**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**

1. **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。

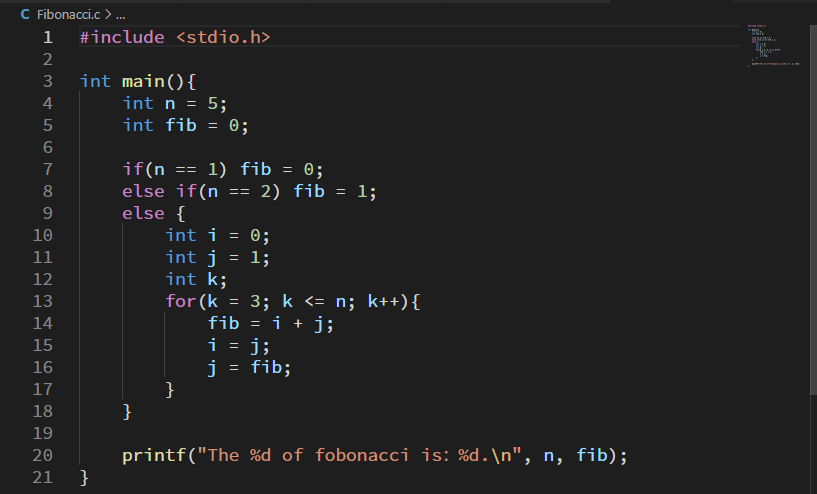
1. 进入riscv-gnu-tool文件夹

（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。

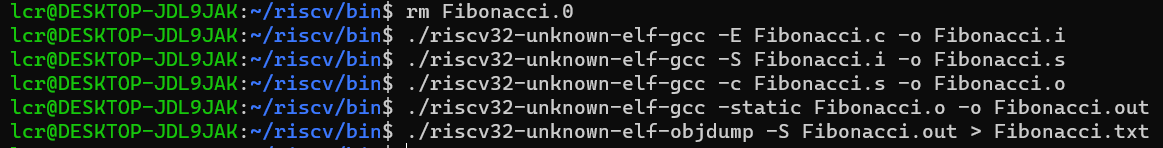
（5）编译：make，等待完成后进入/home/lcr/riscv即可使用riscv编译工具链。

**2、Fibonacci by C**

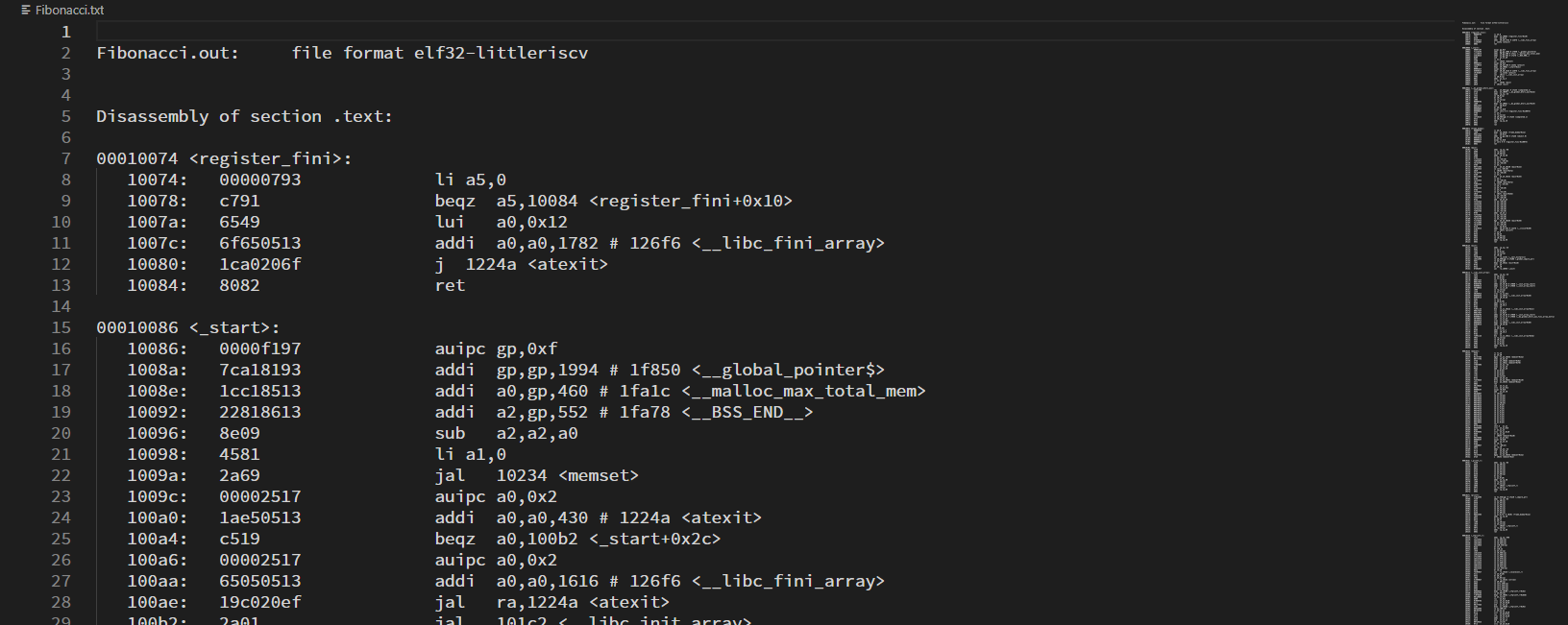
使用c语言编写带有一条printf语句的Fibonacci程序如四个测试程序文件加下的Fibonacci.c文件，内容如图：



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。

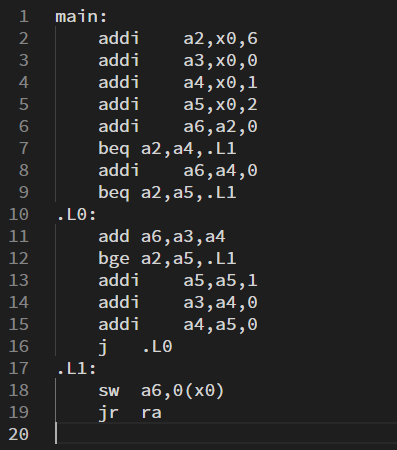


最终生成的结果如下图，文件为四个测试程序文件夹下的Fibonacci.txt，中间生成文件也在该文件夹下。



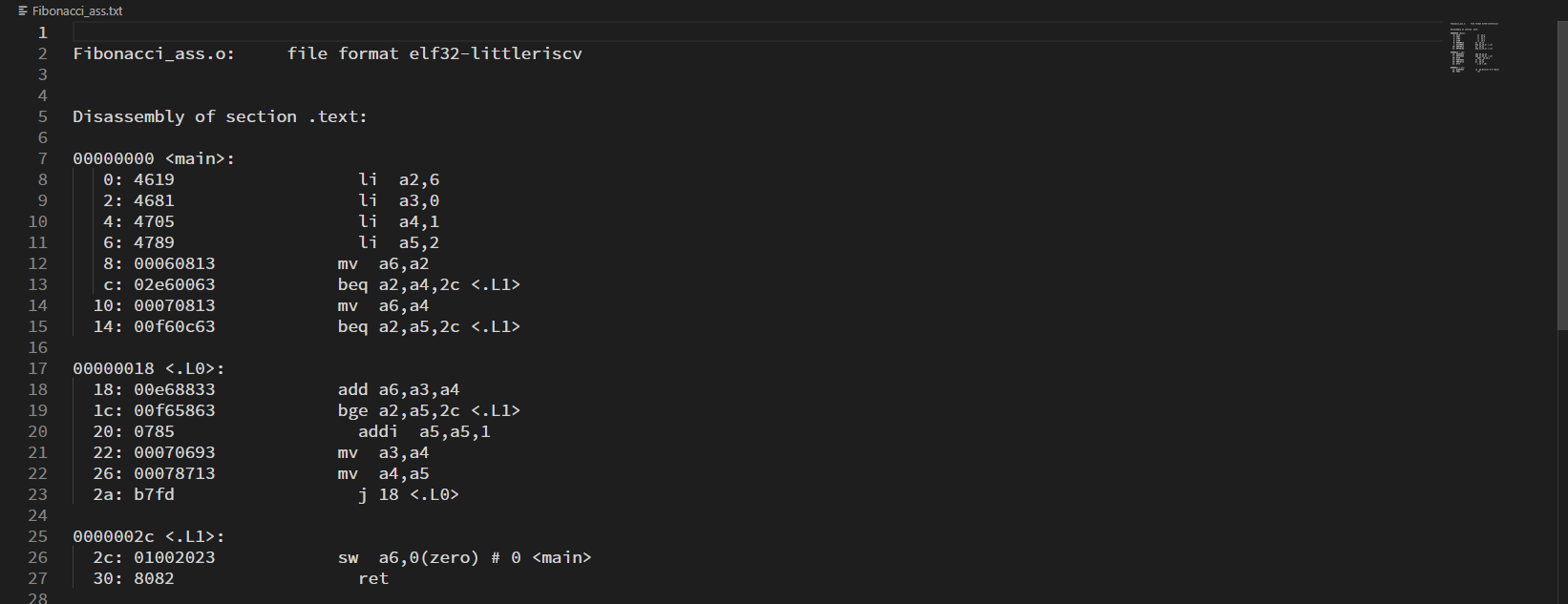
**3、Fibonacci by RV**

使用RV汇编编写的Fibonacci程序如下图。



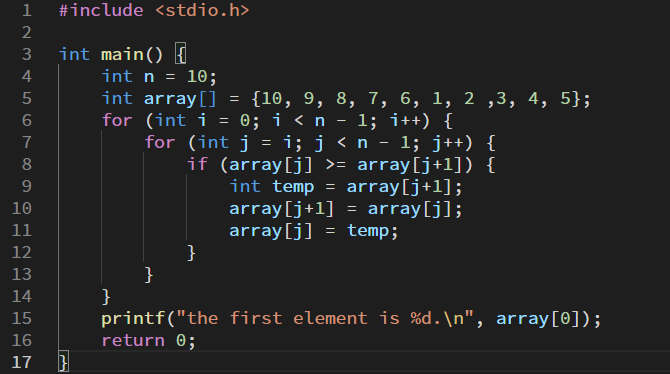
使用工具链对其进行编译生成可执行文件，并对可执行文件反汇编得到汇编指令。文件分别为四个测试程序文件夹下Fibonacci\_ass.s和Fibonacci\_ass.txt。



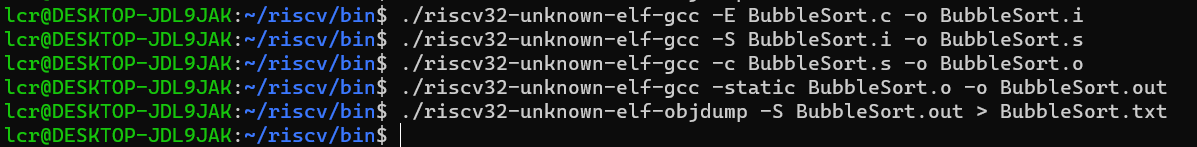


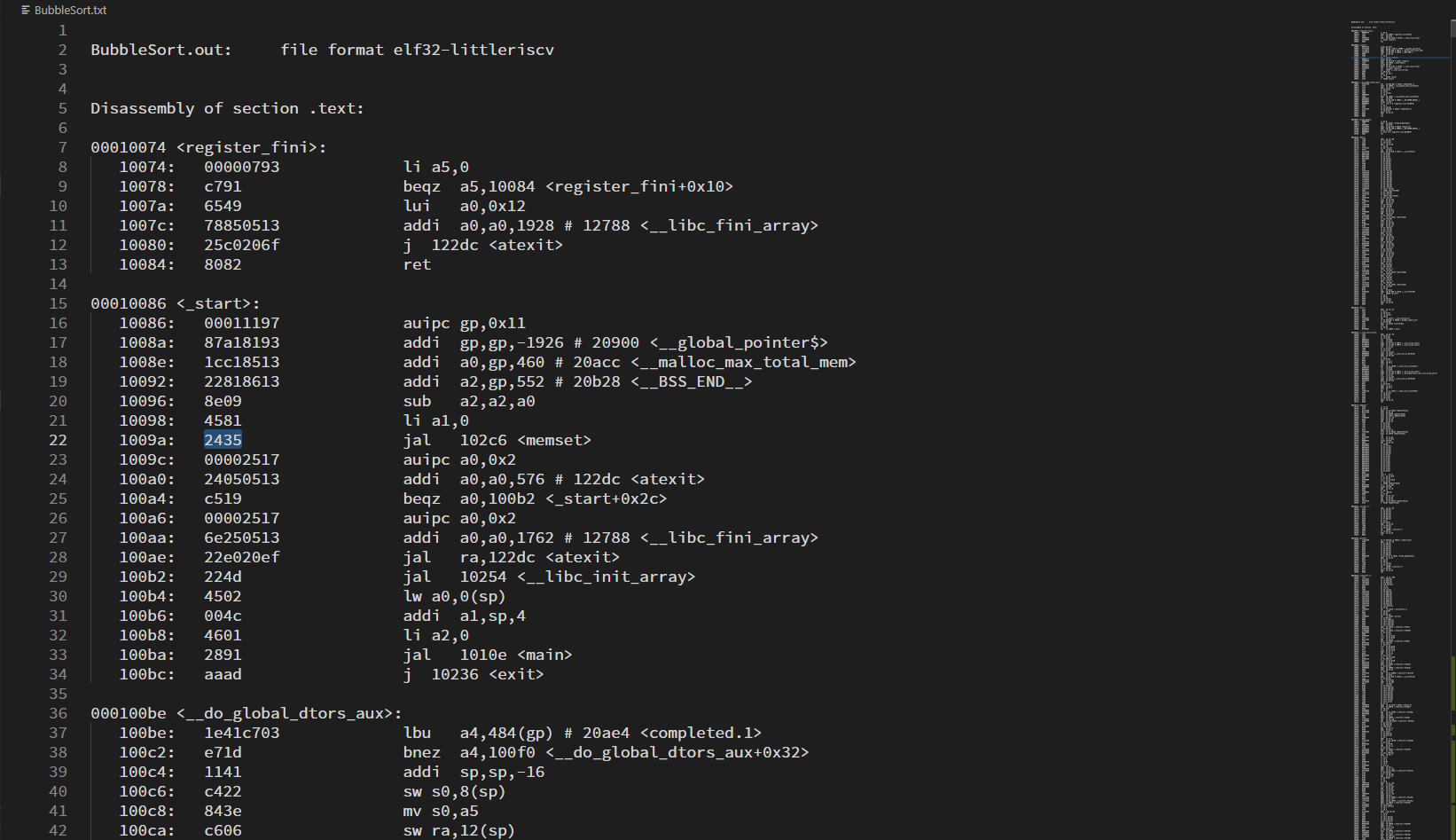
**4、Bubble Sort by C**

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：



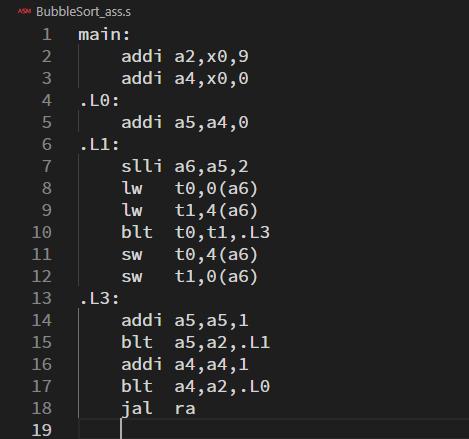
使用工具链进行编译，获得可执行文件，并将可执行文件反汇编获得汇编指令，其命令和结果如下图。文件为四个测试程序下的BubbleSort.txt。中间结果也在该文件夹下。





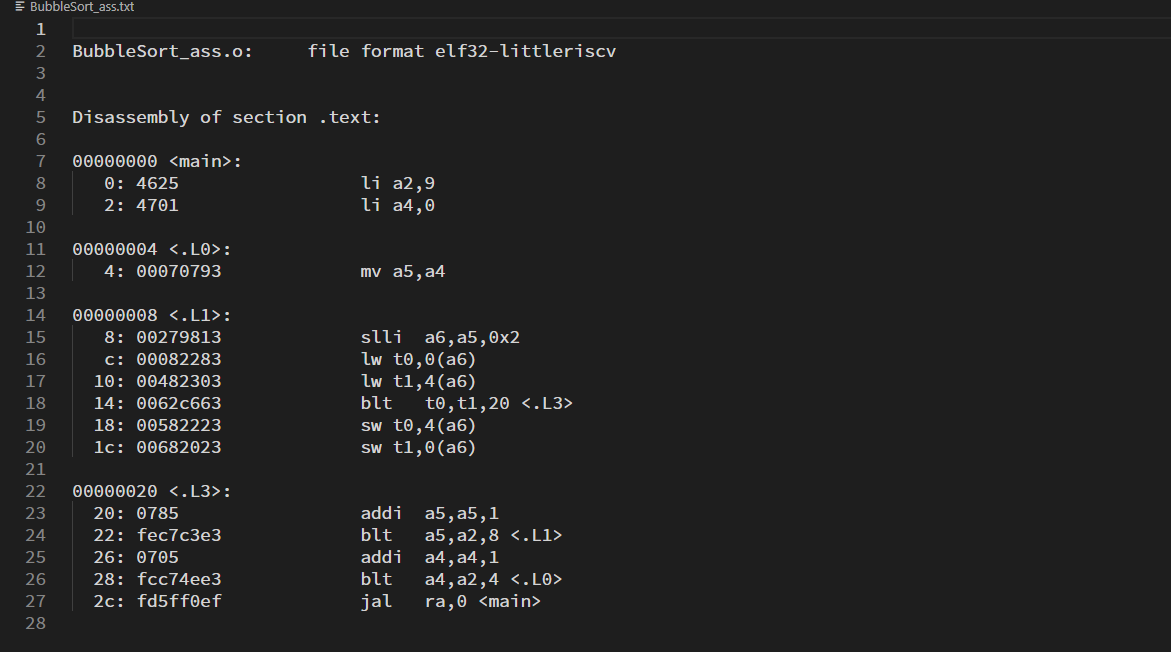
**5、Bubble Sort by RV**

使用RV汇编编写的Bubble Sort程序如下图。



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。





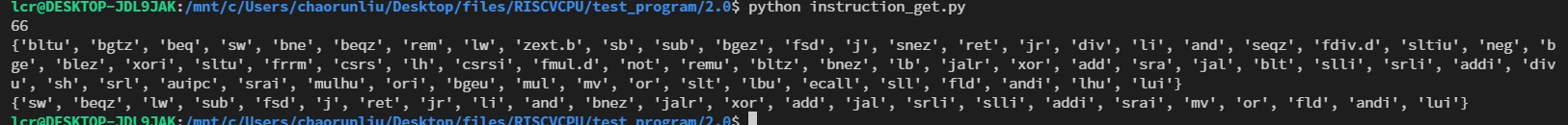
**（二）重新确定目标指令集**

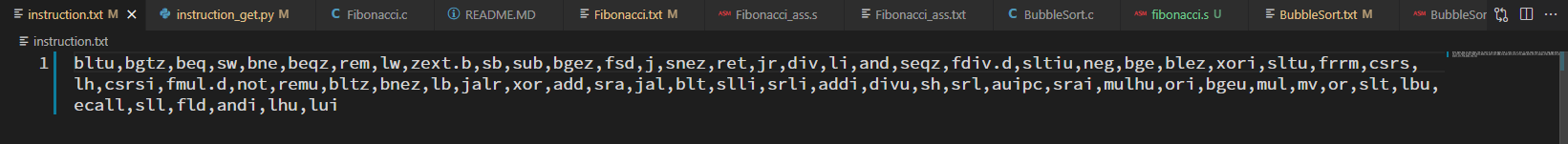
**1、上周初步设计的指令集**

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。

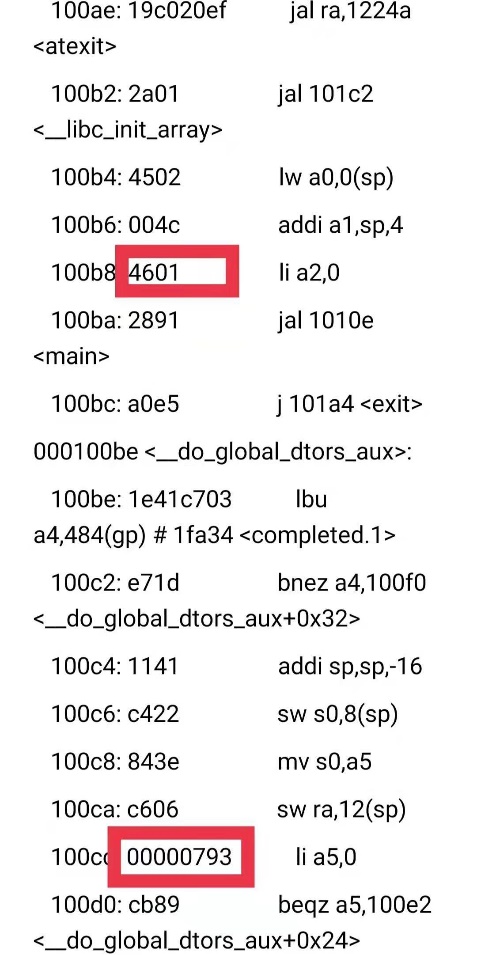


图1 指令li对应的机器码的位数不同

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。

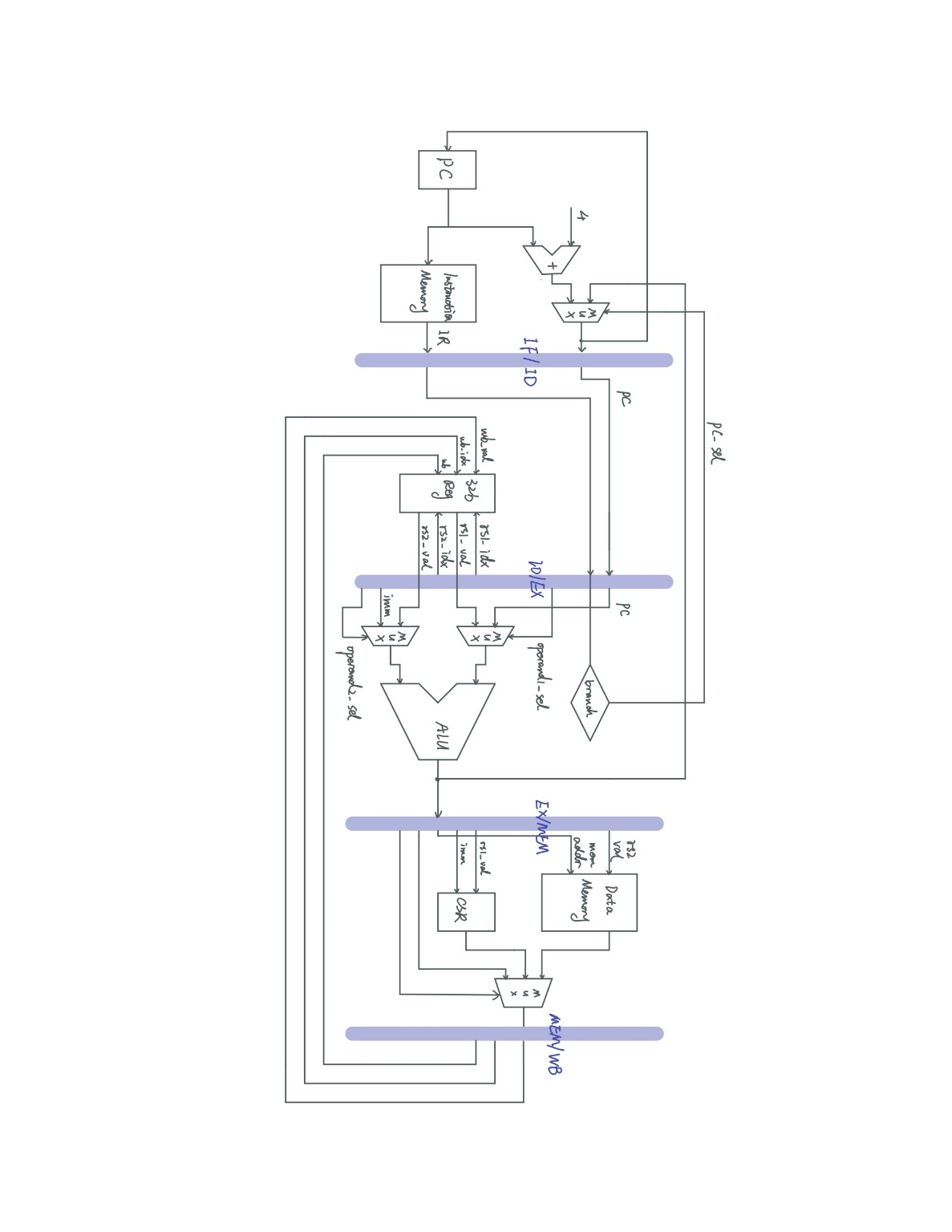
（2）代码编写过程中只专注于能通过仿真实现功能，而没有考虑到最后在硬件映射上的一些问题，如在译码阶段直接使用单级译码，相比多级译码而言，布局布线等更复杂，对硬件不够友好。

（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 |

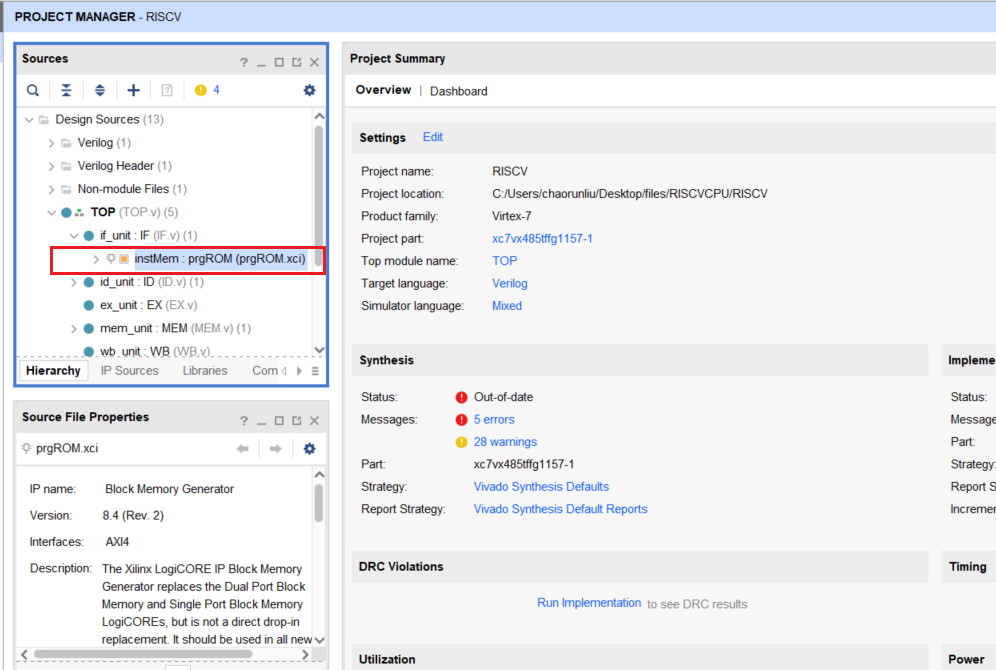
（6）WB写回

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。

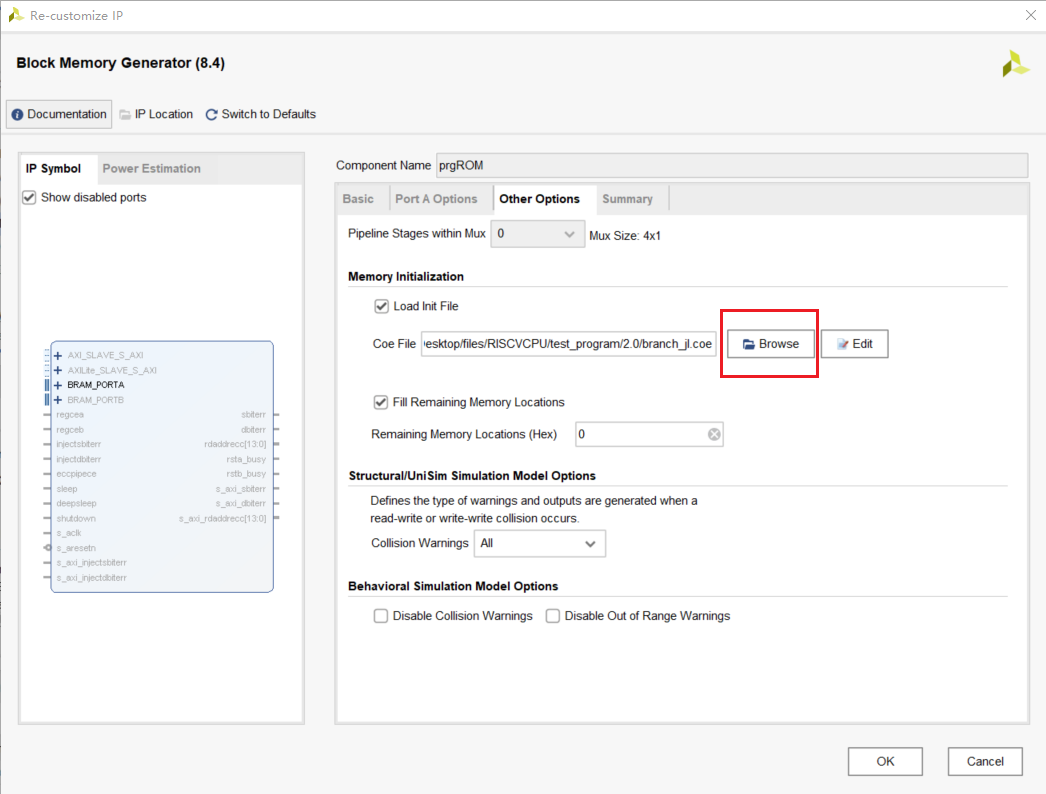
|  |  |
| --- | --- |
| **信号名** | **功能** |
| **input** | |
| rd\_val[31:0] | 需要写回rd寄存器的值 |
| rd\_idx[4:0] | 写回rd寄存器的洗标 |
| op\_type[4:0] | 当前指令操作类型 |
| rs1\_val[31:0] | 寄存器rs1的值，用于csr相关操作 |
| imm[31:0] | 立即数的值，用于csr相关操作 |
| csr\_idx[11:0] | csr寄存器下标 |
| **output** | |
| wb | 是否需要写回寄存器的选择信号 |
| wb\_idx[4:0] | 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。



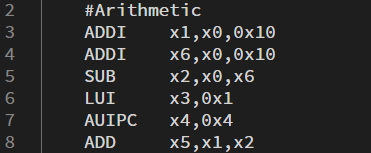
选择ip核



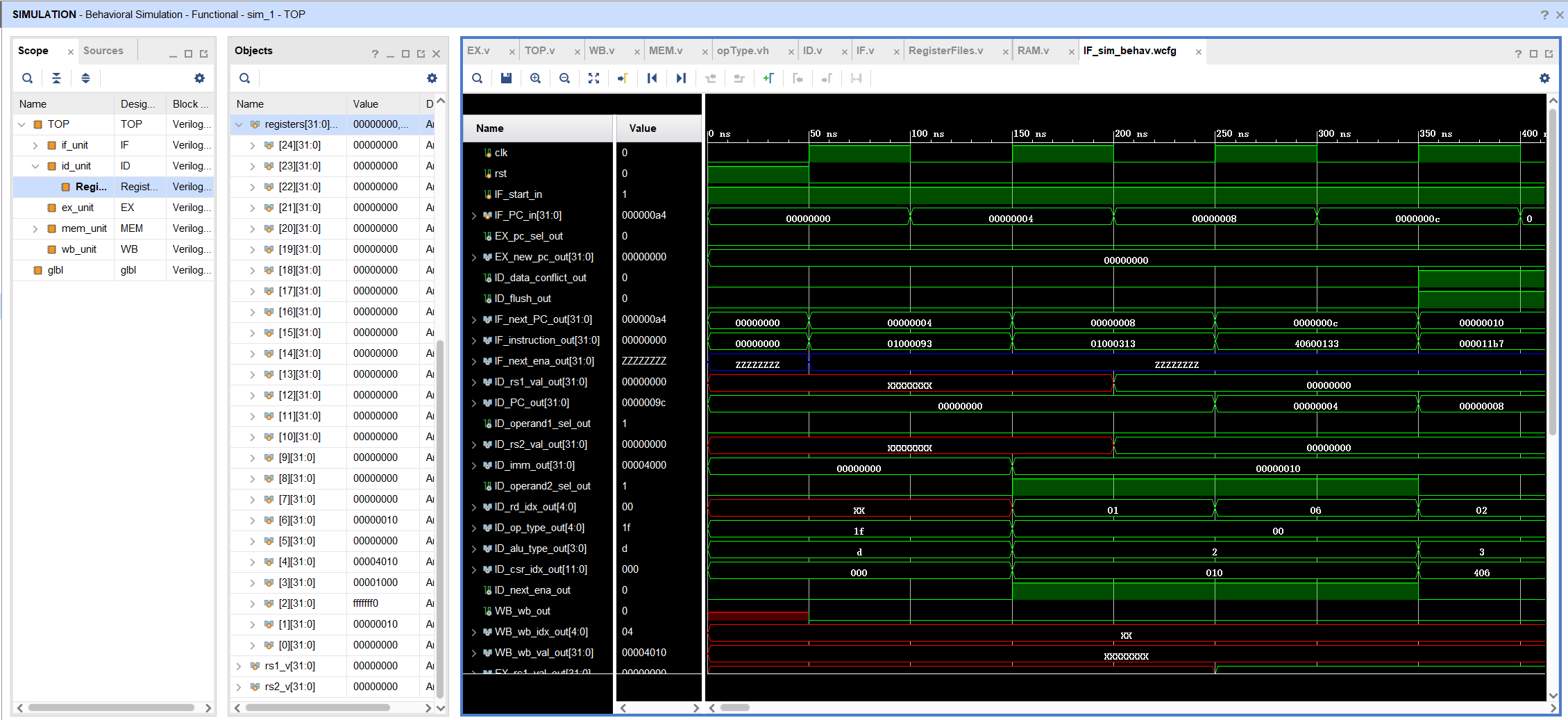
加载coe

**（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。



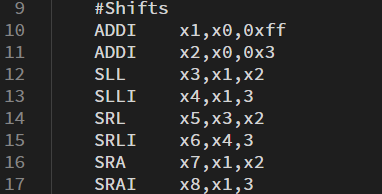
arithmetic测试指令



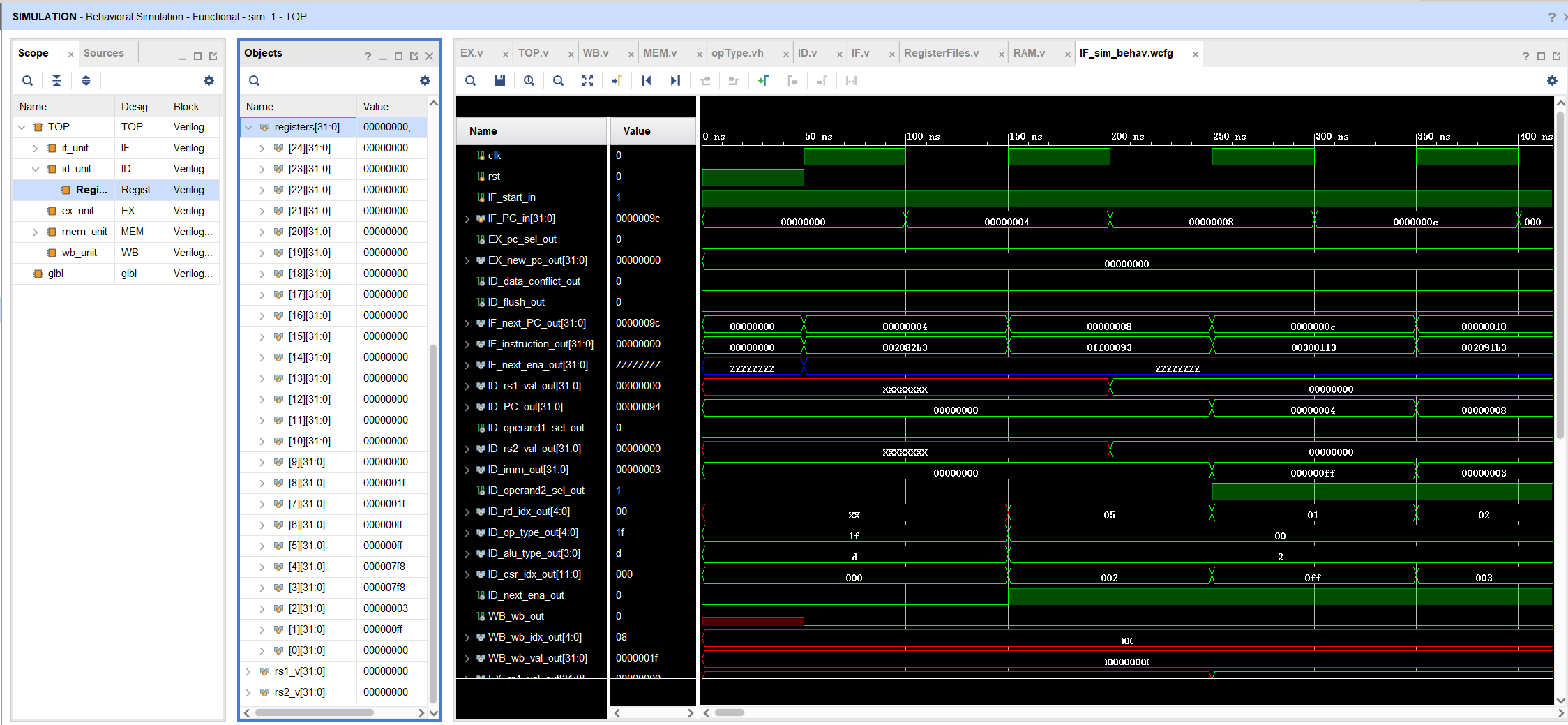
arithmetic测试指令仿真结果

**（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。



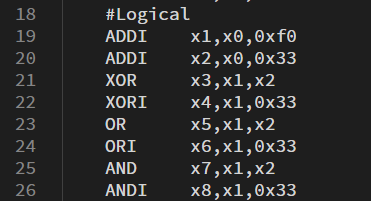
shifts测试指令



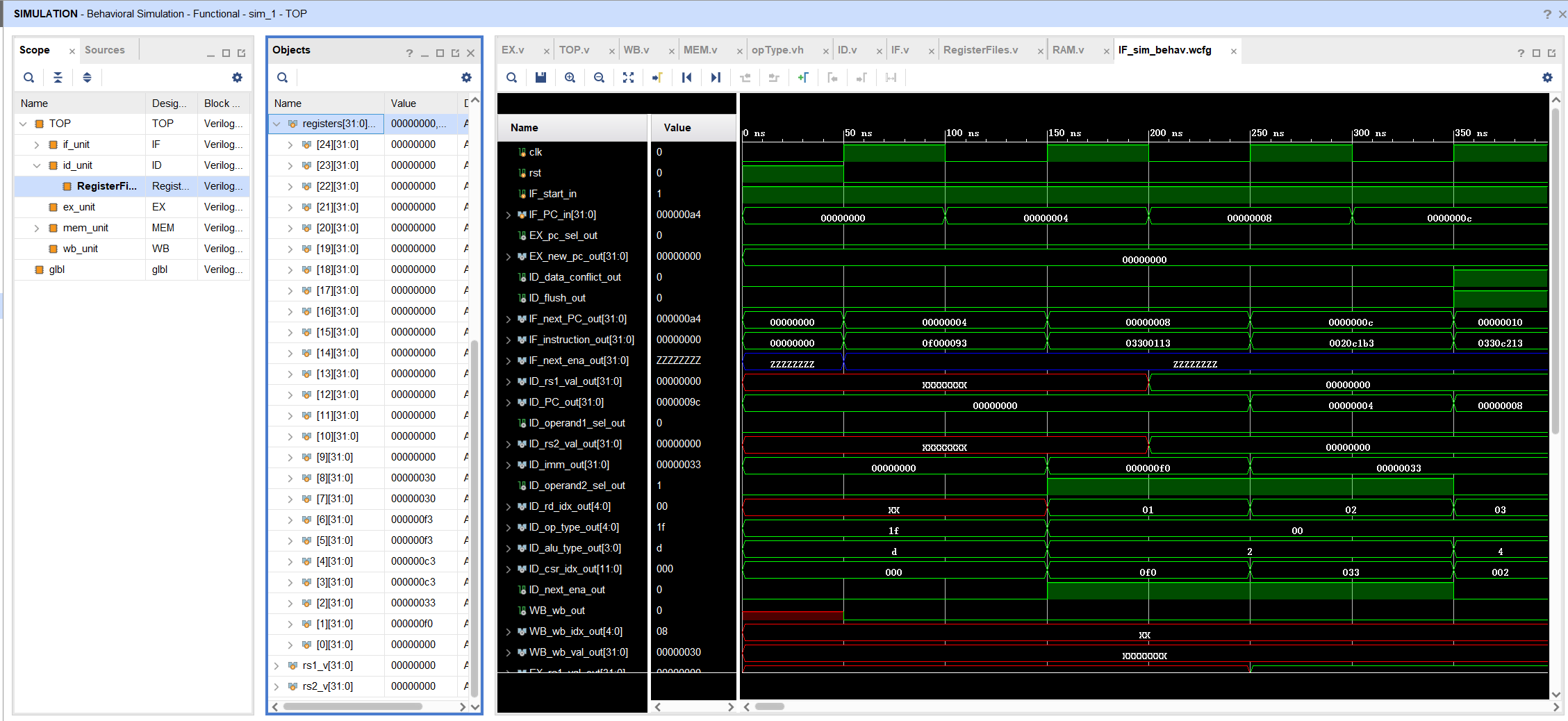
shifts测试指令仿真结果

**（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。



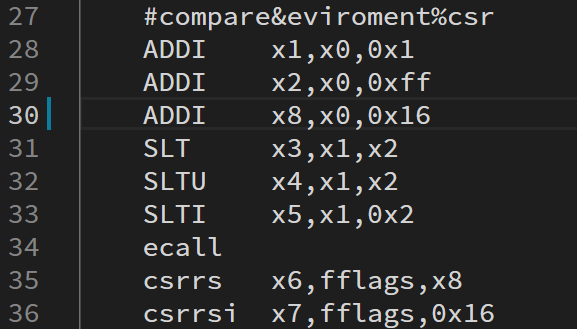
logical测试指令



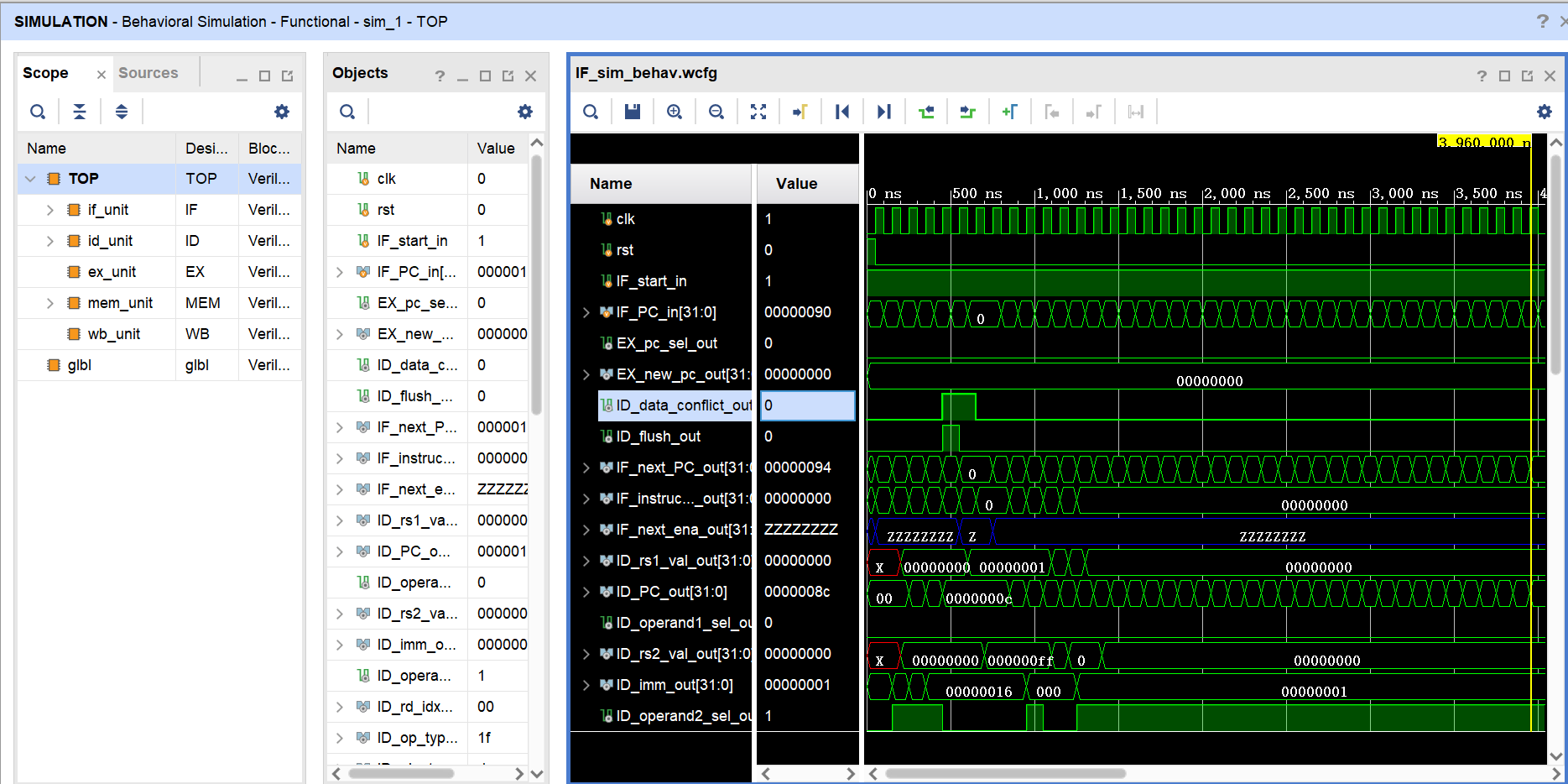
logical测试指令仿真结果

**（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。



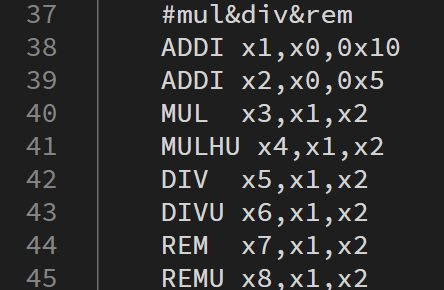
compare environment csr测试指令



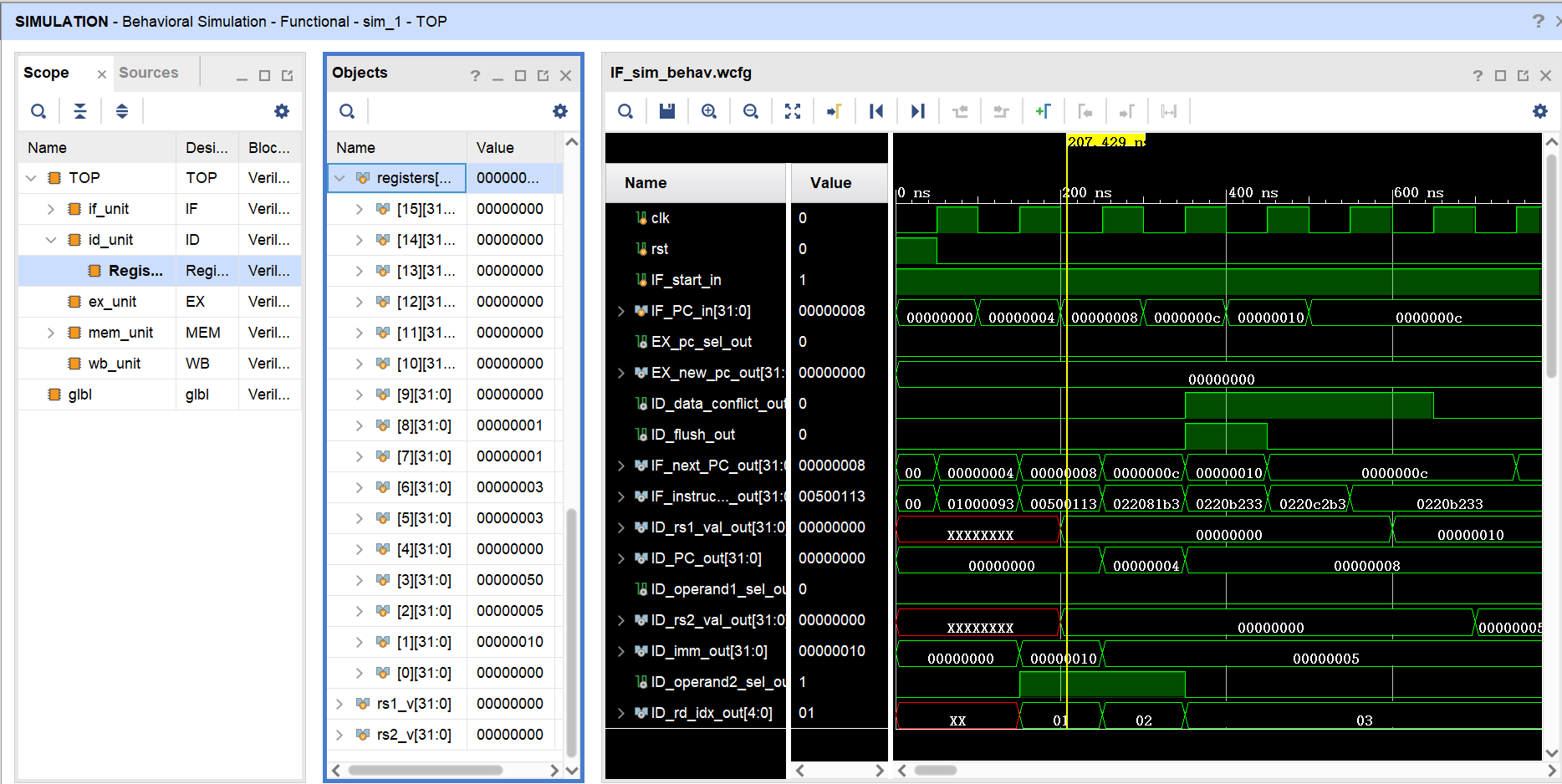
compare environment csr测试指令仿真结果

**（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。



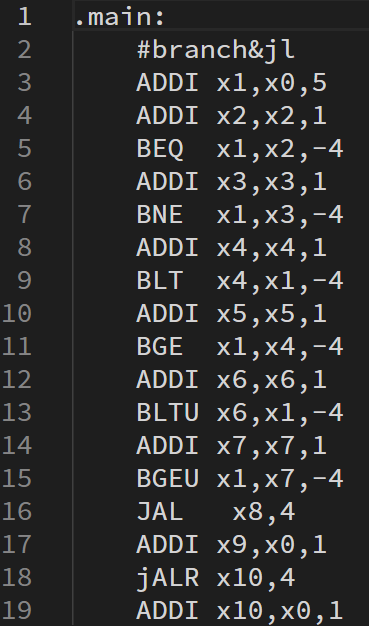
mul div rem测试指令



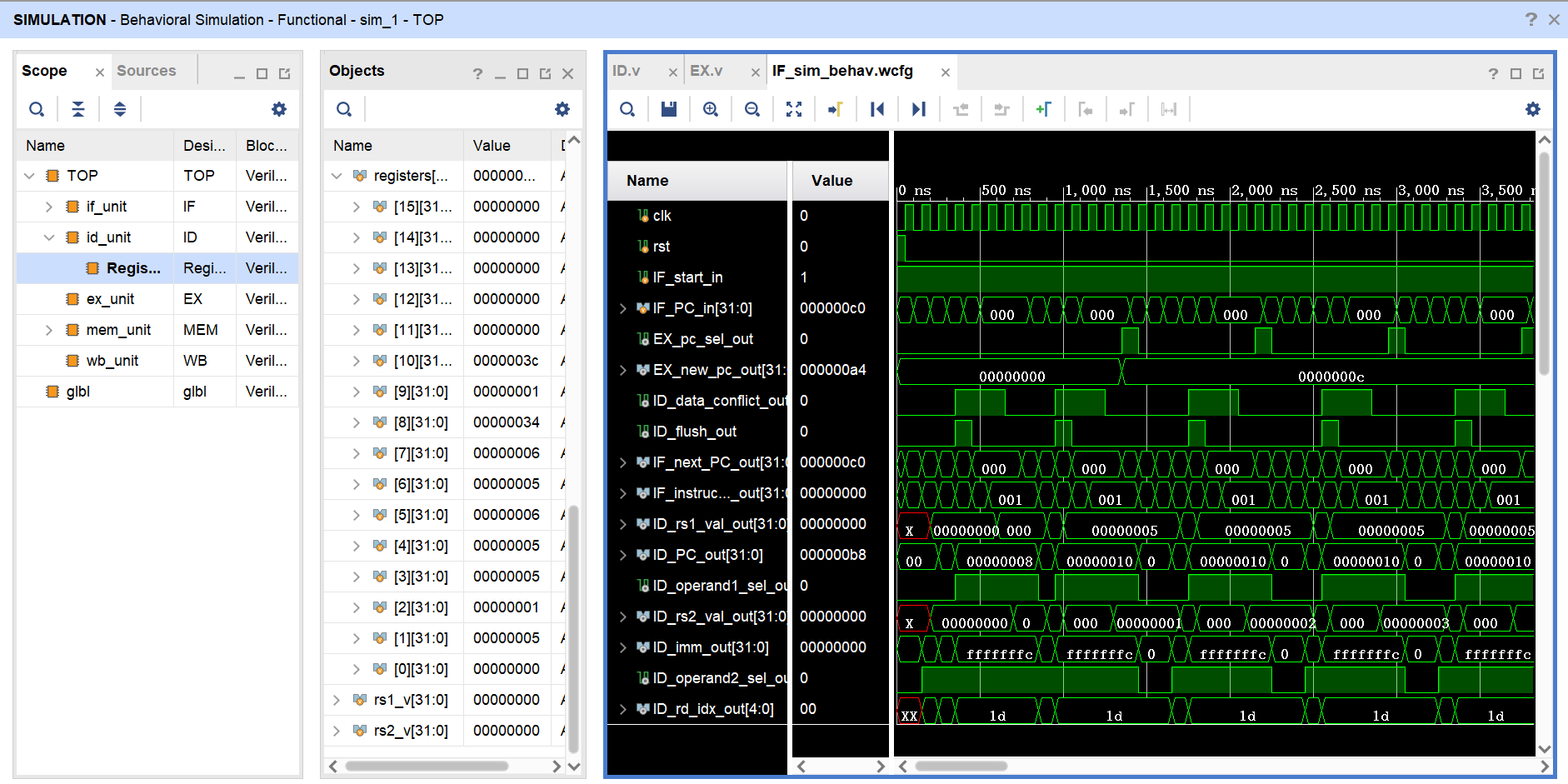
mul div rem测试指令仿真结果

**（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。



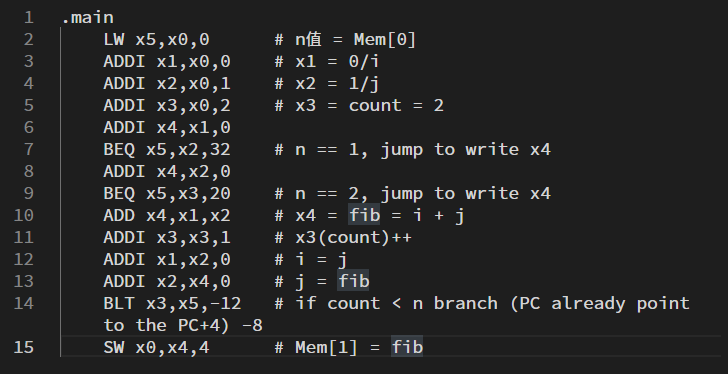
branch jl测试指令



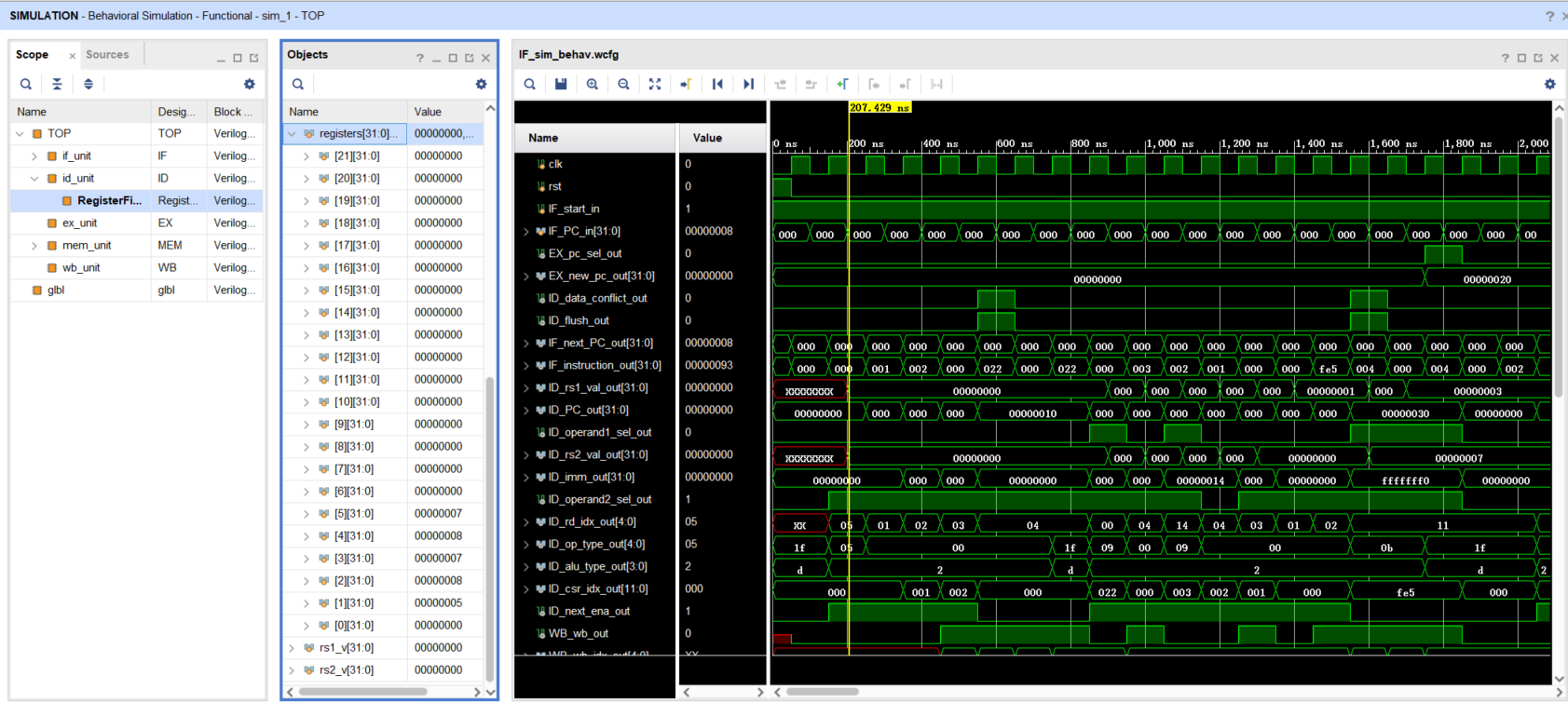
branch jl测试指令仿真结果

**（七）使用带有访存功能的fibonacci数列程序测试访存指令**

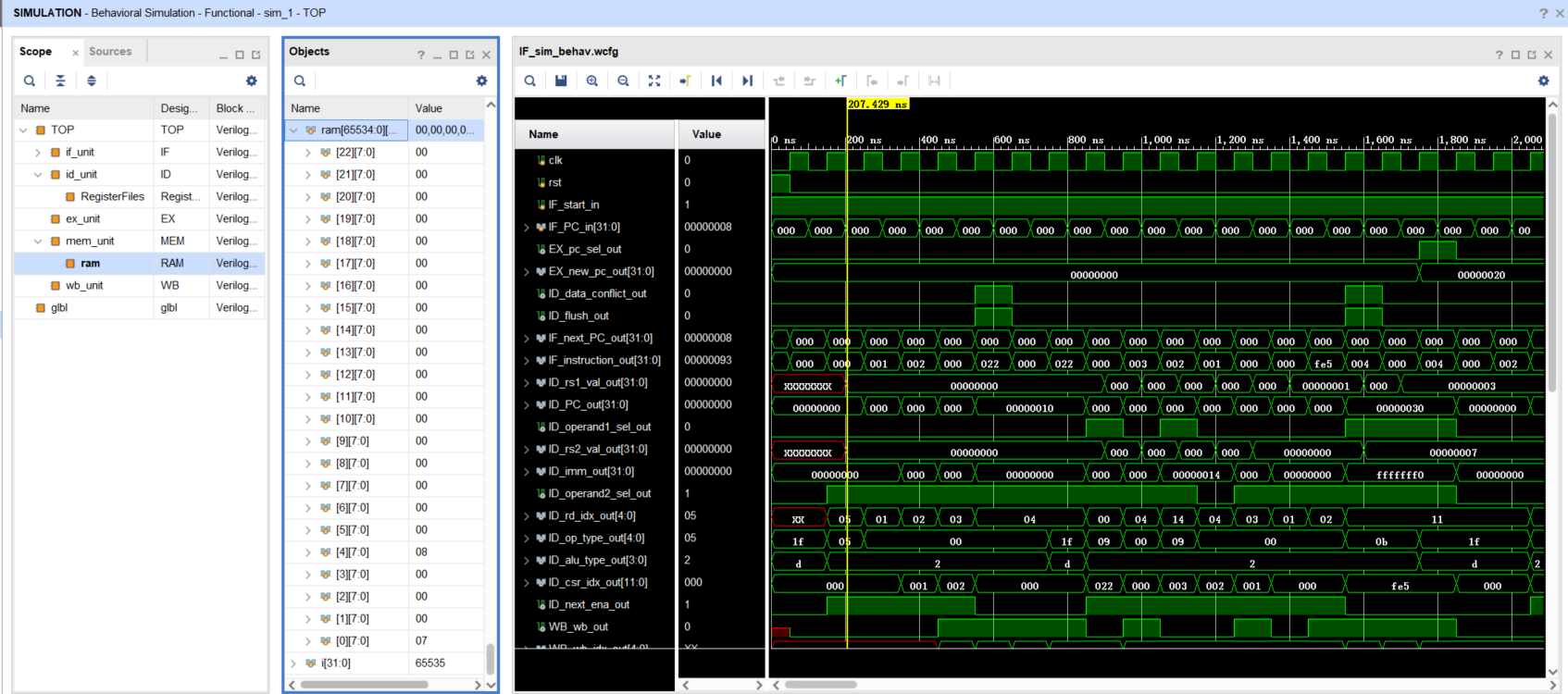
fibonacci测试指令如图5.15所示，使用riscv-gnu-tools将其转成二进制码。选择加载fibonacci.coe文件，运行仿真，仿真结果如图5.16和图5.17。从仿真结果图中可以看出，所有寄存器的值均正确并且内存中数值正常，说明fibonacci程序可以正常运行。



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