

江南大学

# 基于 RISC-V 架构的 45 条指令 单周期 CPU 设计

学 科：计 算 机 组 成 原 理

班 级：计 科 1805

成 员：翁 诗 浩 1033180519

梁 奉 迪 1033180526

韩 梅 1033180507

2020 年 12 月 23 日

## 摘要

对于 RISC-V 架构, 本文实现了一种基于此架构的 45 条指令单周期非流水线的 CPU。实现方式是事先设计好指令所需的组件、各类控制信号的取值、各类指令的数据通路, 然后基于 Vivado2018.3 用 Verilog 语言进行硬件描述将以上思路实现。其中, 用 Vivado2018.3 自带的行为仿真进行 45 条指令的行为级测试, 所有指令测试均通过, 打包生成 IP, 连接 PYQN 开发板和相应的 IO 接口, 将打包好的设计生成比特流, 在 PYNQ 开发云 1 平台运行事先写好在 CPU 里的二分查找算法的指令, 结果与行为仿真一致, 运行成功。

小组成员贡献度表:

姓名	学号	贡献度
翁诗浩	1033180519	34%
梁奉迪	1033180526	33%
韩梅	1033180507	33%

## 目录

一、指令选取.....	4
1.1 前言.....	4
1.2 R 型指令 .....	4
1.3 I 型指令 .....	5
1.4 U 型指令 .....	5
1.5 S 型指令 .....	6
1.6 B 型指令 .....	6
1.7 J 型指令 .....	6
二、各部件功能设计.....	7
2.1 取指 .....	7
2.1.1 PC.....	7
2.1.2 pcAdder.....	7
2.1.3 inst Mem.....	8
2.2 译码 .....	9
2.2.1 ID.....	9
2.3 访存 .....	10
2.3.1 regFile.....	10
2.4 立即数扩展 .....	11
2.4.1 IE.....	11
2.5 ALU.....	11
2.5.1 ALU.....	11
2.6 存储器.....	12
2.6.1 dataMem.....	12
2.7 多路选择器.....	13
2.7.1 mux_From_rs1_PC_To_ALU.....	13
2.7.2 mux_From_rs2_IE_To_ALU.....	13
2.7.3 mux_From_PC_rs1_To_PC.....	14
2.7.4 mux_From_rs2_To_mem.....	14
2.7.5 mux_From_ALU_mem_ToReg.....	15
三、控制信号的设计.....	16
3.1 ALUctr[5:0].....	16
3.2 jump.....	17
3.3 branch[1:0].....	17
3.4 exop[2:0].....	17
3.5 zero.....	17
3.6 mrs2andie_ctr[1:0].....	18
3.7 mrs1andpc_ctr.....	18
3.8 maluandmem_ctr[2:0].....	18
3.9 mrs1andpc_ctr2.....	18
3.10 mrs2_ctr[1:0].....	19
四、数据通路设计.....	19
4.1 RTL 版 .....	19

4.1 手绘版.....	20
五、指令验证.....	20
5.1 声明初值.....	20
5.2 R 型指令测试结果 .....	20
5.3 I 型指令测试结果 .....	29
5.4 U 型指令测试结果 .....	37
5.5 S 型指令测试结果 .....	38
5.6 B 型指令测试结果 .....	40
5.7 J 型指令测试结果 .....	43
六、云平台运行二分查找算法.....	43
6.1 二分查找汇编代码及初始化.....	43
6.1.1 算法描述.....	43
6.1.2 RISC-V Assembly 插件撰写代码 .....	44
6.1.3 翻译成十六进制指令.....	45
6.1.4 初始化存储器.....	45
6.2 行为仿真测试.....	46
6.3 各部件连线图.....	46
6.4 平台测试结果.....	47
七、心得体会.....	47
7.1 翁诗浩心得: .....	47
7.2 梁奉迪心得: .....	48
7.3 韩梅心得: .....	48
八、参考文献.....	48

## 一、指令选取

### 1.1 前言

本次设计选取 RV32I 和 RV32M 大部分可实现的指令，其中囊括 R 型、I 型、U 型、S 型、B 型、J 型指令

### 1.2 R 型指令

R 型指令格式：

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	

编号	指令	名称
1	add rd, rs1, rs2	加
2	and rd, rs1, rs2	与
3	or rd, rs1, rs2	取或
4	xor rd, rs1, rs2	异或
5	srl rd, rs1, rs2	逻辑右移
6	sll rd, rs1, rs2	逻辑左移
7	slt rd, rs1, rs2	小于则置位
8	sltu rd, rs1, rs2	无符号小于则置位
9	div rd, rs1, rs2	除法
10	divu rd, rs1, rs2	无符号除法
11	mul rd, rs1, rs2	乘
12	mulh rd, rs1, rs2	高位乘
13	mulhsu rd, rs1, rs2	高位有符号-无符号乘.
14	mulhu rd, rs1, rs2	高位无符号乘
15	rem rd, rs1, rs2	求余数
16	remu rd, rs1, rs2	求无符号数的余数
17	sra rd, rs1, rs2	算术右移
18	sub rd, rs1, rs2	减

## 1.3 I 型指令

I 型指令格式

31	20 19	15 14	12 11	7 6	0
imm[11:0]		rs1	funct3	rd	opcode

编号	指令	名称
19	addi rd, rs1, immediate	加立即数
20	ori rd, rs1, immediate	或立即数
21	andi rd, rs1, immediate	与立即数
22	slli rd, rs1, shamt	立即数逻辑左移
23	slti rd, rs1, immediate	小于立即数则置位
24	sltiu rd, rs1, immediate	无符号小于立即数则置位
25	srai rd, rs1, shamt	立即数算术右移
26	xori rd, rs1, immediate	立即数异或
27	lb rd, offset(rs1)	字节加载
28	lbu rd, offset(rs1)	无符号字节加载
39	lh rd, offset(rs1)	半字加载
30	lhu rd, offset(rs1)	无符号半字加载
31	lw rd, offset(rs1)	字加载
32	lwu rd, offset(rs1)	无符号字加载
33	jalr rd, offset(rs1)	跳转并寄存器链接

## 1.4 U 型指令

U 型指令格式

31	12 11	7 6	0
imm[31:12]		rd	opcode

编号	指令	名称
34	auipc rd, immediate	PC 加立即数
35	lui rd, immediate	高位立即数加载

## 1.5 S 型指令

S 型指令格式

31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]	rs2			rs1		funct3	imm[4:0]			opcode	

编号	指令	名称
36	sw rs2, offset(rs1)	存字
37	sb rs2, offset(rs1)	存字节
38	sh rs2, offset(rs1)	存半字

## 1.6 B 型指令

B 型指令格式

31	30	25	24	20	19	15	14	12	11	8	7	6	0
imm[12]	imm[10:5]		rs2	rs1	funct3	imm[4:1]		imm[11]		opcode			

编号	指令	名称
39	beq rs1, rs2, offset	相等时分支
40	bge rs1, rs2, offset	大于等于时分支
41	bgeu rs1, rs2, offset	无符号大于等于时分支
42	blt rs1, rs2, offset	小于时分支
43	bltu rs1, rs2, offset	无符号小于时分支
44	bne rs1, rs2, offset	不相等时分支

## 1.7 J 型指令

J 型指令格式

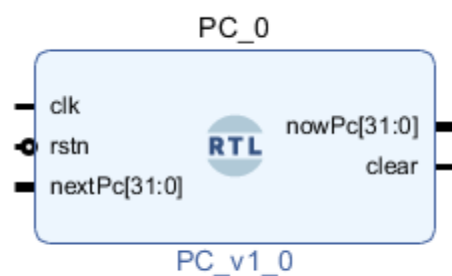
31	30	21	20	19	12	11	7	6	0
imm[20]	imm[10:1]		imm[11]		imm[19:12]		rd	opcode	

编号	指令	名称
45	jal rd, offset	跳转并链接

## 二、各部件功能设计

### 2.1 取指

#### 2.1.1 PC

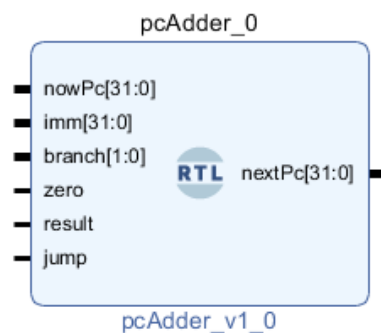


**主要功能描述：** 时钟每翻转一次 PC 就会指向下一条指令的地址。

**部分代码逻辑：**

```
always @(posedge clk) begin
    if(~clear)
        nowPc<=32'b0;
    else
        nowPc<=nextPc;
end
```

#### 2.1.2 pcAdder



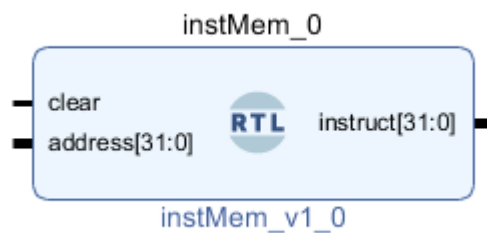
**主要功能描述：** PC 的加法器，控制下一条指令的地址是否要加上立即数，即判断是否要跳转并执行 PC 指针的跳转，输入当前指令地址（nowPc）、立即数



(imm)和判断条件(branch、zero...),输出下一条指令的地址(nextPc)。  
部分代码逻辑:

```
always @(*) begin
    if(branch==2'b01&&(zero||jump))
        nextPc<=nowPc+imm;
    else if(branch==2'b10&&(~zero||jump))
        nextPc<=nowPc+imm;
    else if(branch==2'b11&&result)
        nextPc<=nowPc+imm;
    else if(jump)
        nextPc<=nowPc+imm;
    else
        nextPc<=nowPc+4'h4;
end
```

### 2.1.3 inst Mem



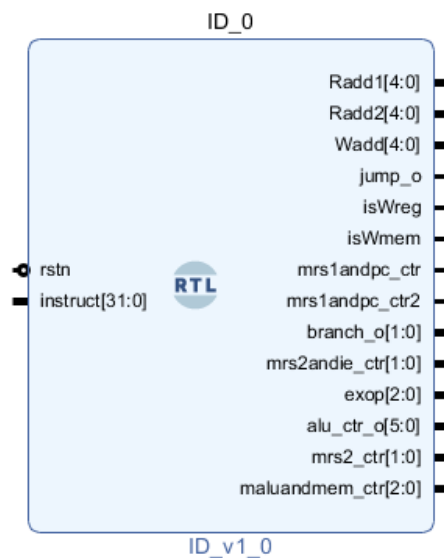
**主要功能描述:** 存储指令的寄存器,输入一个 32 位的指令地址,输出一条 32 位的指令。

部分代码逻辑:

```
always @(*) begin
    if(~clear)
        instruct<=0;
    else
        instruct<=instMem_text[address[9:2]];
end
```

## 2.2 译码

### 2.2.1 ID



**主要功能描述：**指令译码器，输入一条指令（instruct），输出整个CPU所有外部控制信号的取值。

**部分代码逻辑：**

```
case(f7)
  7'b0000000: begin
    case(f3)
      3'b000: begin
        alu_ctr_o <= 6'b000000; //add
      end
      3'b111: begin
        alu_ctr_o <= 6'b000001; //and
      end
      3'b110: begin
        alu_ctr_o <= 6'b000010; //or
      end
      3'b100: begin
        alu_ctr_o <= 6'b000011; //xor
      end
      3'b101: begin
        alu_ctr_o <= 6'b000100; //srl
      end
      3'b001: begin
        alu_ctr_o <= 6'b000101; //sll
      end
    end
  end
```

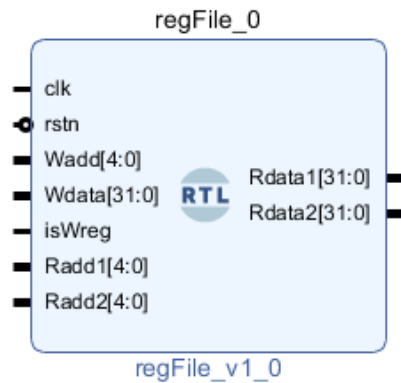
```

        3'b010: begin
            alu_ctr_o <= 6'b000110; //slt
        end
        3'b011:begin
            alu_ctr_o <= 6'b000111; //sltu
        end
    endcase
end

```

## 2.3 访存

### 2.3.1 regFile



**主要功能描述:** 寄存器堆，用于读取或存储寄存器，输入读取或存储的地址以及使能信号，输出读出的数据。

**部分代码逻辑:**

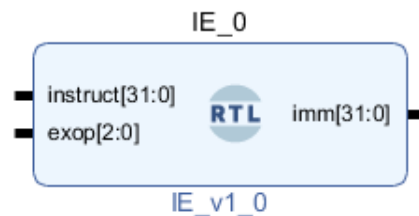
```

always @(*) begin//读
    if(~rstn)begin
        Rdata1<=32'b0;
        Rdata2<=32'b0;
    end
    else begin
        Rdata1<=regF[Radd1];
        Rdata2<=regF[Radd2];
    end
end
end

```

## 2.4 立即数扩展

### 2.4.1 IE



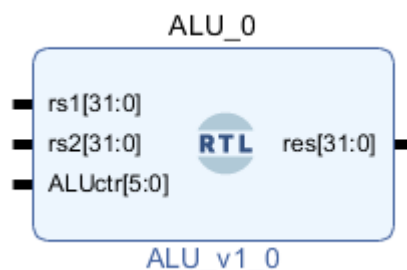
**主要功能描述：**立即数扩展器，输入指令（instruct）和立即数扩展的控制信号（exop）输出扩展后的 32 位立即数（imm）。

**部分代码逻辑：**

```
3'b100: begin // jal
    imm <= {{12{instruct[31]}}, instruct[19:12], instruct[20],
    instruct[30:21], 1'b0};
end
3'b101: begin // slli
    imm <= {{27{instruct[31]}}, instruct[24:20]};
end
3'b110: begin // 无符号扩展 31:20 立即数
    imm <= {20'b0 , instruct[31:20]};
end
```

## 2.5 ALU

### 2.5.1 ALU



**主要功能描述：**算数处理单元，两个数据输入端口（rs1、rs2），一个控制信号端口（ALUctr），输出计算结果

**部分代码逻辑：**

```
6'b000011:begin
    res<=rs1^rs2;
```



```

dataR<=mem[address[9:2]];
end

```

## 2.7 多路选择器

### 2.7.1 mux\_From\_rs1\_PC\_To\_ALU



**主要功能描述：**多路选择器，选择 rs1 还是 pc 的值到 ALU 进行计算，输入两个值和信号，输出选择后的值。

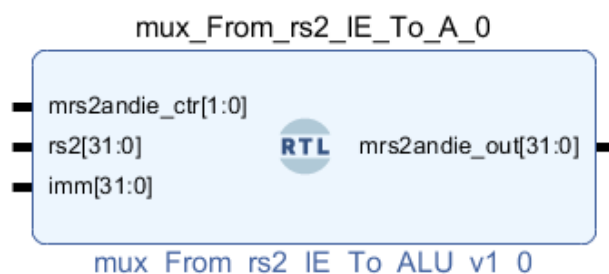
**部分代码逻辑：**

```

case (mrs1andpc_ctr)
  1'b0:begin
    mrs1andpc_out<=rs1;
  end
  1'b1:begin
    mrs1andpc_out<=pc;
  end
endcase

```

### 2.7.2 mux\_From\_rs2\_IE\_To\_ALU



**主要功能描述：**多路选择器，选 rs2 还是立即数（imm）到 ALU 进行计算，输入两个值和信号，输出选择后的值。

**部分代码逻辑：**

```

case (mrs2andie_ctr)

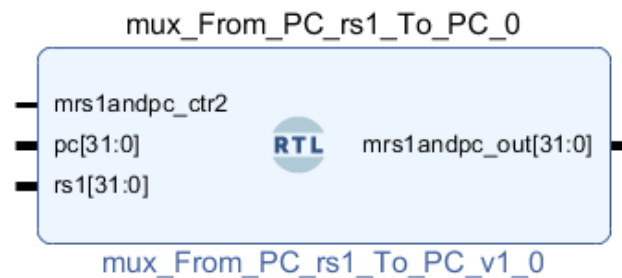
```

```

2'b00:begin
    mrs2andie_out<=rs2;
end
2'b01:begin
    mrs2andie_out<=4'h4;
end
2'b10:begin
    mrs2andie_out<=imm;
end
endcase

```

### 2.7.3 mux\_From\_PC\_rs1\_To\_PC



**主要功能描述:** 多路选择器，选 pc 还是 rs1 的值到 pc，此选择器用于特定 J 型指令 jalr，输入为两个值和控制信号，输出为选择的值。

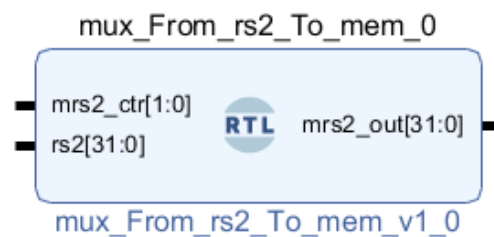
**部分代码逻辑:**

```

case(mrs1andpc_ctr2)
1'b0:begin
    mrs1andpc_out<=pc;
end
1'b1:begin
    mrs1andpc_out<=rs1;
end
endcase

```

### 2.7.4 mux\_From\_rs2\_To\_mem

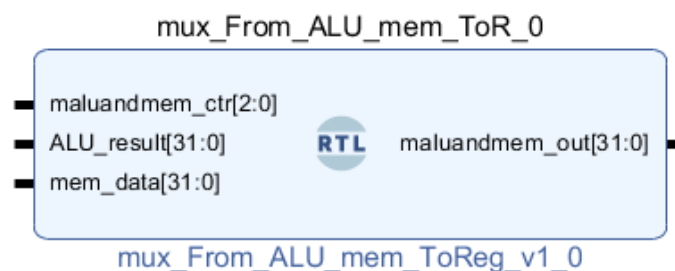


**主要功能描述：**变化 rs2 送到存储器，因为有指令需要变化存储前数据的位数，所以设置此器件，输入为一个数据值和控制信号，输出为要送到存储器的值。

**部分代码逻辑：**

```
case (mrs2_ctr)
  1'b00:begin
    mrs2_out<=rs2;
  end
  2'b01:begin
    mrs2_out <= {{24{rs2[7]}}, rs2[7:0]};
  end
  2'b11:begin
    mrs2_out <= {{16{rs2[15]}}, rs2[15:0]};
  end
endcase
```

## 2.7.5 mux\_From\_ALU\_mem\_ToReg



**主要功能描述：**多路选择器，选择 ALU 的输出还是存储器的输出送到寄存器，此选择器区分 load 等指令和其他指令，输入为两个数据值和控制信号，输出为要送到寄存器的值。

**部分代码逻辑：**

```
case (maluandmem_ctr)
  3'b000:begin
    maluandmem_out<=ALU_result;
  end

  3'b001:begin
    maluandmem_out<=mem_data;
  end

  3'b010: begin
    maluandmem_out <= {{24{mem_data[7]}}, mem_data[7:0]};
  end

  3'b011: begin
```



```

        maluandmem_out <= {{16{mem_data[15]}}, mem_data[15:0]}
;
    end
    3'b100: begin
        maluandmem_out <= {24'b0, mem_data[7:0]};
    end
    3'b101: begin
        maluandmem_out <= {16'b0, mem_data[15:0]};
    end
endcase

```

### 三、控制信号的设计

#### 3.1 ALUctr[5:0]

信号功能：6 位控制信号控制 ALU 执行何种运算

信号取值	运算
000000	res<=rs1+rs2
000001	res<=rs1&rs2
000010	res<=rs1 rs2
000011	res<=rs1^rs2
000100	res<=rs1>>rs2
000101	res<=rs1<<rs2
000110	if(rs1<rs2)res<=32'b0000_0000_0000_0000_0000_0000_0001;else res<=0;
000111	if(rs1<rs2)res<=32'b0000_0000_0000_0000_0000_0000_0001;else res<=0;
001000	res<=rs1/rs2;
001010	res<=rs1*rs2
001011	res<=rs1*rs2>>32
001110	res<=rs1%rs2
010001	res<=rs1-rs2
010010	res<=rs1<<rs2
010011	if(rs1<rs2)res<=32'b0000_0000_0000_0000_0000_0000_0001;else res<=0;
010101	res<=(rs1>=rs2)
010110	res<=(rs1<rs2)
010111	res<=rs2

### 3.2 jump

**信号功能：**单值控制信号，为 1 时代表指令地址需要执行跳转，为 0 时代表此控制信号无效。

取值	操作
1	$\text{nextPc} \leftarrow \text{nowPc} + \text{imm}$
0	None

### 3.3 branch[1:0]

**信号功能：**双值控制信号，控制分支类型。

取值	操作
01	$\text{if}(\text{alu} == 1 \mid \mid \text{jump} == 1) \text{ nextPc} \leftarrow \text{nowPc} + \text{imm}$
10	$\text{if}(\text{alu} == 0 \mid \mid \text{jump} = 1) \text{ nextPc} \leftarrow \text{nowPc} + \text{imm}$
11	$\text{If}(\text{result}) \text{ nextPc} \leftarrow \text{nowPc} + \text{imm}$

### 3.4 exop[2:0]

**信号功能：**三值控制信号，控制立即数扩展的类型。


取值	操作
000	$\text{imm} \leftarrow \{ \{20\{\text{instruct}[31]\} \}, \text{instruct}[31:20] \}$
001	$\text{imm} \leftarrow \{ \text{instruct}[31:12], 12'b0 \}$
010	$\text{imm} \leftarrow \{ \{20\{\text{instruct}[31]\} \}, \text{instruct}[31:25], \text{instruct}[11:7] \}$
011	$\text{imm} \leftarrow \{ \{20\{\text{instruct}[31]\} \}, \text{instruct}[7], \text{instruct}[30:25], \text{instruct}[11:8], 1'b0 \}$
100	$\text{imm} \leftarrow \{ \{12\{\text{instruct}[31]\} \}, \text{instruct}[19:12], \text{instruct}[20], \text{instruct}[30:21], 1'b0 \};$
101	$\text{imm} \leftarrow \{ \{27\{\text{instruct}[31]\} \}, \text{instruct}[24:20] \}$
110	$\text{imm} \leftarrow \{ 20'b0, \text{instruct}[31:20] \}$

### 3.5 zero

**信号功能：**单值控制信号，代表 ALU 结果是否为 0。


取值	操作
1	ALU 结果为 0
0	ALU 结果不为 0

### 3.6 mrs2andie\_ctr[1:0]

信号功能：双值控制信号，控制  mux\_From\_rs2\_IE\_To\_ALU (mux\_F 多路选择器，选 rs2 还是立即数到 ALU。


取值	操作
00	选择 rs2
01	赋定值 4
10	选择 imm (立即数)

### 3.7 mrs1andpc\_ctr

信号功能：单值控制信号，控制  mux\_From\_rs1\_PC\_To\_ALU (mux\_F 多路选择器，选择 rs1 还是 pc 的值到 ALU。


取值	操作
0	选择 rs1
1	选择 pc

### 3.8 maluandmem\_ctr[2:0]

信号功能：三值控制信号，控制  mux\_From\_ALU\_mem\_ToReg0:r 多路选择器，选择 ALU 的输出还是存储器的输出到 reg。

取值	操作
000	alu 结果
001	存储器结果 取 32 位
010	存储器结果 有符号扩展低 8 成 32 位
011	存储器结果 有符号扩展低 16 成 32 位
100	存储器结果 无符号扩展低 8 成 32 位
101	存储器结果 无符号扩展低 16 成 32 位

### 3.9 mrs1andpc\_ctr2

信号功能：单值控制信号，控制  mux\_From\_PC\_rs1\_To\_PC (mux\_F 多路选择器，选择 rs1 的结果还是 PC 的结果送到 PC

取值	操作
0	pc 的结果
1	rs1 的结果

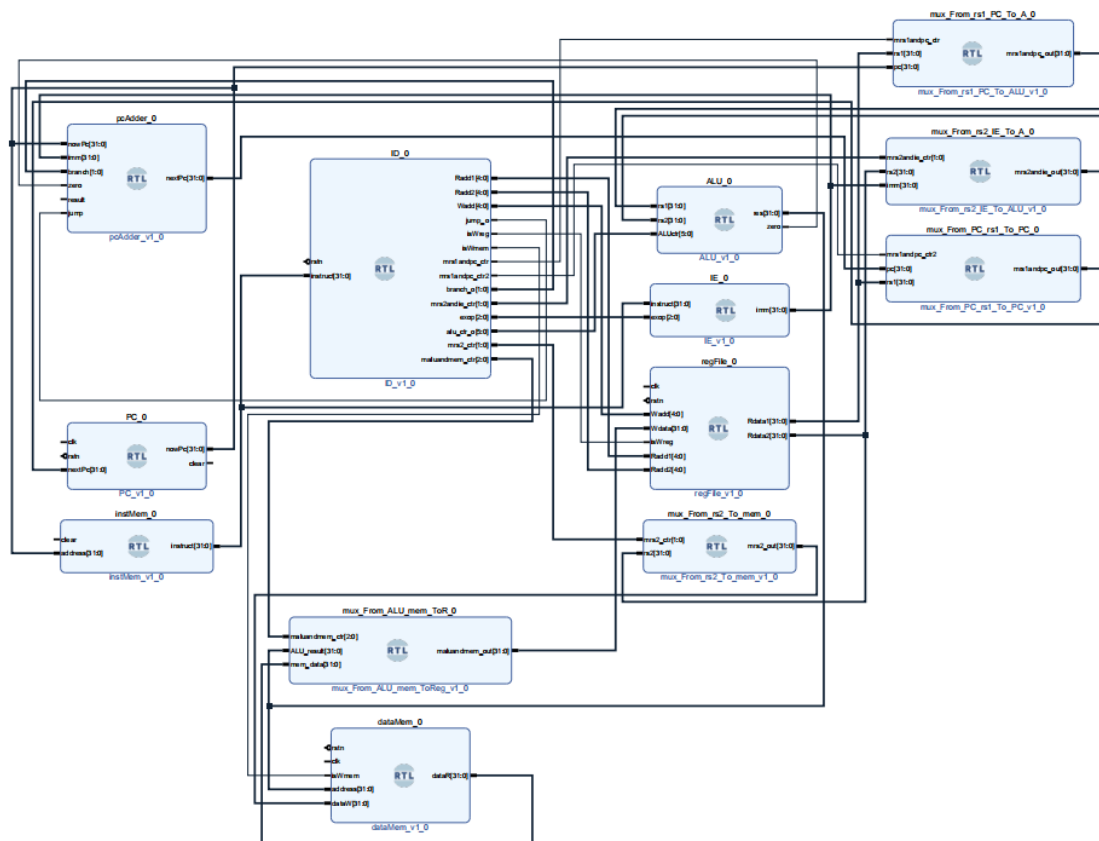
### 3.10 mrs2\_ctr[1:0]

信号功能：控制 ● `mux_From_rs2_To_mem (mux_`，变化 rs2 送到存储器

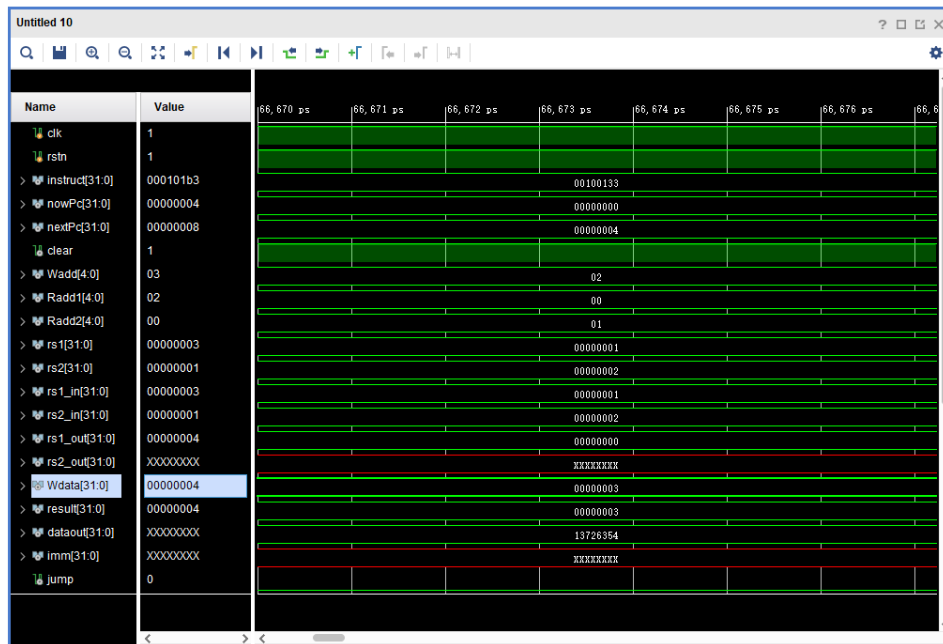
取值	操作
00	rs2 直接送到 mem
01	有符号取 rs2 低 8 位进行有符号扩展送到 mem
11	有符号取 rs2 低 16 位进行有符号扩展送到 mem

## 四、数据通路设计

### 4.1 RTL 版





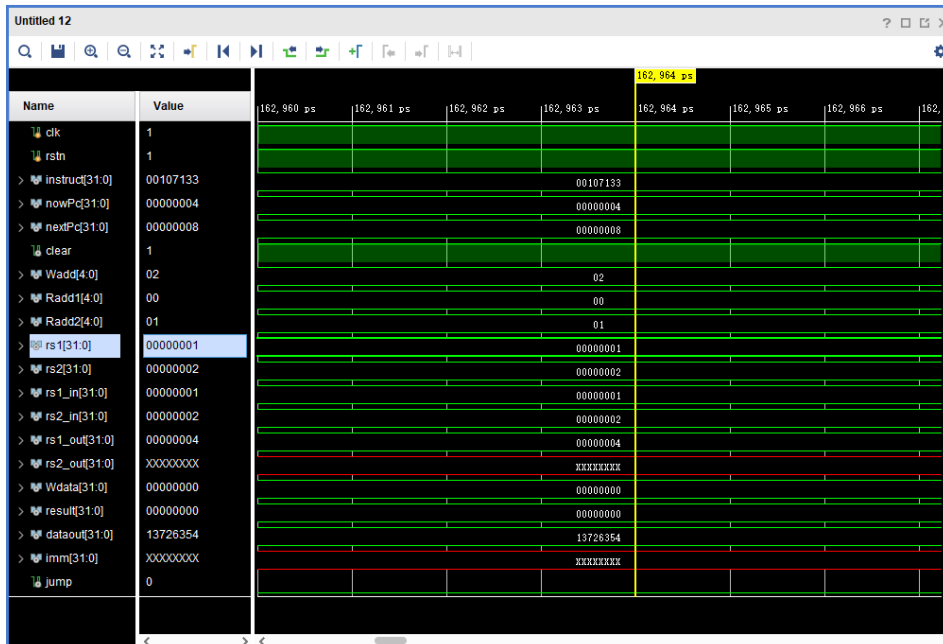


2. and

十六进制: 00107133

解释:  $x[rd] = x[rs1] \& x[rs2]$

测试结果:  $1\&2=0$  (Wdata) 结果正确

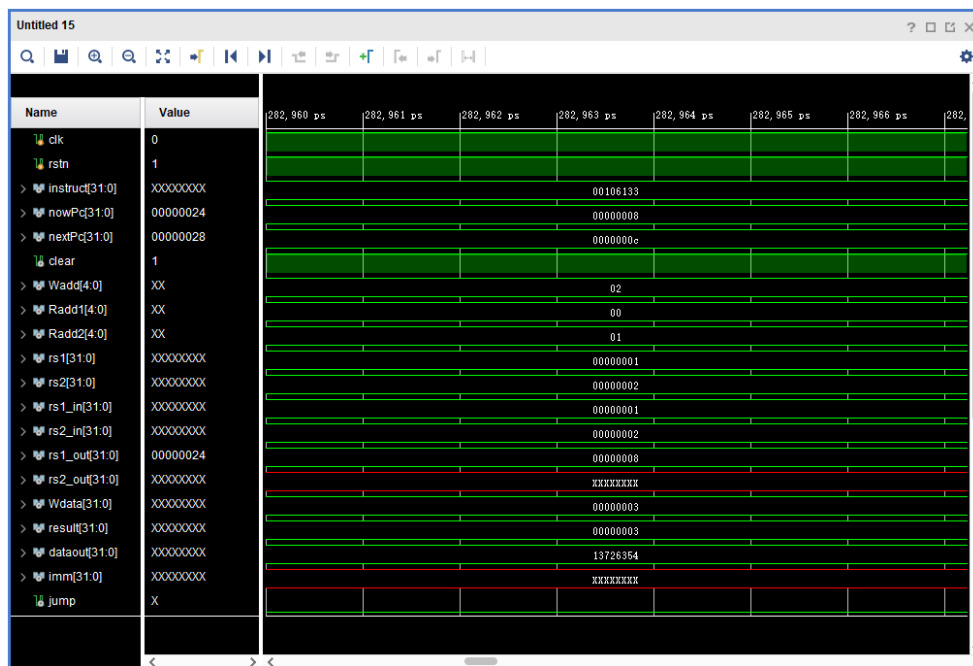


3. or

十六进制: 00106133

解释:  $x[rd]=x[rs1] \mid x[rs2]$

测试结果:  $1 \mid 2=3$  (Wdata) 结果正确

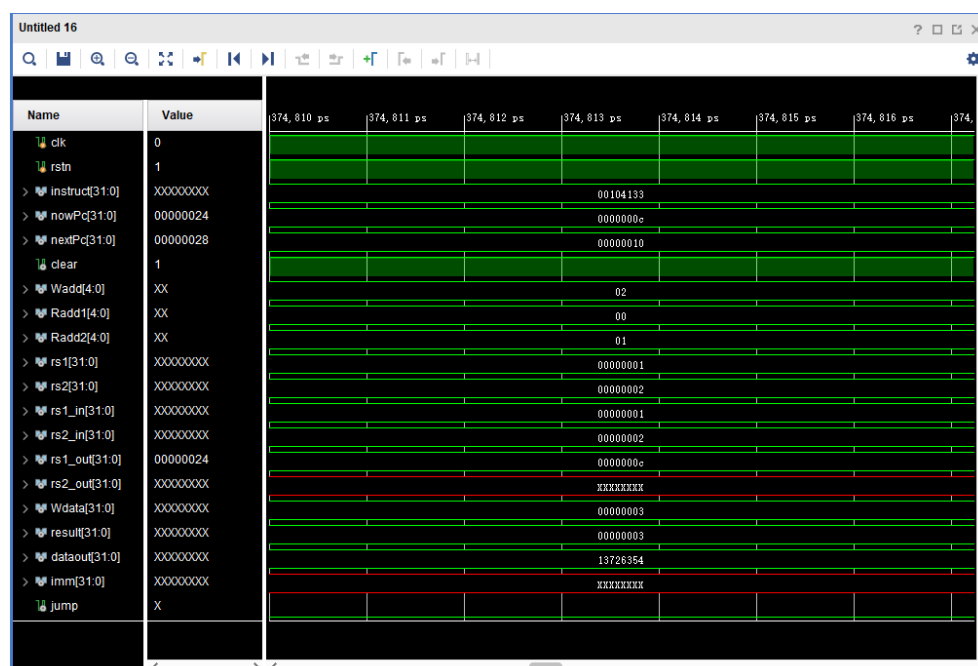


#### 4. xor

十六进制: 00104133

解释:  $x[rd] = x[rs1] \oplus x[rs2]$

测试结果:  $1 \oplus 2 = 3$  (Wdata) 结果正确

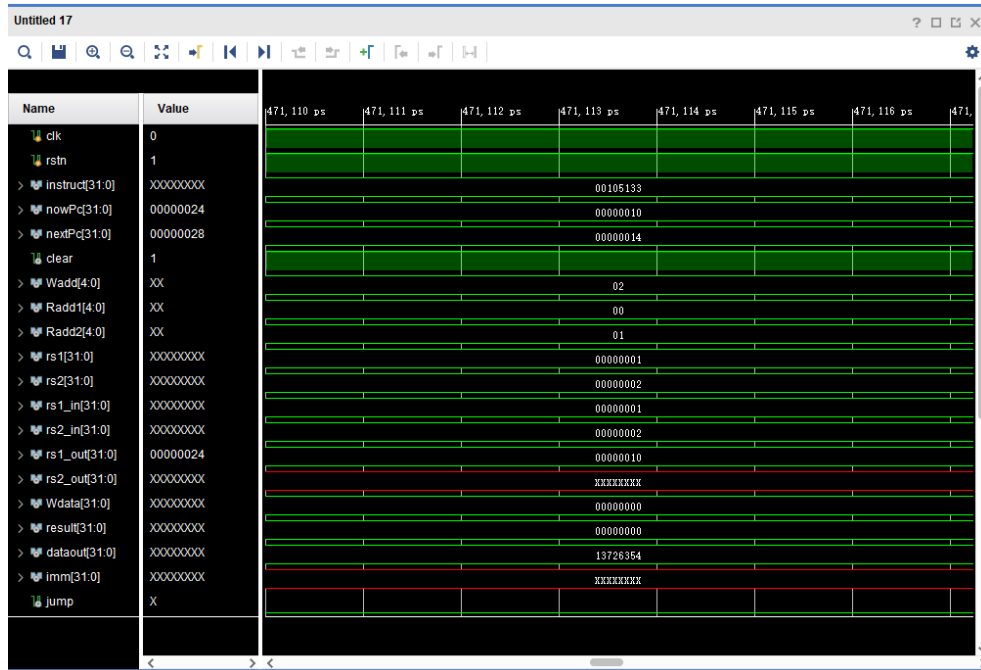


#### 5. srl

十六进制: 00105133

解释:  $x[rd] = (x[rs1] \gg ux[rs2])$  (无符号)

测试结果:  $1 \gg 2 = 0$  (Wdata) 结果正确

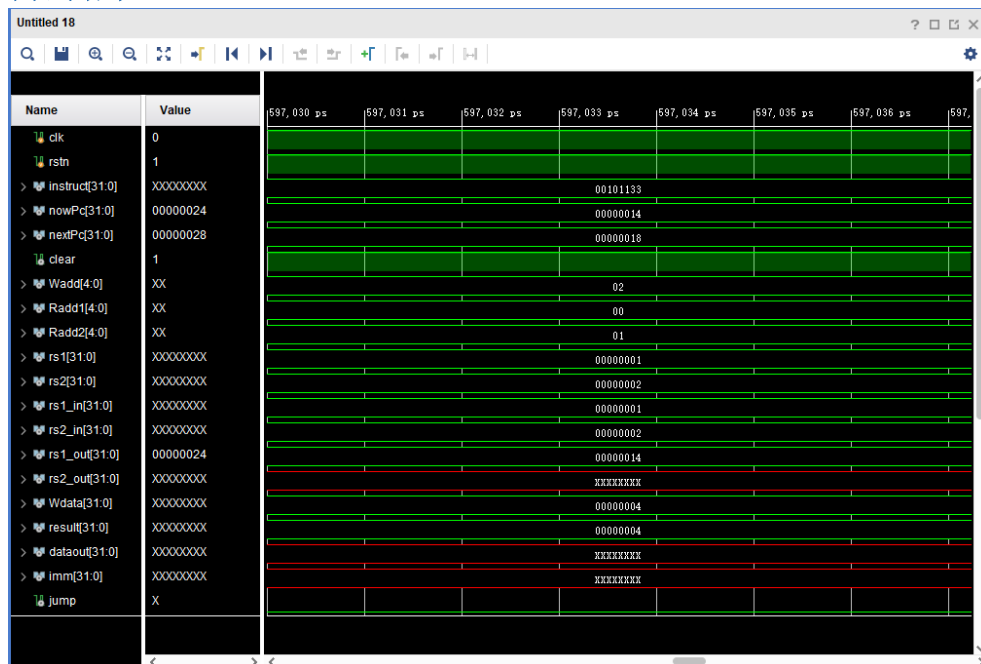


## 6. sll

十六进制: 00105133

解释:  $x[rd] = x[rs1] \ll x[rs2]$

测试结果:  $1 \ll 2 = 4$  (Wdata) 结果正确



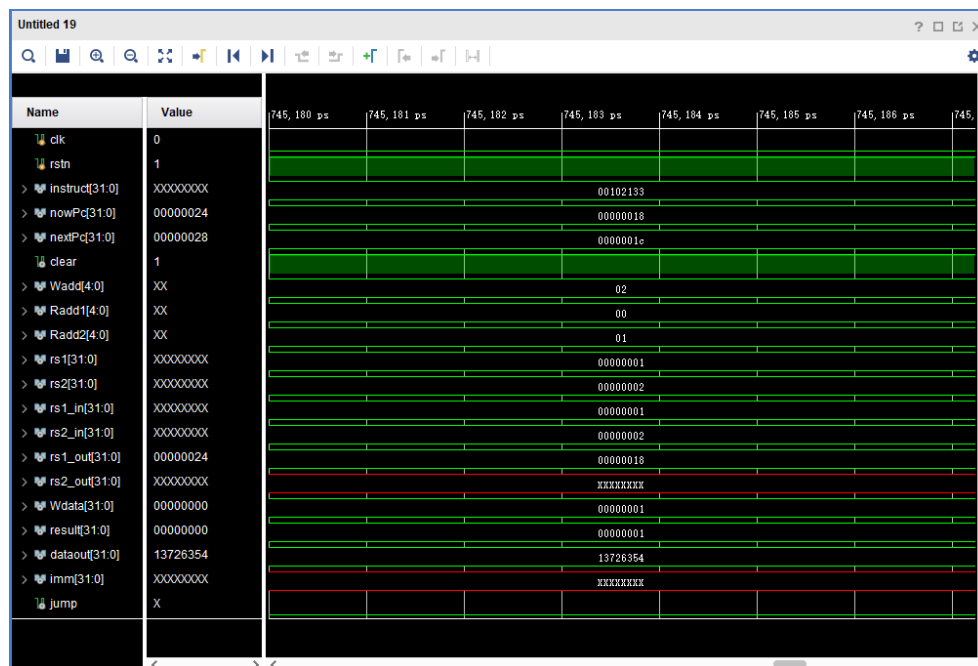
## 7. slt

十六进制: 00102133

解释:  $\text{if}(x[rs1] < x[rs2]) x[rd] \leq 1; \text{else } x[rd] \leq 0;$  (有符号)

测试结果:  $1 < 2$  res=1 (Wdata) 结果正确



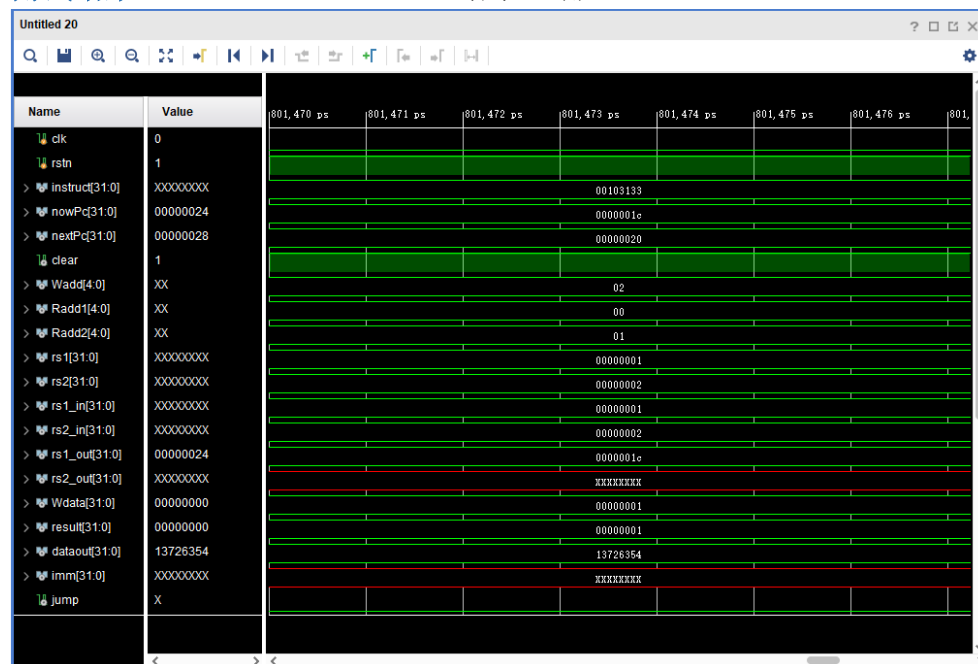


## 8. sltu

十六进制: 00103133

解释: if( $x[rs1] < x[rs2]$ )  $x[rd] \leq 1$ ; else  $x[rd] \leq 0$ ; (无符号)

测试结果:  $1 < 2$  res=1 (Wdata) 结果正确

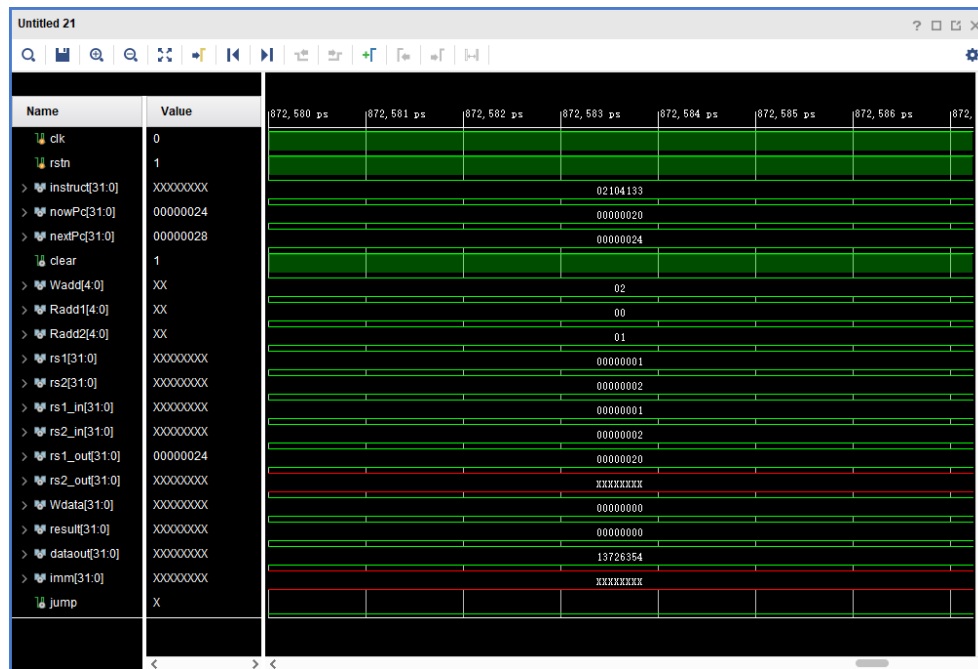


## 9. div

十六进制: 02104133

解释:  $x[rd] = x[rs1] / x[rs2]$  (有符号)

测试结果:  $1/2 = 0$  (Wdata) 结果正确

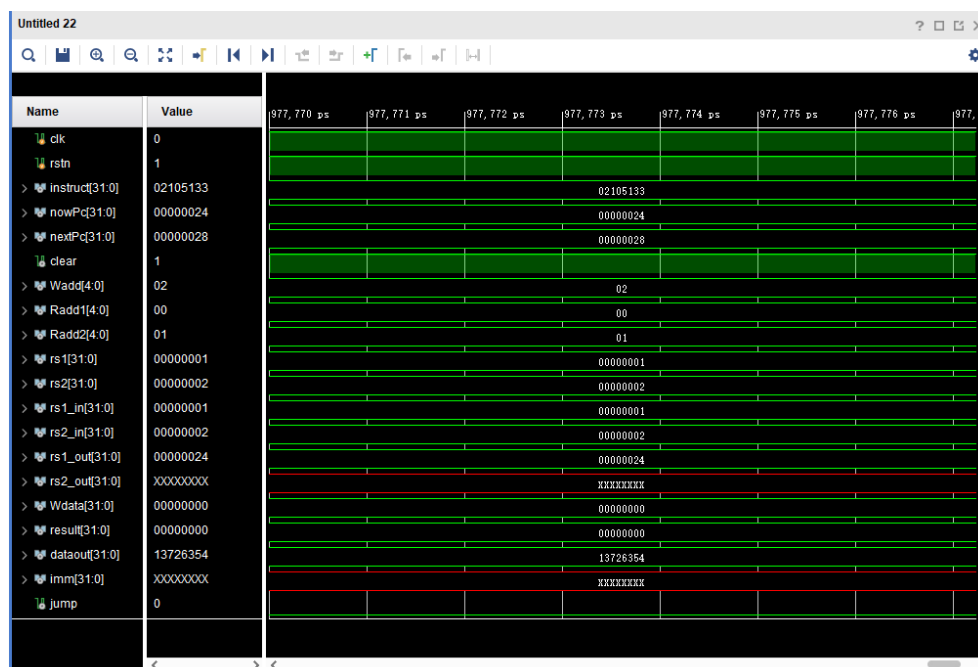


## 10. divu

十六进制: 02105133

解释:  $x[rd] = x[rs1] / x[rs2]$  (无符号)

测试结果:  $1/2=0$  (Wdata) 结果正确

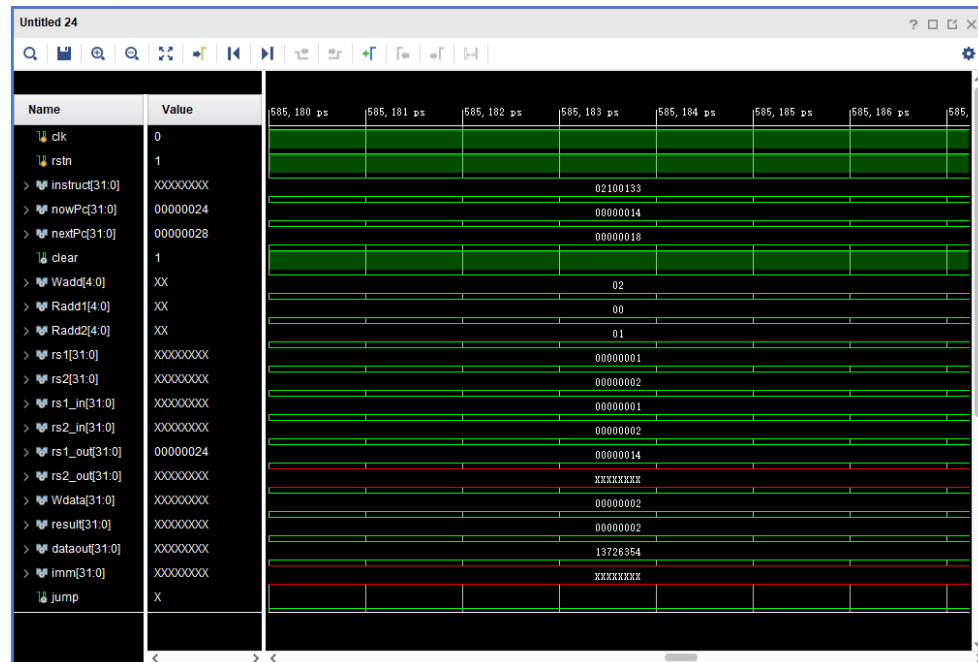


## 11. mul

十六进制: 02100133

解释:  $x[rd] = x[rs1] * x[rs2]$

测试结果:  $1*2=2$  (Wdata) 结果正确

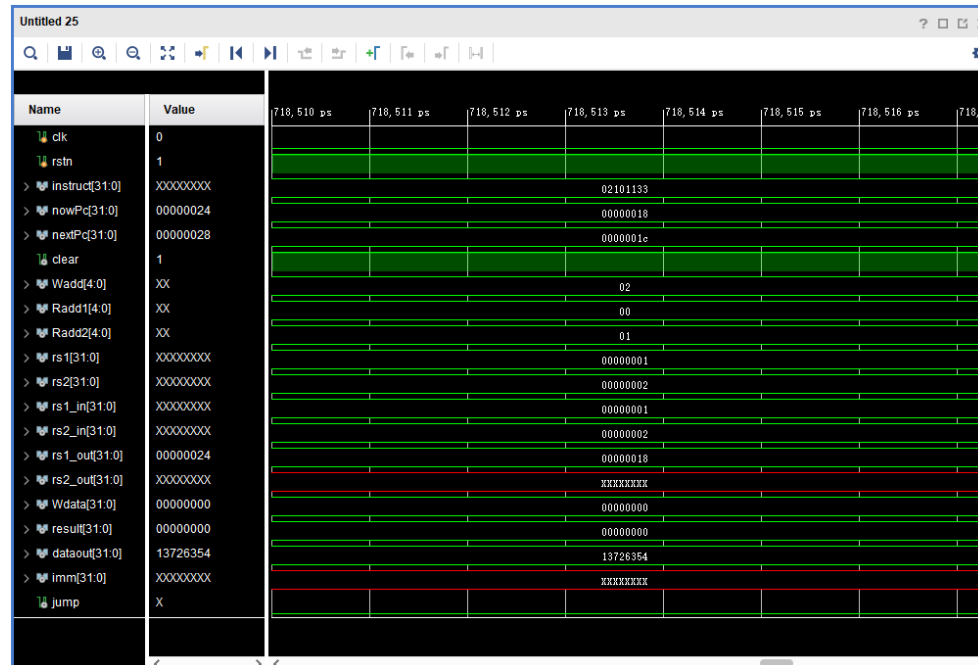


## 12. mulh

十六进制: 02101133

解释:  $x[rd] = (x[rs1] * x[rs2] \text{ (有符号)}) \gg 32 \text{ (有符号)}$

测试结果:  $(1*2) \gg 32 = 0$  (Wdata) 结果正确

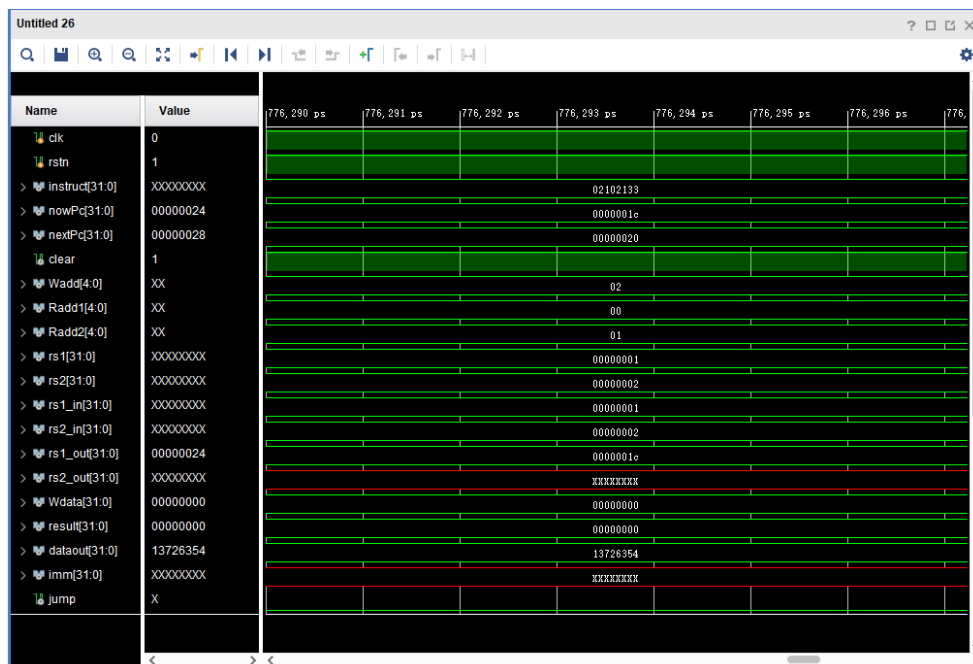


## 13. mulhsu

十六进制: 02102133

解释:  $x[rd] = (x[rs1] * x[rs2] \text{ (无符号)}) \gg 32 \text{ (有符号)}$

测试结果:  $(1*2) \gg 32 = 0$  (Wdata) 结果正确

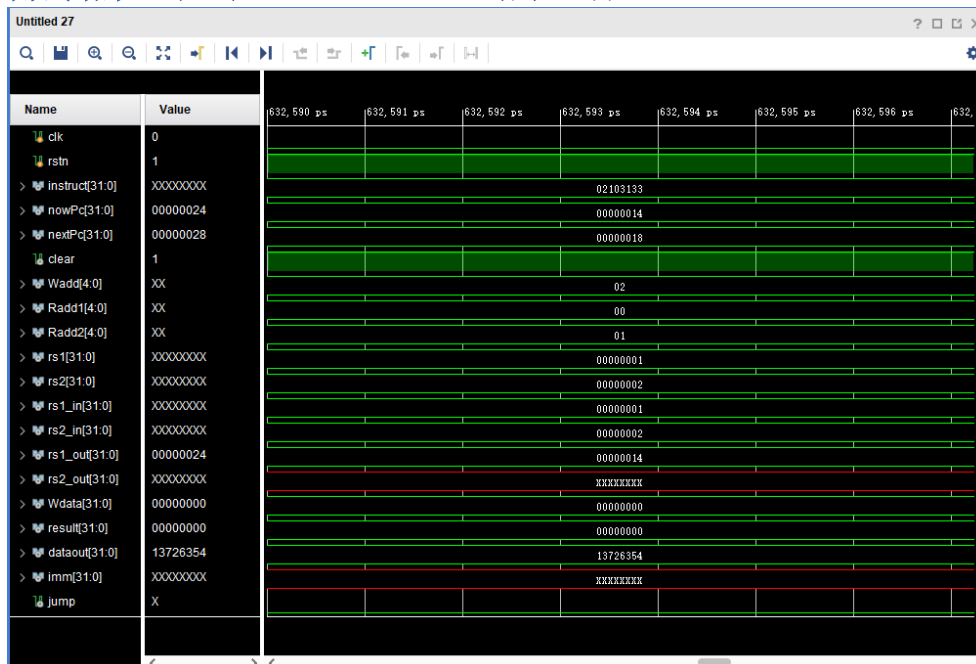


## 14. mulhu

十六进制: 02103133

解释:  $x[rd] = (x[rs1] * x[rs2] \text{ (无符号)}) \gg 32 \text{ (无符号)}$

测试结果:  $(1 * 2) \gg 32 = 0$  (Wdata) 结果正确

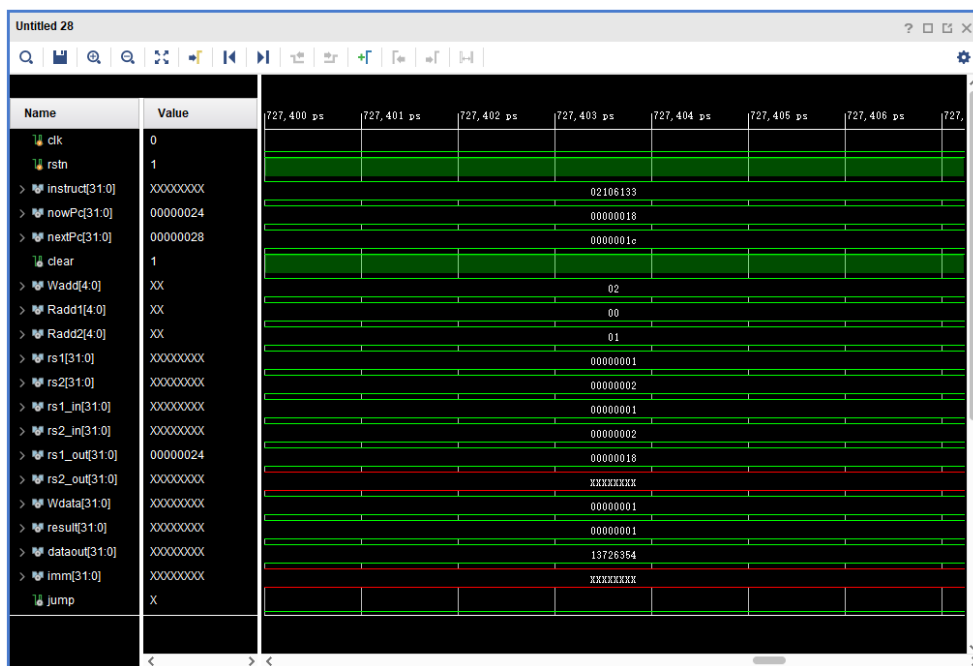


## 15. rem

十六进制: 02106133

解释:  $x[rd] = x[rs1] \% x[rs2] \text{ (有符号)}$

测试结果:  $1 \% 2 = 1$  (Wdata) 结果正确

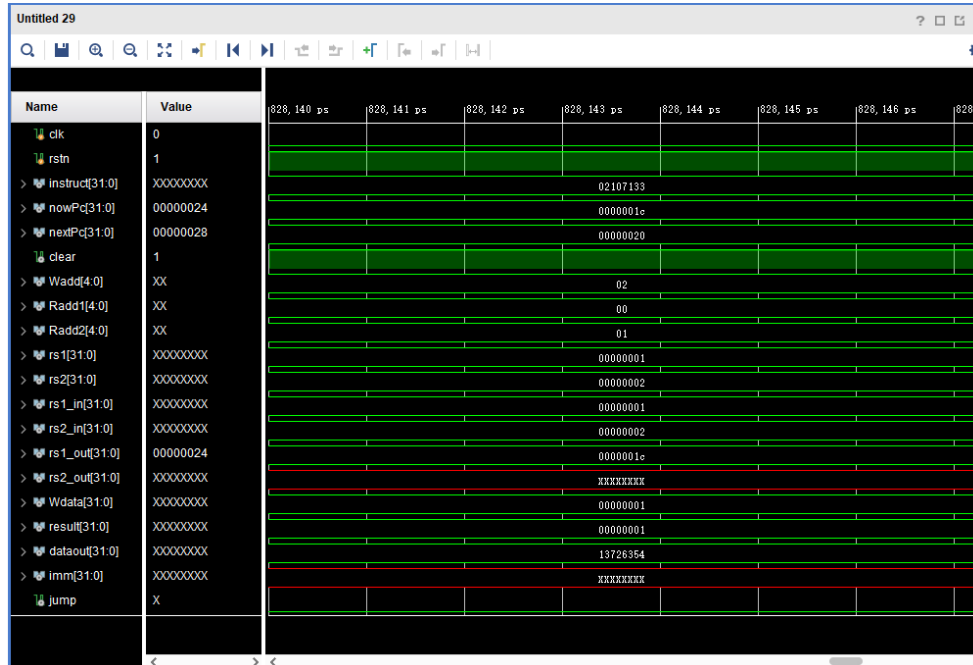


## 16. remu

十六进制: 02107133

解释:  $x[rd] = x[rs1] \% x[rs2]$  (无符号)

测试结果:  $1 \% 2 = 1$  (Wdata) 结果正确

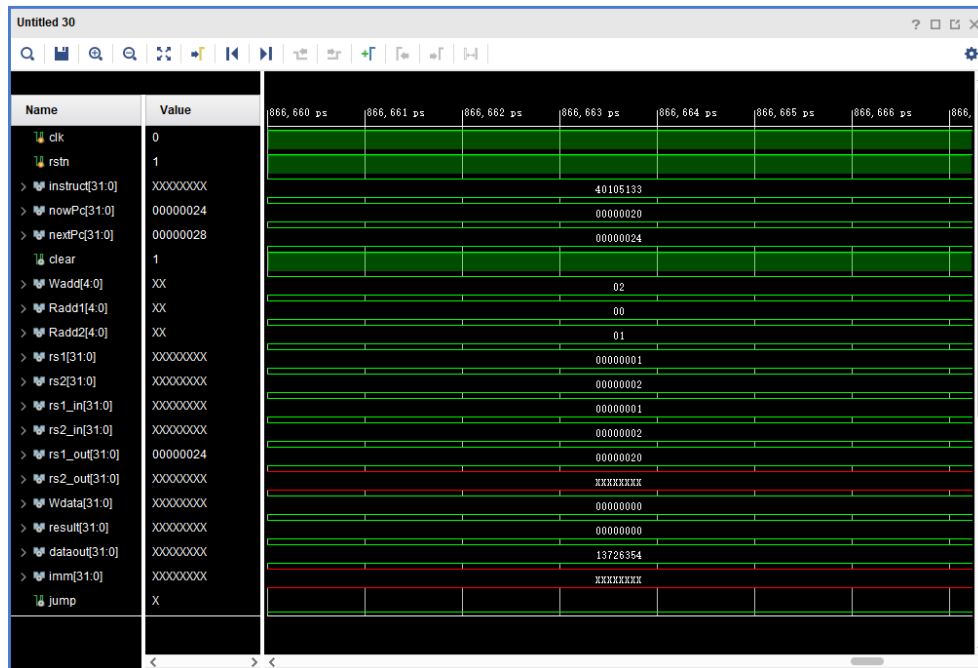


## 17. sra

十六进制: 40105133

解释:  $x[rd] = x[rs1] \gg x[rs2]$  (有符号)

测试结果:  $1 \gg 2 = 0$  (Wdata) 结果正确

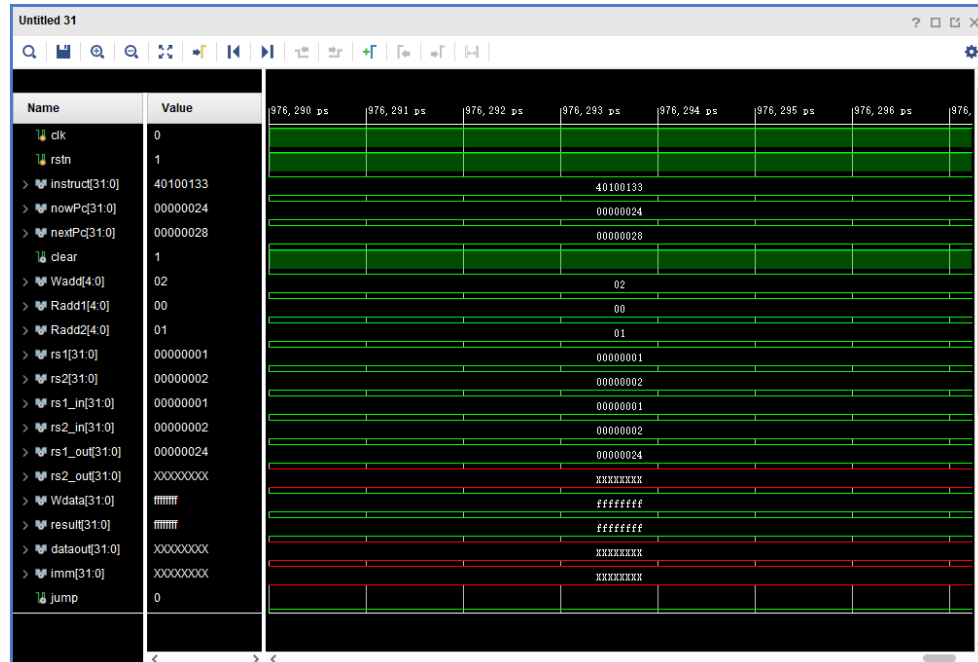


## 18. sub

十六进制: 40100133

解释:  $x[rd] = x[rs1] - x[rs2]$  (有符号)

测试结果:  $1 - 2 = \text{ffffffff}$  (-1 的补码) (Wdata) 结果正确



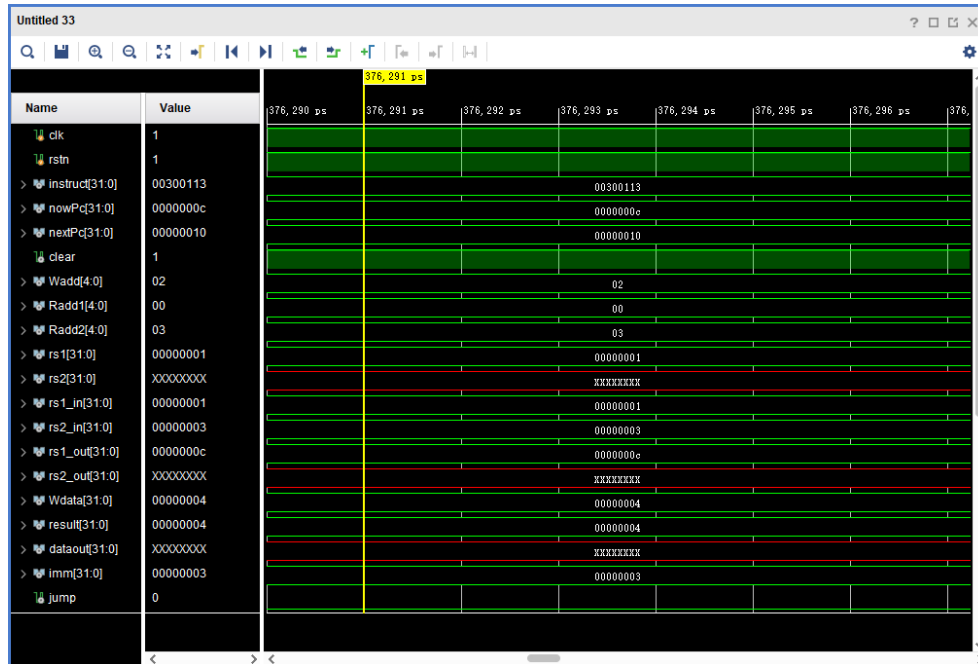
## 5.3 I 型指令测试结果

## 19. addi

十六进制: 00300113

解释:  $x[rd] = x[rs1] + \text{sext}(\text{immediate})$

测试结果: 1+3=4 (Wdata) 结果正确

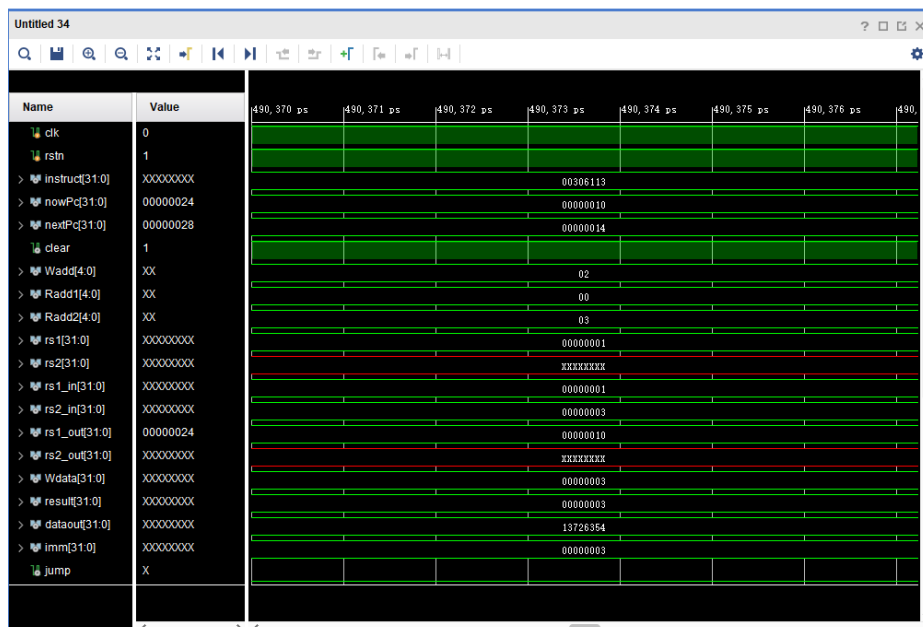


## 20. ori

十六进制: 00306113

解释:  $x[rd] = x[rs1] \mid \text{sext}(\text{immediate})$

测试结果: 1|3=3 (Wdata) 结果正确

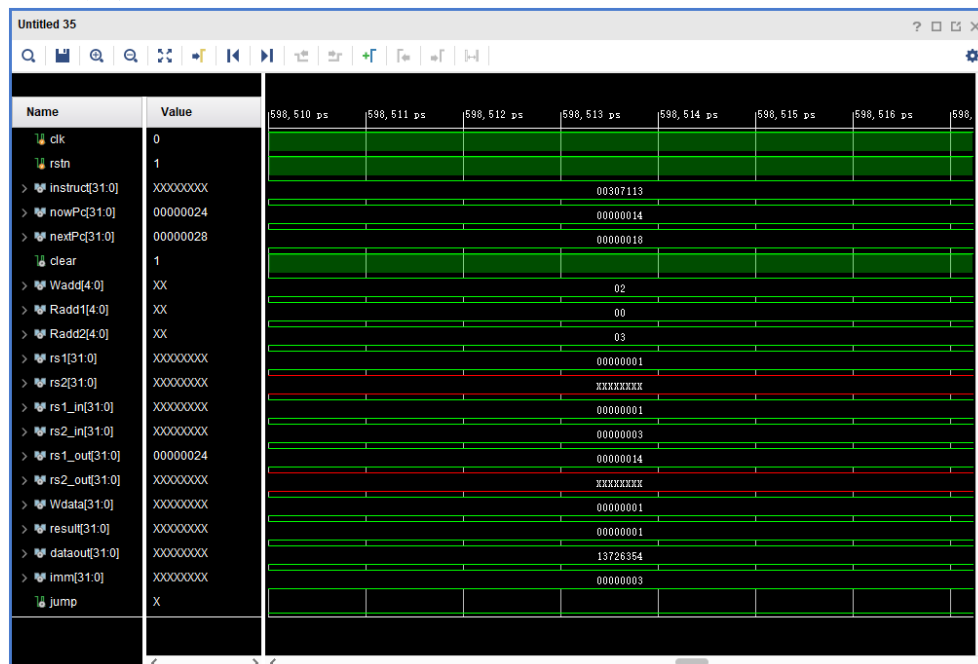


## 21. andi

十六进制: 00307113

解释:  $x[rd] = x[rs1] \& \text{sext}(\text{immediate})$

测试结果:  $1 \& 3 = 1$  (Wdata) 结果正确

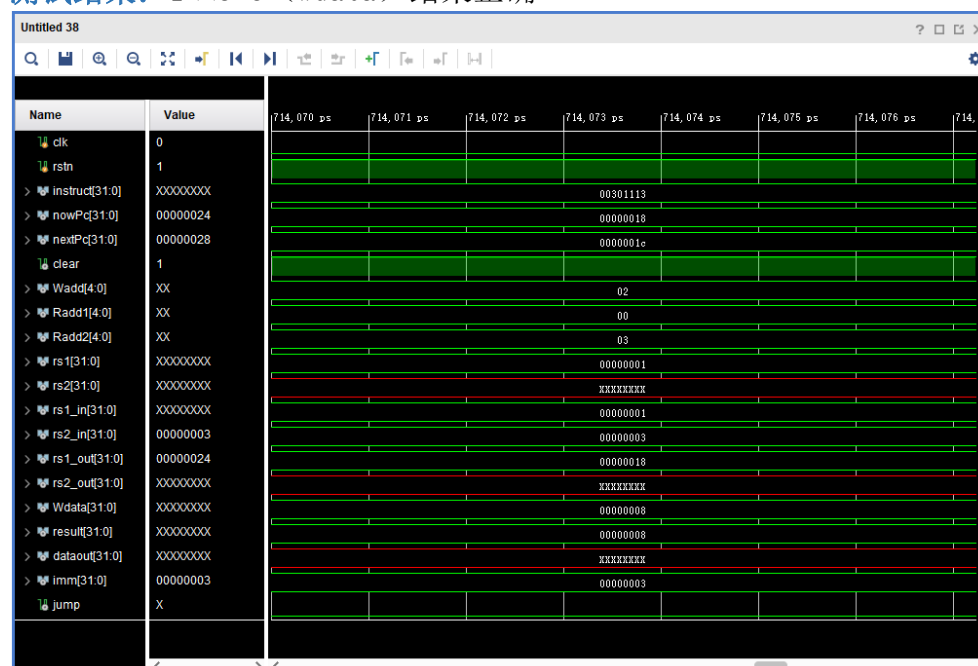


## 22. slli

十六进制: 00301113

解释:  $x[rd] = x[rs1] \ll \text{shamt}$

测试结果:  $1 \ll 3 = 8$  (Wdata) 结果正确



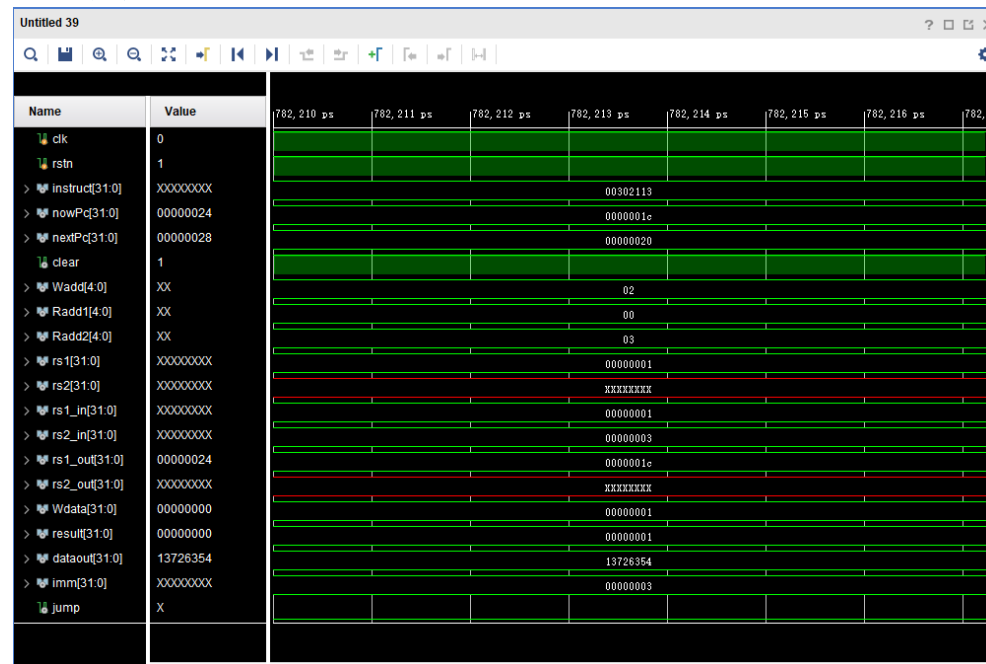
## 23. slti

十六进制: 00302113

解释:  $x[rd] = x[rs1] < \text{sext}(\text{immediate})$  (有符号) ? 1:0



测试结果:  $1 < 3$  rd=1 (Wdata) 结果正确

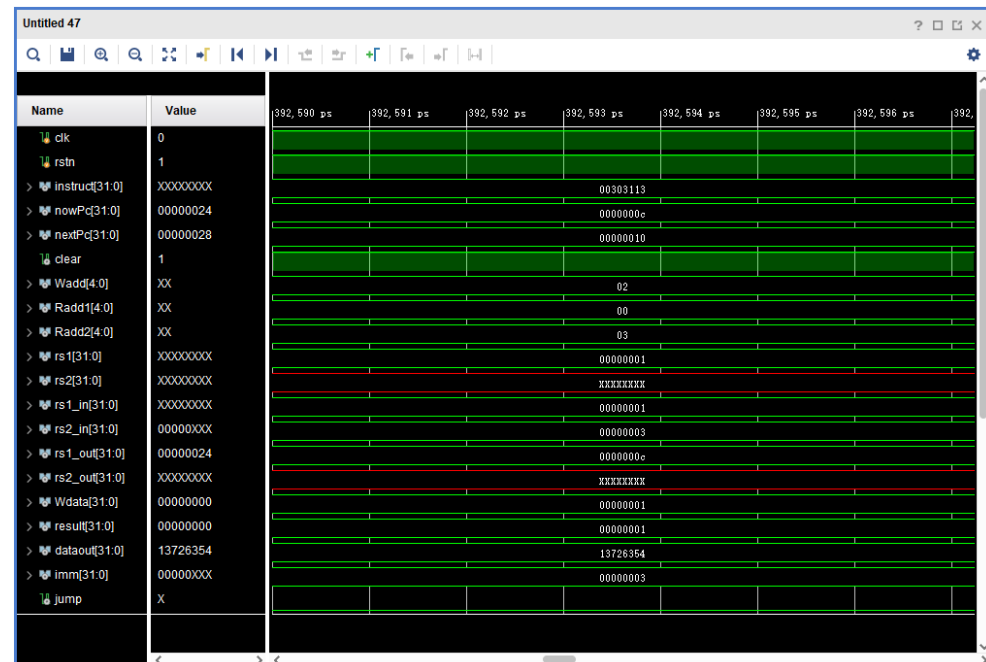


## 24. sltiu

十六进制: 00303113

解释:  $x[rd] = x[rs1] < \text{sext}(\text{immediate})$  (无符号) ? 1:0

测试结果:  $1 < 3$  rd=1 (Wdata) 结果正确

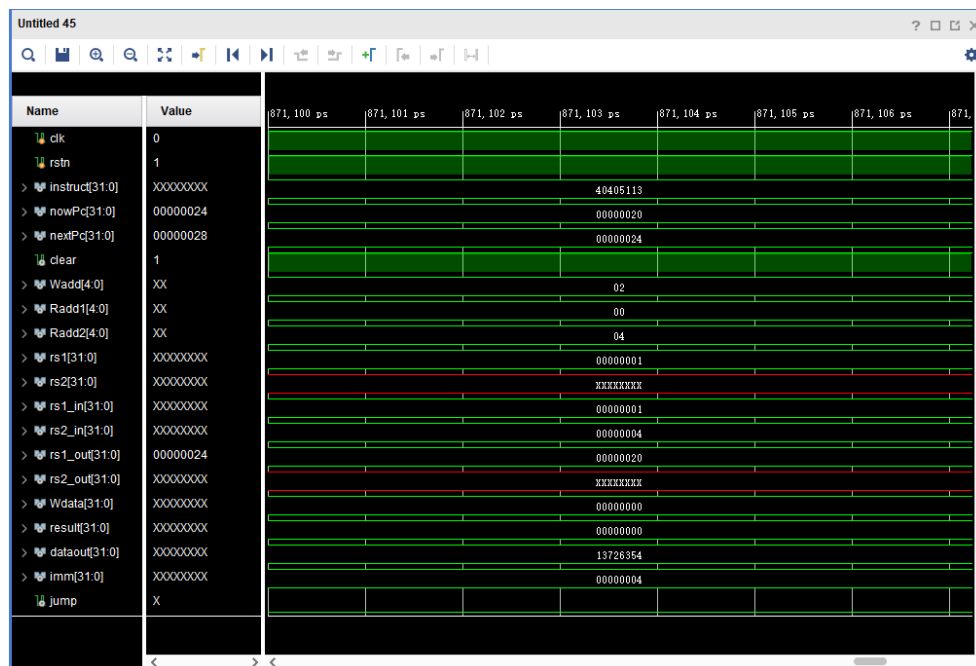


## 25. srai

十六进制: 40405113

解释:  $x[rd] = x[rs1] \gg \text{shamt}$  (有符号)高位用 rs1 的最高位填充

测试结果:  $1 \gg 3 = 0$  (Wdata) 结果正确

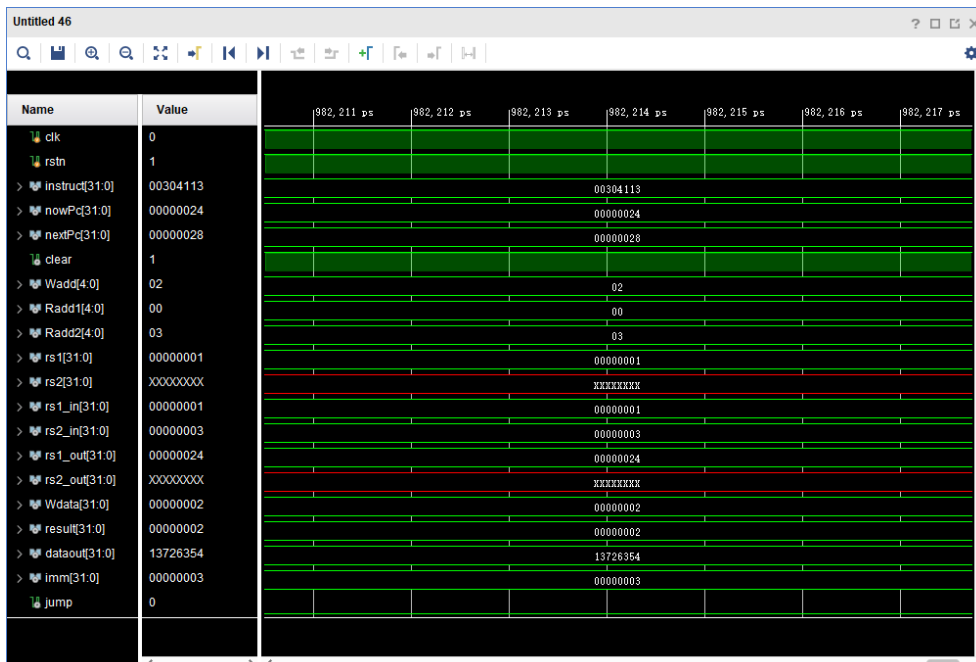


## 26. xori

十六进制: 00304113

解释:  $x[rd] = x[rs1] \hat{\text{sext}}(\text{immediate})$

测试结果:  $1 \hat{=} 3 = 2$  (Wdata) 结果正确

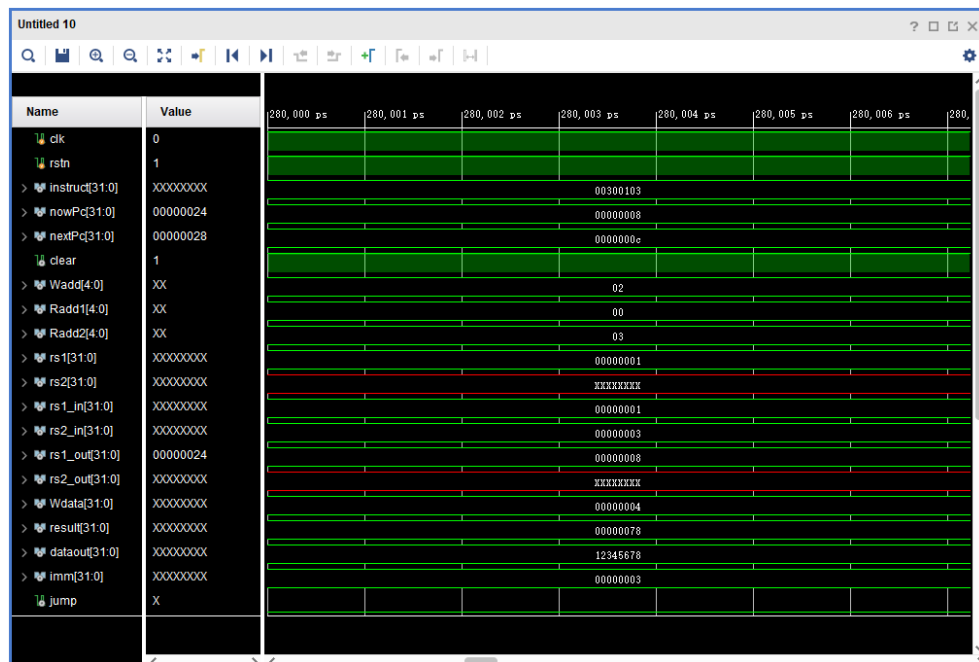


## 27. lb

十六进制: 00300103

解释:  $x[rd] = \text{mem}(x[rs1] + \text{sext}(\text{offset}) (\text{有符号})) [7:0]$

测试结果:  $\text{mem}(1+3)=\text{mem}(4)=12345678$  取八位为 78 (result) 结果正确

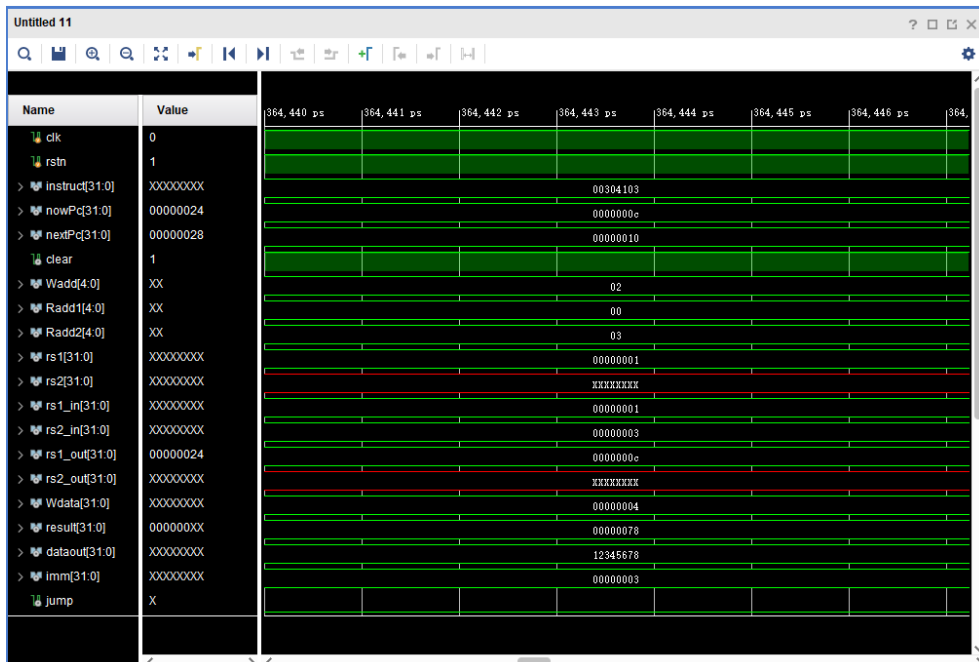


## 28. lbu

十六进制: 00304103

解释:  $x[\text{rd}] = \text{mem}(x[\text{rs1}] + \text{sext}(\text{offset}) \text{ (无符号)})[7:0]$

测试结果:  $\text{mem}(1+3)=\text{mem}(4)=12345678$  取八位为 78 (result) 结果正确

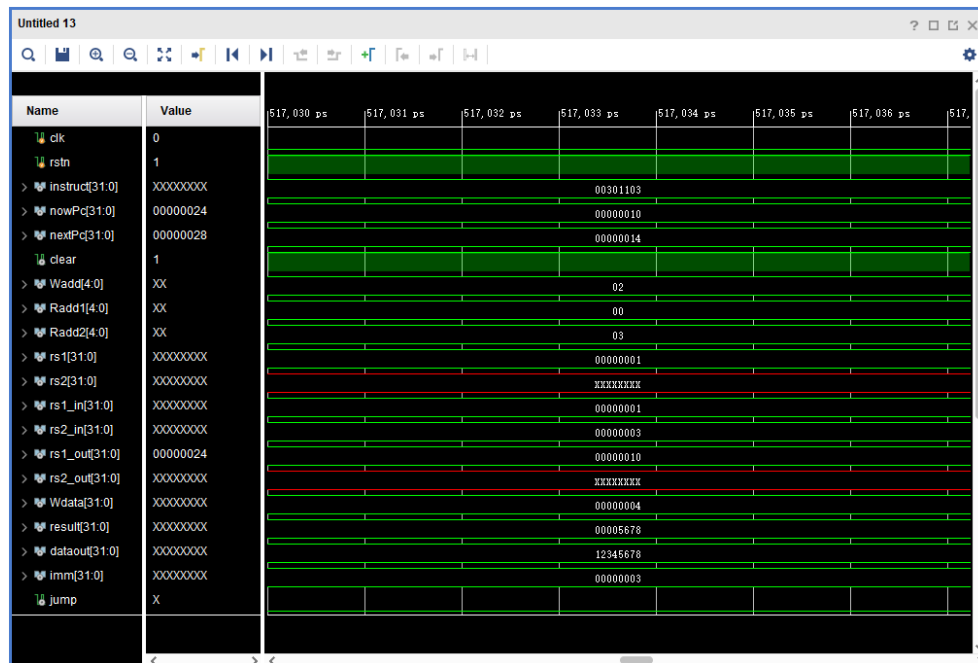


## 29. lh

十六进制: 00301103

解释:  $x[\text{rd}] = \text{mem}(x[\text{rs1}] + \text{sext}(\text{offset}) \text{ (有符号)})[15:0]$

测试结果:  $\text{mem}(1+3)=\text{mem}(4)=$ 第二个数为 12345678 取 16 位为 5678 (result)  
结果正确

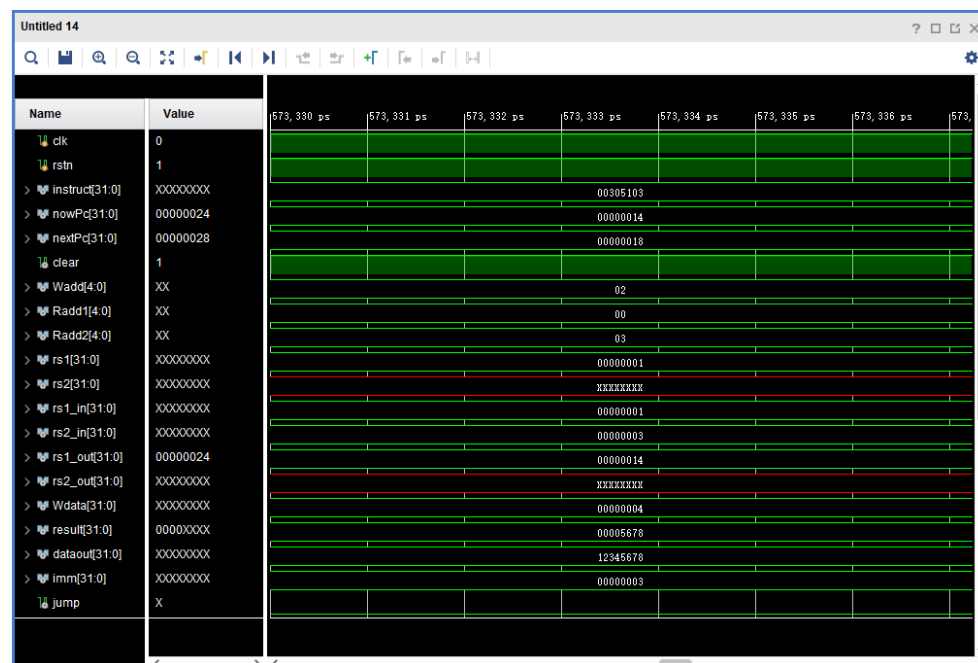


### 30. lhu

十六进制: 00305103

解释:  $x[\text{rd}] = x[\text{mem}](x[\text{rs1}] + \text{sext}(\text{offset})$  (无符号)) [15:0]

测试结果:  $\text{mem}(1+3)=\text{mem}(4)=$ 第二个数为 12345678 取 16 位为 5678 (result)  
结果正确

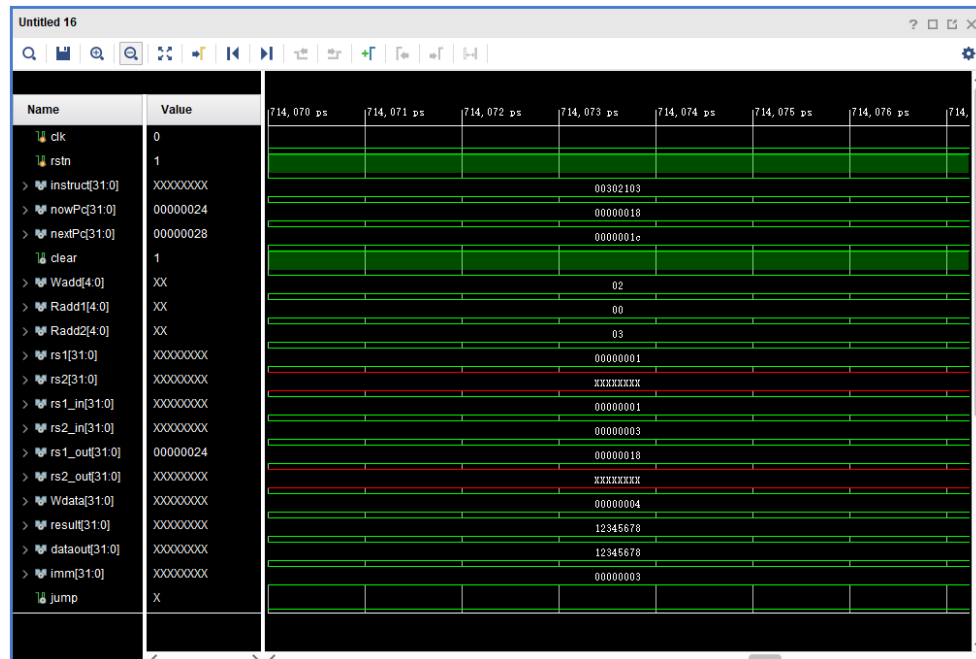


### 31. lw

十六进制: 00302103

解释:  $x[\text{rd}] = \text{mem}(x[\text{rs1}] + \text{sext}(\text{offset})$  (有符号)) [32:0]

**测试结果:**  $\text{mem}(1+3)=\text{mem}(4)=$ 第二个数为 12345678 取 32 位为 12345678(result)  
结果正确

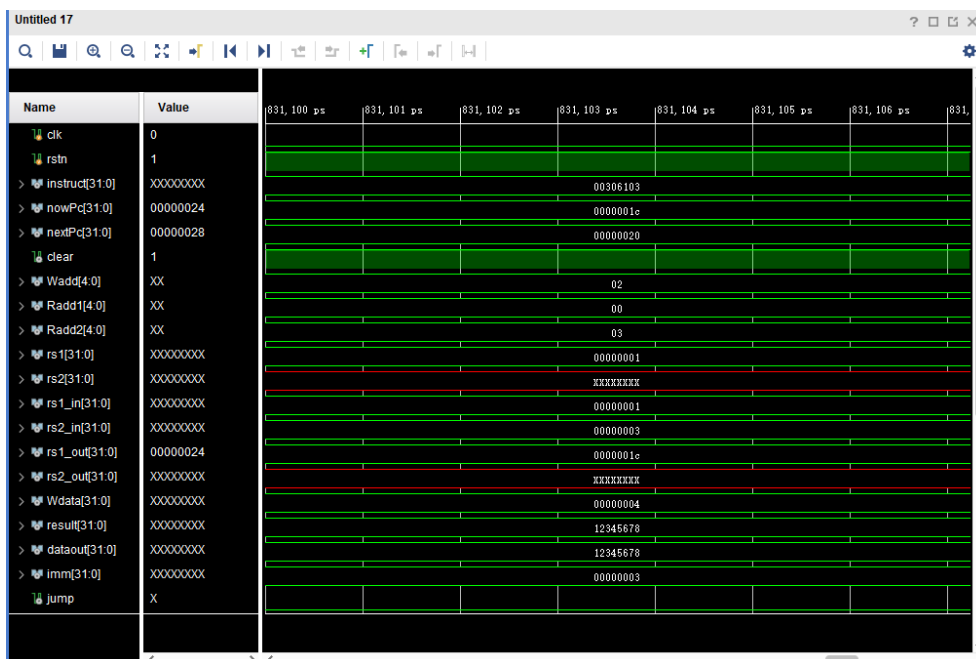


## 32. lwu

**十六进制:** 00306103

**解释:**  $x[\text{rd}] = \text{mem}(x[\text{rs1}] + \text{sext}(\text{offset}))$  (无符号) [32:0]

**测试结果:**  $\text{mem}(1+3)=\text{mem}(4)=$ 第二个数为 12345678 取 32 位为 12345678(result)  
结果正确

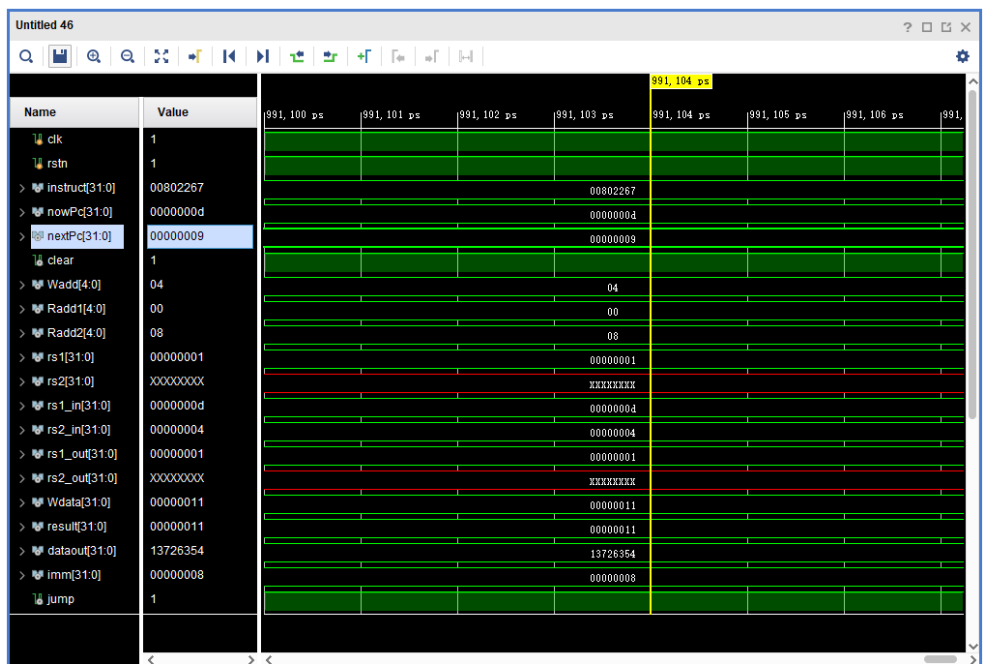


## 33. jalr

**十六进制:** 00802267

**解释:**  $x[\text{rd}]=\text{pc}+4, \text{pc}=(x[\text{rs1}] + \text{sext}(\text{offset}))$

测试结果:  $rd=d+4=11$ ,  $nextPc=(1+8)=9$  (十六进制) 结果正确



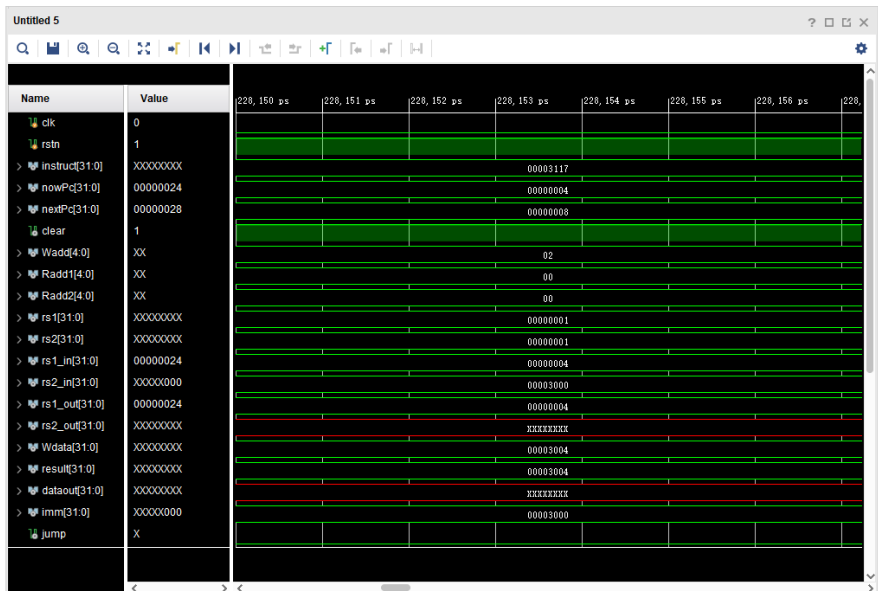
## 5.4 U 型指令测试结果

### 34. auipc

十六进制: 00003117

解释:  $x[rd]=pc+ sext(immediate)<<12$

测试结果:  $PC + 3<<12=3004$  (十六进制) (Wdata) 结果正确

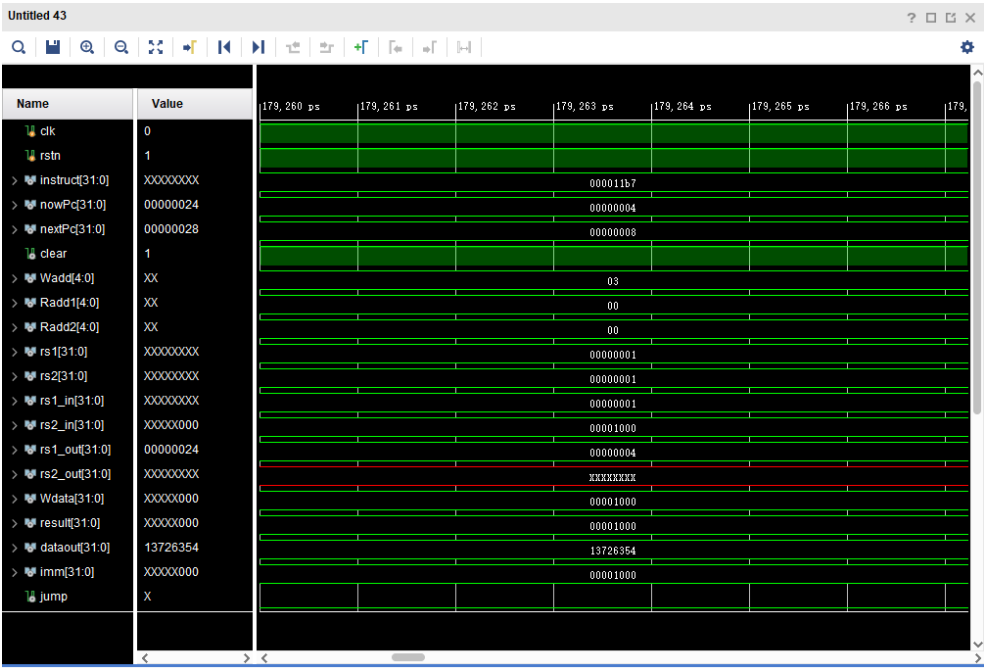


### 35. lui

十六进制: 000011b7

解释:  $x[rd] = sext(immediate)<<12$

测试结果： rd=1<<12=1000(十六进制) (Wdata) 结果正确



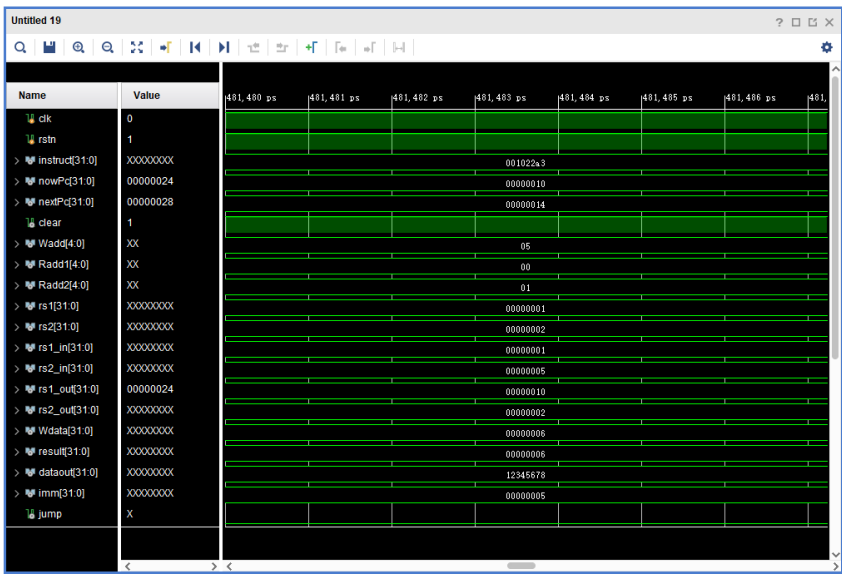
## 5.5 S 型指令测试结果

36. sw

十六进制：001022A3

解释：mem(x[rs1]+s sext(offset))=x[rs2][31:0]

测试结果：mem(1+5=6)=rs2=2 (rs2\_out) 结果正确

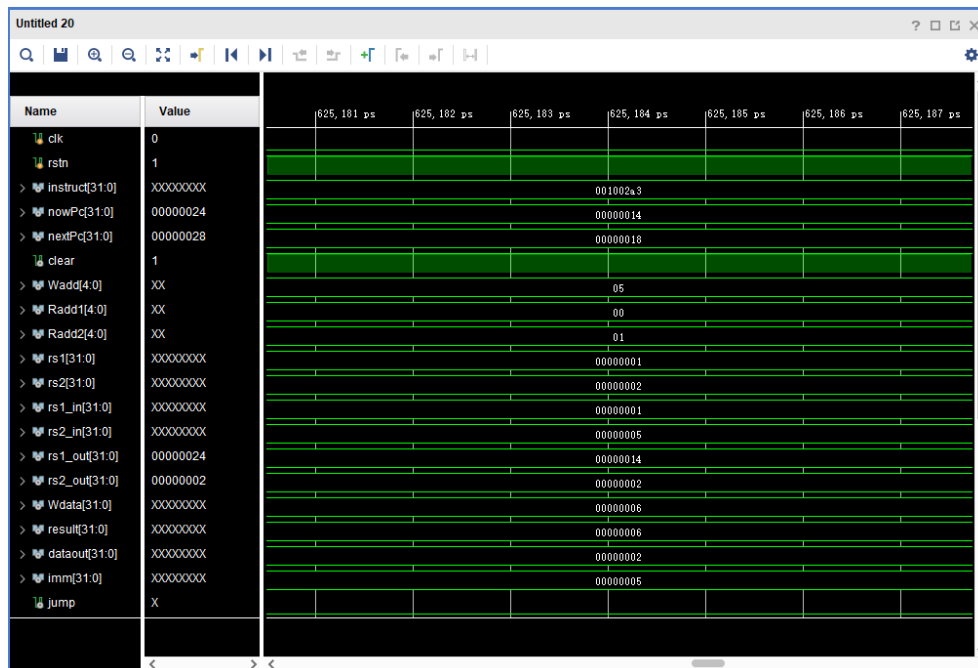


37. sb

十六进制：001002A3

解释：mem(x[rs1]+s sext(offset))=x[rs2][7:0]

测试结果:  $\text{mem}(1+5=6)=\text{rs2}=2$  (rs2\_out) 结果正确

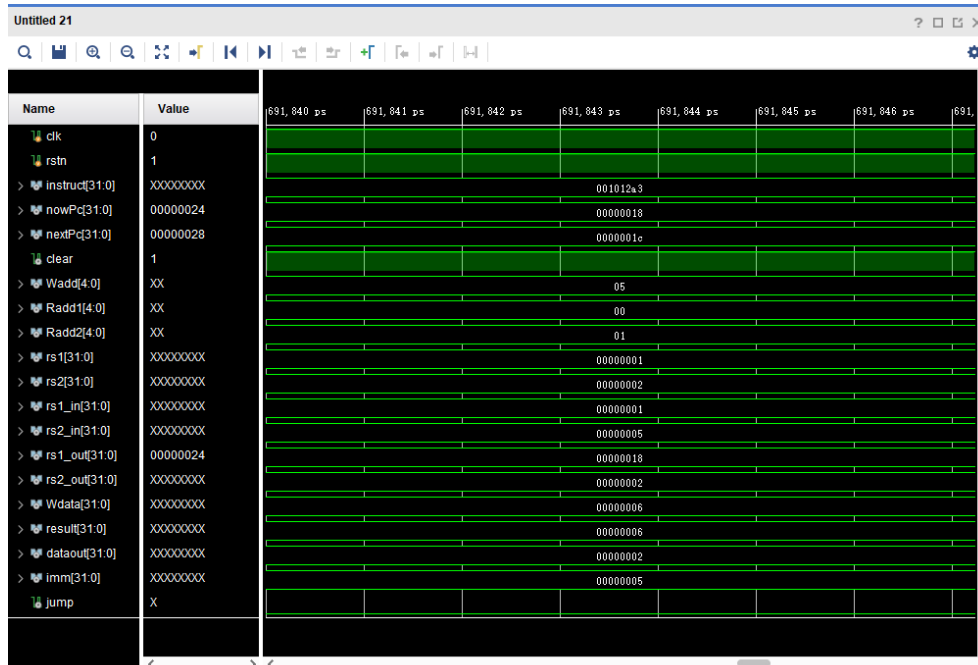


38. sh

十六进制: 001012A3

解释:  $\text{mem}(x[\text{rs1}]+\text{sxt}(\text{offset}))=x[\text{rs2}][15:0]$

测试结果:  $\text{mem}(1+5=6)=\text{rs2}=2$  (rs2\_out) 结果正确





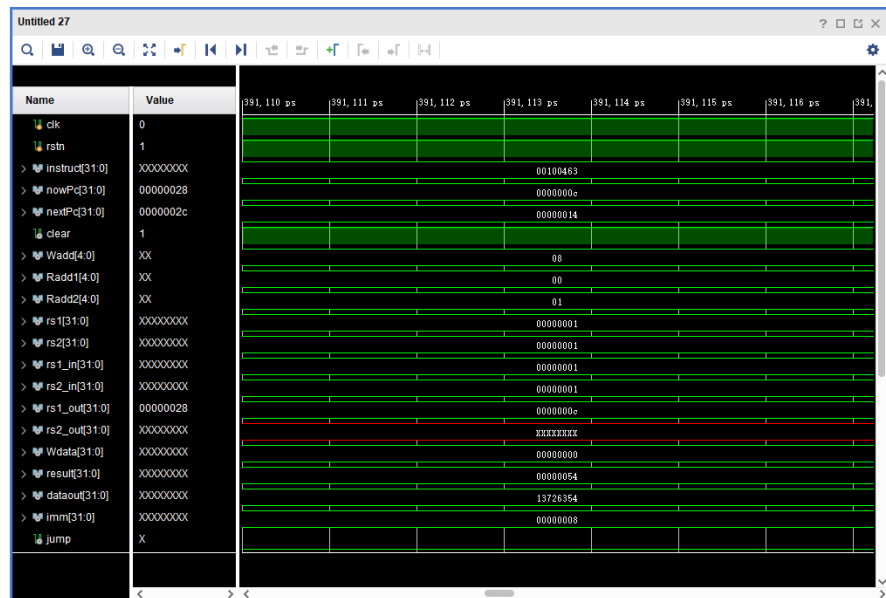
## 5.6 B 型指令测试结果

### 39. beq

十六进制: 00100463

解释: if( $x[rs1] == x[rs2]$ )  $pc += sext(offset)$

测试结果: nextPc=12+8=14(十六进制) 结果正确

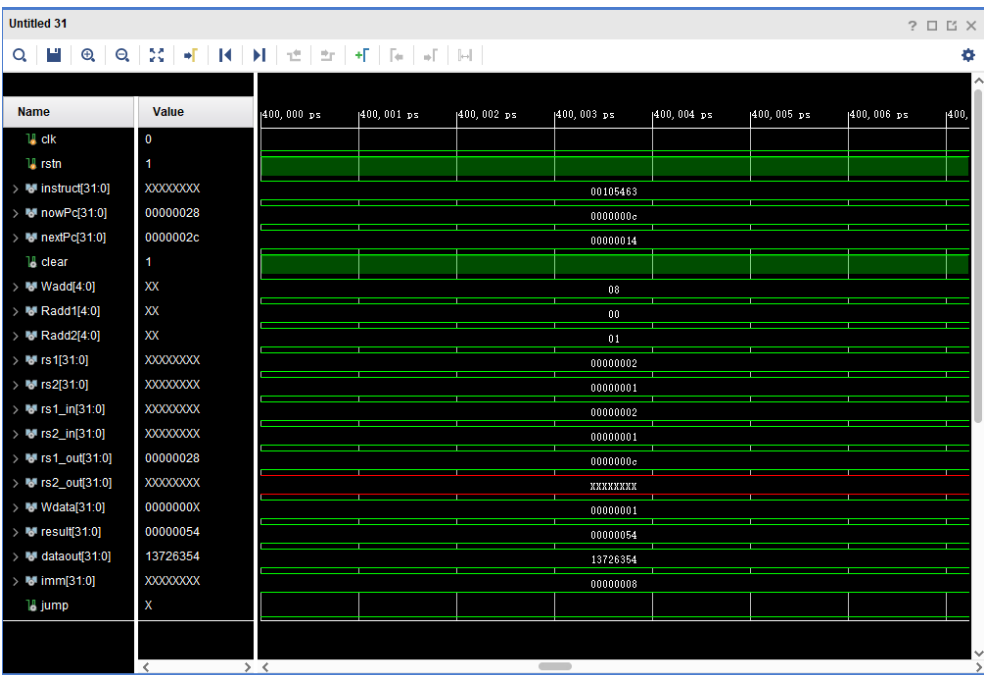


### 40. bge

十六进制: 00105463

解释: if( $x[rs1] \geq x[rs2]$  (有符号))  $pc += sext(offset)$

测试结果: nextPc=12+8=14(十六进制) 结果正确

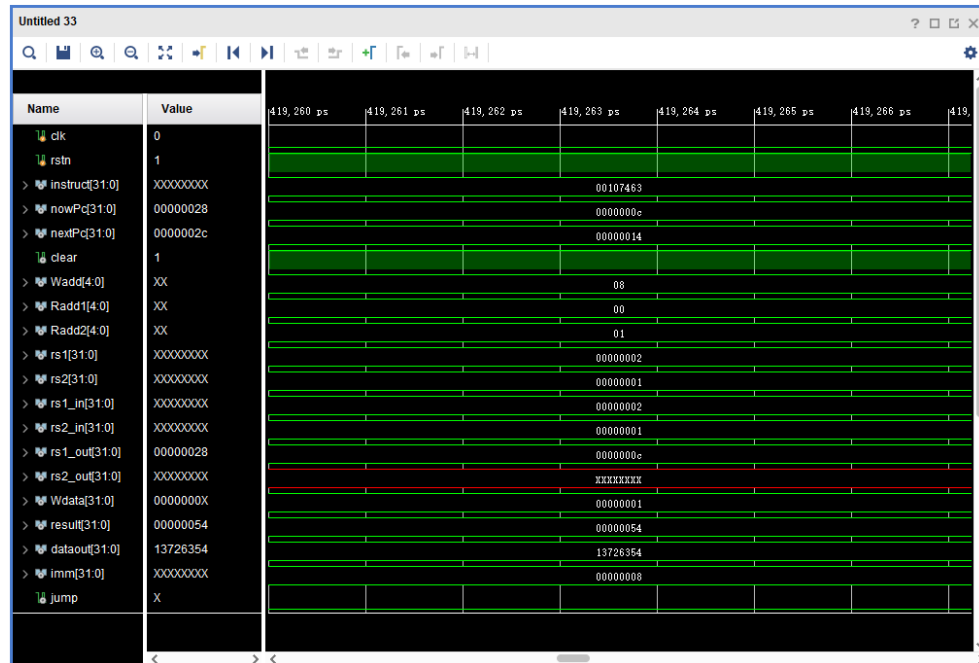


## 41. bgeu

十六进制: 00107463

解释: if( $x[rs1] \geq x[rs2]$  (无符号))  $pc += sext(offset)$

测试结果:  $nextPc = 12 + 8 = 14$  (十六进制) 结果正确

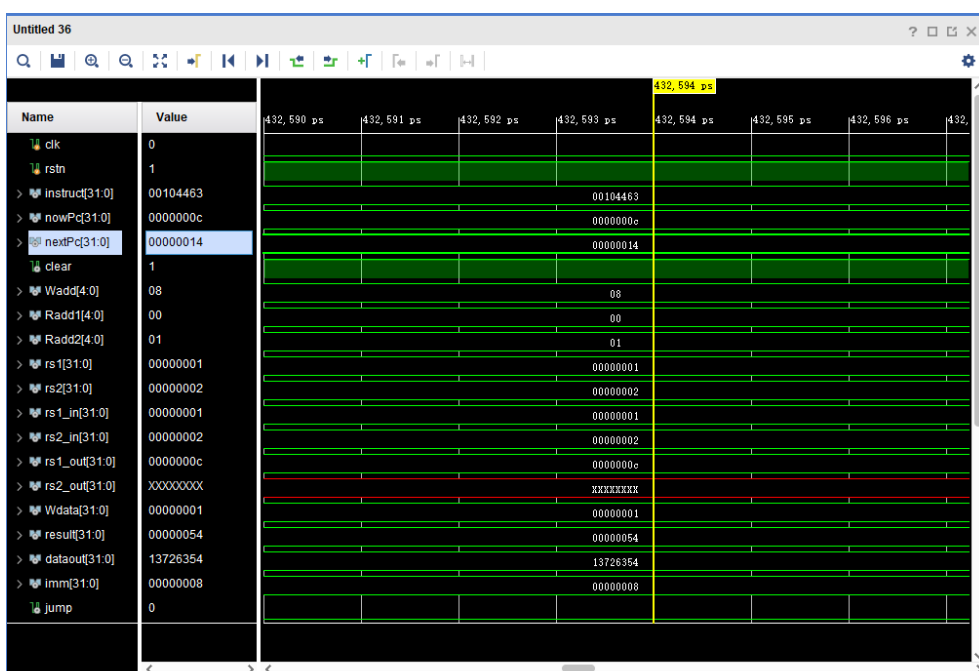


## 42. blt

十六进制: 00104463

解释: if( $x[rs1] < x[rs2]$  (有符号))  $pc += sext(offset)$

测试结果:  $nextPc = 12 + 8 = 14$  (十六进制) 结果正确

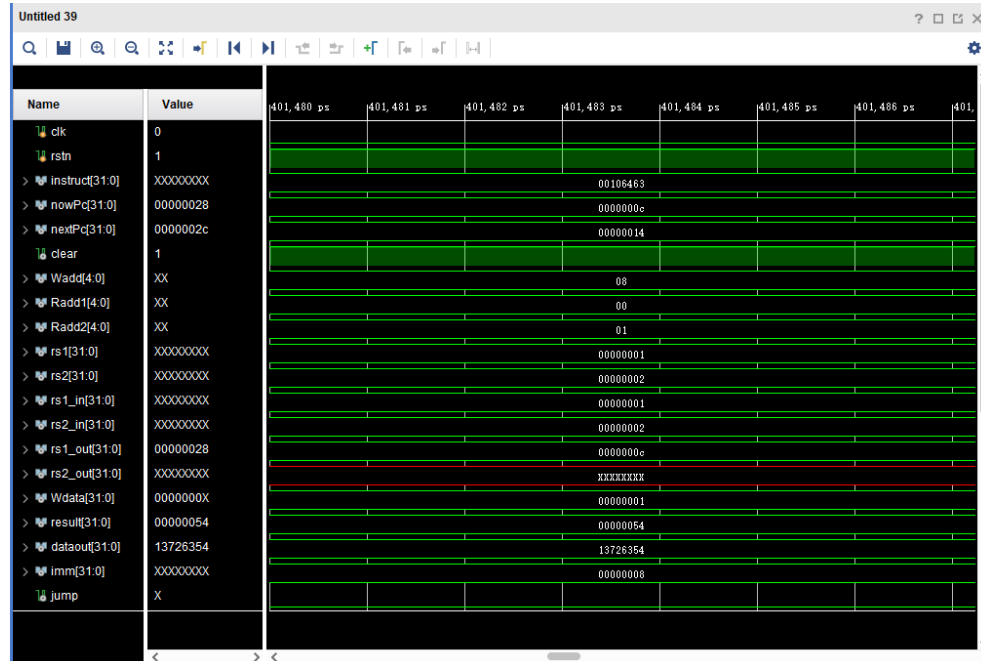


### 43. bltu

十六进制: 00106463

解释: if ( $x[rs1] < x[rs2]$  (无符号))  $pc += sext(offset)$

测试结果:  $nextPc = 12 + 8 = 14$  (十六进制) 结果正确

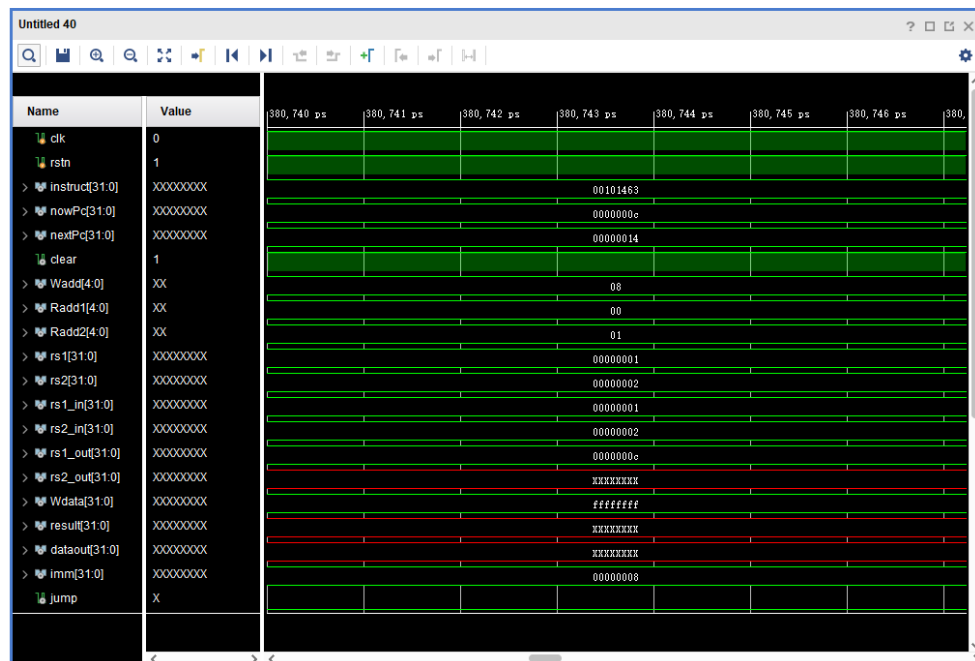


### 44. bne

十六进制: 00101463

解释: if ( $x[rs1] \neq x[rs2]$ )  $pc += sext(offset)$

测试结果:  $nextPc = 12 + 8 = 14$  (十六进制) 结果正确



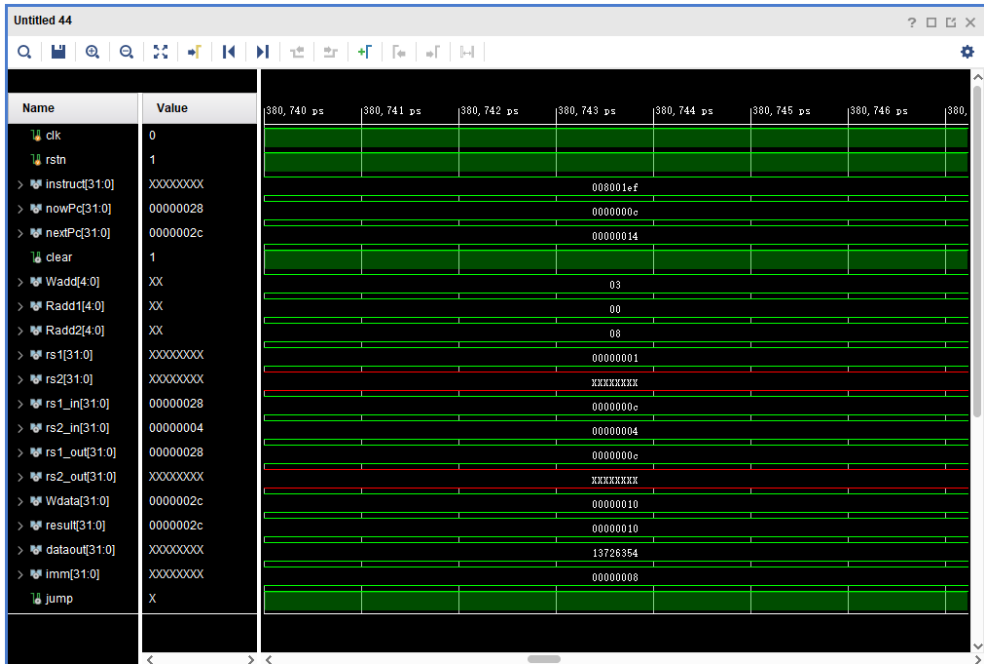
5.7 J 型指令测试结果

45. jal

十六进制: 008001EF

解释:  $x[rd]=pc+4, pc+=s\text{ext}(\text{offset})$

测试结果:  $rd=c+4=10, \text{nextPc}=12+8=14$  (十六进制) 结果正确



六、云平台运行二分查找算法

6.1 二分查找汇编代码及初始化

6.1.1 算法描述

在数组: 1 2 3 4 5 6 7 8 9 10 中用二分查找算法查找 3 所在下标的位置。如果运行正确, 结果应输出 2

## 6.1.2 RISC-V Assembly 插件撰写代码

```
1 #RISC-V二分查找
2 #M[0] ~ M[9] 存放十个数组数据
3 #x0存放0,x1寄存器存放start初始化0,x2存放end初始化9,x3存放middle初始化0,
4 #x4放2,x5放temp初始化0.x6存放Memory, 初始化0,x7放search (不初始化),x8放pc,初始化为0,x9放查找到的值的下标初值为0,x11初始化为4
5 lui x0,0 //00000000000000000000000000000000 00000037
6 lui x1,0 //00000000000000000000000000000000 000000b7
7 lw x2,x0 //00000001000000000000000000000000 02002103
8 lui x3,0 //00000000000000000000000000000000 00000187
9 lw x4,x0 //00000000010000000000000000000000 00402203
10 lui x5,0 //00000000000000000000000000000000 000002B7
11 lui x6,0 //00000000000000000000000000000000 00000337
12 lw x7,x0 //00000001000000000000000000000000 00802383
13 lui x8,0 //00000000000000000000000000000000 00000437
14 lui x9,0 //00000000000000000000000000000000 00000487
15 lui x10,0 //00000000000000000000000000000000 00000537
16 lw x11,x0 //00000001100000000000000000000000 00c02583
17 add x9,x0,x0 #初始化x9为0 PC = 0 //00000000000000000000000000000000 00010110011
18 add x5,x1,x2 #temp = start + end PC = 4 //000000000001000100000001010110011
19 div x3,x5,x4 #middle = temp / 2 PC = 8 //000000010010000101100000110110011
20 mul x3,x3,x11 #middle = middle * 4 PC =12 //000000010101100011000000110110011
21 lw x6,x0,x3 #x6 = M[0*x3] PC =16 //000000000000000001010001100000011
22 beq x7,x6 #search == M[middle]? PC =20 //00000001001100110000001001100011 //相等: PC + 36
23 #不相等:
24 #如果M[middle] > search: end = middle ;
25 bge x6,x7,OFFSET # M[middle] > search,(PC+8),PC = 24 //0000000001100110101010001100011 8: 0 0000 0000 1000
26 blt x6,x7,OFFSET # M[middle] < search,(PC+16),PC = 28 //0000000001100110100100001100011
27 add x2,x3,x0 #end = middle,PC = 32 //00000000000110000000000000000000 000110011
28 div x2,x2,x11 #PC = 36 //000000010101100010100000100110011
29 jal x8,OFFSET #PC = 40 跳到PC=40-36 //1111110111011111111111010001101111
30 add x1,x3,x0 #start = middle,PC = 44 //00000000011000000000000000000000 000110011
31 div x1,x1,x11 #PC = 48 //00000001010110000110000000000000 000110011
32 jal x8,OFFSET #PC = 52 跳到PC=48-48 //11111101000111111111010001101111
33 #程序结束标志:
34 add x9,x3,x0 #PC = 56 //00000000011000000000000000000000 000110011
35 div x9,x9,x11 #PC = 60 //000000010101101001100010010110011
36
```

具体代码如下:

```
lui x0,0
lui x1,0
lw x2,x0
lui x3,0
lw x4,x0
lui x5,0
lui x6,0
lw x7,x0
lui x8,0
lui x9,0
lui x10,0
lw x11,x0
add x9,x0,x0
add x5,x1,x2
div x3,x5,x4
mul x3,x3,x11
lw x6,x0,x3
beq x7,x6
bge x6,x7,OFFSET
blt x6,x7,OFFSET
add x2,x3,x0
div x2,x2,x11
jal x8,OFFSET
add x1,x3,x0
```

```
div x1,x1,x11
jal x8,OFFSET -
add x9,x3,x0
div x9,x9,x11
```

### 6.1.3 翻译成十六进制指令

将上述代码翻译成十六进制指令，并初始化指令寄存器：

```
instMem_text[0]<=32'h00000037;
instMem_text[1]<=32'h000000b7;
instMem_text[2]<=32'h02002103;
instMem_text[3]<=32'h000001B7;
instMem_text[4]<=32'h00402203;
instMem_text[5]<=32'h000002B7;
instMem_text[6]<=32'h00000337;
instMem_text[7]<=32'h00802383;
instMem_text[8]<=32'h00000437;
instMem_text[9]<=32'h000004B7;
instMem_text[10]<=32'h00000537;
instMem_text[11]<=32'h00C02583;
instMem_text[12]<=32'h000004B3;
instMem_text[13]<=32'h001102B3;
instMem_text[14]<=32'h0242C1B3;
instMem_text[15]<=32'h02B181B3;
instMem_text[16]<=32'h0001A303;
instMem_text[17]<=32'h02730263;
instMem_text[18]<=32'h00735463;
instMem_text[19]<=32'h00734863;
instMem_text[20]<=32'h00300133;
instMem_text[21]<=32'h02B14133;
instMem_text[22]<=32'hFDDFF46F;
instMem_text[23]<=32'h003000B3;
instMem_text[24]<=32'h02B0C0B3;
instMem_text[25]<=32'hFD1FF46F;
instMem_text[26]<=32'h003004B3;
instMem_text[27]<=32'h02B4C4B3;
```

### 6.1.4 初始化存储器

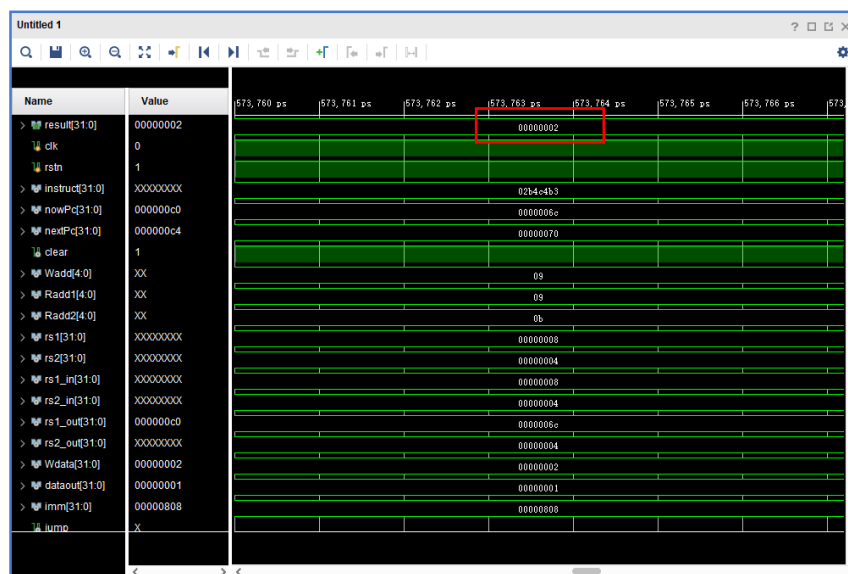
```
mem[0]<=32'd1;
mem[1]<=32'd2;
mem[2]<=32'd3;
```

```

mem[3]<=32'd4;
mem[4]<=32'd5;
mem[5]<=32'd6;
mem[6]<=32'd7;
mem[7]<=32'd8;
mem[8]<=32'd9;
mem[9]<=32'd10;

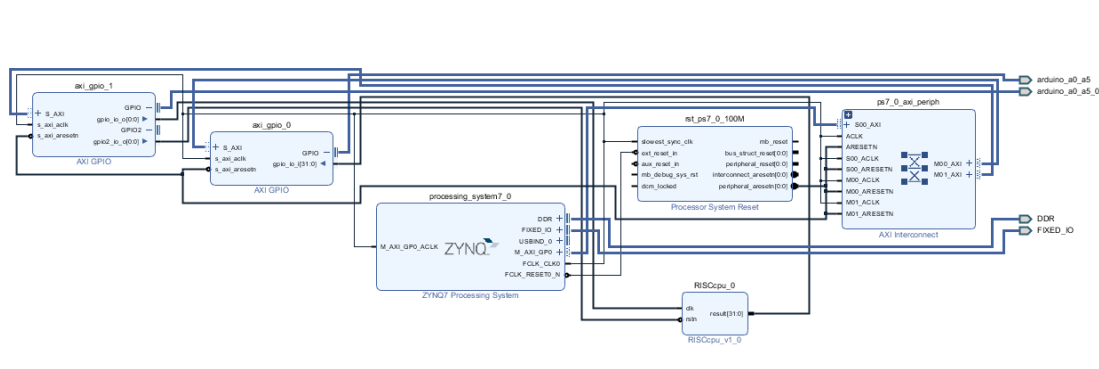
```

## 6.2 行为仿真测试

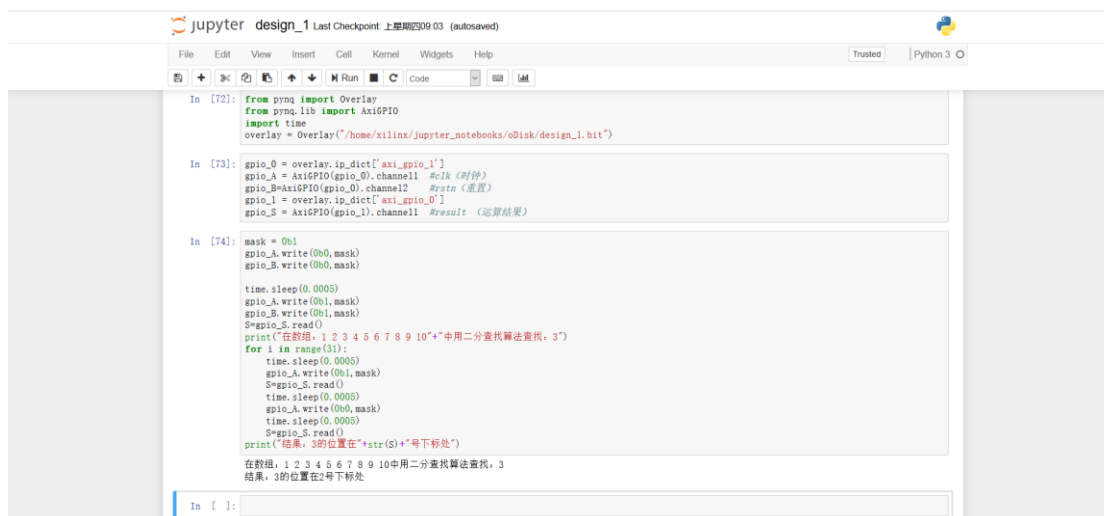


其中 result 结果为 2，结果正确。

## 6.3 各部件连线图



## 6.4 平台测试结果



```

In [72]: from pyq import Overlay
         from pyq.lib import AxiGPIO
         import time
         overlay = Overlay("/home/xilinx/jupyter_notebooks/oDisk/design_1.bit")

In [73]: gpio_0 = overlay.ip_dict['axi_gpio_1']
         gpio_A = AxiGPIO(gpio_0).channel1 #clk (时钟)
         gpio_B=AxiGPIO(gpio_0).channel2  #rstn (重置)
         gpio_1 = overlay.ip_dict['axi_gpio_0']
         gpio_S = AxiGPIO(gpio_1).channel1 #result (运算结果)

In [74]: mask = 0b1
         gpio_A.write(0b0,mask)
         gpio_B.write(0b0,mask)

         time.sleep(0.0005)
         gpio_A.write(0b1,mask)
         gpio_B.write(0b1,mask)
         S=gpio_S.read()
         print("在数组: 1 2 3 4 5 6 7 8 9 10中用二分查找算法查找, 3")
         for i in range(31):
             time.sleep(0.0005)
             gpio_A.write(0b1,mask)
             S=gpio_S.read()
             time.sleep(0.0005)
             gpio_A.write(0b0,mask)
             time.sleep(0.0005)
             S=gpio_S.read()
         print("结果: 3的位置在"+str(S)+"号下标处")
         在数组: 1 2 3 4 5 6 7 8 9 10中用二分查找算法查找, 3
         结果: 3的位置在2号下标处

```

结果分析:

平台运行结果与上述行为仿真结果一致，此 CPU 运行二分查找算法成功！

## 七、心得体会

### 7.1 翁诗浩心得:

这次实验花了许多时间，小组也一起熬了很多夜，主要还是因为要从头开始学一些东西，比如 Verilog 编程，Vivado 封装线路设计，Python 测试代码撰写。

但是，收获是非常多的，经过这次实验，我了解了如何用 VHDL 设计一个硬件，并且亲身实践地完成了，中途遇到好多好多困难，有时候真的会脾气非常暴躁，但是当我做完了整个 CPU，并且在云平台运行测试二分算法成功，那种成就感是非常强的。

经过本次实验，然我进一步理解了 RISC-V 架构的 CPU 的数据通路，以及各类控制信号，还有基于 RISC-V 架构的 CPU 指令的含义，当然还有 Verilog 硬件描述语言该如何写，它是并行执行的一种语言，与我们传统的 C++有着本质区别，与其说是一门语言，更像是一种硬件设计的描述手段。

当然，这次实验还是有些遗憾的，比如没有像真正设计 CPU 一样，把寄存器地址进行划分，而是做了通用处理，也没有能够挑战一下流水线的设计。



## 7.2 梁奉迪心得:

在本次实验中,通过对译码器的编写对 CPU 整体架构以及数据通路和指令格式有了非常深刻的认识。并掌握了 Verilog 的书写。人工实现了汇编的编写、汇编到二进制代码的实现。

但还存在不足之处:译码编写比较混乱,MUX 选取较多,容易遗漏控制信号。从一开始的什么也不会跌跌撞撞一步步完成,还是比较有成功感和收获的。

## 7.3 韩梅心得:

在单周期 cpu 设计的整个实验中,我对于 cpu 的实现过程和整体架构有了更进一步的体会和更全面的了解,之前听完课对于讲的知识还有些茫然,而这个实验就是对于课堂上学到知识的一次很好的应用和实践,有些课堂上不懂的地方在实验的过程中都有了更好的理解。

当然也遇到了非常多的困难,像云平台测试、汇编代码的运行等都是经过很多次的尝试才成功,不过我从中感受到了合作的力量,提高了我分工合作的意识。

## 八、参考文献

- [1]David Patterson, Andrew Waterman. RISC-V 手册[M]. 加州大学伯克利分校:加州伯克利, 2017:9.
- [2]thundersnark. E203 蜂鸟 RISC-V 处理器代码阅读笔记 之指令译码模块 e203\_exu\_decode.v[EB/OL]. <https://blog.csdn.net/thundersnark/article/details/105879902>, 2020-05-01.
- [3]南工小王子. Vivado 加上 VsCode 让你的生活更美好[EB/OL]. [https://blog.csdn.net/qz\\_39498701/article/details/84668833](https://blog.csdn.net/qz_39498701/article/details/84668833), 2019-06-04.
- [4]jzj1993. Xilinx Vivado 的使用详细介绍(1): 创建工程、编写代码、行为仿真、Testbench[EB/OL]. <https://blog.csdn.net/jzj1993/article/details/45533729>, 2015-05-06.
- [5]杨小帆\_. 基于 RISC-V 架构的单周期处理器设计(含所有格式指令)——控制信号选取及代码结构分析[EB/OL]. <https://yangfan.blog.csdn.net/article/details/103353982>, 2019-12-02.