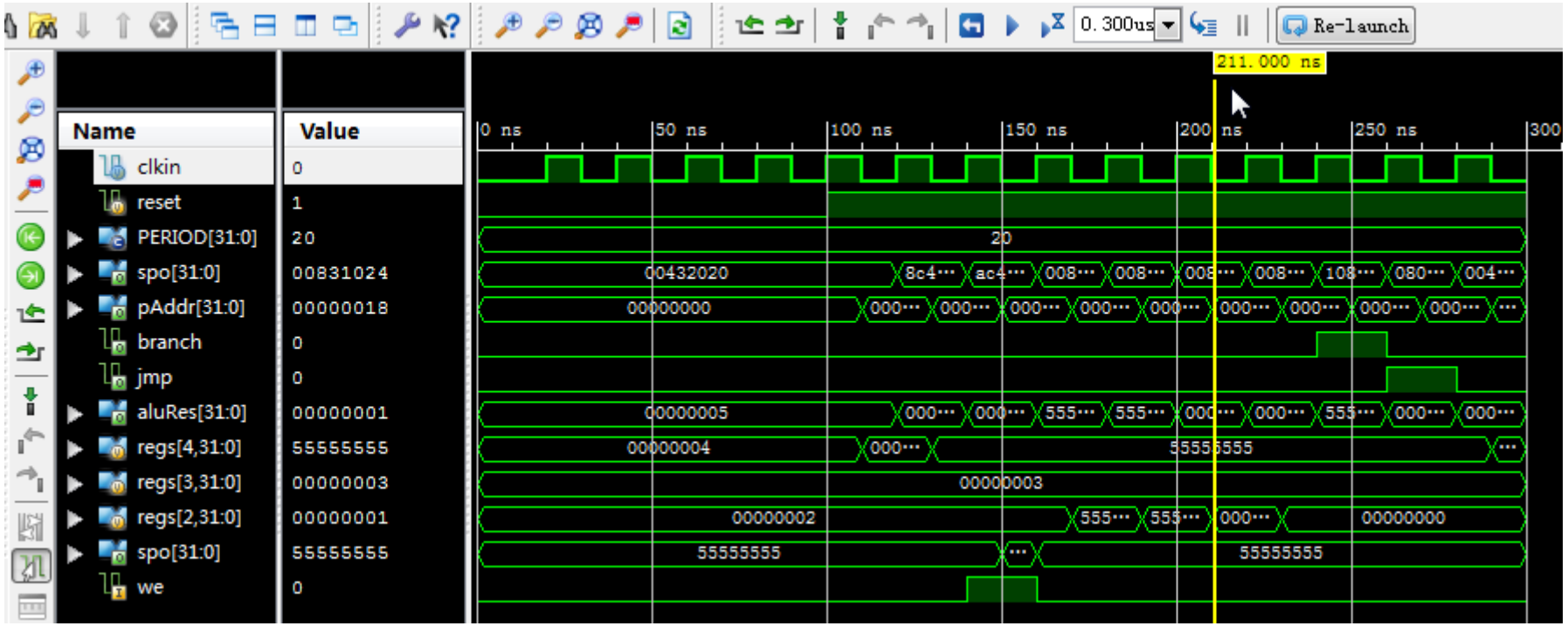
▶ 顶层仿真

- 观察结果：复位后第6个时钟周期，应该执行 and $2, $4, $3 （机器码:00831024）
  - 周期时钟上升沿（200ns处）从ROM读指令，PC指针pAddr = 0x0000 0014，控制器根据机器码产生对应的控制信号。
  - 此时$2，$3，$4寄存器的值为：0x55555557，3，0x55555555。



▶ 顶层仿真

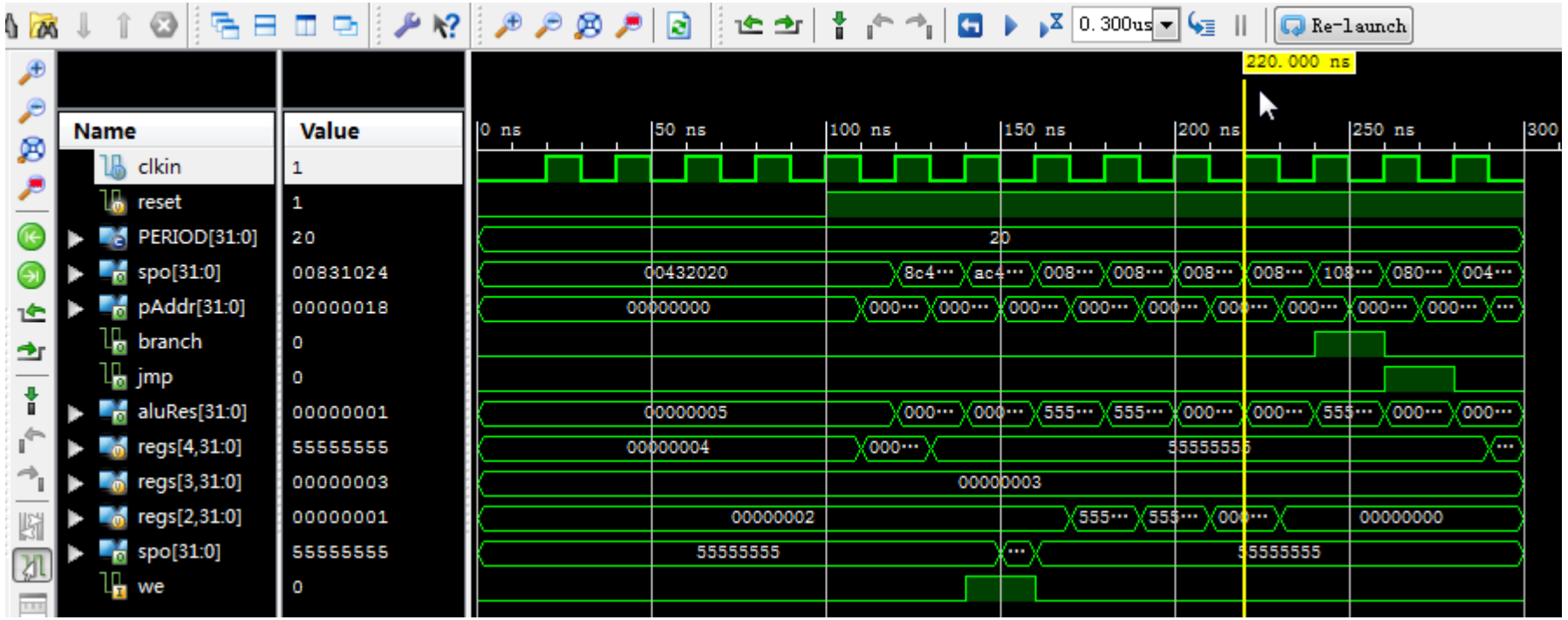
- 观察结果：复位后第6个时钟周期，应该执行 and $2, $4, $3 （机器码:00831024）
  - 周期时钟下降沿（210ns处）执行指令，写Reg、改变PC指针，执行后，如211ns处：
    - 机器码spo = 00831024；$4&S3 → $2=0x00000001，（之前$2，$3，$4寄存器的值为：0x55555557, 3, 0x55555555）；从仿真可知：reg[2,31:0]=1，PC指针pAddr = 0x0000 0018，指向了下一个地址。



由此可见，and 指令被正确执行。

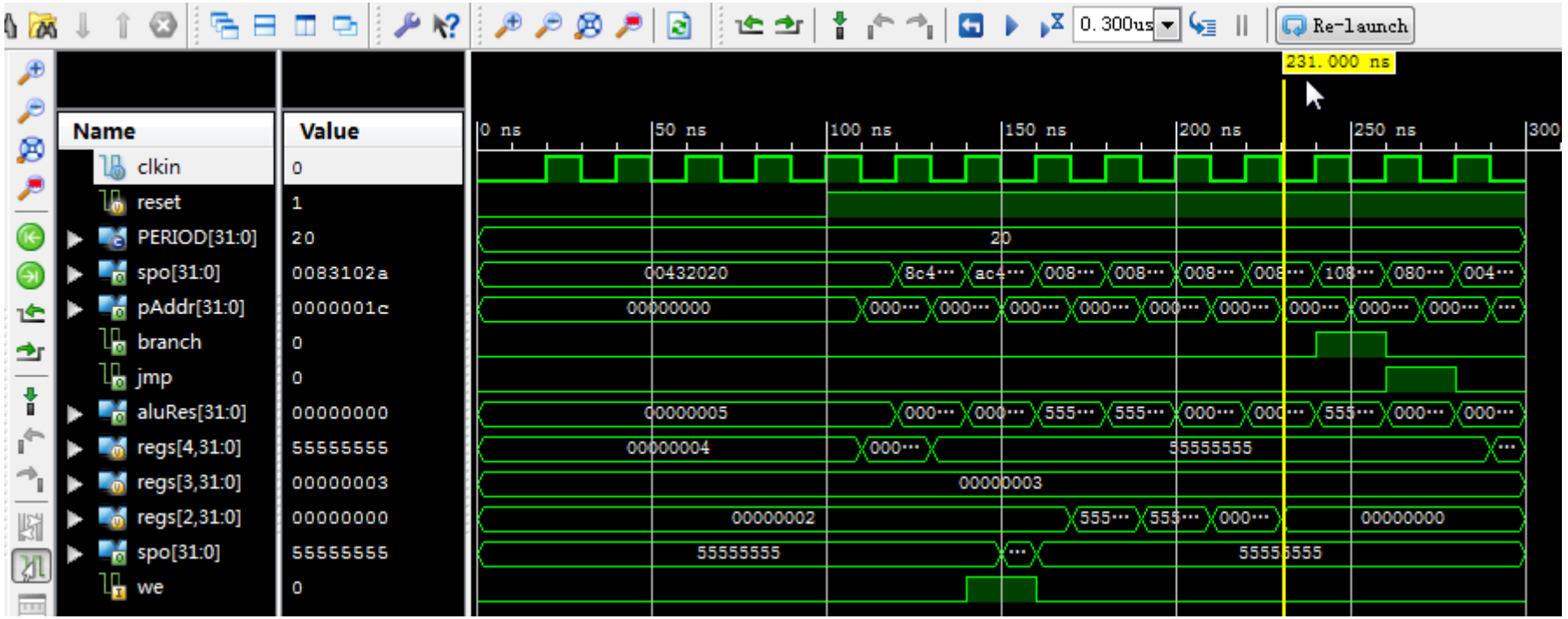
▶ 顶层仿真

- 观察结果：复位后第7个时钟周期，应该执行 slt $2, $4, $3 （机器码:0083102a）
  - 周期时钟上升沿（220ns处）从ROM读指令，PC指针pAddr = 0x0000 0018，控制器根据机器码产生对应的控制信号。
  - 此时$2，$3，$4寄存器的值为：1，3，0x55555555。



▶ 顶层仿真

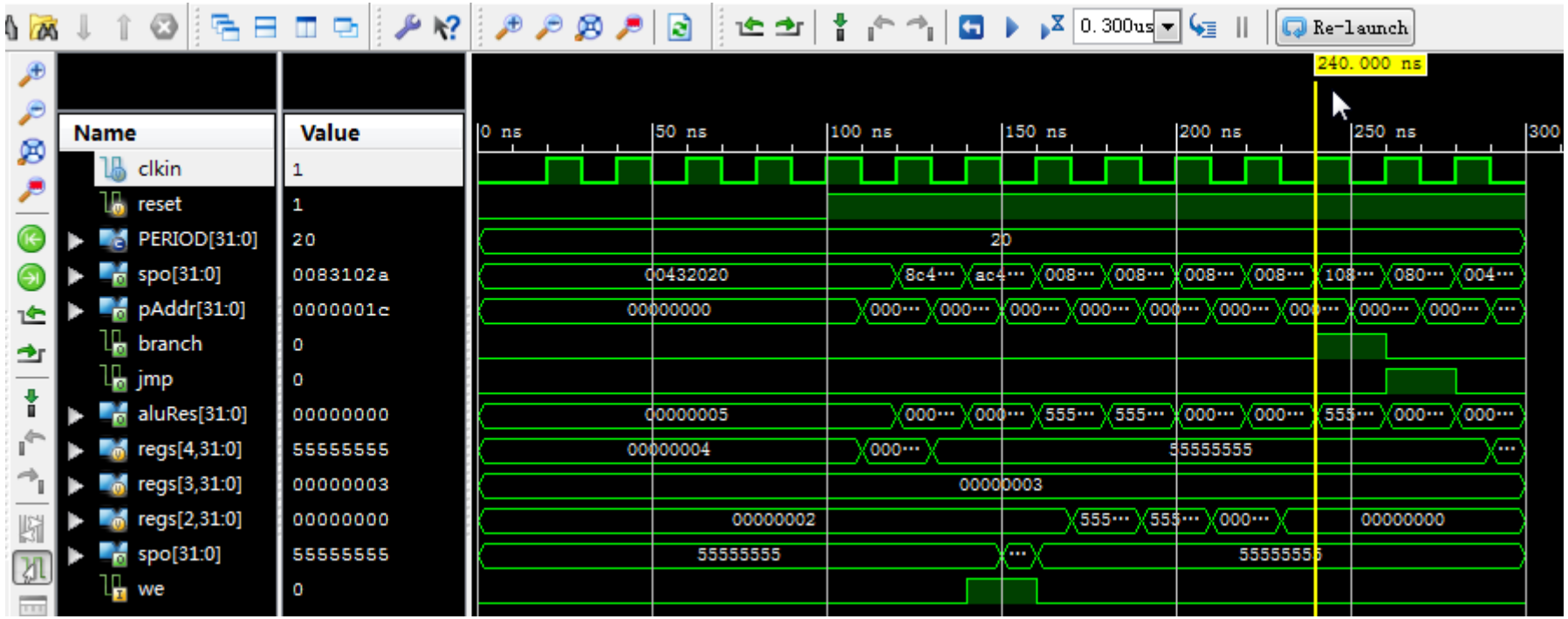
- 观察结果：复位后第7个时钟周期，应该执行 slt $2, $4, $3 （机器码:0083102a ）
  - 周期时钟下降沿（230ns处）执行指令，写Reg、改变PC指针，执行后，如231ns处：
    - 机器码spo = 0083102a；$4!<S3，所以 $2=0，（之前$2，$3，$4寄存器的值为：1, 3, 0x55555555）；从仿真可知：reg[2,31:0]=0，PC指针pAddr = 0x0000 001c，指向了下一个地址。



由此可见，slt 指令被正确执行。

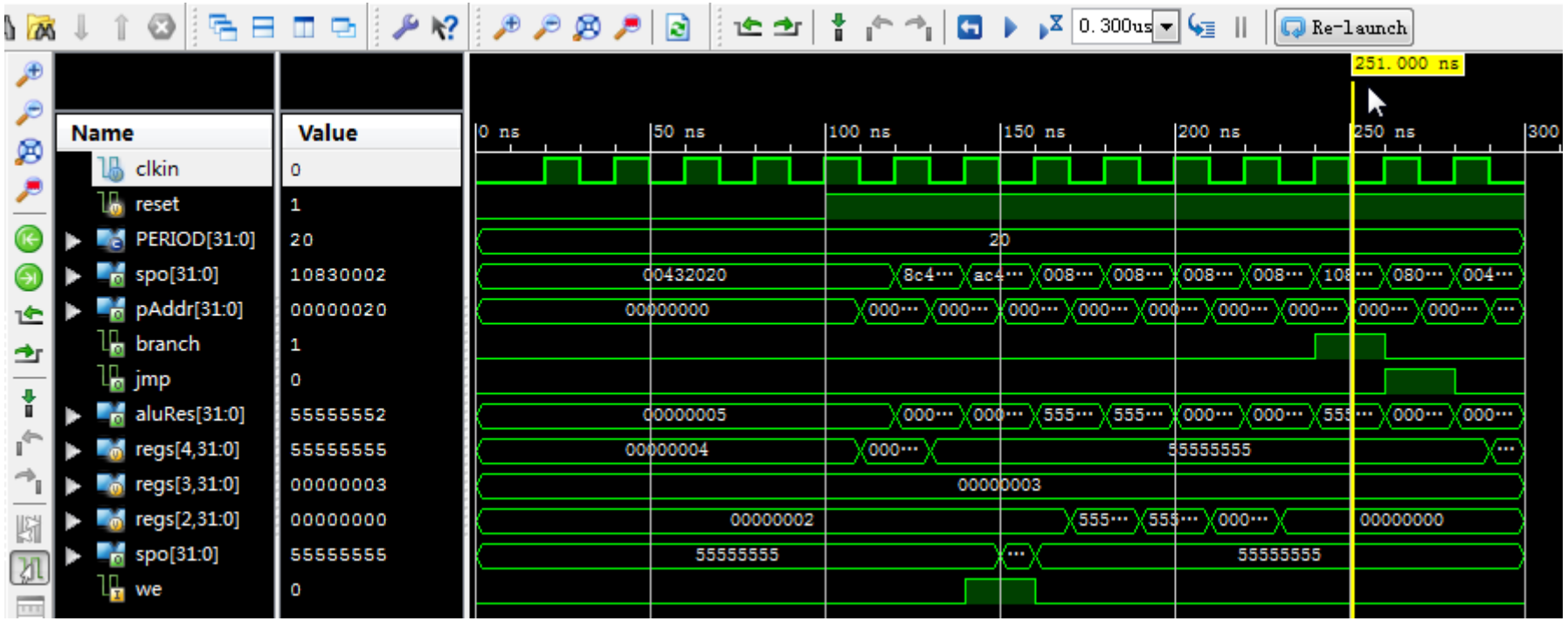
▶ 顶层仿真

- 观察结果：复位后第8个时钟周期，应该执行 beq $4, $3, exit（机器码:10830002）
  - 周期时钟上升沿（240ns处）从ROM读指令，PC指针pAddr = 0x0000 001c，控制器根据机器码产生对应的控制信号。
  - 此时$4，$3寄存器的值为：0x55555555，3。



▶ 顶层仿真

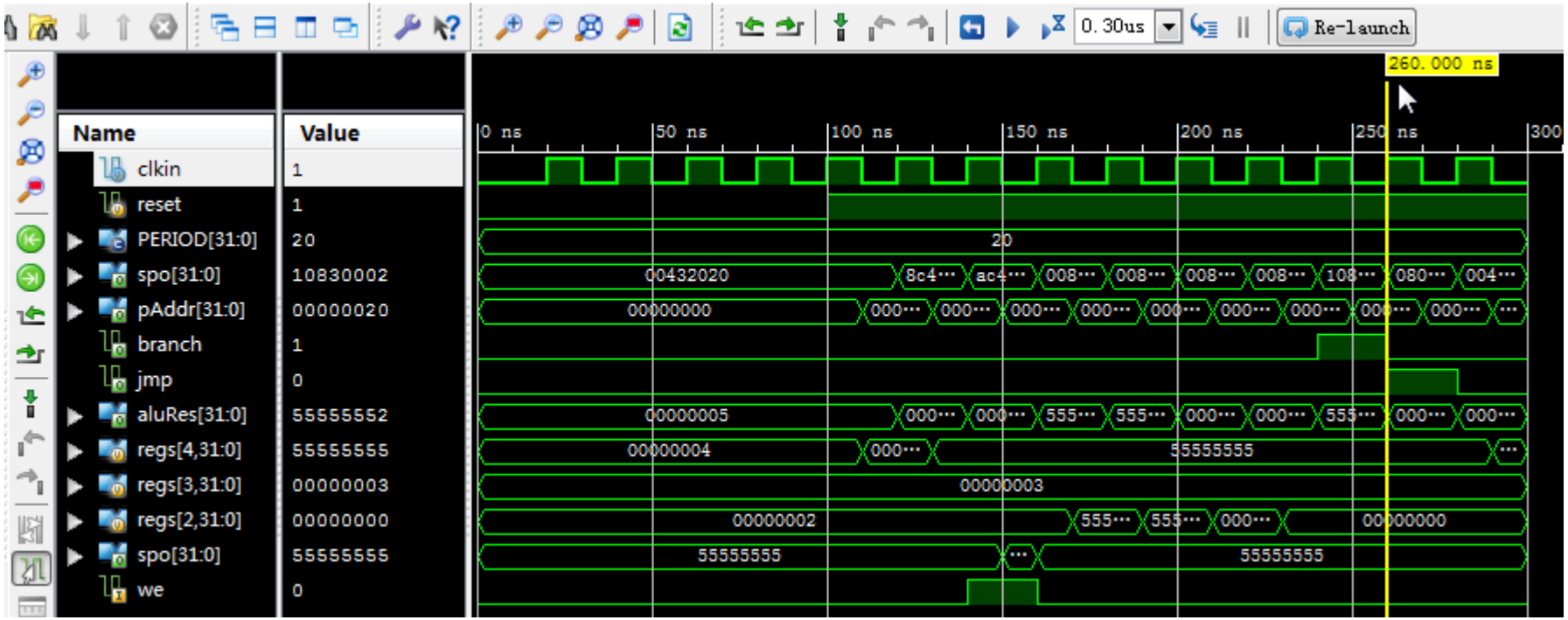
- 观察结果：复位后第8个时钟周期，应该执行 beq $4, $3, exit（机器码:10830002）
  - 周期时钟下降沿（250ns处）执行指令，写Reg、改变PC指针，执行后，如251ns处：
    - 机器码spo = 10830002；
    - $4!=S3，所以顺序执行，（之前$4，$3寄存器的值为：0x55555555，3）；从仿真可知：reg[2,31:0]=0，PC指针pAddr = 0x0000 001c + 4 = 0x0000 0020 ，顺序执行。



由此可见，beq指令顺序部分被正确执行。
如何仿真验证跳转部分？

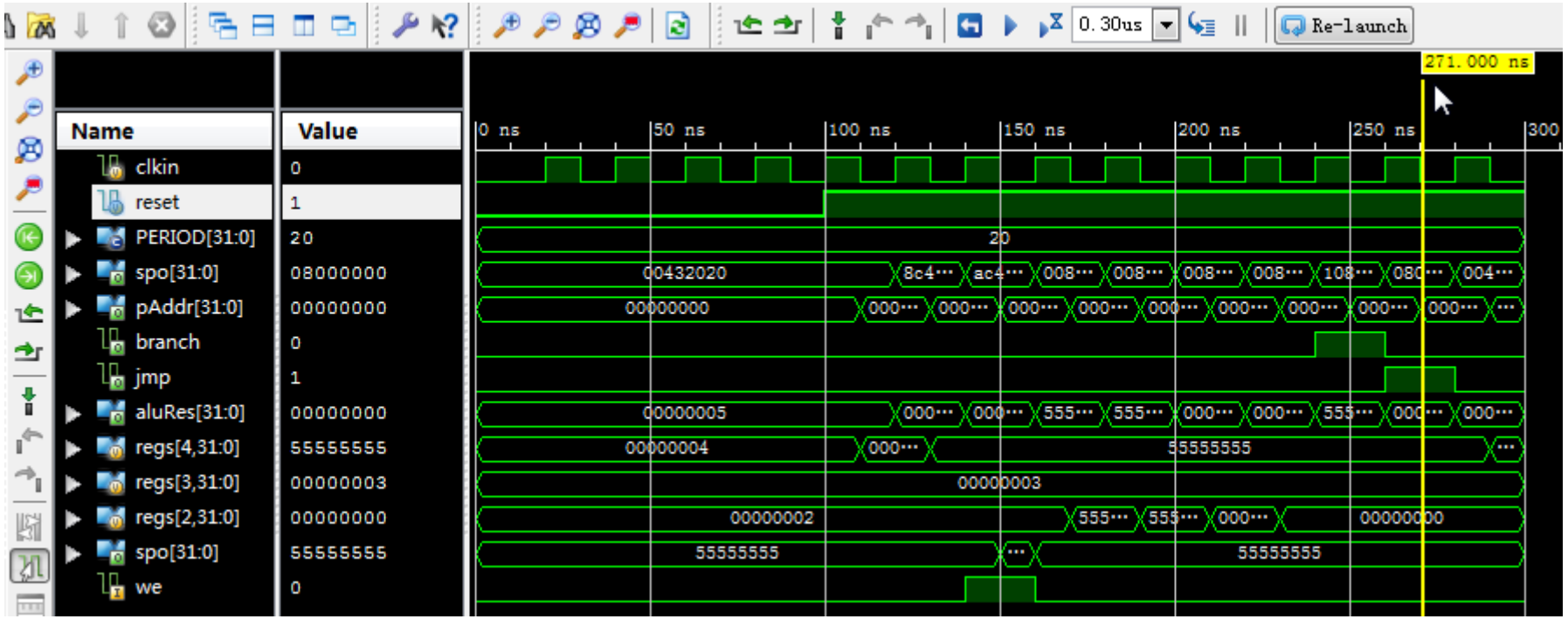
▶ 顶层仿真

- 观察结果：复位后第9个时钟周期，应该执行 j main （机器码:08000000）
  - 周期时钟上升沿（260ns处）从ROM读指令，PC指针pAddr = 0x0000 0020，控制器根据机器码产生对应的控制信号。



▶ 顶层仿真

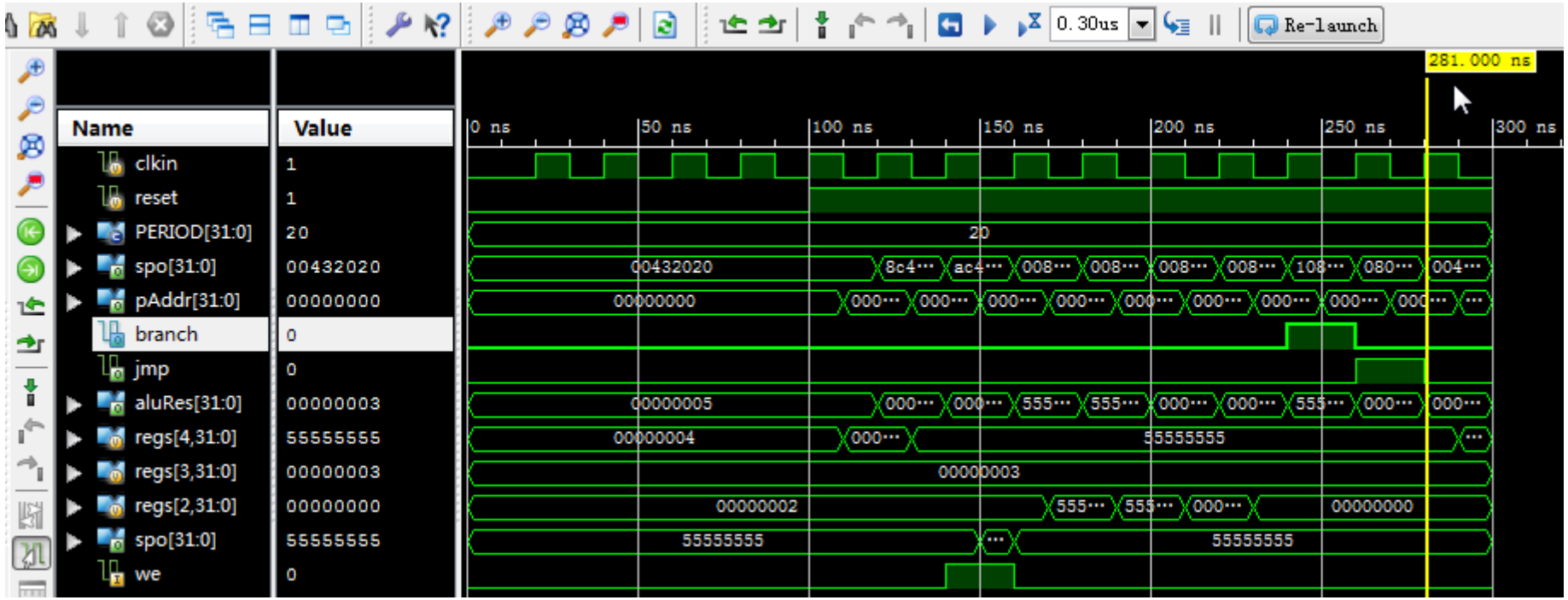
- 观察结果：复位后第9个时钟周期，应该执行 j main （机器码:08000000）
  - 周期时钟下降沿（270ns处）执行指令，写Reg、改变PC指针，执行后，如271ns处：
    - 机器码spo = 08000000；
    - 直接跳转到main，即跳到00000000地址处；从仿真可知：PC指针pAddr = 0x0000 0000 ，实现无条件跳转。



由此可见，j指令被正确执行。

▶ 顶层仿真

- 观察结果：复位后第10个时钟周期，由于之前的 j main 指令，应该执行0000000处的第一条指令 add $4, $2, $3 （机器码:00432020）
  - 周期时钟上升沿（280ns处），从ROM读指令，此后，如281ns处，PC指针pAddr = 0x0000 0000，机器码spo = 00432020，表明已经跳转到main，重新执行程序段。



即程序中的代码会被被循环不断执行。至此，所有指令的功能仿真都已完成，仿真结果说明CPU功能正确。

顶层仿真

- 如果在前述仿真过程中发现某一指令功能不正确，需要查找原因、修改代码，重新进行仿真；
- 查找错误原因的方法
  - 观测相关信号时序
  - 设置断点
  - ……

Thanks