一、实验目的

1. 掌握单周期CPU数据通路图的构成、原理及其设计方法；
2. 掌握单周期CPU的实现方法，代码实现方法；
3. 认识和掌握指令与CPU的关系；
4. 掌握测试单周期CPU的方法。

二、实验内容

2.1 概述

设计一个单周期CPU，该CPU至少能实现所需要的9条指令操作。整个 CPU 设计使用 Verilog 语言进行开发，使用 Vivado 进行集成开发，开发板为 PYNQ Z2，要实现的九条指令包括：add，slt，sltu，ori，lui，lw，sw，beq，jal。既有算术、逻辑运算指令，又有取数、存数指令；既有条件转移指令，又有无条件转移指令；既有对带符号数判断大小的指令，又有对无符号数判断大小的指令。单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。分析每条指令的功能；根据指令的功能给出所需的元件，并考虑如何将他们互连；确定每个元件所需控制信号的取值，汇总所有指令涉及的控制信号，生成反映指令和控制信号之间的关系表；根据关系表，得到每个控制信号的逻辑表达式，据此设计控制电路。

2.2 测试指令

该 CPU 设计以实际的 RISC-V 指令系统 RV32I 为基准，选取指令集六种指令格式中具有代表性的九条指令来进行实现，六种指令格式如下所示。

|  |  |  |
| --- | --- | --- |
| 格式 | 指令 | 操作 |
| R- 型 | add | rd，rs1，rs2 |
| R- 型 | slt | rd，rs1，rs2 |
| R- 型 | sltu | rd，rs1，rs2 |
| I- 型 | ori | rd，rs1，imm12 |
| I- 型 | lw | rd，rs1，imm12 |
| U- 型 | lui | rd，imm20 |
| S- 型 | sw | rs1，rs2，imm12 |
| B- 型 | beq | rs1，rs2，imm12 |
| J- 型 | jal | rd，imm20 |

2.3 指令功能概述

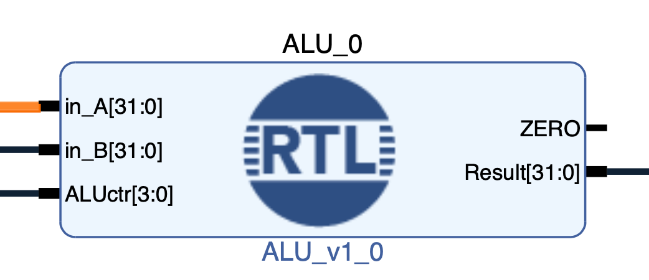
|  |  |  |
| --- | --- | --- |
| 指令 | 功能 | 说明 |
| add rd, rsl, rs2 | PC←PC+4  R[rd]←R[rs1] + R[rs2] | 从 PC 所指的内存单元中取指令，并 PC 加 4  从rsl、rs2 中取数后相加，结果送rd (不进行溢出判断) |
| slt rd, rs1, rs2 | if( R[rs1]< R[rs2]) R[rd]←1  else R[rd]← 0 | 从 rs1、rs2 中取数后按带符号整数来判断两数大小，小于则 rd 中置 1  否则，rd 中清 0 (不进行溢出判断) |
| sltu rd, rs1, rs2 | if(R[rs1] < R[rs2]) R[rd]←1  else R[rd]← 0 | 从 rs1、rs2 中取数后按无符号数来判断两数大小，小于则 rd 中置 1  否则，rd 中清 0 (不进行溢出判断) |
| ori rd, rs1, imm12 | R[rd]←R[rs1] l SEXT ( imm12) | 从 rs1 取数、将 imm12 进行符号扩展，然后两者按位或，结果送 rd |
| lui rd, imm20 | R[rd]←imm20 II 000H | rd高20位为imm20，低12位为0，符号 ll 表示 “拼接” |
| lw rd, rs1, imm12 | Addr←R[rs1] + SEXT ( imm12 )  R[rd]←M[Addr] | 从 rs1 取数、将 imm12 进行符号扩展，然后两者相加，  结果作为访存地址 Addr，从 Addr 中取数并送 rd |
| sw rs1, rs2, imm12 | Addr←R[rs1] + SEXT ( imm12)  M[Addr]←R[rs2] | 从 rs1 取数、将 imm12 进行符号扩展，然后两者相加，  结果作为访存地址 Addr ，将 rs2 送 Addr 中 |
| beq rsl, rs2, imm12 | Cond←R[rs1] - R[rs2]  if (Cond eq 0)  PC←PC+(SEXT (imm12) x 2) | 做减法以比较rsl和rs2中内容的大小，并计算下条指令地址，然后根据比较结果修改PC。  转移目标地址采用相对寻址，基准地址为当前指令地址(即PC)，  偏移量为立即数imm12经符号扩展后的值的2倍。  因此在RV321中，beq 指令转移目标的指令范围为当前指令的前1024到后1023条指令。 |
| jal rd, imm20 | R[rd]←PC+4  PC←PC +( SEXT( imm20) x 2) | PC+4的结果送rd但不送PC，然后计算下条指令地址。  转移地址采用相对寻址，基准地址为当前指令地址(即PC)，  偏移量为立即数imm20经符扩展后的值的2倍。  因此在RV32I中，jal 指令转移目标的指令范围为当前指令的前262144到后262143条指令。 |

2.4 设计流程

1. 确定各条指令所需要的部件
2. 确定各条指令中各个部件的输入输出关系
3. 确定数据通路总图
4. 根据各条指令所需要的部件以及总通路图，确定指令操作时间表（真值表）
5. 根据指令操作时间表，设计控制器
6. 根据总通路图以及指令操作时间表，编写Verilog代码

2.5 各组件及设计思路

**2.5.1 ALU（运算器）**

所处位置：

模块功能：

根据外部提供的控制信号和输入的数据执行不同的操作，并将运算后的结果输出

实现思路：

1. 加法：内置加法
2. 减法：内置减法
3. 有符号/无符号小于置1：

对于有符号数，需要检查加法器的符号位和溢出位。

符号位即为结果的符号，溢出位需要通过in\_A[31]^in\_B[31]的结果与结果的符号比较，如果不同，说明溢出。或者结果的第33位产生1，也会出现溢出。

对于无符号数，直接可以使用内置的小于符号。

④ 源操作数B：直接输出B

引脚及控制信号：

输入：

input [31:0] in\_A：寄存器1的值

input [31:0] in\_B：寄存器2的值或立即数

ALUctr：控制信号

输出：

output reg[31:0] Result：输出计算结果和信号

output reg Zero：结果为0时将Zero置1

内部变量：

reg [32:0] temp\_add; 33位的临时加法结果

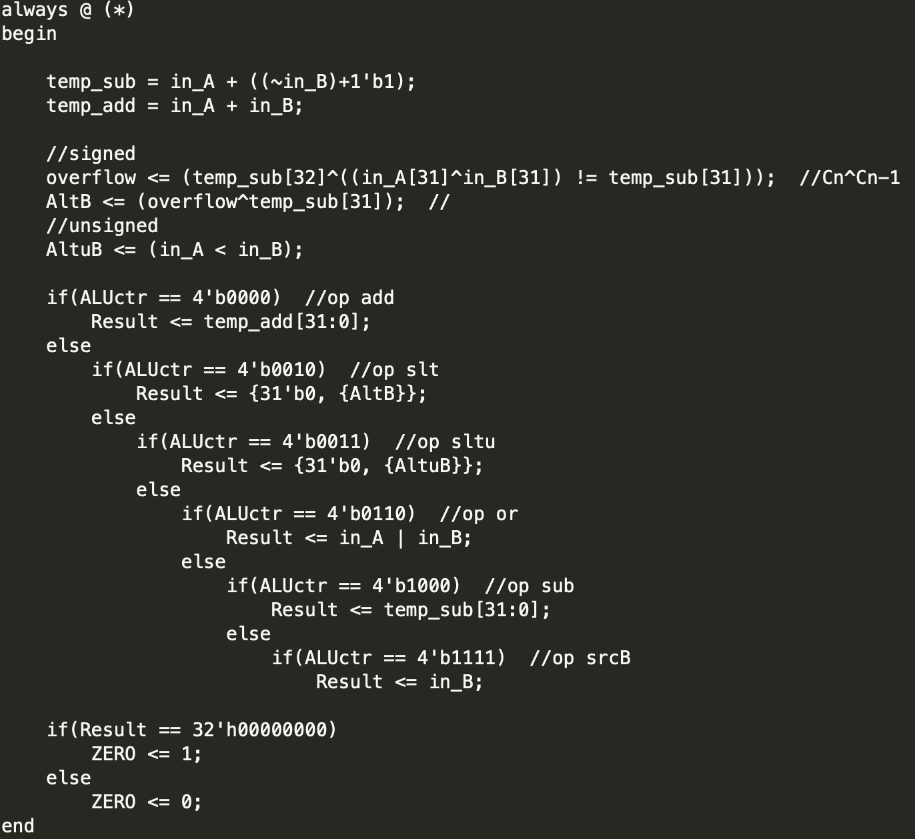
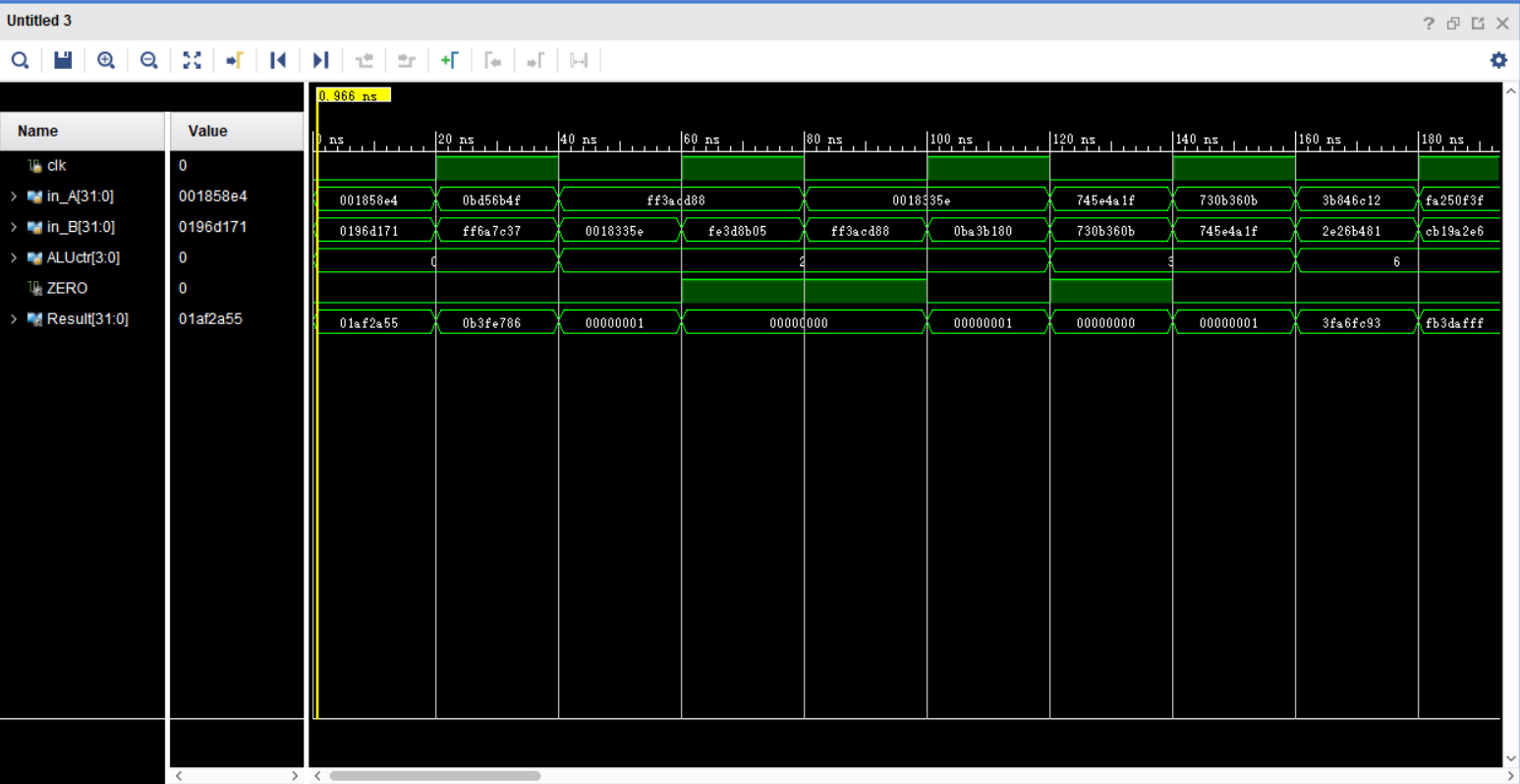
reg [32:0] temp\_sub; 33位的临时减法结果

reg AltB; 带符号整数小于置1

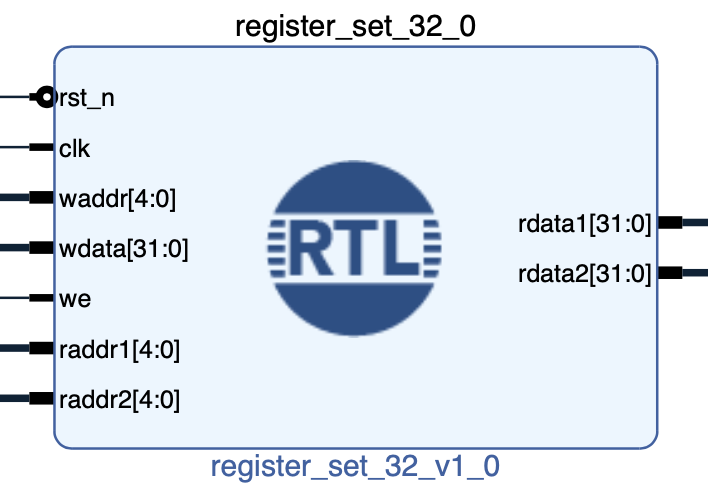
reg overflow; 溢出符号

reg AltuB; 无符号整数小于置1

实现代码：



模拟图：

**2.5.2 Register（寄存器组）**

所处位置：

模块功能：

寄存器用于CPU中数据的暂存

实现思路：

一共由32个32位寄存器组成，参考了材料中的示例：

1. 写入是在时钟脉冲的控制下，下降沿时写入
2. 读出的数据没有限制，端口根据读地址的选择，始终保持当前待读数据
3. 虽然设置了读使能，但是可以在测试中始终保持为1，即允许读入

引脚及控制信号：

输入：

waddr：5位，写地址

wdata：32位，写入的数据

we：1位，写使能。取1时为允许写入，0时为不能写入

输出：

raddr1：5位，读地址a

rdata1: 32位，读数据a

re1：1位，读使能，在测试中始终保持1，为允许读入

raddr2：5位，读地址a

rdata2： 32位，读数据a

re2： 1位，读使能，在测试中始终保持1，为允许读入

控制信号：

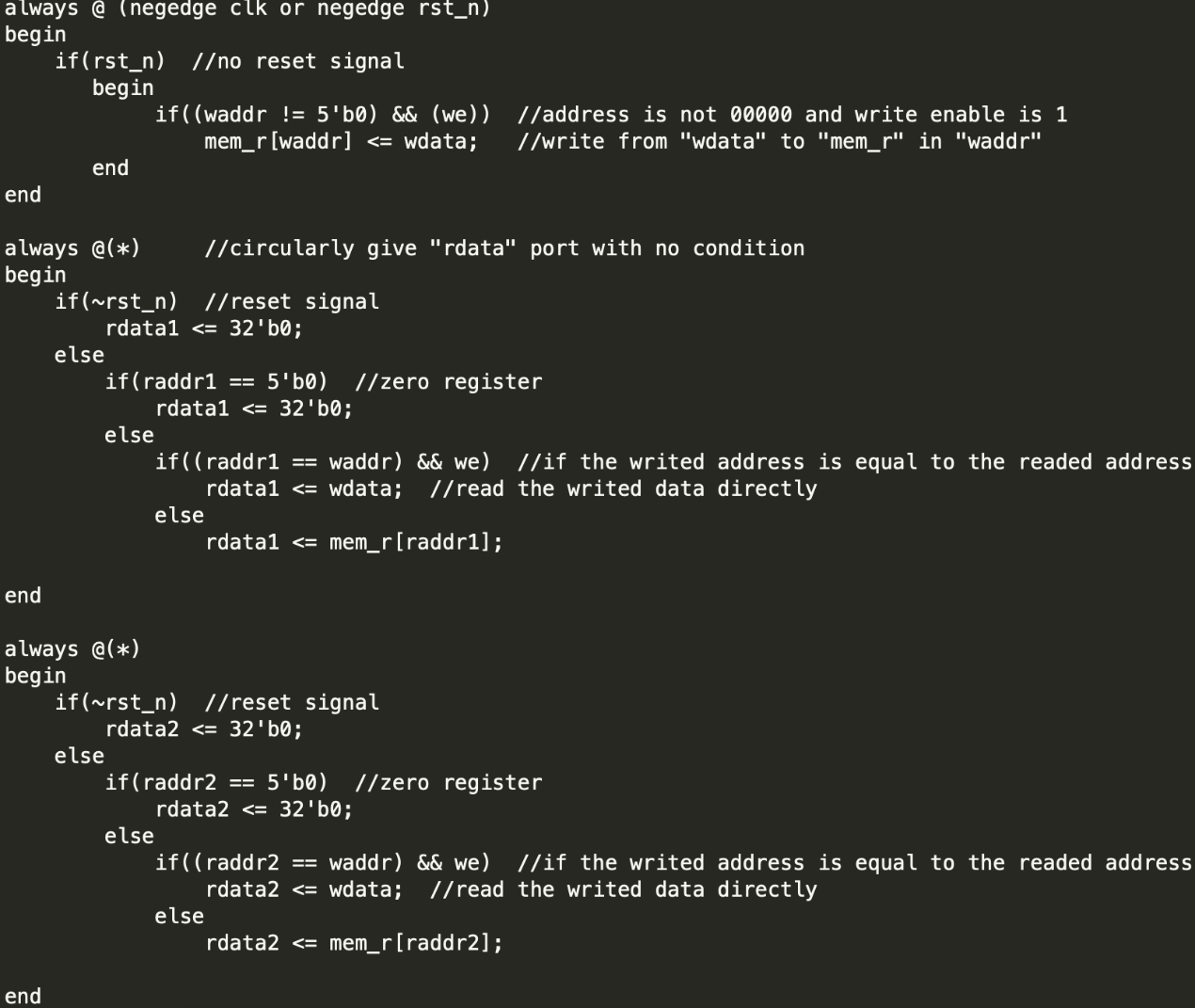
rst\_n：reset重置，将当前地址的寄存器置0

clk：时钟脉冲

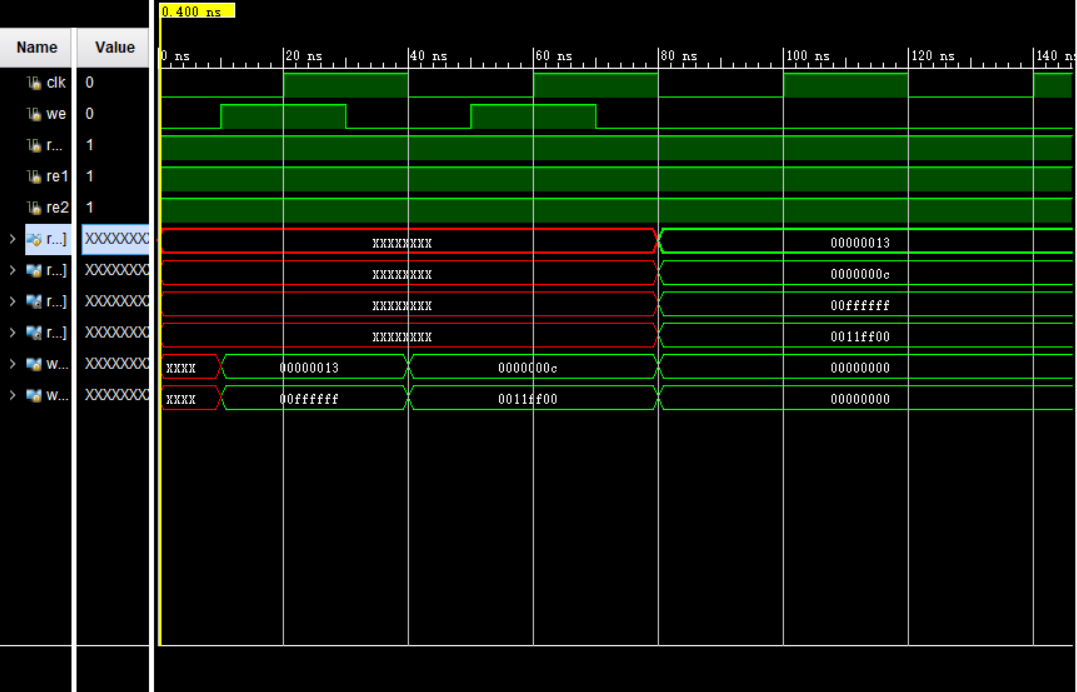
内部变量：

reg[31:0] mem\_r[0:31] 32位数组，类型为32位寄存器，通过将地址作为数组下标，送入到变量中，实现读和写

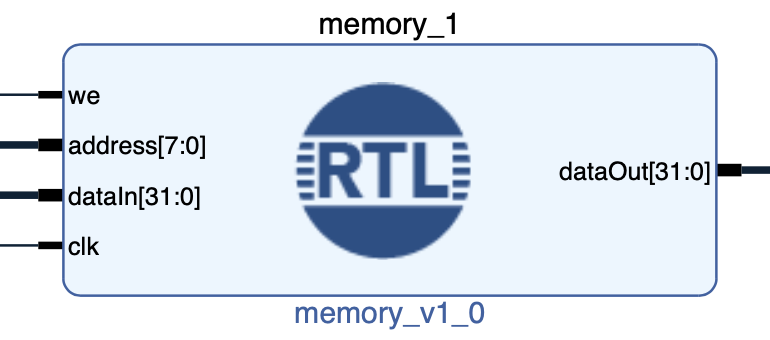
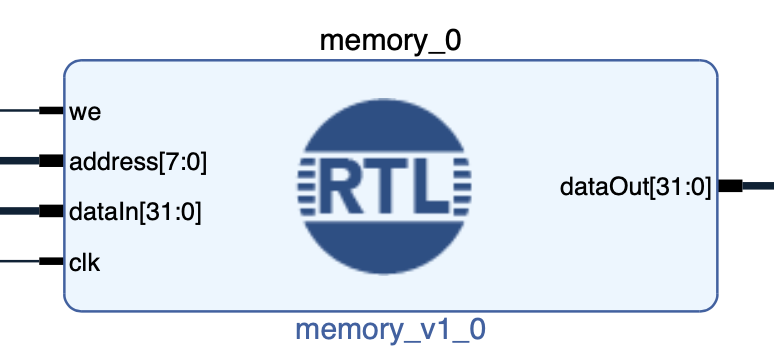
实现代码：



模拟图：



**2.5.3 Memery储存器**



所处位置：

模块功能：

数据存储器，CPU 可从存储器中读取指定地址处的数据，也可将数据写入到指定地址处（指令储存器和数据储存器均为MEM）

实现思路：

类似于寄存器组的实现过程

引脚及控制信号：

输入：

input address：读取和写入数据的地址

input dataIn：写入到储存器的数据

输出：

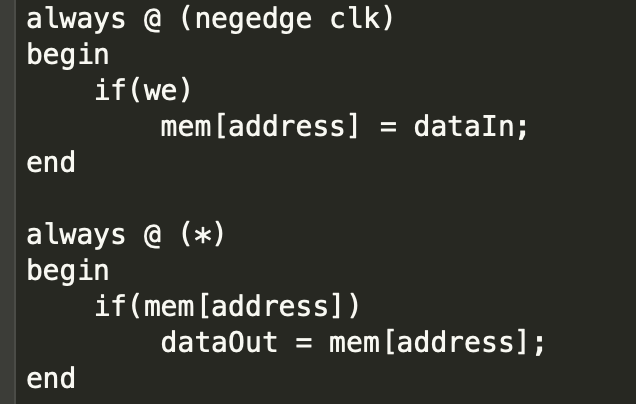
output dataOut：根据address地址从储存器输出的数据

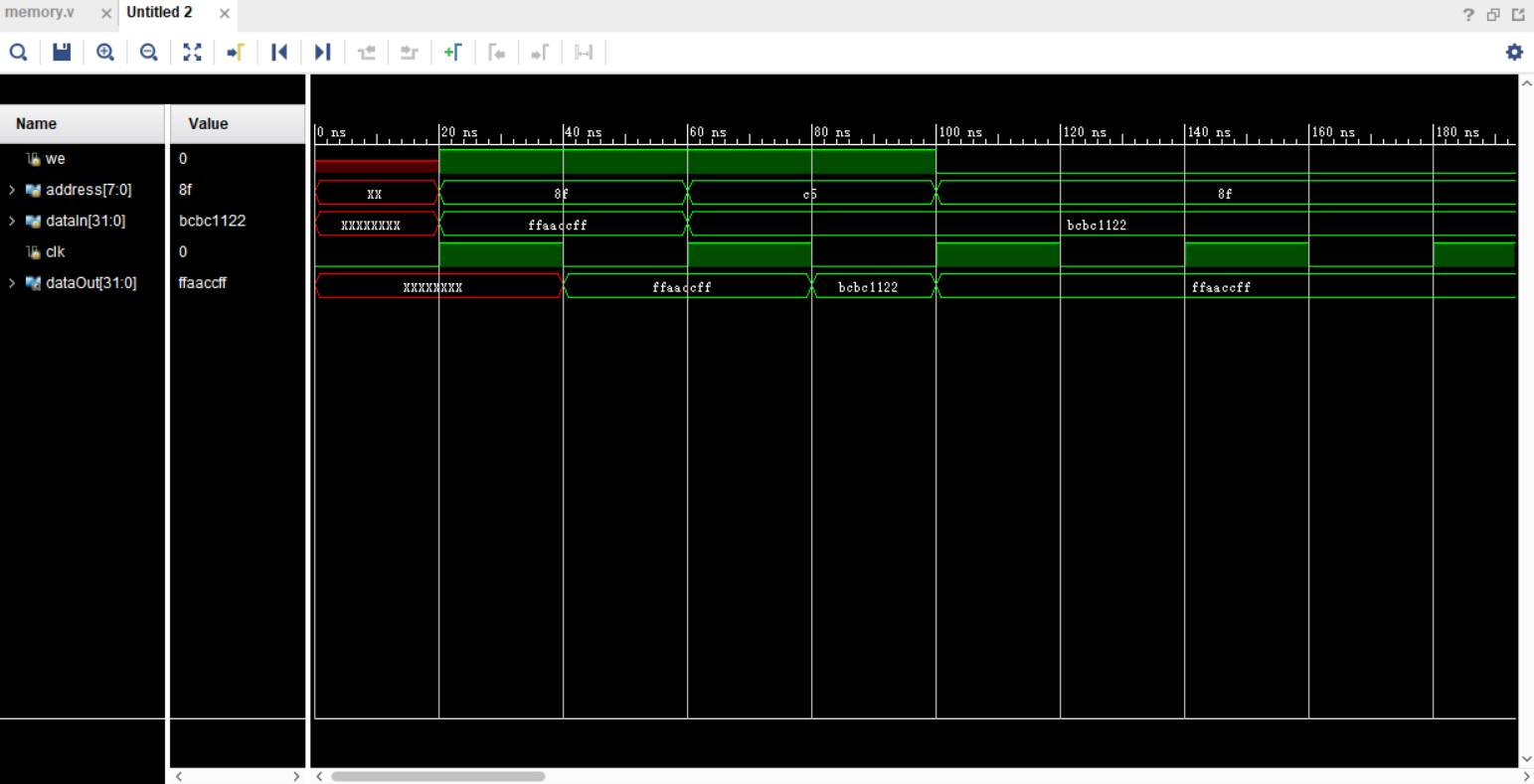
控制信号：

we：储存器写使能，置1时有信息写入到储存器

clk：时钟脉冲

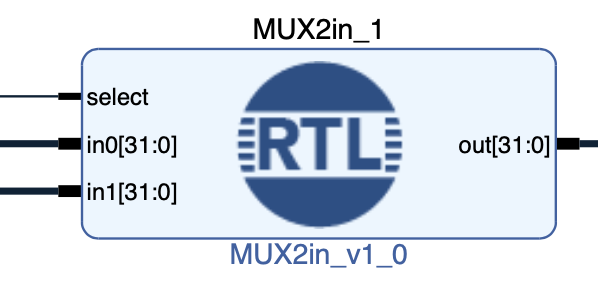
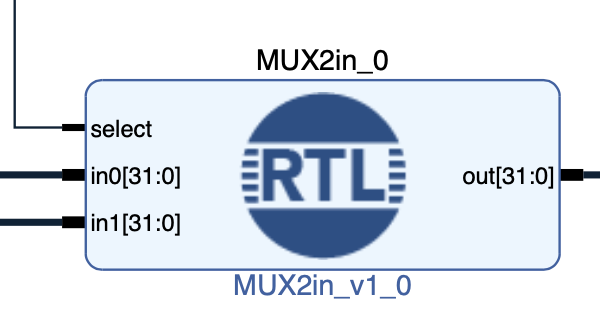
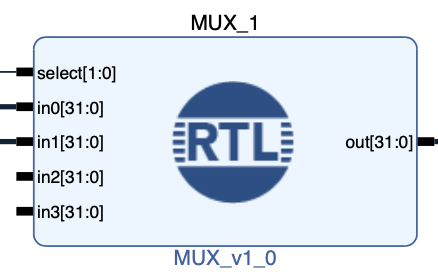
实现代码：



模拟图：

**2.5.4 MUX（四选一/二选一多路选择器）**

所处位置：



模块功能：

将ALU的A端口输入数据的多路选择器，ALU的B端口输入数据的多路选择器，以及ALU和Memery之间的多路选择器整合为一个，在数据通路中放置多个实现不同的功能

1. ALU的A端口输入数据
2. ALU的B端口输入数据
3. ALU运算后的结果作为下周期写入寄存器的数据

实现思路：

内部由if/else实现

引脚及控制信号：

输入：

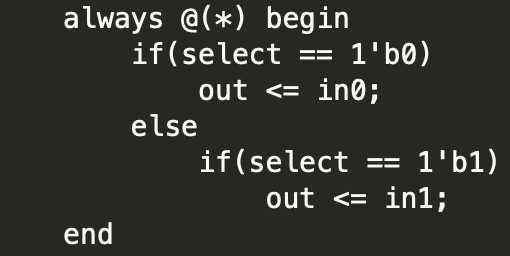
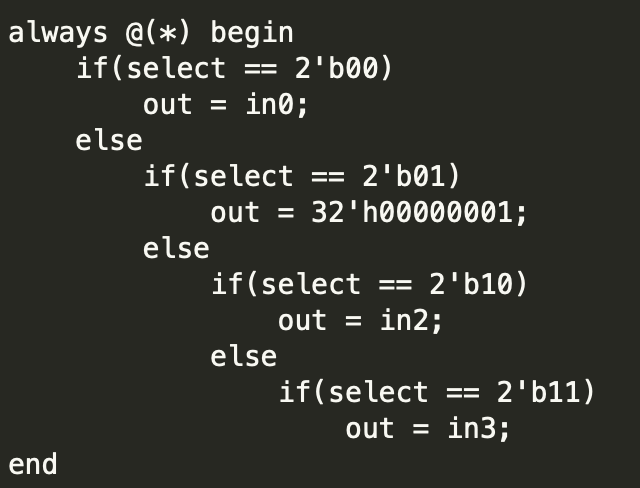
input [1:0]select：2位选择口

input [31:0] in0~in3：4个32位数据口

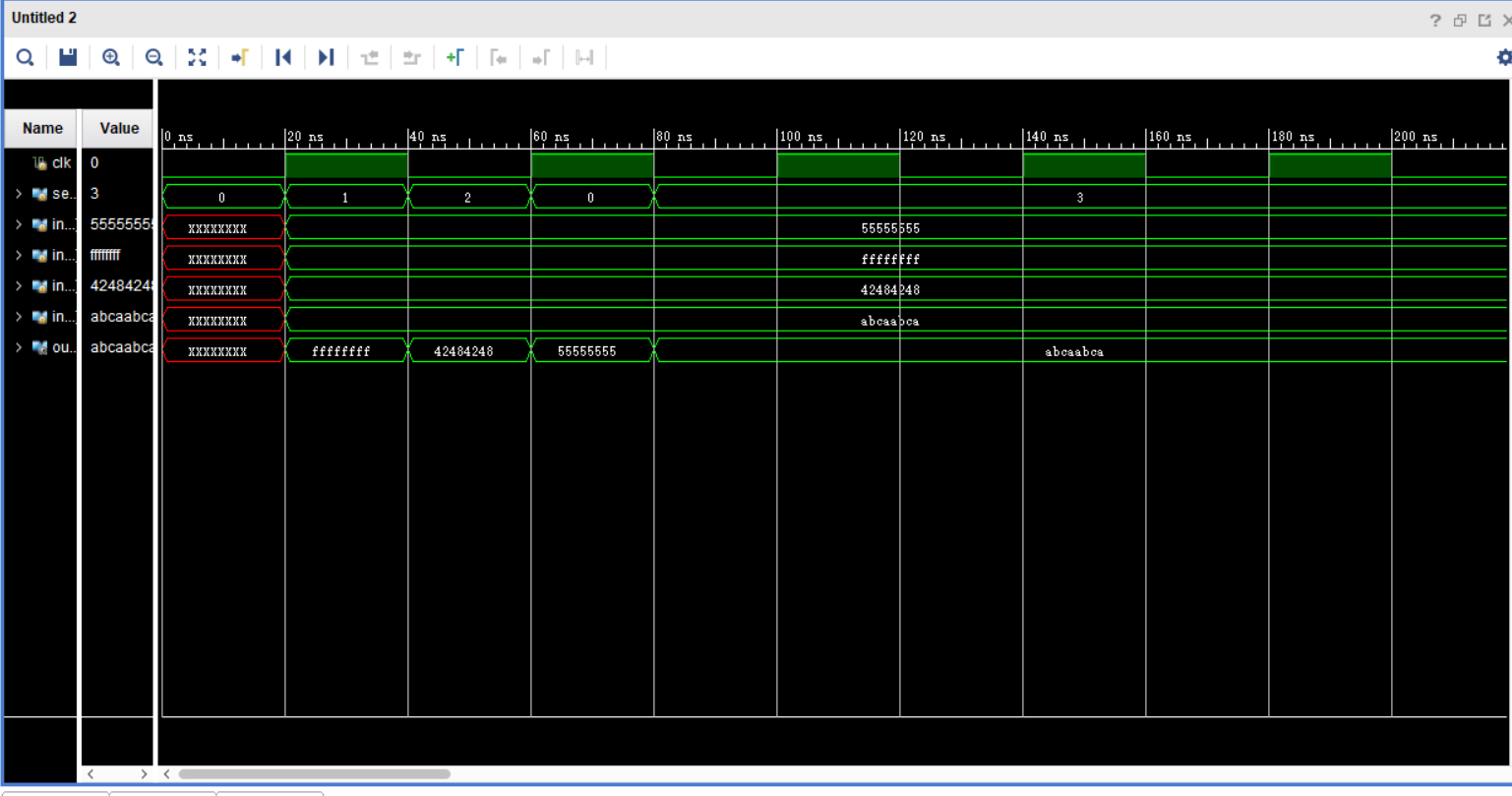
输出：

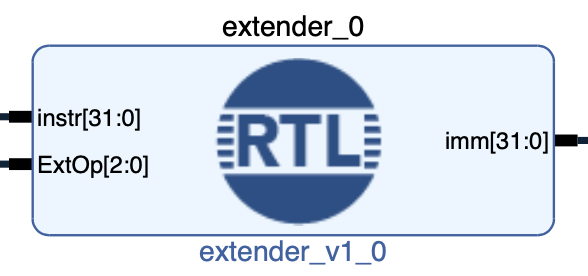
output [31:0]out：32位输出数据口

实现代码：



模拟图：



**2.5.5 Extender（立即数扩展器）**

所处位置：

模块功能：

根据ExtOp控制信号对输入的指令中的立即数进行扩展

实现思路

使用长为5的32位数组存储五种立即数，根据输入的选择信号ExtOp选择应当输出的元素

imm <= imms[ExtOp]

引脚及控制信号：

输入：

input [31:0] instr

输出：

output reg[31:0] imm

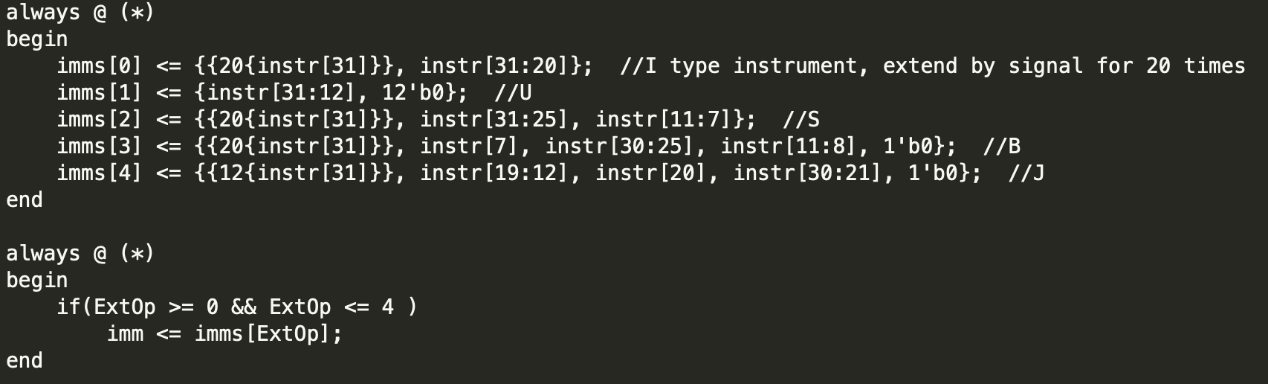
控制信号：

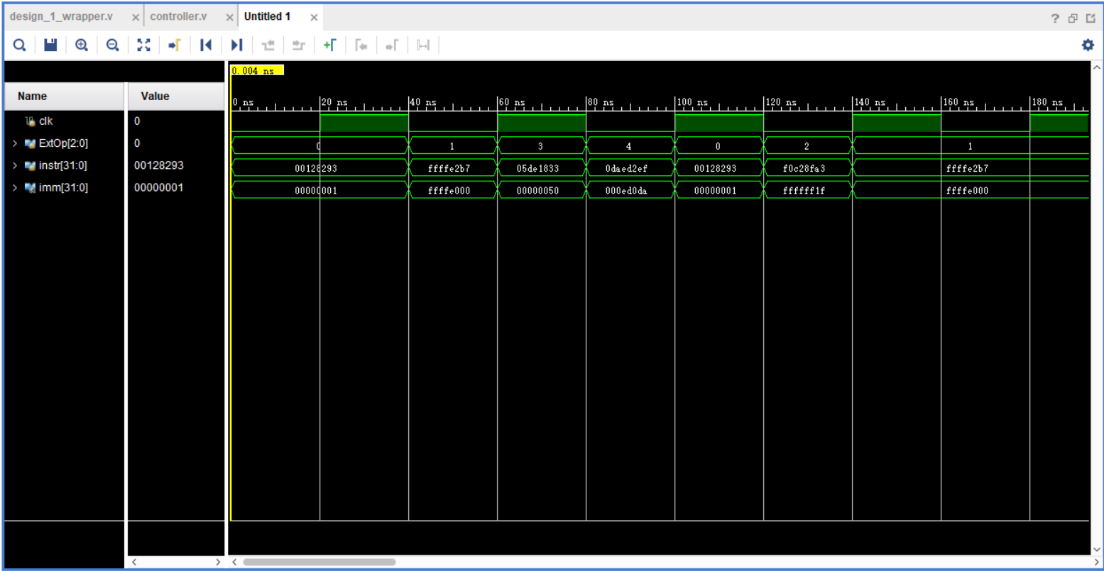
input [2:0] ExtOp

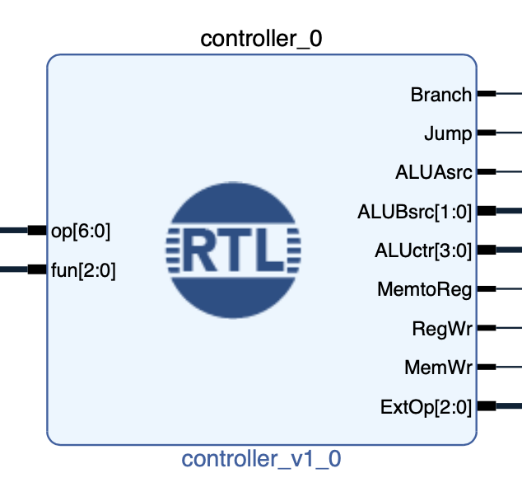
**内部变量：**

reg[31:0] imms[0:4]; //i u s b j

实现代码：



模拟图：

**2.5.6 Controller（控制器）**

所处位置：

模块功能：

控制器是作为CPU控制信号产生的器件，通过解析op和fun得到该指令的各种控制信号，根据各控制信号的指令逻辑表达式得到控制信号的取值

实现思路：

参照教材，使用内部变量表示各个指令的信号，对引脚和控制信号进行设计

引脚和控制信号：

输入：

input [6:0] op //opcode 输入信号

input [2:0] fun //funct3 输入信号

输出：

output reg[1:0] ALUBsrc 控制ALU B端输入口数据的MUX部件

output reg[3:0] ALUctr 控制ALU内部计算种类

output reg MemtoReg 控制ALU运算后的结果作为下周期写入寄存器的数据所使用的MUX部件

output reg RegWr 控制输入寄存器的We写使能信号

output reg MemWr 控制数据存储器的We写使能信号

output reg[2:0] ExtOp 控制扩展器Extender的ExtOp控制信号

内部变量：

reg Rtype

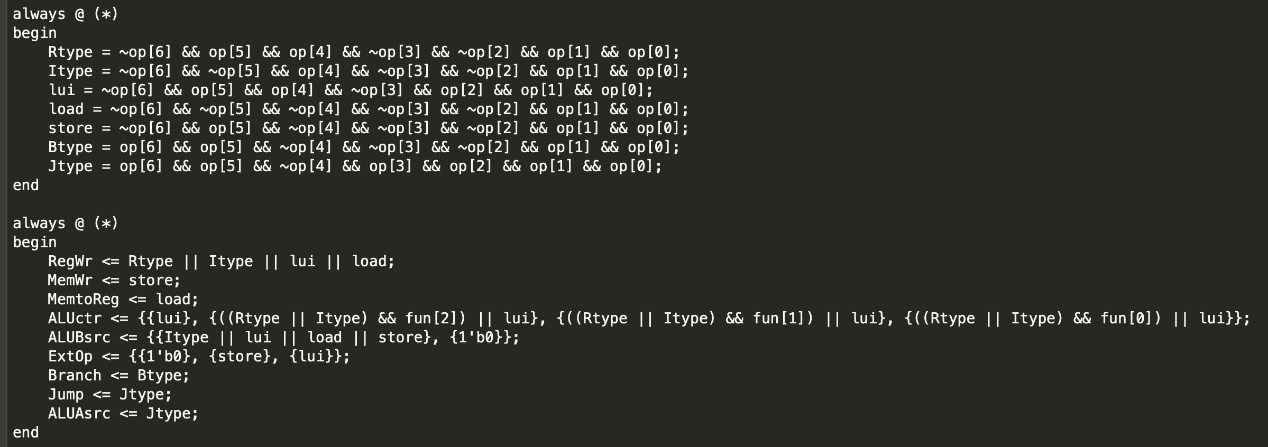
reg Itype

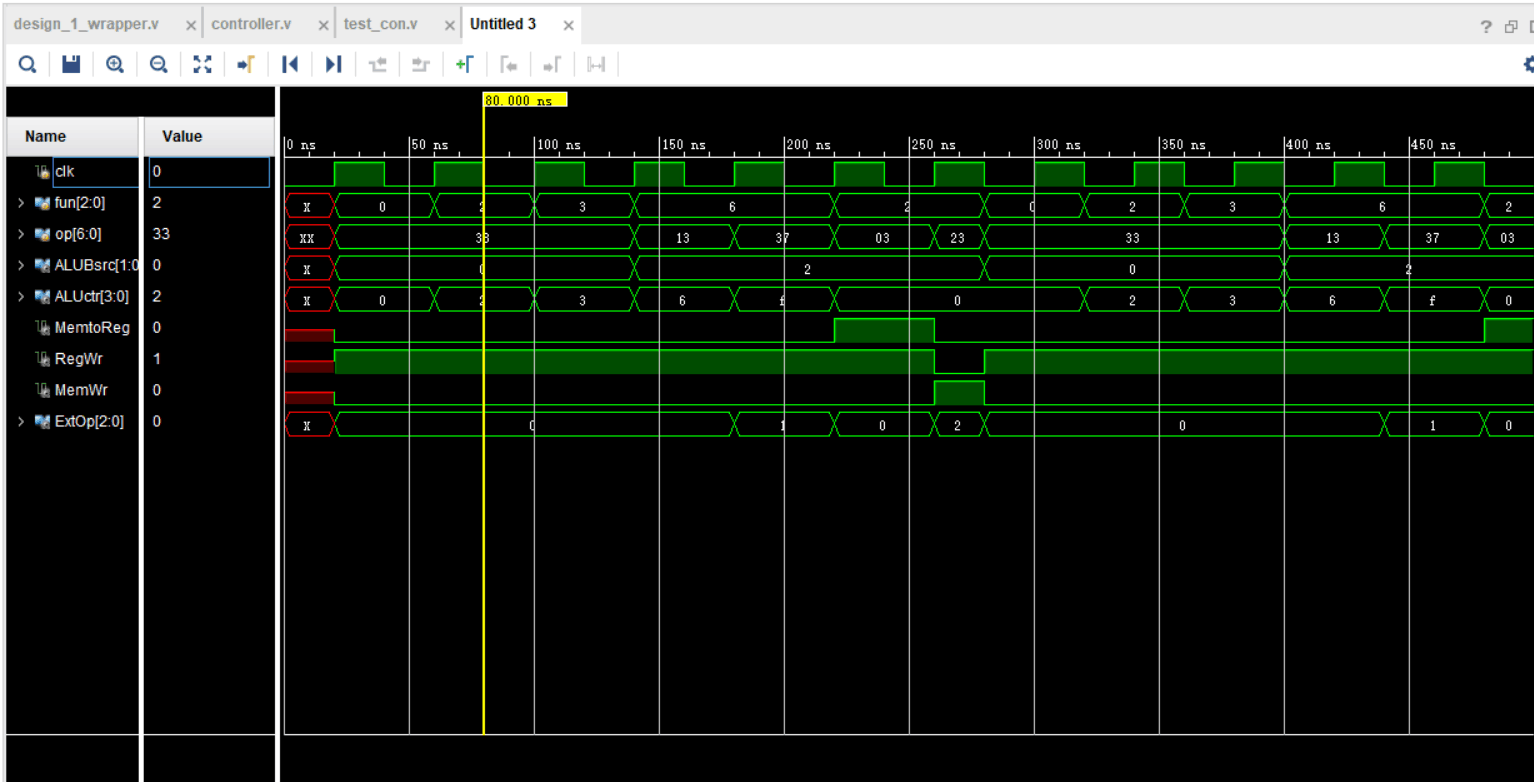
reg lui

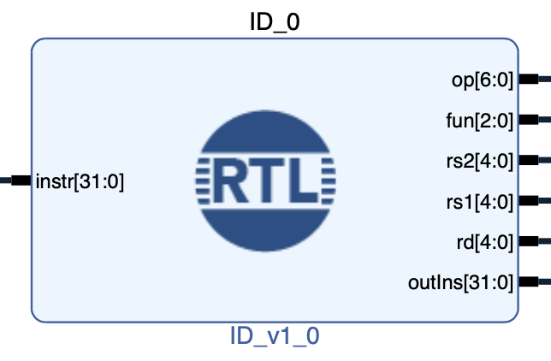
reg load

reg store

实现代码：



模拟图：

**2.5.7 ID（指令译码器）**

所处位置：

模块功能：

根据输入的指令对指令进行译码并将译码后的控制信号输出

实现思路：

根据教材内容和已完成部件的指令输入端口进行设计实现

引脚和控制信号：

输入：

input [31:0] instr：待译码的指令输入

输出：

output [6:0] op：输出后输入到Controller控制器的op控制信号

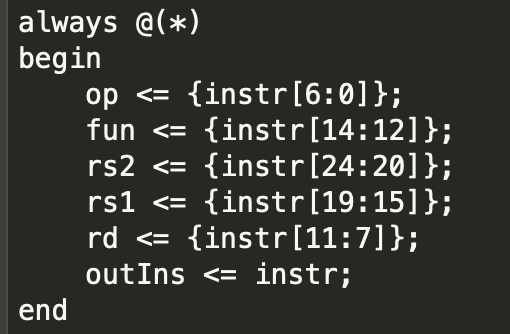
output [2:0] fun：输出后输入到Controller控制器的fun控制信号

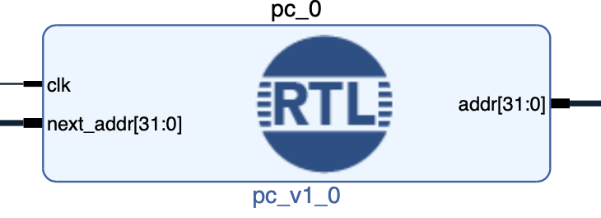
output [4:0] rs2：寄存器中读取地址

output [4:0] rs1：寄存器中读取地址

output [4:0] rd：寄存器中写入地址

output [31:0] outlns：扩展器中输入的待扩展指令的立即数

实现代码：

**2.5.8 PC（指令计数器）**

所处位置：

模块功能：

根据输入的时钟周期和下指令地址控制整个系统的时钟周期，并在下个周期开始时将当前指令地址更新为前周期的下指令地址

实现思路：

为实现稳定输出，在时钟信号的下降沿更新

引脚及控制信号**:**

输入：

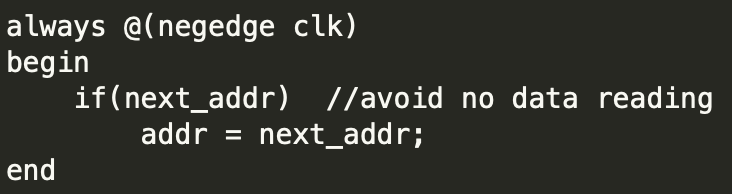
clk：时钟周期信号

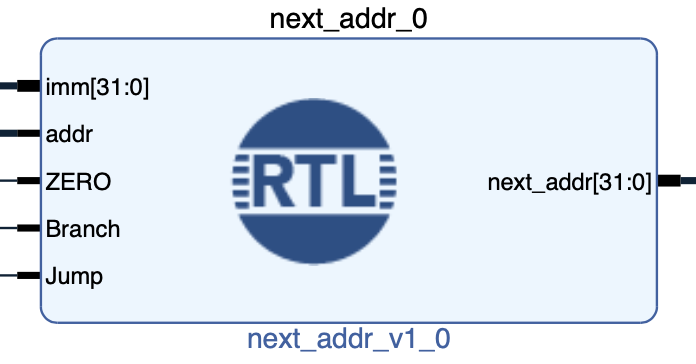
[31:0] next\_addr：目标地址，可能是跳转地址或者是下一条指令的地址

输出：

[31:0] addr：当前指令地址

实现代码：



**2.5.9 next\_addr（指令更新器）**

所处位置：

模块功能：

根据输入的控制信号以及数据计算下一条指令的地址

实现思路：

branch 表示当前指令是否是分支指令，imm 为立即数扩展，jump 表示当前指令是否是无条件跳转指令，addr 为当前周期的程序计数器也既当前正在执行的指令的地址，zero 表示 ALU 运算后的结果是否为零

引脚及控制信号：

输入：

[31:0] addr：当前指令地址

[31:0] imm：立即数扩展

branch

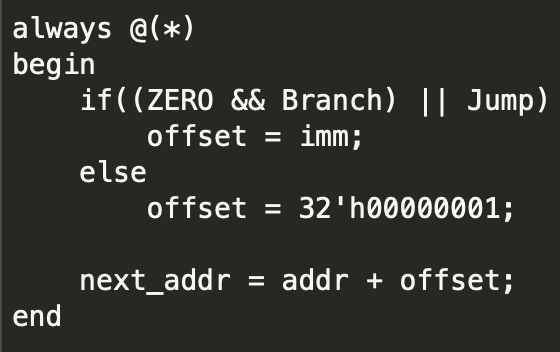
zero

jump

//branch 表示当前指令是否是分支指令，ce 为置零位，imm 为立即数扩展，jump 表示当前指令是否是无条件跳转指令，pc 为当前周期的程序计数器也既当前正在执行的指令的地址，zero 表示 ALU 运算后的结果是否为零。根据branch、zero、jump的值判断是否需要跳转，当branch、zero同时为1或者jump为1时跳转，否则PC+4

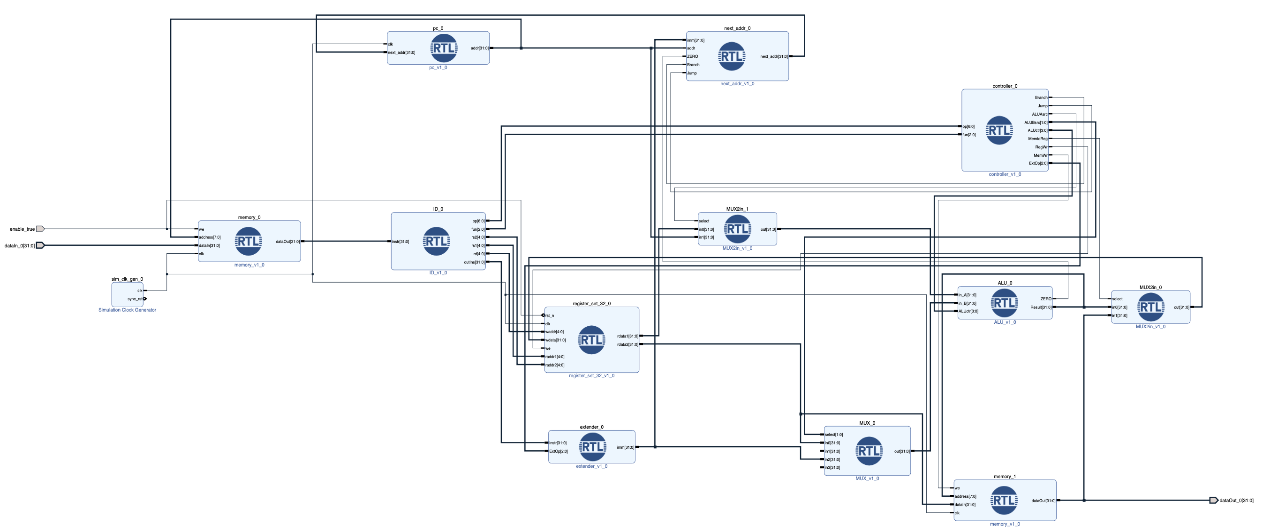
输出：

[31:0] next\_addr：下一条指令地址



实现代码：

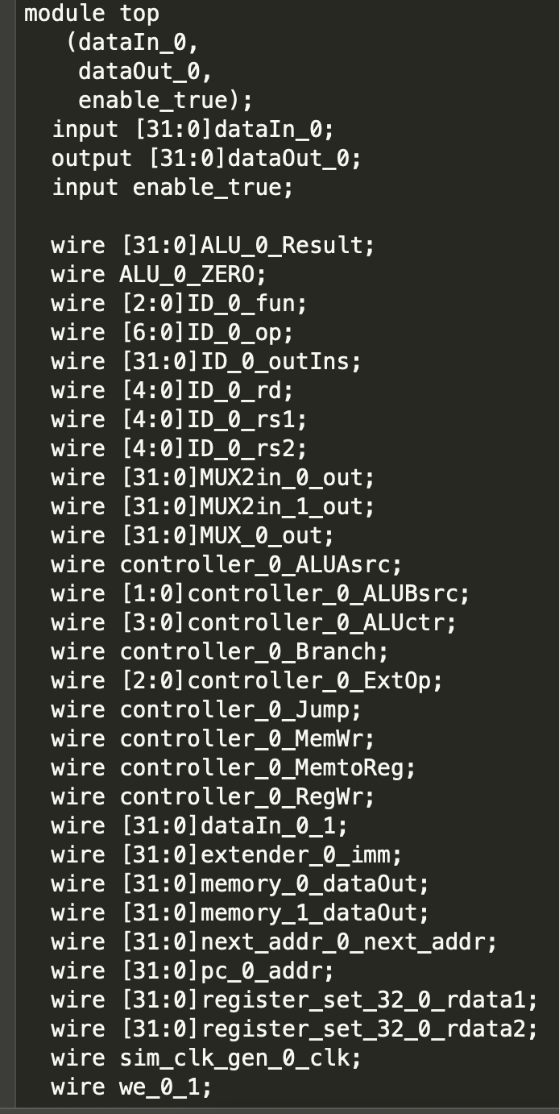
**2.5.10 数据通路图**



**2.5.11 顶层模块设计**

顶层模块的主要目的是将上述文件中的输入输出端口进行连接和例化，将上述各个模块连接成一个完整系统。

下面为顶层模块部分代码。



2.6 指令测试

顶层模块测试数据

16进制。

寄存器地址：2位 00-20

存储器地址：2位 00-ff

1.向0x01寄存器、0x02寄存器中写数：

ori 0x01, 0x00, 0x072 R[0x01] <- R[0x00] | SEXT(0x072)

0000,0111,0010, 0000,0 110, 0000,1 001,0011

ori 0x02, 0x00, 0x053 R[0x02] <- R[0x00] | SEXT(0x053)

0000,0101,0011, 0000,0 110, 0001,0 001,0011

2.向0x03、0x04寄存器中写数：

lui 0x03, 0xac894

1010,1100,1000,1001,0100, 0001,1 011,0111

lui 0x04, 0xbd633

1011,1101,0110,0011,0011, 0010,0 011,0111

3.以1做地址，2做数据，存入存储器：

sw 0x01, 0x03, 0x000

0000,000 0,0011, 0000,1 010, 0000,0 010,0011

sw 0x02, 0x04, 0x000

0000,000 0,0100, 0001,0 010, 0000,0 010,0011

state：

------

memory：

0x72 0xac89 4000

0x53 0xbd63 3000

register：

0x01 0x0000 0072

0x02 0x0000 0053

0x03 0xac89 4000

0x04 0xbd63 3000

------

4.将地址0x72和0x53的数据重新写回寄存器0x05、0x06：

lw 0x05, 0x00, 0x72

0000,0111,0010, 0000,0 010, 0010,1 000,0011

lw 0x06, 0x00, 0x53

0000,0101,0011, 0000,0 010, 0011,0 000,0011

state：

------

memory：

0x72 0xac89 4000

0x53 0xbd63 3000

register：

0x01 0x0000 0072

0x02 0x0000 0053

0x03 0xac89 4000

0x04 0xbd63 3000

0x05 0xac89 4000

0x06 0xbd63 3000

------

5.比较寄存器0x05和0x06的内容，小于置0x07为1

slt 0x07, 0x05, 0x06

0000,000 0,0110, 0010,1 010, 0011,1 011,0011

state：

———

memory：

0x72 0xac89 4000

0x53 0xbd63 3000

register：

0x01 0x0000 0072

0x02 0x0000 0053

0x03 0xac89 4000

0x04 0xbd63 3000

0x05 0xac89 4000

0x06 0xbd63 3000

0x07 0x0000 0001

------

6.寄存器0x07送入存储器地址0x2f

sw 0x00, 0x07, 0x2f

0000,001 0,0111, 0000,0 010, 0111，1 010,0011

state：

------

memory：

0x72 0xac89 4000

0x53 0xbd63 3000

0x2f 0x0000 0001

register：

0x01 0x0000 0072

0x02 0x0000 0053

0x03 0xac89 4000

0x04 0xbd63 3000

0x05 0xac89 4000

0x06 0xbd63 3000

0x07 0x0000 0001

------

7.将寄存器0x05、0x06结果相加，送入0x05

add 0x05, 0x05, 0x06

0000,000 0,0110, 0010,1 000, 0010,1 011,0011

state：

------

memory：

0x72 0xac89 4000

0x53 0xbd63 3000

0x2f 0x0000 0001

register：

0x01 0x0000 0072

0x02 0x0000 0053

0x03 0xac89 4000

0x04 0xbd63 3000

0x05 0x69ec 7000

0x06 0xbd63 3000

0x07 0x0000 0001

------

8.以0xa1作为地址将0x05中数据送入存储器

sw 0x00, 0x05, 0xa1

0000,101 0,0101, 0000,0 010, 0000,1 010,0011

state：

------

memory：

0x72 0xac89 4000

0x53 0xbd63 3000

0x2f 0x0000 0001

0xa1 0x69ec 7000

register：

0x01 0x0000 0072

0x02 0x0000 0053

0x03 0xac89 4000

0x04 0xbd63 3000

0x05 0x69ec 7000

0x06 0xbd63 3000

0x07 0x0000 0001

------

9.读出全部存储器中文件：

lw 0x08, 0x00, 0x072

lw 0x08, 0x00, 0x053

lw 0x08, 0x00, 0x02f

lw 0x08, 0x00, 0x0a1

0000,0111,0010, 0000,0 010, 0100,0 000,0011

0000,0101,0011, 0000,0 010, 0100,0 000,0011

0000,0010,1111, 0000,0 010, 0100,0 000,0011

0000,1010,0001, 0000,0 010, 0100,0 000,0011

10.指令测试二进制、16进制码：

00000111001000000110000010010011 07206093

00000101001100000110000100010011 05306113

10101100100010010100000110110111 ac8941b7

10111101011000110011001000110111 bd633237

00000000001100001010000000100011 0030a023

00000000010000010010000000100011 00412023

00000111001000000010001010000011 07202283

00000101001100000010001100000011 05302303

00000000011000101010001110110011 0062a3b3

00000010011100000010011110100011 027027a3

00000000011000101000001010110011 006282b3

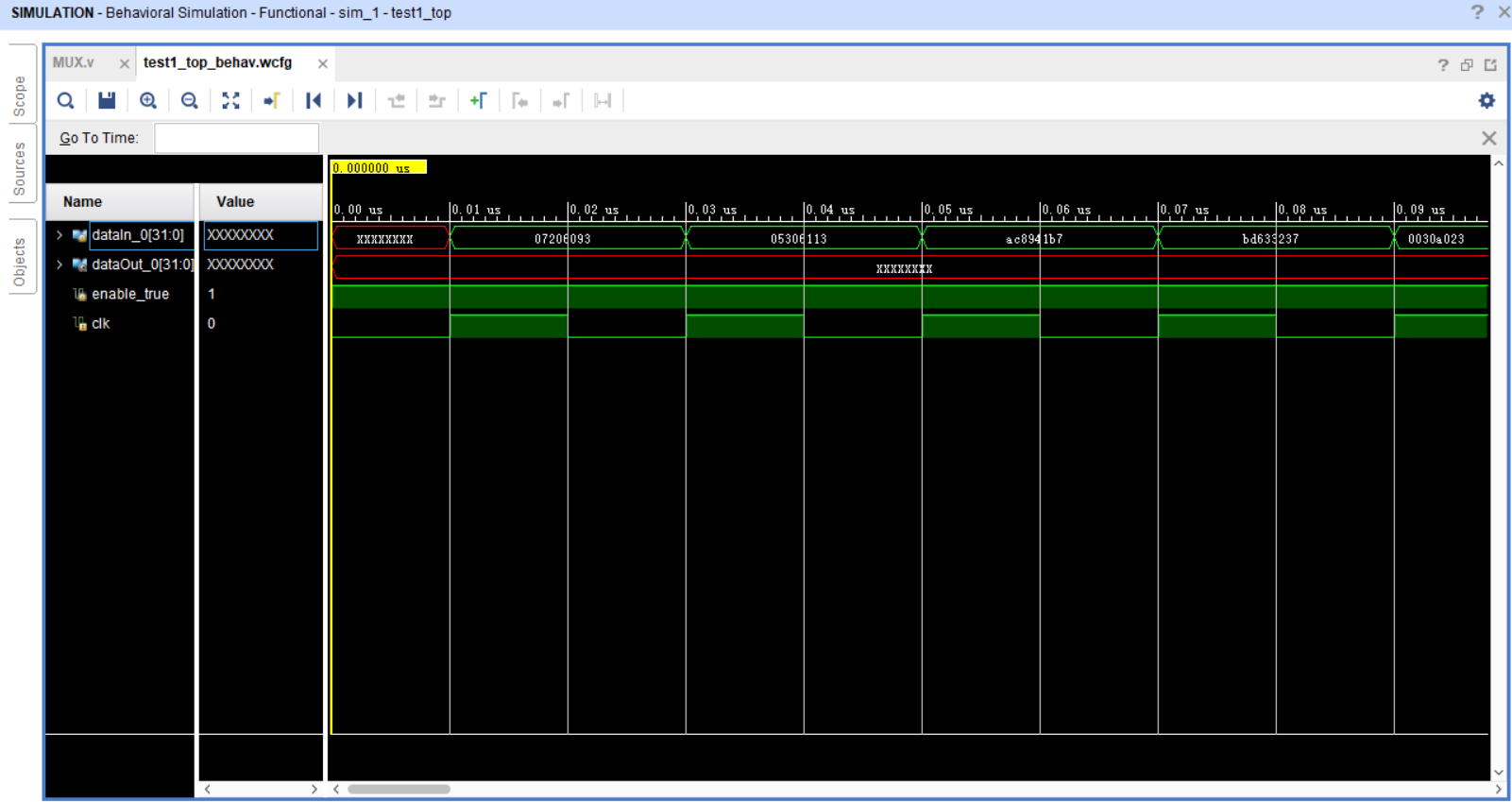
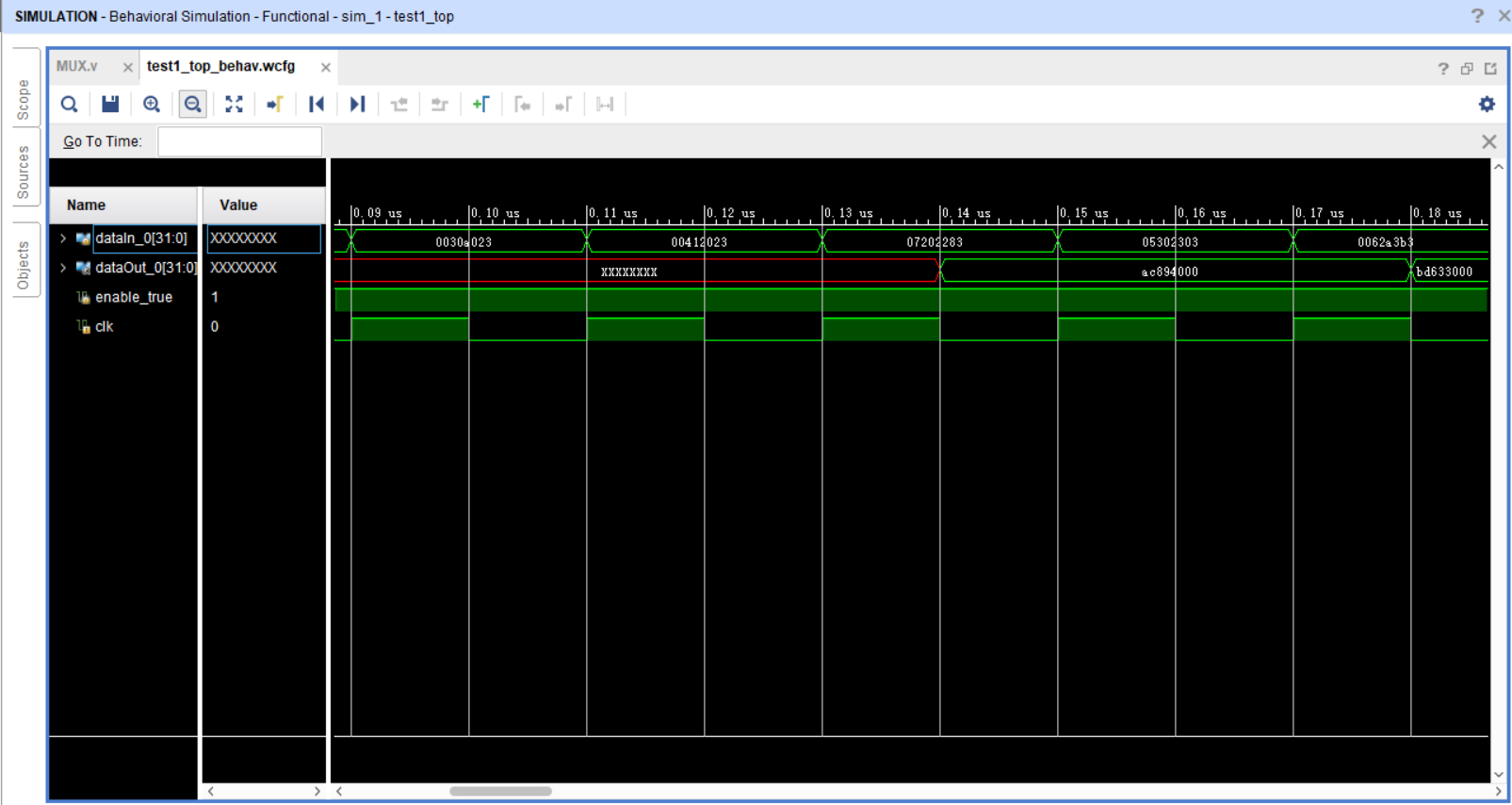
00001010010100000010000010100011 0a5020a3

00000111001000000010010000000011 07202403

00000101001100000010010000000011 05302403

00000010111100000010010000000011 02f02403

00001010000100000010010000000011 0a102403

2.7 指令仿真图

