计算机体系结构实验 第三周实验报告

组员：许诗瑶 20023105、刘朝润 20023114、刘晓航 20020070

一、实验要求

（一）C 语言与 RV 汇编指令分别编写 计算第 n 个斐波那契数 和 冒泡排序 测试程序，其中 C 语言程序编写时至少添加一条 printf 语句显示输出程序的执行结果。
（二）对测试程序涉及的指令整理，确定目标指令集。
（三）基于上周设计实现的 5 级流水线优化改进，并实现除访存、浮点和 16 位长外的所有指令。

二、实验环境

编程语言：Verilog
IDE：Vivado 2018.3
编译工具链：riscv-gnu-tool
工程版本控制及代码托管：Github 平台

三、实验内容

（一）C 语言与 RV 汇编指令编写测试程序

1、工具链

（1）使用 git clone 工具链源码，因为使用了 submodule 所以可以使用—recursive 在 clone 时同时 clone submodule，故命令为 git clone --recursive https://github.com/riscv/ riscv-gnu-toolchain。
（2）由于使用的环境时 win10 下的 wsl2（windows subsystem linux 2）的 ubuntu20.04，故须西在相关的包，命令：sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev。
（3） 进入 riscv-gnu-tool 文件夹
（4）选择配置参数，这里我开启了 32i 和 cfdm 四个拓展，并使用了 32bit hard float，并将生成的文件放在/home/lcr/riscv 文件夹下，故配置参数为./configure --prefix=/home/lcr/riscv --with-arch=rv32imfdc --with-abi=ilp32d。
（5）编译：make，等待完成后进入/home/lcr/riscv 即可使用 riscv 编译工具链。

2、Fibonacci by C

使用 c 语言编写带有一条 printf 语句的 Fibonacci 程序如四个测试程序文件加下的 Fibonacci.c 文件，内容如图：

```c
#include <stdio.h>

int main(){
    int n = 5;
    int fib = 0;

    if(n == 1) fib = 0;
    else if(n == 2) fib = 1;
    else {
        int i = 0;
        int j = 1;
        int k;
        for(k = 3; k <= n; k++){
            fib = i + j;
            i = j;
            j = fib;
        }
    }

    printf("The %d of fobonacci is: %d.\n", n, fib);
}
```

使用工具链进行编译，并最终反汇编可执行文件获得最终的汇编程序。其中由于使用printf函数，该函数依赖于静态库libc，故在第四步的时候使用-static参数静态链接libc，这样可以在反汇编代码中包含 printf 的机器码。

```
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ rm Fibonacci.0
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./riscv32-unknown-elf-gcc -E Fibonacci.c -o Fibonacci.i
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./riscv32-unknown-elf-gcc -S Fibonacci.i -o Fibonacci.s
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./riscv32-unknown-elf-gcc -c Fibonacci.s -o Fibonacci.o
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./riscv32-unknown-elf-gcc -static Fibonacci.o -o Fibonacci.out
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./riscv32-unknown-elf-objdump -S Fibonacci.out > Fibonacci.txt
```

最终生成的结果如下图，文件为四个测试程序文件夹下的 Fibonacci.txt，中间生成文件也在该文件夹下。

```
Fibonacci.out:     file format elf32-littleriscv


Disassembly of section .text:

00010074 <register_fini>:
   10074:   00000793            li a5,0
   10078:   c791                beqz  a5,10084 <register_fini+0x10>
   1007a:   6549                lui   a0,0x12
   1007c:   6f650513            addi  a0,a0,1782 # 126f6 <__libc_fini_array>
   10080:   1ca0206f            j  1224a <atexit>
   10084:   8082                ret

00010086 <_start>:
   10086:   0000f197            auipc gp,0xf
   1008a:   7ca18193            addi  gp,gp,1994 # 1f850 <__global_pointer$>
   1008e:   1cc18513            addi  a0,gp,460 # 1fa1c <__malloc_max_total_mem>
   10092:   22818613            addi  a2,gp,552 # 1fa78 <__BSS_END__>
   10096:   8e09                sub   a2,a2,a0
   10098:   4581                li a1,0
   1009a:   2a69                jal   10234 <memset>
   1009c:   00002517            auipc a0,0x2
   100a0:   1ae50513            addi  a0,a0,430 # 1224a <atexit>
   100a4:   c519                beqz  a0,100b2 <_start+0x2c>
   100a6:   00002517            auipc a0,0x2
   100aa:   65050513            addi  a0,a0,1616 # 126f6 <__libc_fini_array>
   100ae:   19c020ef            jal   ra,1224a <atexit>
```

3、Fibonacci by RV

使用 RV 汇编编写的 Fibonacci 程序如下图。

```
main:
    addi    a2,x0,6
    addi    a3,x0,0
    addi    a4,x0,1
    addi    a5,x0,2
    addi    a6,a2,0
    beq a2,a4,.L1
    addi    a6,a4,0
    beq a2,a5,.L1
.L0:
    add a6,a3,a4
    bge a2,a5,.L1
    addi    a5,a5,1
    addi    a3,a4,0
    addi    a4,a5,0
    j   .L0
.L1:
    sw  a6,0(x0)
    jr  ra
```

使用工具链对其进行编译生成可执行文件，并对可执行文件反汇编得到汇编指令。文件分别为四个测试程序文件夹下 Fibonacci_ass.s 和 Fibonacci_ass.txt。

```
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./riscv32-unknown-elf-gcc -c Fibonacci_ass.s -o Fibonacci_ass.o
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./riscv32-unknown-elf-objdump -S Fibonacci_ass.o > Fibonacci_ass.txt
```

Fibonacci_ass.txt

```
Fibonacci_ass.o:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <main>:
   0: 4619                li  a2,6
   2: 4681                li  a3,0
   4: 4705                li  a4,1
   6: 4789                li  a5,2
   8: 00060813          mv  a6,a2
   c: 02e60063          beq a2,a4,2c <.L1>
  10: 00070813          mv  a6,a4
  14: 00f60c63          beq a2,a5,2c <.L1>

00000018 <.L0>:
  18: 00e68833          add a6,a3,a4
  1c: 00f65863          bge a2,a5,2c <.L1>
  20: 0785                addi  a5,a5,1
  22: 00070693          mv  a3,a4
  26: 00078713          mv  a4,a5
  2a: b7fd                j 18 <.L0>

0000002c <.L1>:
  2c: 01002023          sw  a6,0(zero) # 0 <main>
  30: 8082                ret
```

4、Bubble Sort by C

使用 c 语言编写带有一条 printf 语句的 Bubble Sort 程序如四个测试程序文件加下的 BubbleSort.c 文件，内容如图：

```c
#include <stdio.h>

int main() {
    int n = 10;
    int array[] = {10, 9, 8, 7, 6, 1, 2 ,3, 4, 5};
    for (int i = 0; i < n - 1; i++) {
        for (int j = i; j < n - 1; j++) {
            if (array[j] >= array[j+1]) {
                int temp = array[j+1];
                array[j+1] = array[j];
                array[j] = temp;
            }
        }
    }
    printf("the first element is %d.\n", array[0]);
    return 0;
}
```

使用工具链进行编译，获得可执行文件，并将可执行文件反汇编获得汇编指令，其命令和结果如下图。文件为四个测试程序下的 BubbleSort.txt。中间结果也在该文件夹下。

```
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./riscv32-unknown-elf-gcc -E BubbleSort.c -o BubbleSort.i
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./riscv32-unknown-elf-gcc -S BubbleSort.i -o BubbleSort.s
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./riscv32-unknown-elf-gcc -c BubbleSort.s -o BubbleSort.o
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./riscv32-unknown-elf-gcc -static BubbleSort.o -o BubbleSort.out
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./riscv32-unknown-elf-objdump -S BubbleSort.out > BubbleSort.txt
lcr@DESKTOP-JDL9JAK:~/riscv/bin$
```

```
BubbleSort.txt

BubbleSort.out:     file format elf32-littleriscv


Disassembly of section .text:

00010074 <register_fini>:
   10074:   00000793            li a5,0
   10078:   c791                beqz  a5,10084 <register_fini+0x10>
   1007a:   6549                lui   a0,0x12
   1007c:   78850513            addi  a0,a0,1928 # 12788 <__libc_fini_array>
   10080:   25c0206f            j  122dc <atexit>
   10084:   8082                ret

00010086 <_start>:
   10086:   00011197            auipc gp,0x11
   1008a:   87a18193            addi  gp,gp,-1926 # 20900 <__global_pointer$>
   1008e:   1cc18513            addi  a0,gp,460 # 20acc <__malloc_max_total_mem>
   10092:   22818613            addi  a2,gp,552 # 20b28 <__BSS_END__>
   10096:   8e09                sub   a2,a2,a0
   10098:   4581                li a1,0
   1009a:   2435                jal   102c6 <memset>
   1009c:   00002517            auipc a0,0x2
   100a0:   24050513            addi  a0,a0,576 # 122dc <atexit>
   100a4:   c519                beqz  a0,100b2 <_start+0x2c>
   100a6:   00002517            auipc a0,0x2
   100aa:   6e250513            addi  a0,a0,1762 # 12788 <__libc_fini_array>
   100ae:   22e020ef            jal   ra,122dc <atexit>
   100b2:   224d                jal   10254 <__libc_init_array>
   100b4:   4502                lw a0,0(sp)
   100b6:   004c                addi  a1,sp,4
   100b8:   4601                li a2,0
   100ba:   2891                jal   1010e <main>
   100bc:   aaad                j  10236 <exit>

000100be <__do_global_dtors_aux>:
   100be:   1e41c703            lbu   a4,484(gp) # 20ae4 <completed.1>
   100c2:   e71d                bnez  a4,100f0 <__do_global_dtors_aux+0x32>
   100c4:   1141                addi  sp,sp,-16
   100c6:   c422                sw s0,8(sp)
   100c8:   843e                mv s0,a5
   100ca:   c606                sw ra,12(sp)
```

5、Bubble Sort by RV

使用 RV 汇编编写的 Bubble Sort 程序如下图。

BubbleSort_ass.s

```
1   main:
2       addi a2,x0,9
3       addi a4,x0,0
4   .L0:
5       addi a5,a4,0
6   .L1:
7       slli a6,a5,2
8       lw   t0,0(a6)
9       lw   t1,4(a6)
10      blt  t0,t1,.L3
11      sw   t0,4(a6)
12      sw   t1,0(a6)
13  .L3:
14      addi a5,a5,1
15      blt  a5,a2,.L1
16      addi a4,a4,1
17      blt  a4,a2,.L0
18      jal  ra
19
```

使用工具链生成可执行文件并反汇编成汇编代码，结果如下图。

```
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./riscv32-unknown-elf-gcc -c BubbleSort_ass.s -o BubbleSort_ass.o
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./riscv32-unknown-elf-objdump -S BubbleSort_ass.o > BubbleSort_ass.txt
```

BubbleSort_ass.txt

```
1
2  BubbleSort_ass.o:     file format elf32-littleriscv
3
4
5  Disassembly of section .text:
6
7  00000000 <main>:
8     0: 4625                li a2,9
9     2: 4701                li a4,0
10
11 00000004 <.L0>:
12    4: 00070793            mv a5,a4
13
14 00000008 <.L1>:
15    8: 00279813            slli  a6,a5,0x2
16    c: 00082283            lw t0,0(a6)
17   10: 00482303            lw t1,4(a6)
18   14: 0062c663            blt   t0,t1,20 <.L3>
19   18: 00582223            sw t0,4(a6)
20   1c: 00682023            sw t1,0(a6)
21
22 00000020 <.L3>:
23   20: 0785                addi  a5,a5,1
24   22: fec7c3e3            blt   a5,a2,8 <.L1>
25   26: 0705                addi  a4,a4,1
26   28: fcc74ee3            blt   a4,a2,4 <.L0>
27   2c: fd5ff0ef            jal   ra,0 <main>
28
```

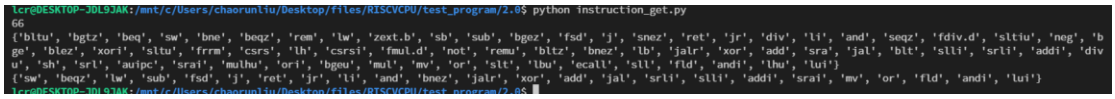
（二）重新确定目标指令集

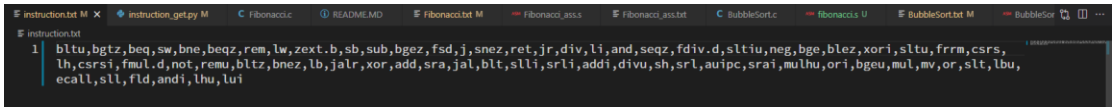
1、上周初步设计的指令集

在 RISC-V 手册中 base Interger instructions: RV32I 的 47 条指令的基础上，我们增加了 8 条扩展指令完成乘、除、取余数。以及为了更好的完成 for 循环跳转，设计了 2 条自定义扩展指令，我们初步完成了指令集设计。详见文件 ISA/ISA Definition_v1.xlsx。

2、根据本周要求调整并重新确定指令集

利用生成 Fibonacci.txt， Fibonacci_ass.txt,BubbleSort.txt,BubbleSort_ass.txt 文件分析其中用到的指令，由于其中两个使用了 printf 语句的反汇编文件程度在 21000 行左右，因此人工分析较慢，所以我们编写了一个用于指令分析的 python 文件，它可以将这四个文件中用到的指令写入 instruction.txt 文件中。文件见本次实验测试程序及代码文件夹下的 instruction_get.py，运行结果如下图：





从中我们共获得 66 个伪汇编指令，接下来我们将伪汇编指令转化为指令成为我们本次实验的指令集（因浮点、16 位指令不在此次实验要求中故而没有实现）。如下为 66 条指令具体内容：

Li

基础指令	指令描述	指令格式
等同于 lui/addi	Load Immediate 使用尽可能少的指令将常量加载到 x[rd]中	

Bltz

基础指令	指令描述	指令格式
等同于 blt	Branch if Less Than Zero 小于零时分支	blt rs1, x0, offset

Bgtz

基础指令	指令描述	指令格式
等同于 blt	Branch if Greater Than Zero 大于零时分支	blt x0, rs2, offset

Frrm

基础指令	指令描述	指令格式
等同于 csrrs	Floating-Point Read Rounding Mode 把浮点舍入模式的值写入 x[rd]	csrrs rd, frm, x0

Ret

基础指令	指令描述	指令格式
扩展为 jalr	Return 从子程序返回	jalr x0, 0（x1）

J

基础指令	指令描述	指令格式
等同于 jal	Jump 把 PC 设置成当前值加上符号位扩展的 offset	jal x0, offset

Seqz

基础指令	指令描述	指令格式
扩展为 sltiu	Set if Equal to Zero 如果 x[rs1]等于 0，向 x[rd]写入 1，否则写入 0	sltiu rd, rs1,1

Neg

基础指令	指令描述	指令格式

扩展为 sub	Negate 取反，把寄存器 x[rs2]的二进制补码写入 x[rd]	sub rd，x0，rs2

Beqz

基础指令	指令描述	指令格式
等同于 beq	Branch if Equal to Zero 等于零时分支	beq rs1，x0，offset

Zext.b

基础指令	指令描述	指令格式
等同于 addi	Load Immediate 使用尽可能少的指令将常量加载到 x[rd]中	addi rd，rs1，imme

Blez

基础指令	指令描述	指令格式
等同于 bge	Branch if Less Than or Equal to Zero 小于等于零时分支	bge x0，rs2，offset

Jr

基础指令	指令描述	指令格式
等同于 jalr	Jump Register	jarl x0，0(rs1)

Mv

基础指令	指令描述	指令格式
扩展为 addi	Move 把寄存器 x[rs1]复制到 x[rd]中	addi rd，rs1，imme

Not

基础指令	指令描述	指令格式
扩展为 xori	NOT 把寄存器 x[rs1]对于 1 的补码（按位取反的值）写到 x[rd]中	xori rd，rs1，-1

Csrs

基础指令	指令描述	指令格式
等同于 csrrs	Control and Status Register Set 对于 x[rs1]中的每一个为 1 的位，把控制状态寄存器 crs 的对应位清零	csrrs x0, csr, rs1

Csrsi

基础指令	指令描述	指令格式
等同于 csrrsi	Control and Status Register Set Immediate 对于五位的零扩展的立即数中的每一个为 1 的位，把控制状态寄存器 crs 的对应位清零	csrrsi x0, csr, zimm

Snez

基础指令	指令描述	指令格式
等同于 sltu	Set if Not Equal to Zero 如果 x[rs1]不等于 0，向 x[rd]写入 1，否则写入 0	sltu rd, x0, rs2

Bnez

基础指令	指令描述	指令格式
等同于 bne	Branch if Not Equal to Zero 不等于零时分支	bne rs1, x0, offset

Bgez

基础指令	指令描述	指令格式
等同于 bge	Branch if Greater Than or Equal to Zero 大于等于零时分支	bge rs1, x0, offset

结合伪指令转成的基础指令可以看出，该测试程序共有 49 条基础指令。对比我们之前设计的指令集，我们增加了 fsd、fld、fmul.d、fdiv.d 这四条双精度浮点数指令。

同时，分析 Fibonacci.out 文件我们发现，部分指令的机器码有时是 4 位，有时 8 位。对应的指令位数是 16 位和 32 位。如图 1 所示。经过实验可以确定，指令的位数根据编译器开启的拓展不同而变化。开启编译器的 c 拓展，部分指令有 16 位和 32 位两种形式。一旦关闭编译器的 c 拓展，部分指令就只有 32 位一种形式了。分析其设计目的是为了压缩指令存

储。

图 1 指令 li 对应的机器码的位数不同

所以，我们在原有指令集的基础上又增加了 24 条 16 位的指令，同时删去了一些多余的指令。目前，4 条双精度浮点指令和这 24 条 16 位指令还未经过测试。下一步实验将进行功能实现与测试。

重新调整后的指令集详见附件中 ISA/ISADefinitoin_v2.xlsx。

（三）5 级流水线 CPU 的优化改进

1、优化动机：上周实现了包含取指 IF、译码 ID、执行 EX、访存 MEM、写回 WB 的 5 级流水线，并成功运行了手写 RV 汇编指令的 Fibonacci 程序。但上周实现过程中，存在如下问题：

（1）我们首先确定了整体架构并绘制出简单数据通路，但在具体实现时发现设计思路的一些细节问题，以至于真正代码的编写与初步设计有很大出入，测试调试过程中不断修改代码，导致最后实际实现工程的代码逻辑有些混乱，如有些模块的功能冗余或者部分信号重复等。

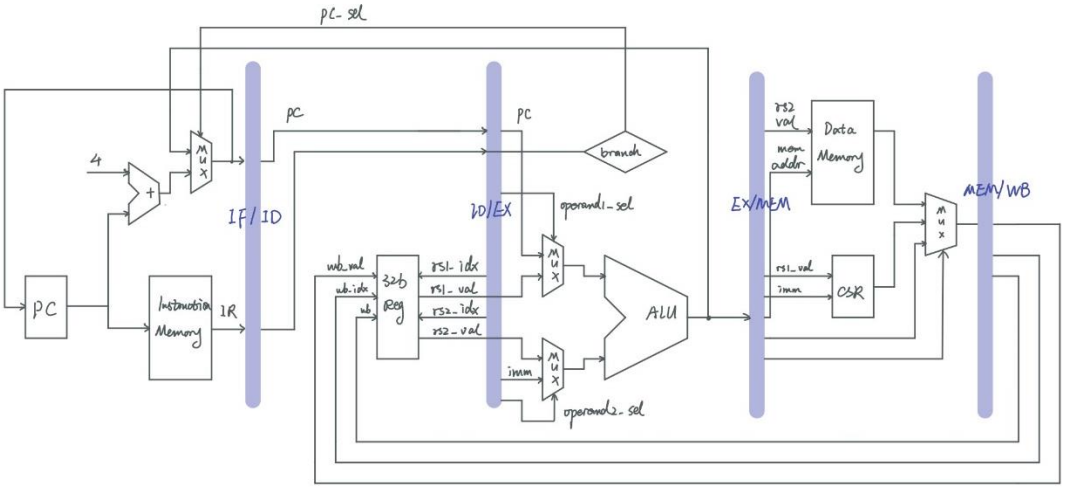
（2）代码编写过程中只专注于能通过仿真实现功能，而没有考虑到最后在硬件映射上的一些问题，如在译码阶段直接使用单级译码，相比多级译码而言，布局布线等更复杂，对硬件不够友好。

（3）上周实现的目标指令集为 RV32I，本周增加实现 RVM 指令集（乘、除、取余共 8 条指令）。

2、5 级流水线 CPU 实现

（1）5 级流水线整体结构

如下图所示为整体实现的基本架构，5 级流水线为取指 IF、译码 ID、执行 EX、访存 MEM 和写回 WB，每级为一模块，每个模块固定包含时钟 clk、重启 rst、开始 start 三个输入信号和一个表示下一级是否可执行的 next_ena 输出信号。最终由顶层文件 TOP 将各级模块互联。



（2）IF 取指

根据当前 PC 从程序存储器中取出对应指令，并通过 PC 选择信号确定下一条指令的 PC 值为 PC+4 或是分支跳转指令后的 PC。

信号名	功能
input	
PC[31:0]	当前需取指的 PC 值
PC_sel	PC 选择信号，下一条指令选择 PC+4 或是由 EX 传来的分支指令
PC_jump[31:0]	EX 传来的分支跳转后的指令 PC 值
data_conflic	有无数据冲突
flush	是否需要冲刷本级操作
output	
next_PC[31:0]	下一条指令的 PC 值
instruction[31:0]	以当前 PC 从程序存储器中取出的指令

（3）ID 译码

译码阶段主要功能是实现对 IF 取得的指令译码其操作数及其操作，在上周的实现时，译码过程为单级译码，即直接对每条指令一一匹配确定其相关的功能及参数，没有考虑到综合实现的难度及复杂度。因此本周对译码阶段重新实现，通过多级译码，对指令的不同字段逐步分析出指令的类型等，依次确定出指令的功能等。

信号名	功能
input	
next_PC[31:0]	由 IF 传来的下一条指令的 PC 值
IR[31:0]	指令寄存器，为 IF 取得的当前指令
wb_idx[4:0]	WB 传来的需要写回的寄存器的地址
wb_val[31:0]	WB 传来的需要写回的寄存器的值
wb	WB 传来的是否写回的选择信号
pc_sel	表示有无分支跳转需要冲刷本级
output	
rs1_val[31:0]	对当前指令译码后并从寄存器文件中得到的寄存器 rs1 的值
PC[31:0]	当前指令的 PC 值
operand1_sel	操作数 1 的选择信号，为 0 选择 rs1 值，为 1 选择 PC 值

rs2_val[31:0]	对当前指令译码后并从寄存器文件中得到的寄存器 rs2 的值
imm[31:0]	经过指令要求的相关扩展后的 32 位立即数
operand2_sel	操作数 2 的选择信号，为 0 选择 rs2 值，为 1 选择 imm 值
rd_idx[4:0]	对当前指令译码出的写回寄存器 rd 的下标
op_type[4:0]	对当前指令译码出的操作类型，可详见 opType.vh 头文件
alu_type[3:0]	对当前指令译码出的 alu 操作类型，可详见 opType.vh 头文件
csr_idx[11:0]	对 csr 指令译码出的 csr 寄存器下标
data_conflict	译码检测当前指令是否与之前已在执行的指令有数据冲突
flush	是否有数据冲突需要冲刷前两级操作

（4）EX 执行

执行阶段主要是通过两个多路选择器对两个操作数进行选择，分别有 rs1 op rs2、rs1 op imm、PC op imm 和不做计算（lui）搭配，通过译码的 alu_type 选择进行的计算操作类型，最后再通过 op_type 对特殊的指令处理，如判断分支指令是否跳转等。

信号名	功能
input	
rs1_val[31:0]	由 ID 传来的寄存器 rs1 的值
PC[31:0]	由 ID 传来的 PC 的值
operand1_sel	由 ID 传来的操作数 1 的选择信号
rs2_val[31:0]	由 ID 传来的寄存器 rs2 的值
imm[31:0]	由 ID 传来的 imm 的值
operand2_sel	由 ID 传来的操作数 2 的选择信号
rd_idx[4:0]	由 ID 传来的目标写回寄存器的下标
op_type[4:0]	由 ID 传来的指令操作类型，可详见 opType.vh 头文件
alu_type[3:0]	由 ID 传来的指令 alu 操作类型，可详见 opType.vh 头文件
csr_idx[11:0]	由 ID 传来的 csr 寄存器的下标
output	
rs1_val_out[31:0]	rs1 寄存器的值
PC_out[31:0]	PC 值
rs2_val_out[31:0]	rs2 寄存器的值
imm_out[31:0]	立即数 imm 的值
rd_idx_out[4:0]	目标写回寄存器 rd 的下标
op_type_out[4:0]	当前指令的操作类型
csr_idx_out[11:0]	csr 寄存器下标
ex_output[31:0]	alu 执行结果
new_pc[31:0]	对于跳转指令为目标分支的 PC 值
pc_sel	PC 是否跳转的选择信号
mask[3:0]	对于 store 指令，通过掩码 mask 表示写回一字节或半字或一字
store_ena	是否可以往数据存储器中存储数据

（5）MEM 访存

MEM 阶段主要执行对数据存储器的 load 和 store 操作。

信号名	功能
input	
ex_output[31:0]	由 EX 传来的 ALU 计算出的值
rd_idx[4:0]	目标写回寄存器的下标
op_type[4:0]	当前指令操作类型
rs1_val[31:0]	寄存器 rs1 的值
rs2_val[31:0]	寄存器 rs2 的值
imm[31:0]	立即数的值
PC[31:0]	当前 PC 值
csr_idx[11:0]	csr 相关寄存器的下标
mask[3:0]	用于辅助 store 指令功能的掩码
store_ena	是否对数据存储器存储
output	
rd_val[31:0]	确定最终写回寄存器的值（可能是 alu 执行出来的，可能是从存储器中 load 的值）
rd_idx_out[4:0]	写回寄存器的下标
op_type_out[4:0]	指令操作类型
rs1_val_out[31:0]	rs1 的值
imm_out[31:0]	立即数的值
csr_idx_out[11:0]	csr 寄存器下标

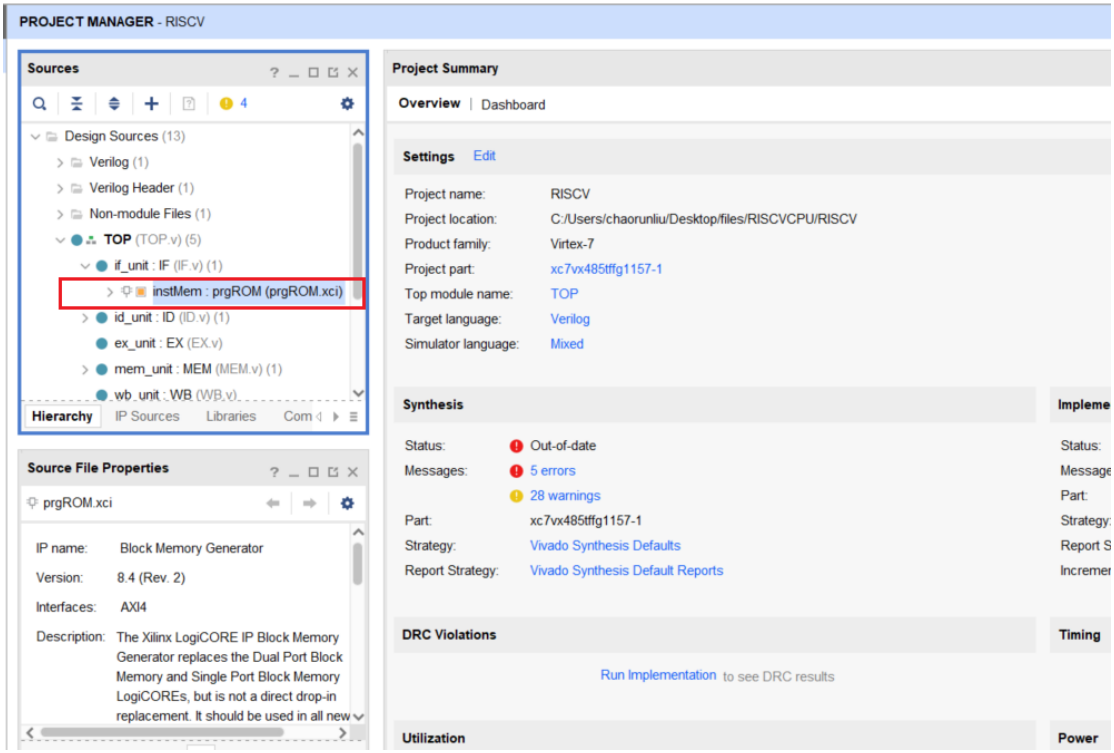
（6）WB 写回

WB 主要是对目标寄存器 rd 的写回，以及对 csr 寄存器相关的操作。

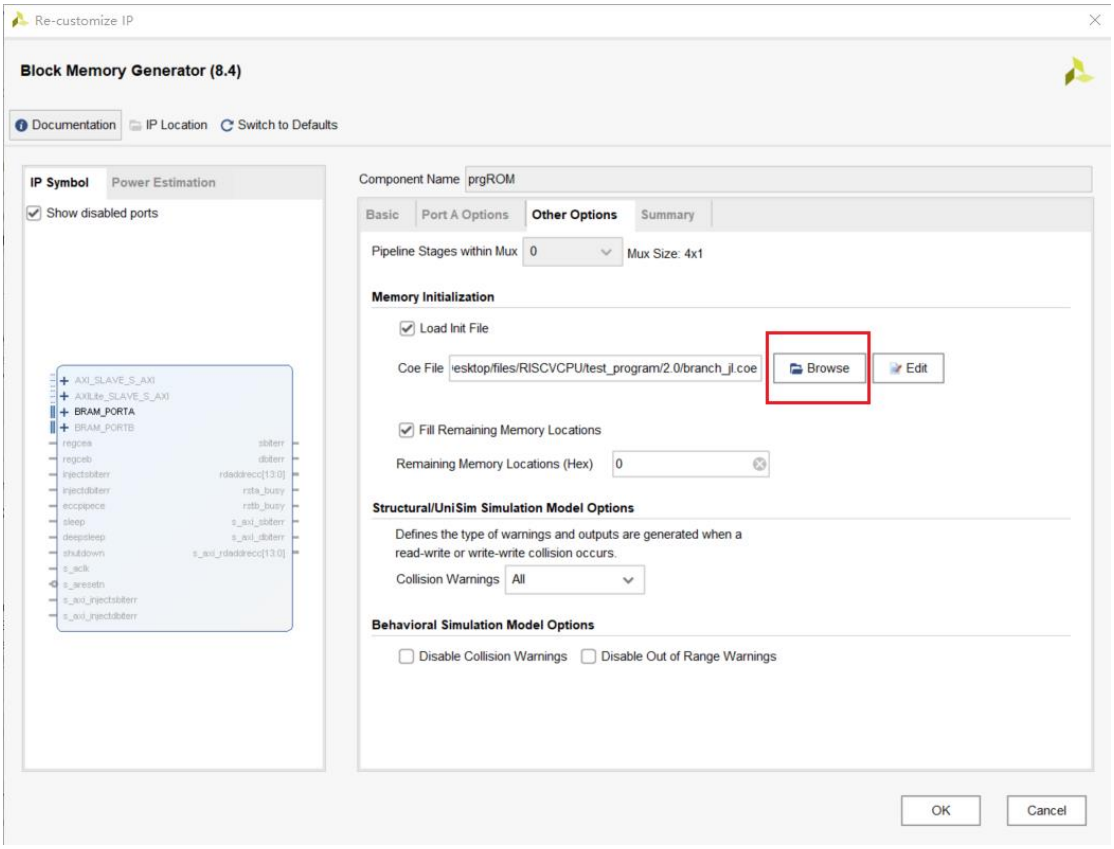
信号名	功能
input	
rd_val[31:0]	需要写回 rd 寄存器的值
rd_idx[4:0]	写回 rd 寄存器的洗标
op_type[4:0]	当前指令操作类型
rs1_val[31:0]	寄存器 rs1 的值，用于 csr 相关操作
imm[31:0]	立即数的值，用于 csr 相关操作
csr_idx[11:0]	csr 寄存器下标
output	
wb	是否需要写回寄存器的选择信号
wb_idx[4:0]	写回寄存器的下标
wb_val[31:0]	写回寄存器的值

四、实验结果

为了方便测试和看出指令的执行结果，我们将需要实现的指令分为 6 类进行测试，分别是 Arithmetic、Shifts、Logical、compare&environment%csr、branch&jl、mul&div&rem。实验使用 vivado 2018.3。指令寄存器使用 IP 核载入方式如下：双击如图 5.1 中的 instMem，选择 other options 中的 Browser，加载 coe 文件。然后点击 ok 和 generate。实验使用的 coe 文件均在本次实验测试程序及代码文件夹下。仿真工程文件在 RISCV 文件夹下。



选择 ip 核



加载 coe

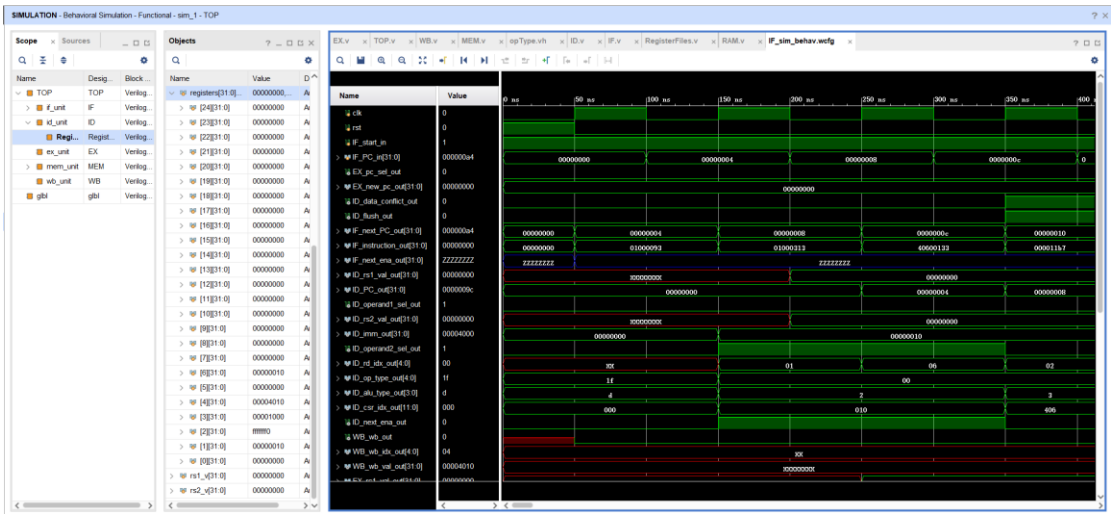
（一）测试算数指令

Arithmetic 测试的指令如图 5.3 所示，使用 riscv-gnu-tools 将其转成二进制码。选

择加载 arithmetic.coe 文件，运行仿真，仿真结果如图 5.4。从图中寄存器的值可以看出所有指令均正常。

```
2       #Arithmetic
3       ADDI    x1,x0,0x10
4       ADDI    x6,x0,0x10
5       SUB     x2,x0,x6
6       LUI     x3,0x1
7       AUIPC   x4,0x4
8       ADD     x5,x1,x2
```

arithmetic 测试指令



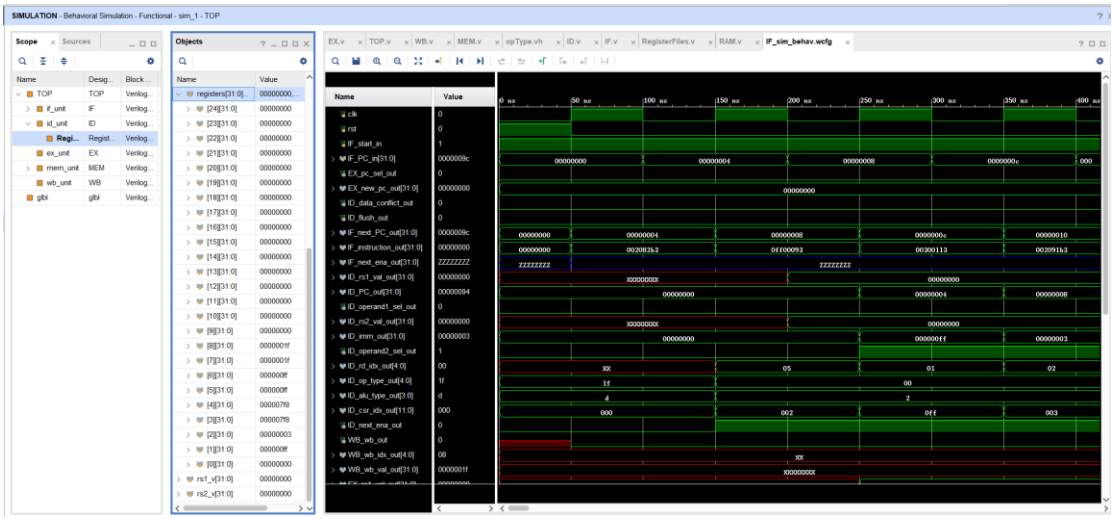
arithmetic 测试指令仿真结果

（二）测试 shifts 指令

Shifts 测试指令如图 5.5 所示，使用 riscv-gnu-tools 将其转成二进制码。选择加载 shifts.coe 文件，运行仿真，仿真结果如图 5.6。从仿真结果图中可以看出，所有寄存器的值均正确，说明 shifts 指令正常。

```
9       #Shifts
10      ADDI    x1,x0,0xff
11      ADDI    x2,x0,0x3
12      SLL     x3,x1,x2
13      SLLI    x4,x1,3
14      SRL     x5,x3,x2
15      SRLI    x6,x4,3
16      SRA     x7,x1,x2
17      SRAI    x8,x1,3
```

shifts 测试指令



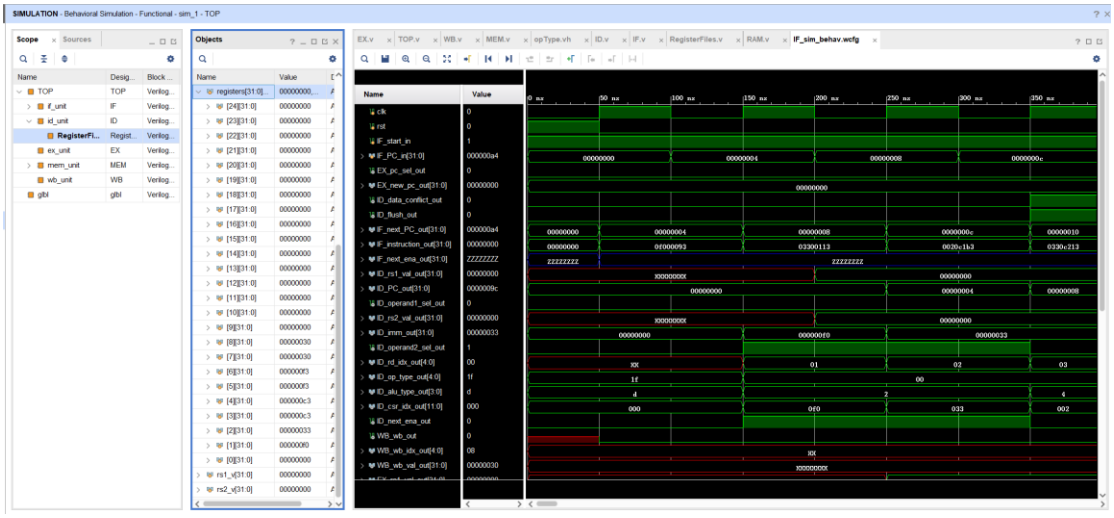
shifts 测试指令仿真结果

（三）测试 logical 指令

Logical 测试指令如图 5.7 所示，使用 riscv-gnu-tools 将其转成二进制码。选择加载 logical.coe 文件，运行仿真，仿真结果如图 5.8。从仿真结果图中可以看出，所有寄存器的值均正确，说明 logical 指令正常。

```
18      #Logical
19      ADDI    x1,x0,0xf0
20      ADDI    x2,x0,0x33
21      XOR     x3,x1,x2
22      XORI    x4,x1,0x33
23      OR      x5,x1,x2
24      ORI     x6,x1,0x33
25      AND     x7,x1,x2
26      ANDI    x8,x1,0x33
```

logical 测试指令



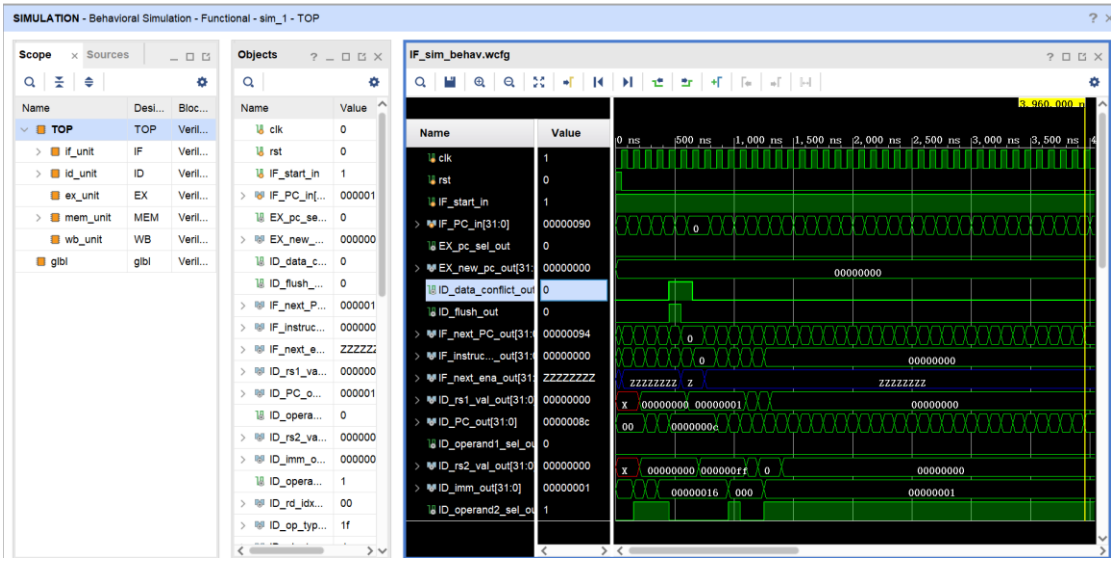
logical 测试指令仿真结果

（四）测试 compare environment csr 指令

compare environment csr 测试指令如图 5.9 所示，使用 riscv-gnu-tools 将其转成二进制码。选择加载 compare_environment_csr.coe 文件，运行仿真，仿真结果如图 5.10。从仿真结果图中可以看出，所有寄存器的值均正确，说明 compare environment csr 指令正常。

```
27      #compare&eviroment%csr
28      ADDI    x1,x0,0x1
29      ADDI    x2,x0,0xff
30      ADDI    x8,x0,0x16
31      SLT     x3,x1,x2
32      SLTU    x4,x1,x2
33      SLTI    x5,x1,0x2
34      ecall
35      csrrs   x6,fflags,x8
36      csrrsi  x7,fflags,0x16
```

compare environment csr 测试指令



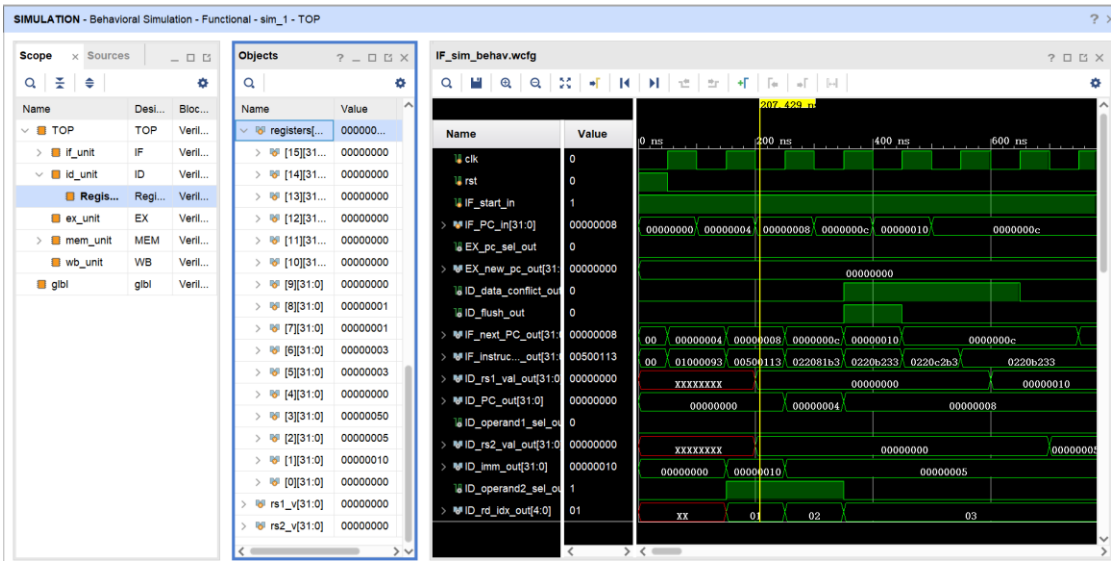
compare environment csr 测试指令仿真结果

（五）测试 mul div rem 指令

mul div rem 测试指令如图 5.11 所示，使用 riscv-gnu-tools 将其转成二进制码。选择加载 mul_div_rem.coe 文件，运行仿真，仿真结果如图 5.12。从仿真结果图中可以看出，所有寄存器的值均正确，说明 mul div rem 指令正常。

```
37      #mul&div&rem
38      ADDI x1,x0,0x10
39      ADDI x2,x0,0x5
40      MUL  x3,x1,x2
41      MULHU x4,x1,x2
42      DIV  x5,x1,x2
43      DIVU x6,x1,x2
44      REM  x7,x1,x2
45      REMU x8,x1,x2
```

mul div rem 测试指令



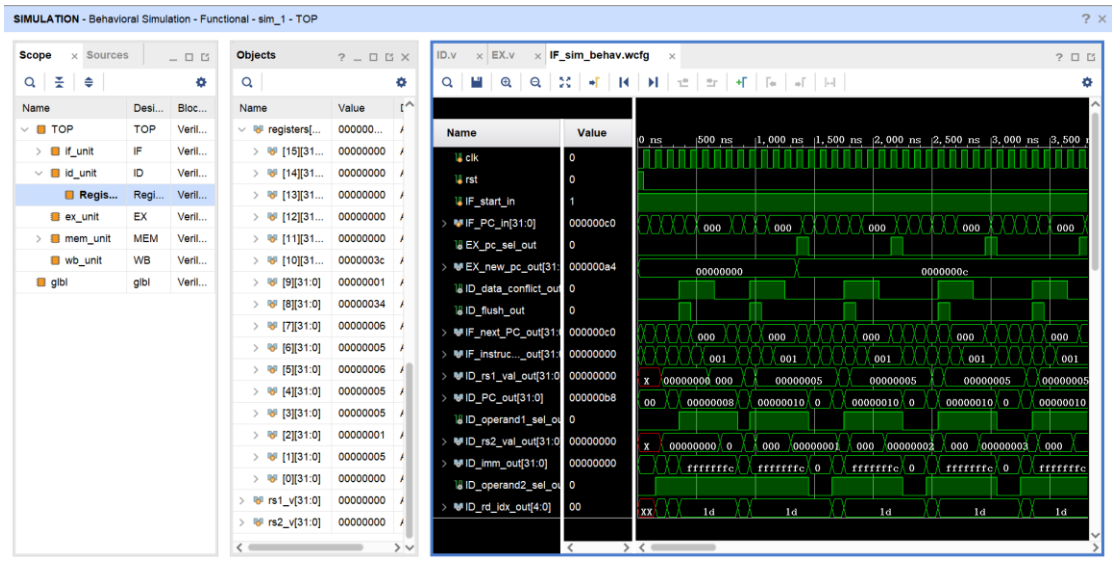
mul div rem 测试指令仿真结果

（六）测试 branch jl 指令

branch jl 测试指令如图 5.13 所示，使用 riscv-gnu-tools 将其转成二进制码。选择加载 branch_jl.coe 文件，运行仿真，仿真结果如图 5.14。从仿真结果图中可以看出，所有寄存器的值均正确，说明 branch jl 指令正常。

```
.main:
    #branch&jl
    ADDI x1,x0,5
    ADDI x2,x2,1
    BEQ  x1,x2,-4
    ADDI x3,x3,1
    BNE  x1,x3,-4
    ADDI x4,x4,1
    BLT  x4,x1,-4
    ADDI x5,x5,1
    BGE  x1,x4,-4
    ADDI x6,x6,1
    BLTU x6,x1,-4
    ADDI x7,x7,1
    BGEU x1,x7,-4
    JAL   x8,4
    ADDI x9,x0,1
    jALR x10,4
    ADDI x10,x0,1
```

branch jl 测试指令



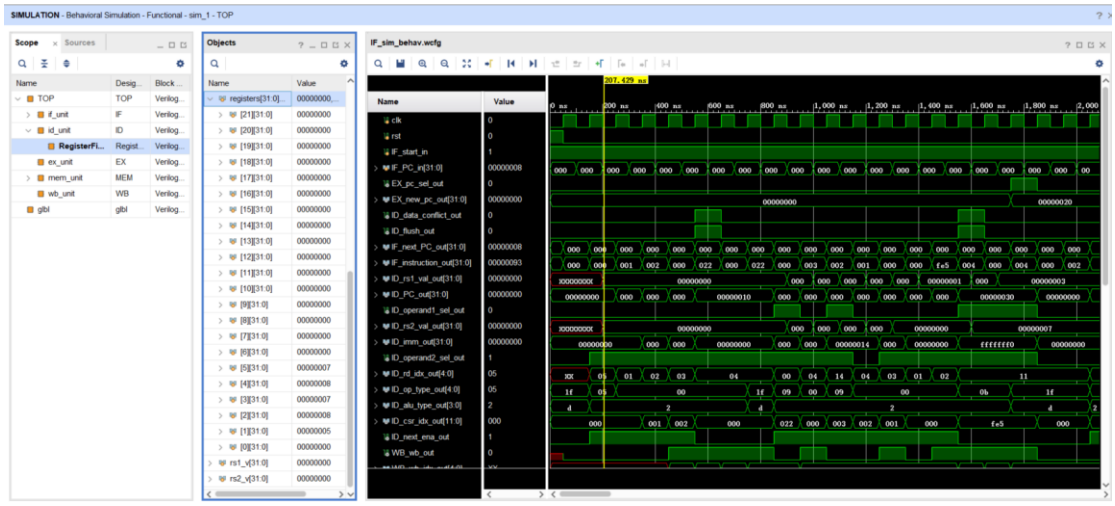
branch jl 测试指令仿真结果

（七）使用带有访存功能的 fibonacci 数列程序测试访存指令

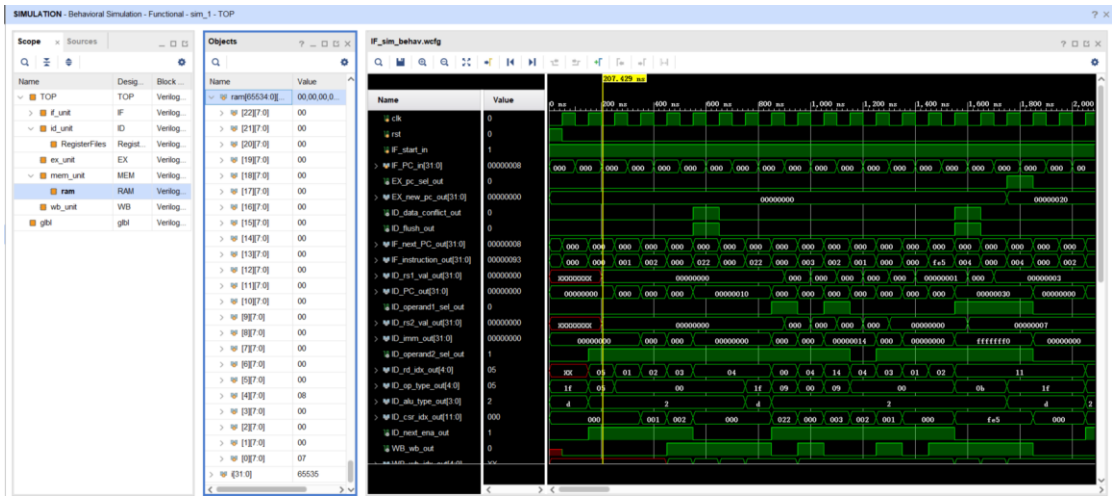
fibonacci 测试指令如图 5.15 所示，使用 riscv-gnu-tools 将其转成二进制码。选择加载 fibonacci.coe 文件，运行仿真，仿真结果如图 5.16 和图 5.17。从仿真结果图中可以看出，所有寄存器的值均正确并且内存中数值正常，说明 fibonacci 程序可以正常运行。

```
.main
    LW x5,x0,0        # n值 = Mem[0]
    ADDI x1,x0,0      # x1 = 0/i
    ADDI x2,x0,1      # x2 = 1/j
    ADDI x3,x0,2      # x3 = count = 2
    ADDI x4,x1,0
    BEQ x5,x2,32      # n == 1, jump to write x4
    ADDI x4,x2,0
    BEQ x5,x3,20      # n == 2, jump to write x4
    ADD x4,x1,x2      # x4 = fib = i + j
    ADDI x3,x3,1      # x3(count)++
    ADDI x1,x2,0      # i = j
    ADDI x2,x4,0      # j = fib
    BLT x3,x5,-12     # if count < n branch (PC already point
    to the PC+4) -8
    SW x0,x4,4        # Mem[1] = fib
```

fibonacci 数列和访存测试指令



fibonacci 数列和访存测试指令仿真测试结果-寄存器



fibonacci 数列和访存测试指令仿真测试结果-数据存储器

五、实验总结

（一）实验分工

许诗瑶：5 级流水线的优化改进代码的重新编写

刘朝润：工具链的研究使用、测试程序指令集的初步筛选、5 级流水线测试与调试

刘晓航：指令集的调整比较及指令细节的整理

（二）实验遇到的问题

1、译码阶段对指令分级译码需要提前将指令对应要求的信号一一整理出来后从大范围到小范围逐类区分，并且要保证每个指令在区分时有唯一辨识度，不会重复或错误译码

2、拓展 RVM 指令集时，需要注意到与 RV32I 直接 alu 计算不同，需要先将计算结果的寄存器扩展至 64 位存储运算结果，再根据指令具体内容与要求取 32 位值。

（三）附件文件说明

ISA 文件夹：包含上周与本周两次目标指令集文件

RISCV 文件夹：Vivado 环境下项目工程代码

本次实验测试程序代码文件夹

四个测试程序