10 天实现处理器

——OpenMIPS 成长记

Lei
leishangwen@163.com
2013-12-1
v1.0

OpenMIPS 开始于 2013 年 8 月，目的是开发一款 32 位、兼容 MIPS32 指令集的开源软核处理器，便于老师教学、学生体会理解计算机体系结构课程的相关知识，同时也可以做实际用途。OpenMIPS 将坚持自由软件的理念，保持开源的形式，同时采用商业友好的 LGPL 授权。并且分为两个版本：教学版、实践版，每个版本都使用 VHDL、Verilog HDL 两种语言编写，这样实际是有四个版本，分别命名为（以 1.0 版为例）

- ✓ OpenMIPS_VHDL_study_v1.0　　教学版（VHDL）1.0
- ✓ OpenMIPS_Verilog_study_v1.0　　教学版（Verilog）1.0
- ✓ OpenMIPS_VHDL_practice_v1.0　　实践版（VDHL）1.0
- ✓ OpenMIPS_Verilog_practice_v1.0　　实践版（Verilog）1.0

教学版的主要设想是尽量简单，比如：在一个时钟周期内可以取到指令，完成存储、加载数据，这样处理器的运行情况（比如：流水线的运行）就比较理想化，与教科书相似，代码也很清晰简单，便于使用其进行教学、学术研究和讨论，也有助于各位同学理解课堂上讲授的知识。

实践版的主要设想是使 OpenMIPS 成为一个实际可用的处理器，能够下载到 FPGA 上，运行实际有用的程序，为此，添加了 wishbone 总线接口，使其可以挂接在 wb_conmax 互联矩阵上，这样就能方便的利用 OpenCores 上提供的 SDRAM、Flash、GPIO、UART、LCD 等模块控制器，组成一个 SOPC，完成特定功能，进一步还可为其移植操作系统。

本文档是在整理当初发布 OpenMIPS 时的一个 10 天教程上形成的，主要介绍了 OpenMIPS 教学版（VHDL）的设计思路，利用 10 天时间，从无到有、从小到大，给出了 OpenMIPS 的成长点滴，通过本教程，读者可以更加容易的理解 OpenMIPS 设计原理。也希望广大读者不吝赐教，针对设计中可能存在的问题及时指出。

让我们为了 OpenMIPS 的茁长成长共同努力！

第一天

主要内容

（1）OpenMIPS 介绍
（2）实验环境搭建
（3）实现通用寄存器 Regfile
（4）实现指令存储器 imem
（5）实现数据存储器 dmem

1.1 OpenMIPS 介绍

首先简单介绍一下 MIPS 体系结构，MIPS 即无内锁流水线微处理器（microprocessor without interlocked pipeline），是上世纪 80 年代诞生的 RISC CPU 的重要代表，其设计者 John Hennessy 时任斯坦福大学的教授，John Hennessy 和他的学生探寻了 RISC 体系结构概念，该概念基于以下理论：使用相对简单的指令，结合优秀的编译器以及采用流水线执行指令的硬件，就可以用更少的晶元面积生产更快的处理器。这一概念是如此的成功，以至于 1984 年就成立了 MIPS 计算机系统公司对 MIPS 体系结构进行商业化。在随后的十几年中，MIPS 体系结构在很多方面得到发展，在工作站和服务器系统中应用的非常成功。MIPS 体系结构自身也从 MIPS I、MIPS II、MIPS III、MIPS IV、MIPS V、MIPS32 发展到 MIPS64。其中 MIPS32/64 第一次包含了被称为协处理器 0 的“CPU 控制”功能，MIPS32 是 MIPS II 的超集，而包含了 64 位指令的 MIPS64 是 MIPS IV 的超集，大多数在 1999 年以后设计的 MIPS CPU 都和这两个标准兼容，因此 OpenMIPS 也计划兼容 MIPS32 体系结构。

MIPS 的设计者 John Hennessy 教授为广大处理器学习者所熟悉，还得益于其与 David Patterson 教授合作编写的经典书籍《计算机体系结构——量化研究方法》，该书至今已出了五版，影响了一代一代的处理器设计开发爱好者。John Hennessy 本人也从最初的斯坦福大学教授到如今斯坦福大学第 10 任校长，并与 David Patterson 共同获得 2000 年度约翰·冯·诺依曼奖章。

本教程设计实现的处理器为 OpenMIPS，是一款具有哈佛结构的 32 位标量处理器，兼容 MIPS32 体系结构，这样可以使用现有的 MIPS 编译环境。OpenMIPS 具有以下特点：

- 五级整数流水线，分别是：取指、译码、执行、访存、回写
- 哈佛结构，分开的指令、数据接口
- 32 个 32 位整数寄存器
- 大端模式
- 向量化异常处理，支持精确异常处理
- 8 个外部中断
- 32bit 数据、地址总线宽度
- 单周期乘法
- 支持延迟转移
- 兼容 MIPS32 体系结构，支持 MIPS32 指令集中的所有整数指令
- 大多数指令可以在一个时钟周期内完成
- 可综合

- LGPL 开源

OpenMIPS 的对外接口如图 1.1 所示：

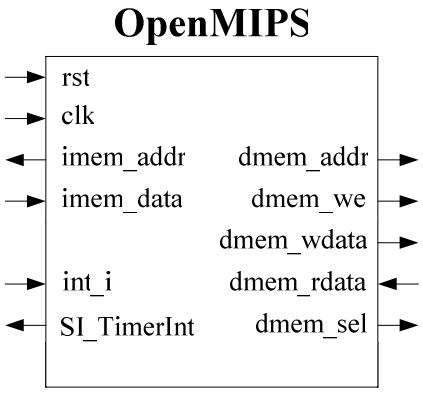


图 1.1　OpenMIPS 接口示意图

各接口描述如表 1.1 所示：

表 1.1　OpenMIPS 接口描述表

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号，高电平有效
2	clk	1	输入	时钟信号
3	imem_addr	32	输出	指令存储器地址输出
4	imem_data	32	输入	指令存储器数据输入
5	int_r	8	输入	中断信号
6	SI_TimerInt	1	输出	时钟中断信号
7	dmem_addr	32	输出	数据存储器地址输出
8	dmem_we	1	输出	数据存储器写使能
9	dmem_wdata	32	输出	数据存储器数据输出
10	dmem_rdata	32	输入	数据存储器数据输入
11	dmem_sel	4	输出	数据存储器字节有效信号

OpenMIPS 内部结构如图 1.2 所示，可见内部包含一个整数单元 IU、一个除法单元 Div、一个寄存器文件单元 Regfile，这三个单元分别对应 iu.vhd、div.vhd、regfile.vhd 三个源文件，其中 IU 内部实现了五级整数流水线，Div 模块实现了 32 位整数除法，采用的是试商法，Regfile 实现了 32 个 32 位整数寄存器。此外还有 OpenMIPS.vhd 文件实现了顶层模块 OpenMIPS。

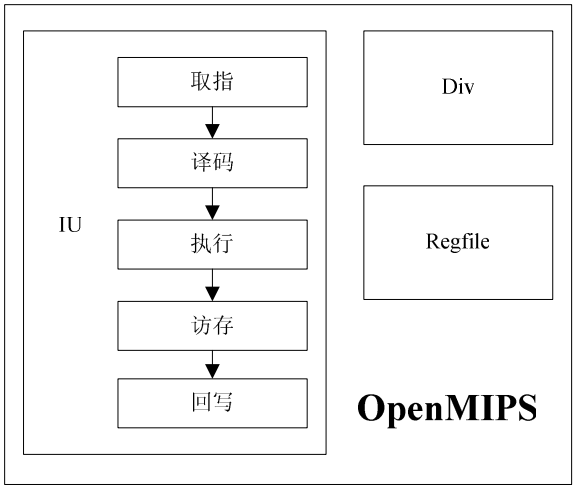


图 1.2 OpenMIPS 内部结构

OpenMIPS 支持 MIPS32 指令集中的所有整数指令，分类如下，指令的详细解释将在后期的实现过程中逐一描述：

（1）算术指令，包含：ADD、ADDI、ADDIU、ADDU、CLO、CLZ、SLT、SLTI、SLTIU、SLTU、SUB、SUBU、MADD、MADDU、MSUB、MSUBU、MUL、MULT、MULTU、DIV、DIVU

（2）分支和转移指令，包含：B、BAL、BEQ、BGEZ、BGEZAL、BGTZ、BLEZ、BLTZ、BLTZAL、BNE、J、JAL、JALR、JR

（3）控制指令，包含：NOP、SSNOP

（4）加载存储指令，包含：LB、LBU、LH、LHU、LL、LW、LWL、LWR、SB、SC、SH、SW、SWL、SWR

（5）逻辑操作指令，包含：AND、ANDI、LUI、NOR、OR、ORI、XOR、XORI

（6）移动指令，包含：MFHI、MFLO、MOVN、MOVZ、MTHI、MTLO

（7）移位指令，包含：SLL、SLLV、SRA、SRAV、SRL、SRLV

（8）自陷指令，包含：SYSCALL、TEQ、TEQI、TGE、TGEI、TGEIU、TGEU、TLT、TLTI、TLTIU、TLTU、TNE、TNEI

（9）特殊指令，包含：MFC0、MTC0、ERET

指令执行周期如表 1.2 所示：

表 1.2　OpenMIPS 指令执行所需周期

指令	执行所需周期
DIV、DIVU	35
MADD、MADDU、MSUB、MSUBU	2
其余指令	1

为了使读者对 OpenMIPS 有一个直观的印象，表 1.3 给出其所有源代码文件及作用说明，其中蓝色标注的模块是为了验证 OpenMIPS 是否正确实现而添加的，通过这三个模块建立一个简单 SOPC。

表 1.3 OpenMIPS 的源文件及其作用说明

源文件	作用
iu.vhd	实现了五级流水线
div.vhd	实现了除法运算
regfile.vhd	实现了通用寄存器模块
stdlib.vhd	一些宏定义、结构体定义、函数定义
OpenMIPS.vhd	OpenMIPS 顶层文件，其中连接 IU、Div、Regfile 模块
imem.vhd	指令存储器，为仿真而添加
dmem.vhd	数据存储器，为仿真而添加
OpenMIPS_min_sopc.vhd	简单 SOPC 的顶层文件，为仿真而添加，其中连接 OpenMIPS、imem、dmem 模块

1.2 实验环境搭建

本教程主要使用 ModelSim10.1a 进行仿真，使用 gcc 进行编译，可以在 http://bbs.eetop.cn/thread-429161-1-1.html 下载，gcc 安装在 Ubuntu11.10 上，读者可以使用 OR1200 提供的 Ubuntu 虚拟机镜像，能够在 ftp://openrisc.opencores.org/virtualbox-image/地址下载到，FTP 的用户名和密码都是 openrisc，登录后会出现如图 1.3 所示界面。

图 1.3 Ubuntu 虚拟机镜像下载

下载最新的那个文件就可以了，笔者使用的是 2011-12-15 版。下载完成后解压该文件，大约 4GB 左右。此时还需要下载 VisualBox 以打开该文件。VisualBox 是一款开源的虚拟机软件，本教程使用的是 4.1.22 版。下载完成后安装 VisualBox，安装完成后打开 VisualBox，界面如图 1.4 所示。

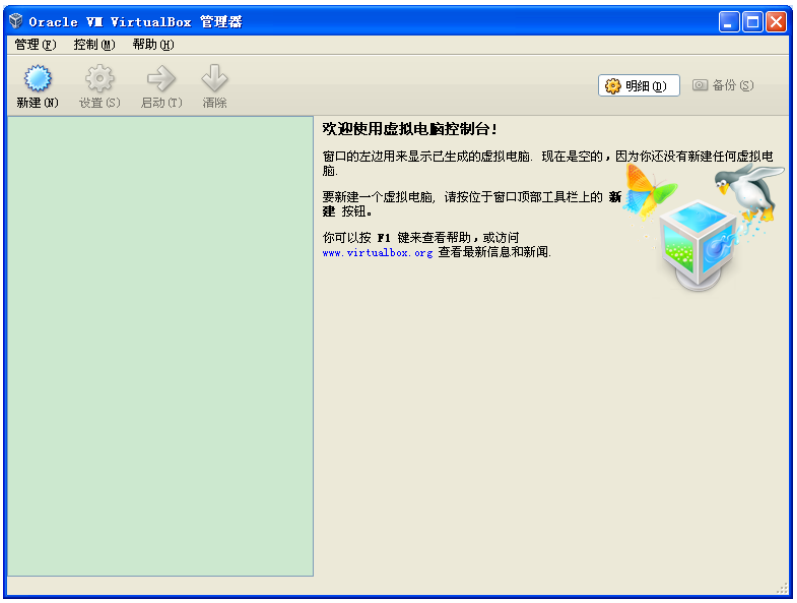


图 1.4 VisualBox 主界面

点击“新建”出现“新建虚拟机”向导，点击“下一步”，出现如下界面：

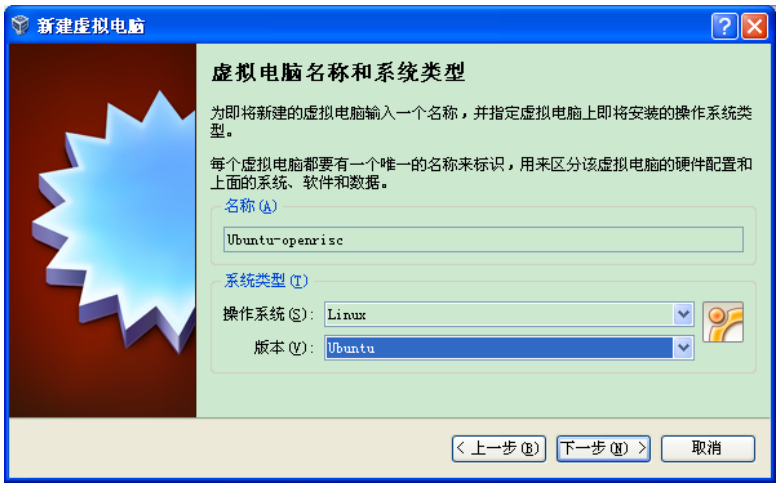


图 1.5 新建虚拟机设置一

此处操作系统选择 Linux，版本选择 Ubuntu，点击下一步，设置内存大小，如图 1.6 所示。

图 1.6 新建虚拟机设置二

内存大小依据个人情况设置，本人设置的是 512M，已经够用了，毕竟我们需要编译的程序都是十分简单的，点击下一步，选择“使用现有的虚拟硬盘”，然后选择解压后的虚拟机文件。

图 1.7 新建虚拟机设置三

点击“下一步”，VisualBox 会将用户刚才的设置都列出来，确认无误后，点击“创建”，这样虚拟机就创建好了。启动虚拟机，显示如图 1.8 所示。

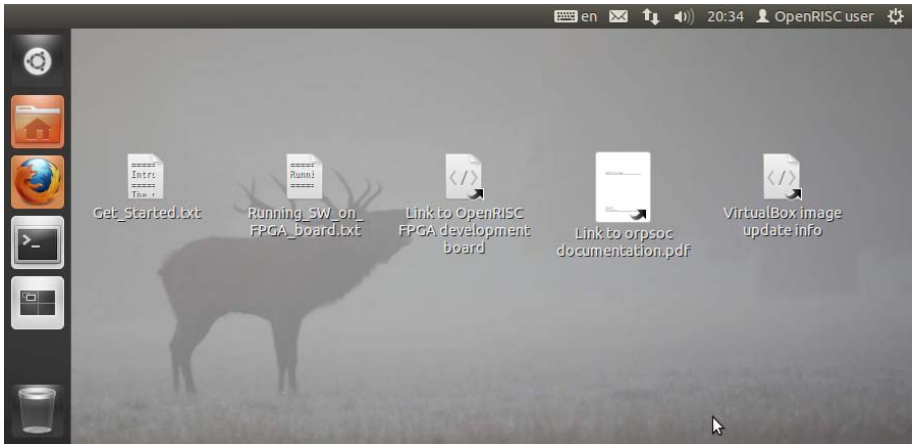


图 1.8 Ubuntu 虚拟机桌面

因为宿主机是 Windows 平台，而且在后面仿真时使用的 ModelSim 也是 Windows 平台的，为了方便文件的传递，这里需要设置虚拟机与宿主机的文件共享。打开 VisualBox 中虚拟机的设置界面，选择“共享文件夹”，如图 1.9 所示。

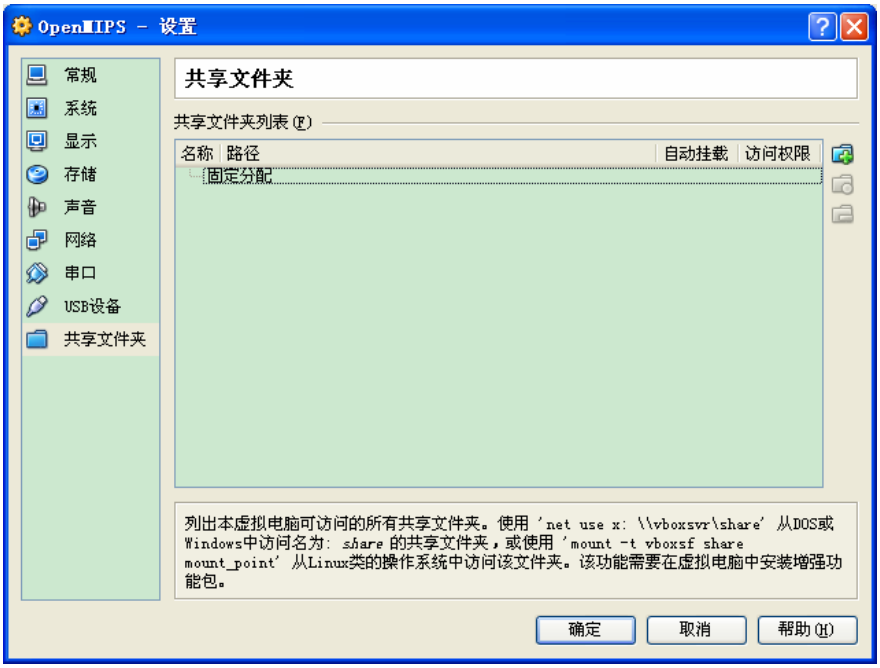


图 1.9 虚拟机与宿主机共享文件夹设置步骤一

点击界面右边的添加文件夹按钮，出现如图 1.10 所示界面：

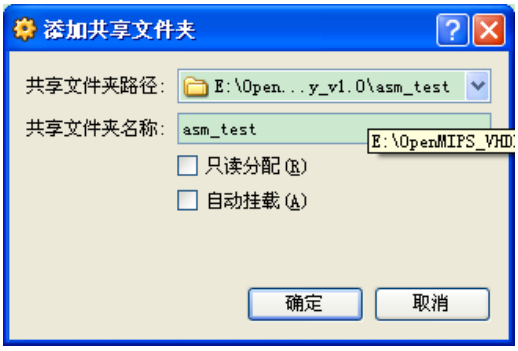


图 1.10 虚拟机与宿主机共享文件夹设置步骤二

在其中选择共享文件夹的路径，设置名称，参考如上设置，然后启动虚拟机，打开终端，输入命令：

```
    sudo mount -t vboxsf asm_test /mnt/
```

该命令的作用是将共享文件夹挂载在/mnt/目录下，sudo 表示以 Root 用户身份执行该命令，终端会提示输入密码，Ubuntu 虚拟机默认 Root 用户的密码是 openrisc。这样就实现了虚拟机与宿主机的文件共享，对虚拟机而言共享文件放在/mnt/路径下，对宿主机而言共享文件放在图 1.10 所示的 E 盘 OpenMIPS_VHDL_study_v1.0/asm_test 文件夹下。

将下载的 gcc 工具压缩包拷贝到 Ubuntu 的/opt 目录下，打开 Ubuntu 的终端，使用如下命令解压缩：

```
cd /opt
tar vfxj mips_linux_toolchain_bin-1.1.tar.bz2
```

在用户主目录下有一个 .bashrc 文件（注意：该文件是隐藏的），在此文件中加入 PATH 的设置如下：

```
    export PATH="$PATH:/opt/mips-4.3/bin"
```

然后注销。在终端中输入 mips-sde-elf，然后按两次 Tab 键，会列出安装的针对 MIPS 平台的所有编译工具，如图 1.11 所示，表示 MIPS 编译环境安装成功。

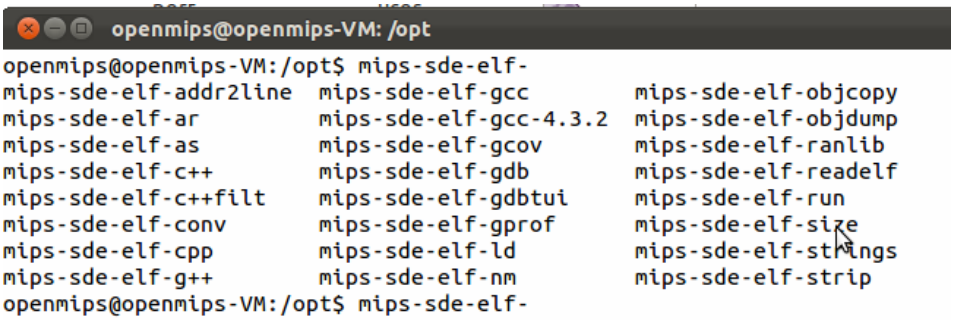


图 1.11 Ubuntu 中安装的编译工具

写一个简单的汇编程序如下，命名为 inst_rom.S

```
    .org 0x0
   .global _start
  .set noat
_start:
  ori $1,$0,0x1100       # r1 = r0 | 0x1100 = 0x1100
```

```
  ori $1,$1,0x0020       # r1 = r1 | 0x0020 = 0x1120
  ori $1,$1,0x4400       # r1 = r1 | 0x4400 = 0x5520
  ori $1,$1,0x0044       # r1 = r1 | 0x0044 = 0x5564
```

其中使用到了 ORI 指令，该指令的作用是将寄存器的值与立即数相或，将结果存入目的寄存器，需要注意的是 MIPS 指令中第一个寄存器是目的寄存器，此处用到的寄存器是 r0、r1，其中 r0 始终为 0，程序的最后结果是使得 r1 为 0x5564。

编写链接文件 ram.ld 如下：

```
MEMORY
      {

      ram    : ORIGIN = 0x00000000, LENGTH = 0x00001000
      }

SECTIONS
{

    .text :
      {
      *(.text)
      } > ram

      .data :
      {
      *(.data)
      } > ram

      .bss :
      {
      *(.bss)
      } > ram

      .stack  ALIGN(0x10) (NOLOAD):
      {
      *(.stack)
      _ram_end = .;
      } > ram
}

ENTRY (_start)
```

这里定义了一个存储块——ram，其起始地址是 0x0，长度是 0x1000，然后指示链接器输出文件包含四个 Section，分别是.text、.data、.bss、.stack，其中.text 从 ram 的起始地址开始存放，后面跟着.data、.bss、.stack，并且输入文件的 Section .text 存放在输出文件的.text 中，输入文件的 Section .data 存放在输出文件的.data 中，输入文件的 Section .bss 存放在输

出文件的.bss 中，输入文件的 Section .stack 存放在输出文件的.stack 中。最后的 Entry 指定程序的入口地址，也就是第一条执行指令的地址是_start 符号的值，从汇编代码中可知_start 符号就是地址 0x0。

编写 Makefile 文件如下：

```
ifndef CROSS_COMPILE
CROSS_COMPILE = mips-sde-elf-
endif
CC = $(CROSS_COMPILE)as
LD = $(CROSS_COMPILE)ld
OBJCOPY = $(CROSS_COMPILE)objcopy
OBJDUMP = $(CROSS_COMPILE)objdump

OBJECTS = inst_rom.o

export  CROSS_COMPILE

# ********************
# Rules of Compilation
# ********************

all: inst_rom.om inst_rom.bin inst_rom.asm

%.o: %.S
    $(CC) -mips32 $< -o $@
inst_rom.om: ram.ld $(OBJECTS)
    $(LD) -T ram.ld $(OBJECTS) -o $@
inst_rom.bin: inst_rom.om
    $(OBJCOPY) -O binary $<  $@
inst_rom.asm: inst_rom.om
    $(OBJDUMP) -D $< > $@
clean:
    rm -f *.o *.om *.bin *.asm
```

将上述 ram.ld、Makefile 两个文件拷贝到汇编程序 inst_rom.S 所在目录，然后打开 Ubuntu 终端，进入 inst_rom.S 所在目录，输入 make all，即可得到需要的二进制文件 inst_rom.bin，反汇编文件 inst_rom.asm，其中 inst_rom.asm 的目的是为了方便查找指令对应的二进制，以便于在 ModelSim 中仿真。本节中给出的简单汇编程序编译后得到的 inst_rom.asm 文件如下：

```
inst_rom.om:     file format elf32-tradbigmips

Disassembly of section .text:

00000000 <_start>:
   0:   34011100    li  at,0x1100
   4:   34210020    ori at,at,0x20
```

```
   8:   34214400    ori at,at,0x4400
   c:   34210044    ori at,at,0x44
Disassembly of section .reginfo:

00000000 <_ram_end-0x20>:
   0:   00000002    srl zero,zero,0x0
    ...
```

inst_rom.bin 文件的内容如图 1.12 所示：

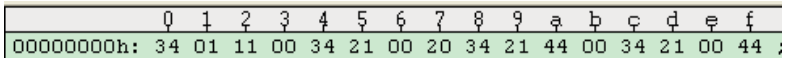


图 1.12 inst_rom.bin 文件的内容

可见 inst_rom.bin 文件的内容就是四条指令对应的二进制。在本教程的仿真中需要使用 VHDL 代码读取文件，以初始化指令存储器，但是 VHDL 要读取的文件对格式有要求，本教程使用以下格式：

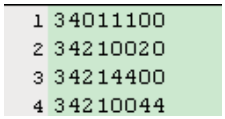


图 1.13 代码初始化指令存储器使用的文件格式

其中每一条指令占用一行，所以需要对 inst_rom.bin 文件进行修改，以得到符合格式的指令存储器初始化文件，笔者制作了一个小程序 Bin2Mem.exe 可以实现此目的，修改 Makefile 如下：

```
ifndef CROSS_COMPILE
CROSS_COMPILE = mips-sde-elf-
endif
CC = $(CROSS_COMPILE)as
LD = $(CROSS_COMPILE)ld
OBJCOPY = $(CROSS_COMPILE)objcopy
OBJDUMP = $(CROSS_COMPILE)objdump

OBJECTS = inst_rom.o

export  CROSS_COMPILE

# ********************
# Rules of Compilation
# ********************

all: inst_rom.om inst_rom.bin inst_rom.asm inst_rom.data

%.o: %.S
    $(CC) -mips32 $< -o $@
```

```
inst_rom.om: ram.ld $(OBJECTS)
    $(LD) -T ram.ld $(OBJECTS) -o $@
inst_rom.bin: inst_rom.om
    $(OBJCOPY) -O binary $<  $@
inst_rom.asm: inst_rom.om
    $(OBJDUMP) -D $< > $@
inst_rom.data: inst_rom.bin
    ./Bin2Mem.exe -f $< -o $@
clean:
    rm -f *.o *.om *.bin *.data *.asm
```

这样，以后再使用 make all 编译文件的时候，就可以得到 inst_rom.data 文件，其中就是每一行对应一条指令。

后续每一天的实验步骤大致相同，修改 OpenMIPS，使其支持新的指令，然后编写一段汇编程序，其中使用到新的指令，在 Ubuntu 中编译该汇编程序，将得到的 inst_rom.data 文件拷贝到 ModelSim 工程中，使用该文件初始化指令存储器，进行仿真验证，以确保新添加的指令实现正确。

上述过程已经多次提及指令存储器，在此加以明确，指令存储器 imem、数据存储器 dmem 都是为了仿真而添加的，他们与 OpenMIPS 共同组成了一个小的 SOPC，如图 1.15 所示。

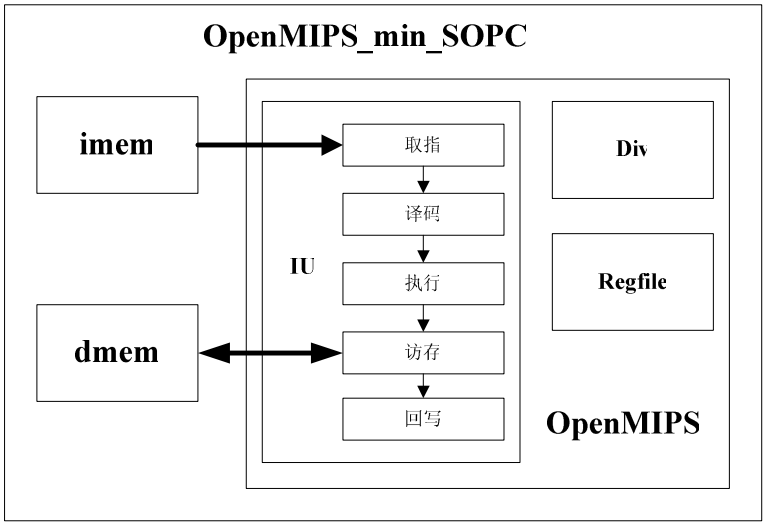


图 1.15 OpenMIPS 与 imem、dmem 一起组成一个小的 SOPC

今天是第 1 天，除了介绍 OpenMIPS 基本情况外，还将实现 Regfile，以及用于仿真验证的指令存储器 imem、数据存储器 dmem。

1.3 实现通用寄存器 Regfile

通用寄存器 Regfile 模块实现了 32 个 32 位整数寄存器，可以同时读出两个寄存器的值，接口如图 1.16 所示，各接口含义如表 1.4 所示。

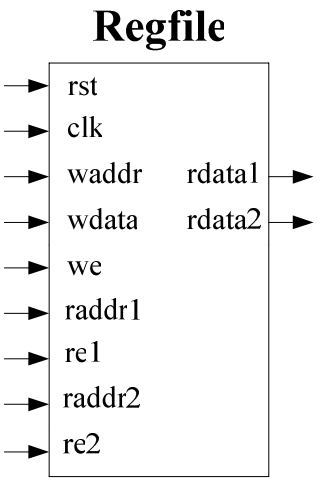


图 1.16 Regfile 模块的接口

表 1.4 Regfile 模块接口描述表

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号，高电平有效
2	clk	1	输入	时钟信号
3	waddr	5	输入	要写入的寄存器地址
4	wdata	32	输入	要写入的数据
5	we	1	输入	写使能信号
6	raddr1	5	输入	要读取的寄存器 1 的地址
7	re1	1	输入	是否要读取寄存器 1，高电平有效
8	raddr2	5	输入	要读取的寄存器 2 的地址
9	re2	1	输入	是否要读取寄存器 2，高电平有效
10	rdata1	32	输出	读取的寄存器 1 的值
11	rdata2	32	输出	读取的寄存器 2 的值

对应的源文件是 regfile.vhd，代码如下：

```
--------------------------------------------------------------------------------
--
-- Entity:  regfile
-- File:    regfile.vhd
-- Author:  Lei Silei
-- Description: register file
--------------------------------------------------------------------------------
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
library std;
```

```
use WORK.stdlib.all;
use WORK.all;

entity regfile is
  port (
    clk   : in  std_logic;
    rst   : in  std_logic;
    waddr  : in  std_logic_vector(4 downto 0);
    wdata  : in  std_logic_vector(31 downto 0);
    we    : in  std_logic;
    raddr1 : in  std_logic_vector(4 downto 0);
    re1   : in  std_logic;
    rdata1 : out std_logic_vector(31 downto 0);
    raddr2 : in  std_logic_vector(4 downto 0);
    re2   : in  std_logic;
    rdata2 : out std_logic_vector(31 downto 0)
  );
end;

architecture rtl of regfile is
  type mem is array(0 to 31) of std_logic_vector(31 downto 0);
  signal regarr : mem;
begin


  process(clk)
  begin
    if rising_edge(clk) then
      if(rst /= '1') then
          if (we = '1') and (waddr /= "00000") then
            --如果是写操作，且目的地址不是r0
              regarr(conv_integer(waddr)) <= wdata;
            end if;
      end if;
    end if;
  end process;

  --根据raddr1的值，给出读操作的寄存器1的值，分四种情况
  --（1）读取寄存器0，那么直接返回0
  --（2）读使能，且写使能，且读地址与写地址相等，那么直接返回要写入的值
  --（3）读使能，返回对应寄存器的值
  --（4）返回0
  rdata1 <= (others => '0') when raddr1 = "00000" else
           wdata    when raddr1 = waddr and re1 = '1' and we = '1' else
```

```
          regarr(conv_integer(raddr1)) when re1 = '1' else
          (others => '0');

  --根据 raddr2 的值，给出读操作的寄存器 2 的值，也分四种情况，同寄存器 1
  rdata2 <= (others => '0') when raddr2 = "00000" else
          wdata    when raddr2 = waddr and re2 = '1' and we = '1' else
          regarr(conv_integer(raddr2)) when re2 = '1' else
          (others => '0');

end;
```

1.4 实现指令存储器 imem

imem 模块作为指令存储器，是只读的，且读操作是时钟异步的，使用在 1.2 节通过 make all 得到的 inst_rom.data 文件初始化其内部存储空间，接口如图 1.17 所示，各接口含义如表 1.5 所示。

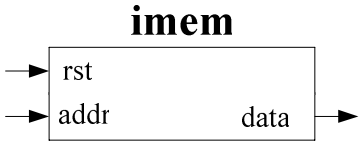


图 1.17 imem 模块的接口

表 1.5 imem 模块接口描述表

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号，高电平有效
2	addr	32	输入	要读取的指令地址信号
3	data	32	输出	读出的指令

imem 模块的代码如下，其中使用到了 textio 库中的函数：

```
--------------------------------------------------------------------------------
--
-- Entity:  imem
-- File:    imem.vhd
-- Author:  Lei Silei
-- Description: Instruction Memory
--------------------------------------------------------------------------------
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use WORK.stdlib.all;
```

```
use ieee.std_logic_textio.all;
library std;
use std.textio.all;

entity imem is
  port (
    rst   : in  std_logic;
      addr  : in  word;
      data  : out word
  );
end;

architecture rtl of imem is

   --此处的 IMEMSIZEINWORD 是在 stdlib.vhd 中定义的 imem 的大小（单位是 word）
    TYPE mem is ARRAY (IMEMSIZEINWORD downto 0) of std_logic_vector(31 downto 0);
    signal mem0: mem;

    --使用 inst_rom.data 文件初始化 imem
    file f1:text is in "inst_rom.data";
begin

    process(rst, addr)
        variable li : line;
        variable k,i : integer;
        variable j : character;
        variable good : boolean;
        variable temp : std_logic_vector(31 downto 0);
        variable temp1: integer range 0 to IMEMSIZEINWORD;
    begin
          if(rst='1') then
              --如果复位信号有效，那么读取 inst_rom.data 文件初始化 imem
              k := 0;
              while not endfile(f1) loop
                 readline(f1,li);
                 i := 0;
                 while i<8 loop
                     read(li,j,good);
                     if(good) then
                        temp(31-i*4 downto 28-i*4) :=
                                     conv_character_to_std_logic_vector(j);
                     end if;
                     i := i+1;
                 end loop;
```

```
                mem0(k) <= temp;
                k := k+1;
            end loop;
            data <= (others=>'0');
          else
            --复位信号无效的时候就是正常读取，依据地址给出指令
            --注意的是地址的最低两位不用，4 字节对齐
             temp1 := conv_integer(addr(IMEMBIT+2 downto 2));
             data <= mem0(temp1);
        end if;

    end process;

end;
```

1.5 实现数据存储器 dmem

dmem 模块作为数据存储器，是可读写的，读操作是时钟异步的，写操作是时钟同步的，并且写操作可以按照字节写入。接口如图 1.18 所示，各接口含义如表 1.6 所示。

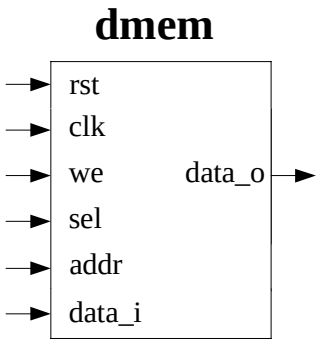


图 1.18 dmem 模块的接口

表 1.6 dmem 模块接口描述表

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号，高电平有效
2	clk	1	输入	时钟信号
3	we	1	输入	写操作信号，高电平表示写操作，低电平表示读操作
4	sel	4	输入	写操作时的字节选择信号
5	addr	32	输入	读写地址
6	data_i	32	输入	要写的数据
7	data_o	32	输出	读出的数据

为了方便实现按照字节写入 dmem，在设计 dmem 的时候使用 4 个 8 位存储器代替一个 32 位存储器，如图 19 所示，读操作时，从 4 个 8 位存储器中各读出一个字节，组合为一个 32 位的数据输出，写操作时，依据 sel 的值，修改其中的特定存储器对应的字节即可。

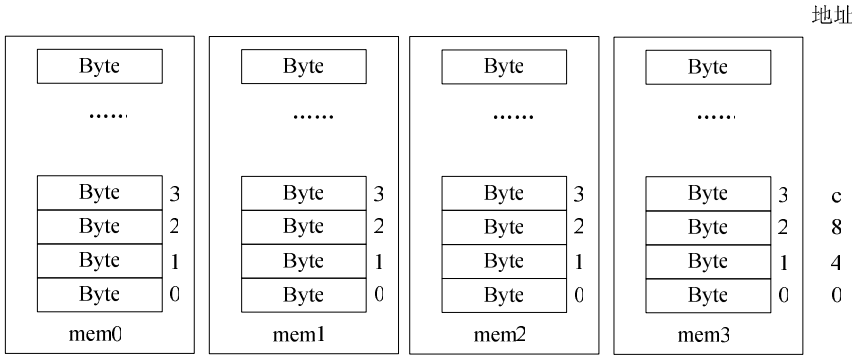


图 19 dmem 由 4 个 8 位存储器构成

dmem 代码如下：

```
--------------------------------------------------------------------------------
--
-- Entity:  dmem
-- File:    dmem.vhd
-- Author:  Lei Silei
-- Description: Data Memory
--------------------------------------------------------------------------------
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use WORK.stdlib.all;
use ieee.std_logic_textio.all;
library std;
use std.textio.all;

entity dmem is
  port (
    clk   : in  std_logic;
    rst   : in  std_logic;
    we    : in  std_logic;
    sel   : in  std_logic_vector(3 downto 0);
     addr  : in  word;
     data_i  : in  word;
     data_o  : out word
  );
```

```
end;

architecture rtl of dmem is

   --定义 4 个 8 位存储器
    TYPE mem is ARRAY (DMEMSIZEINWORD downto 0) of std_logic_vector(7 downto 0);
    signal mem0, mem1, mem2, mem3: mem;
begin


  process(clk)
       variable temp: integer range 0 to DMEMSIZEINWORD;
     begin
     if rising_edge(clk) then
     if (rst='0') then
         temp := conv_integer(addr(DMEMBIT+2 downto 2));
         if(we = '1') then
           --如果是写操作，那么依据 sel 的值确定要写入的字节
           if(sel(3) = '1') then
             mem0(temp) <= data_i(31 downto 24);
           end if;
           if(sel(2) = '1') then
             mem1(temp) <= data_i(23 downto 16);
           end if;
           if(sel(1) = '1') then
             mem2(temp) <= data_i(15 downto 8);
           end if;
           if(sel(0) = '1') then
             mem3(temp) <= data_i(7 downto 0);
           end if;
      end if;
     end if;
     end if;
     end process;

     process(rst, we, addr)
       variable temp: integer range 0 to DMEMSIZEINWORD;
     begin
       if(rst = '1') then
         data_o <= (others => '0');
       elsif(we = '0') then
         --读操作是异步的，注意地址的低 2bit 没有使用
         temp := conv_integer(addr(DMEMBIT+2 downto 2));
         data_o <= mem0(temp) & mem1(temp) & mem2(temp) & mem3(temp);
```

```
       else
         data_o <= (others => '0');
       end if;
     end process;
end;
```

第二天

主要内容

（1）实现五级流水线框架
（2）实现第一条指令——ORI

2.1 实现五级流水线框架

本教程实现的 OpenMIPS 具有五级流水线，对流水线概念不熟悉的同学请阅读计算机体系结构方面的书籍，本教程不再重复解释。OpenMIPS 五级流水线在各个阶段完成的工作如下：

- 取指阶段：从指令存储器 imem 取得指令，修改 pc 的值
- 译码阶段：指令译码，取得指令执行需要的寄存器的值、立即数的值，并判断是否是多周期指令
- 执行阶段：判断并解决数据相关，执行指令操作，判断是否转移
- 访存阶段：如果是加载存储指令，那么读写数据存储器 dmem
- 回写阶段：写目的寄存器。判断是否有异常发生

为了实现五级流水线，使用了 VHDL 中的 record 数据类型，在 stdlib.vhd 中定义如下 record 数据类型，**注意：这里的定义不是 OpenMIPS 完全实现时的定义，只是一个简化版本，可以满足第 2 天的需要，在实现其余指令的时候需要扩展此处的 record 类型定义。**

```
--取指阶段的寄存器
type fetch_reg_type is record
  pc     : pctype;  --要读取的指令地址
end record;

--译码阶段的寄存器
type decode_reg_type is record
  pc     : pctype;  --处于译码阶段的指令地址
  inst   : word;    --处于译码阶段的指令
end record;

--执行阶段的寄存器
type execute_reg_type is record
  rd    : rfatype;        --要写入的目的寄存器
  wreg  : std_logic;       --是否要写入目的寄存器
  rfe1      : std_logic;  --是否要读取源寄存器 1
  rfe2      : std_logic;  --是否要读取源寄存器 2
  rfa1      : rfatype;    --要读取的源寄存器 1 的地址
  rfa2      : rfatype;    --要读取的源寄存器 2 的地址
  reg1      : word;      --读取到的源寄存器 1 的值
```

```
    reg2      : word;      --读取到的源寄存器 2 的值
    imm       : word;      --指令需要的立即数的值
    cnt    : std_logic_vector(1 downto 0);    --是否是多周期指令
    aluop  : std_logic_vector(7 downto 0);  --ALU 的操作类型
    alusel : std_logic_vector(2 downto 0);      --ALU 的运算结果选择信号
     inst_valid : std_logic;  --指令是否有效
  end record;

  --访存阶段的寄存器
  type memory_reg_type is record
    waddr : rfatype;        --要写入的目的寄存器
    wreg  : std_logic;      --是否要写入目的寄存器
    result : word;          --要写入目的寄存器的值
  end record;

  --回写阶段的寄存器
  type write_reg_type is record
    result : word;          --要写入目的寄存器的值
    waddr  : rfatype;       --要写入的目的寄存器
    wreg   : std_logic;     --是否要写入目的寄存器
  end record;

  --流水线各个阶段的寄存器都通过 registers 类型定义
  type registers is record
    f  : fetch_reg_type;
    d  : decode_reg_type;
    e  : execute_reg_type;
    m  : memory_reg_type;
    w  : write_reg_type;
  end record;
```

有了上述 record，可以给出流水线的框架如下：

```
--流水线在整数单元 iu 中定义，对应 iu.vhd 文件
entity iu is
  port (    ......    );
end;

architecture rtl of iu is
  --定义两个信号 r、rin
  signal r, rin : registers;

begin

------------------------------------------------------------------------
---------------------------  组合逻辑过程 comb       -------------------
```

```
--------------------------------------------------------------------------
 comb : process( r, rst, ......)
     variable v    : registers;

 begin

   v := r;

--------------------------------------------------------------------------
-- WRITE BACK STAGE 回写阶段
--------------------------------------------------------------------------

  --依据此时回写阶段寄存器的值，设置 Regfile 模块的写信号
  --也就是依据 r.w 的值，设置 Regfile 模块的写信号，Regfile 是 32 个整数寄存器

--------------------------------------------------------------------------
-- MEMORY STAGE 访存阶段
--------------------------------------------------------------------------

   --依据此时访存阶段寄存器的值，设置对 dmem 模块的读写信号
   --也就是依据 r.m 的值，设置 dmem 模块的读写信号，dmem 是数据存储器

   --设置下一时钟周期回写阶段寄存器的值，也就是 v.w 的值

--------------------------------------------------------------------------
-- EXECUTE STAGE 执行阶段
--------------------------------------------------------------------------

   --依据此时执行阶段寄存器的值，进行指定运算
   --也就是依据 r.e 的值，进行指定运算

   --设置下一时钟周期访存阶段寄存器的值，也就是 v.m 的值

--------------------------------------------------------------------------
-- DECODE STAGE 译码阶段
--------------------------------------------------------------------------

      --依据此时译码阶段寄存器的值，进行指令译码
      --也就是依据 r.d 的值，进行指令译码

      --设置下一时钟周期执行阶段寄存器的值，也就是 v.e 的值

--------------------------------------------------------------------------
-- FETCH STAGE 取指阶段
```

```
------------------------------------------------------------------------

    --依据此时取指阶段寄存器的值，访问 imem 模块，读取指令
    --也就是依据 r.f 的值，访问 imem 模块，读取指令，imem 模块是指令存储器

    --设置下一个时钟周期译码阶段寄存器的值，也就是 v.d 的值

    --设置下一个时钟周期取指阶段寄存器的值，也就是 v.f 的值

------------------------------------------------------------------------
-- OUTPUTS
------------------------------------------------------------------------
     rin <= v;

  end process;

------------------------------------------------------------------------
----------------------------  时序逻辑过程 reg        --------------------
------------------------------------------------------------------------
  reg : process (clk)
  begin
    if rising_edge(clk) then
        if(rst = '1') then
      ......
       else
           r <= rin;--流水线前进一步
     end if;
    end if;
  end process;

end;
```

2.2 实现第一条指令——ORI

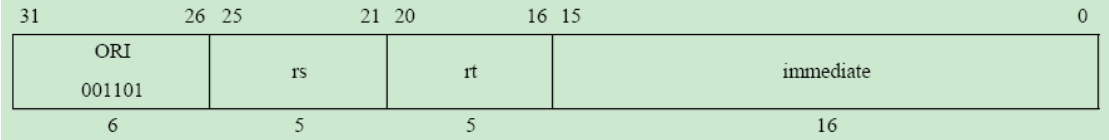
上面的流水线框架可能不易理解，童鞋们可以借助接下来 OpenMIPS 第一条指令 ORI 的实现理解上述流水线框架，另外，此处的流水线框架参考了 LEON3 的实现，因此，理解 OpenMIPS 的流水线也有助于学习 LEON3。

有了上一节的流水线框架，往里面填充具体内容即可实现相应功能，本小节以指令 ORI 的实现为例，完善流水线模块，所有代码都在 iu.vhd 文件中，本说明文档只摘取其中部分进行讲解。ORI 指令的说明如下所示：

指令	用法	作用	说明
ori	ori rt,rs,immediate	rt <- rs or immediate	指令中的 16 位立即数零扩展为 32 位后，

			与 rs 寄存器中的值相或，结果存在 rt 寄存器中

指令格式如下：



为了实现对 ORI 指令的执行，填充流水线如下：

（1）取指阶段

取指阶段需要从 imem 中取出指令，同时修改下一个周期要读取指令的地址，代码如下：

```
--给出 imem 的读取地址
imem_addr <= r.f.pc(31 downto 2) & "00";

--将从 imem 中读取的指令保存到 v.d.inst 中，在一个时钟周期进入译码阶段
v.d.inst := imem_data;
v.d.pc := r.f.pc;

if (rst = '1') then
  v.f.pc := "00000000000000000000000000000000";
else
  --下一条指令地址是当前读取指令地址加 4
  v.f.pc := r.f.pc(31 downto 2) + 1;
end if;
```

（2）译码阶段

译码阶段对指令进行译码，确定源操作数类型（是寄存器还是立即数），并访问 Regfile 模块，读取出对应的寄存器，判断是否是多周期指令，目前认为都是单周期指令，代码如下：

```
--调用过程 inst_decode
inst_decode(r.d.inst, v.e.wreg, v.e.rd, v.e.aluop, v.e.alusel,
             v.e.rfe1, v.e.rfe2, v.e.rfa1, v.e.rfa2, v.e.imm,
              v.e.cnt, v.e.inst_valid);

--依据译码结果，给出 Regfile 的访问信号
rf_o.raddr1 <= v.e.rfa1;   --第一个寄存器的读地址
rf_o.raddr2 <= v.e.rfa2;   --第二个寄存器的读地址
rf_o.ren1 <= v.e.rfe1;     --第一个寄存器的读使能信号
rf_o.ren2 <= v.e.rfe2;     --第二个寄存器的读使能信号

--将从 Regfile 中读取出的寄存器值，传递到执行阶段
v.e.reg1 := rf_i.data1;    --读出的第一个寄存器的值
v.e.reg2 := rf_i.data2;    --读出的第二个寄存器的值
```

在译码过程调用了过程 inst_decode，其中进行指令译码，给出 Regfile 的访问信号，以

及操作类型 aluop，inst_decode 定义参考 iu.vhd 文件，在第二天只对 ORI 指令进行译码。

（3）执行阶段

执行阶段依据译码阶段给出的操作类型 aluop，以及操作数进行运算，代码如下：

```
--操作数选择，是立即数还是寄存器的值
opdata_select(r, v, ex_opdata1, ex_opdata2);

--向访存阶段传递要写的目的寄存器地址、是否要写入目的寄存器
v.m.waddr := r.e.rd;
v.m.wreg := r.e.wreg;

--因为 ORI 是逻辑操作指令，所以此处只是调用过程 logic_op 进行逻辑 OR 运算
--结果存储在 ex_logic_res 中
logic_op(r, ex_opdata1, ex_opdata2, ex_logic_res);

--将逻辑运算的结果传递到访存阶段
 v.m.result := ex_logic_res;
```

其中调用了过程 opdata_select、logic_op，具体定义参考 iu.vhd。

（4）访存阶段

因为只考虑 ORI 这一条指令，没有考虑加载存储指令，所以在第 2 天不用考虑访存阶段对 dmem 的操作，访存阶段只是简单地传递信号到回写阶段代码如下：

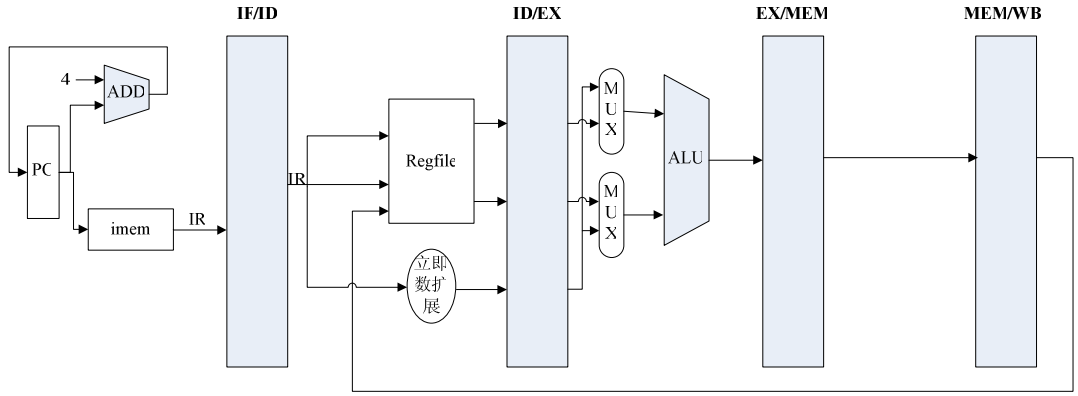
```
--将 Regfile 的写信号传递到回写阶段
v.w.result := r.m.result;
v.w.waddr  := r.m.waddr;
v.w.wreg   := r.m.wreg;
```

（5）回写阶段

进行 Regfile 模块的写操作，代码如下：

```
--依据运算结果进行 Regfile 的写操作
rf_o.wdata <= r.w.result;
rf_o.waddr <= r.w.waddr;
rf_o.wren <= r.w.wreg;
```

经过上述修改后，流水线的数据通路如下，其中不考虑 dmem，以及数据相关问题：

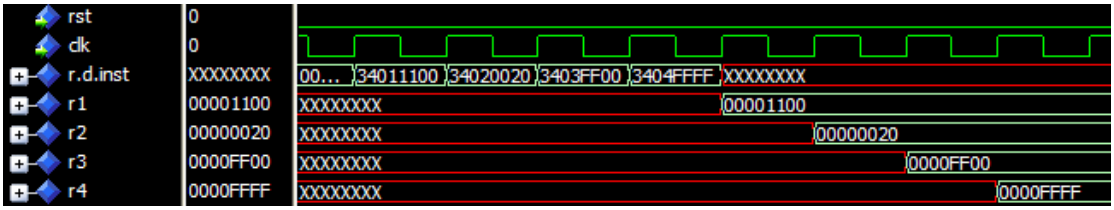


在后续几天中将不断完善上述数据通路。

为了验证是否正确实现 ORI 指令，编写测试程序如下，代码位于 asm_test/Day2 目录下。

```
    .org 0x0
    .global _start
    .set noat
_start:
   ori $1,$0,0x1100       # r1 = r0 | 0x1100 = 0x1100
   ori $2,$0,0x0020       # r2 = r0 | 0x0020 = 0x0020
   ori $3,$0,0xff00       # r3 = r0 | 0xff00 = 0xff00
   ori $4,$0,0xffff       # r4 = r0 | 0xffff = 0xffff
```

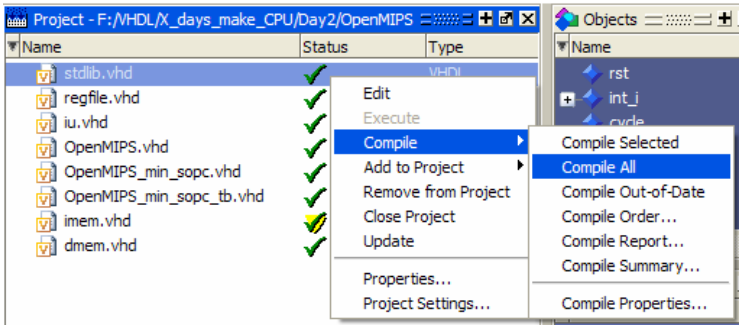
上述程序执行后，会使得 r1 为 0x1100，r2 为 0x0020，r3 为 0xff00，r4 为 0xffff。将上述代码保存为 inst_rom.S 文件，运用第 1 天建立的实验环境编译，得到 inst_rom.data 文件，利用该文件初始化 imem，在 ModelSim 中仿真得到如下结果，可知 ORI 指令实现正确。OpenMIPS 的代码参考 10_Days_make_OpenMIPS/Day2。



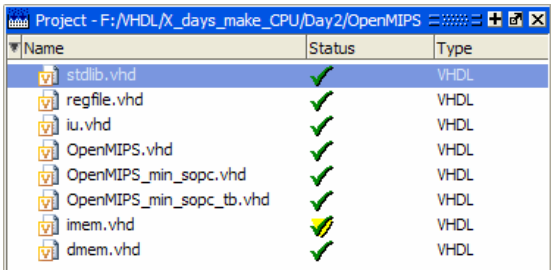
注意：本教程没有解释 OpenMIPS_min_sopc_tb 文件，该文件是 testbench，十分简单，各位同学一看即懂。

不会仿真的同学可以继续往下阅读，会仿真的同学今天就可以休息了^_^。

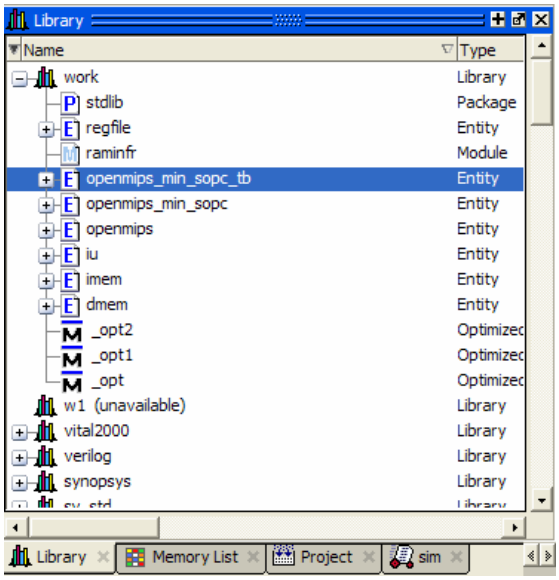
首先使用 ModelSim 新建一个工程，添加 10_Days_make_OpenMIPS/Day2 文件夹下的文件，随便选择一个文件，选中后单击右键，在弹出菜单中选择如下 Compile -> Compile All



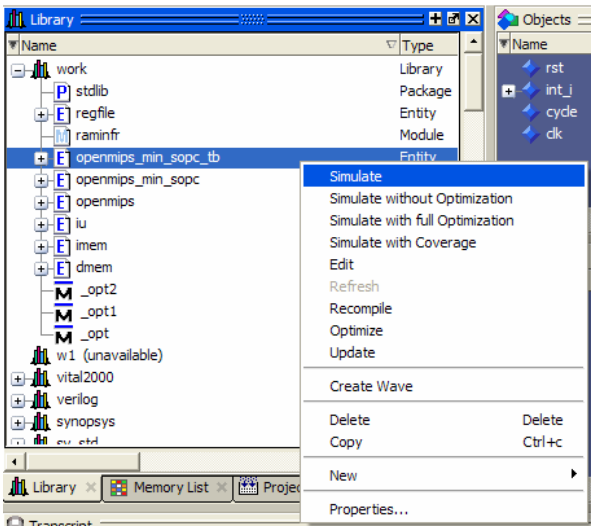
全部编译通过后，显示的都是对勾，如下，有时可能会提示编译出错，这是由于文件编译顺序的原因，再次编译一次应该就没问题了。



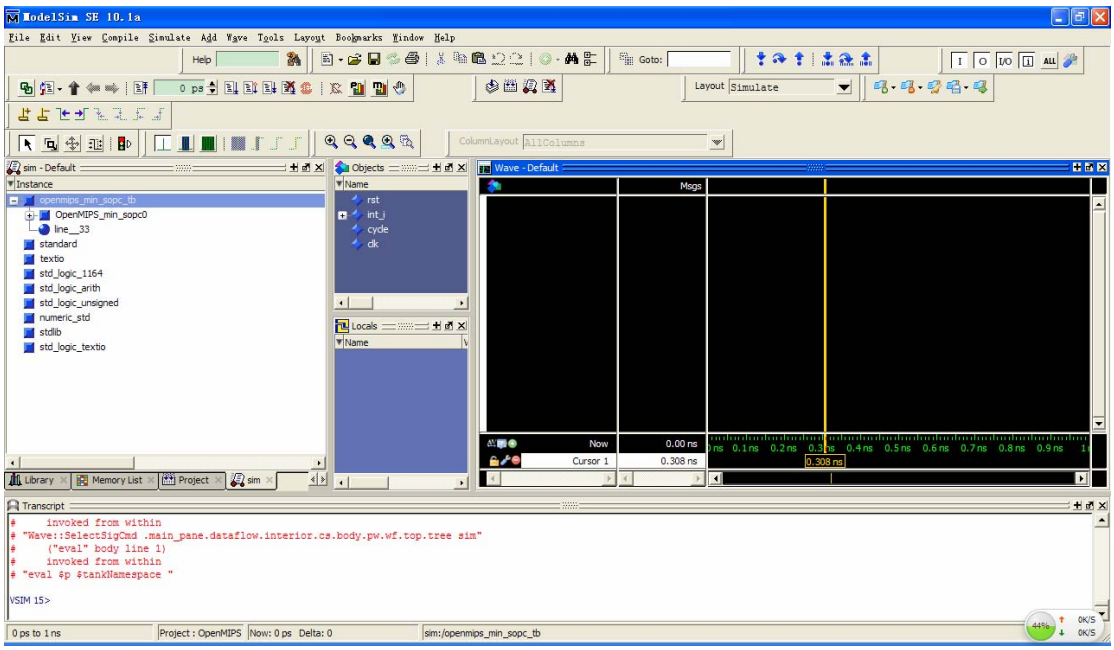
切换到 Library 这个 Tab 窗体，展开 work 如下：



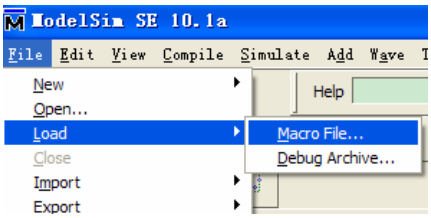
选择 openmips_min_sopc_tb，单击鼠标右键，选择 Simulate



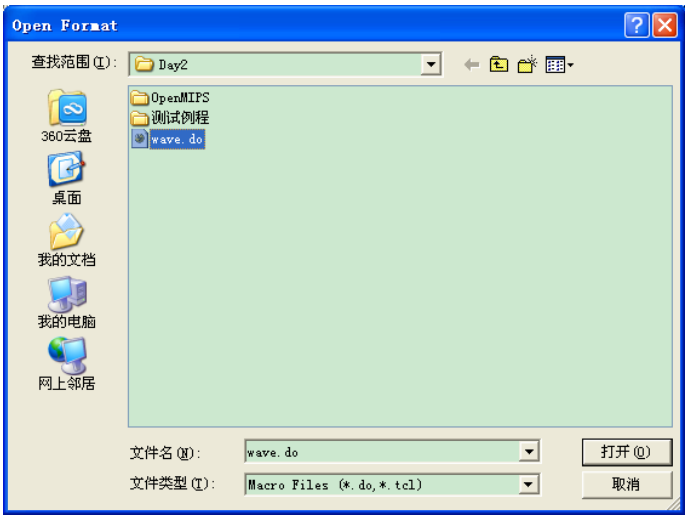
此时会出现波形显示窗口，如下：



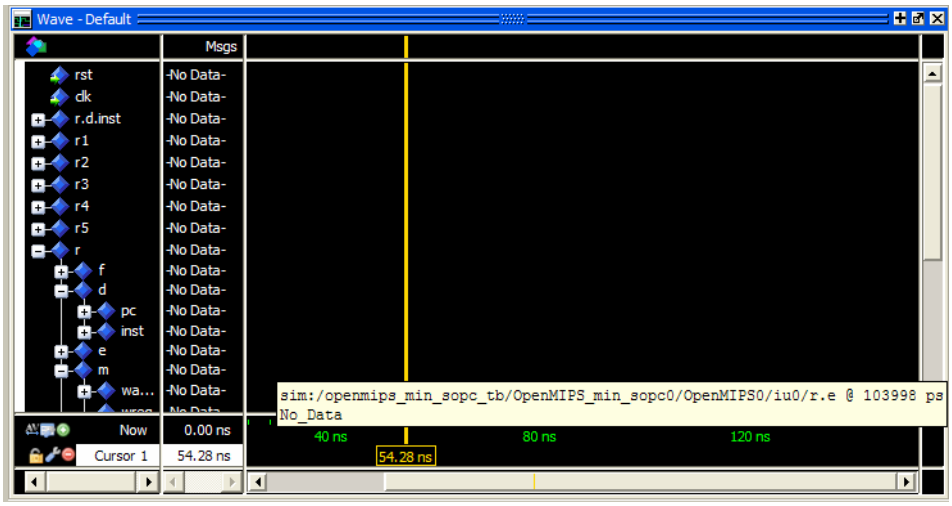
鼠标单击黑色的波形显示窗口，然后选择 file -> load -> Macro File，如下：



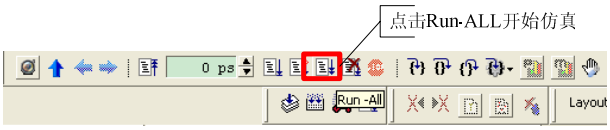
在打开的文件选择对话框中选择 Day2 文件下的 wave.do 文件，如下：



单击打开按钮，此时波形显示窗口就会出现一些要观察的信号，如下：



单击工具栏上的 run-all 按钮即可运行仿真，得到仿真结果。



第三天

主要内容

（1）解决流水线数据相关的问题
（2）实现其余的逻辑操作指令——AND、ANDI、LUI、NOR、OR、XOR、XORI

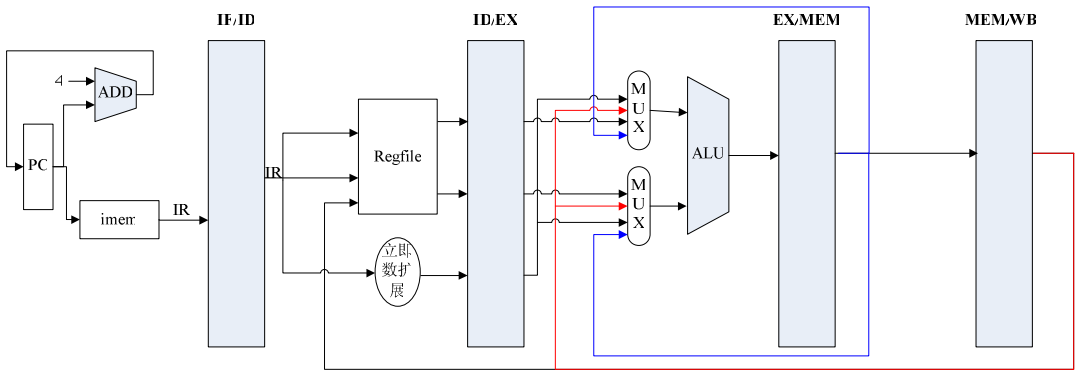
3.1 解决流水线数据相关的问题

流水线数据相关的由来就不再详细论述了，各位同学可以参考《计算机体系结构——量化研究方法》一书。在这里只给出 OpenMIPS 的解决方法，主要是修改在第二天中写成的流水线执行阶段的 opdata_select 过程，修改为如下，添加蓝色的代码：

```
procedure opdata_select(r,v: registers;
                        opdata1 : out word; opdata2: out word) is
  begin
   if r.e.rfe1='0' then
      opdata1 := r.e.imm;                          --源操作数是立即数
   elsif (r.m.waddr = r.e.rfa1 and r.m.wreg = '1' and r.e.rfe1 = '1') then
      opdata1 := r.m.result;                      --与上一条指令存在数据相关
   elsif (r.w.waddr = r.e.rfa1 and r.w.wreg = '1' and r.e.rfe1 = '1') then
      opdata1 := r.w.result;                      --与上上一条指令存在数据相关
   else
    opdata1 := r.e.reg1;                    --不存在数据相关
   end if;

   if r.e.rfe2='0' then
      opdata2 := r.e.imm;                          --源操作数是立即数
   elsif (r.m.waddr = r.e.rfa2 and r.m.wreg = '1' and r.e.rfe2 = '1' ) then
      opdata2 := r.m.result;                      --与上一条指令存在数据相关
   elsif (r.w.waddr = r.e.rfa2 and r.w.wreg = '1' and r.e.rfe2 = '1' ) then
      opdata2 := r.w.result;                      --与上上一条指令存在数据相关
   else
      opdata2 := r.e.reg2;                    --不存在数据相关
   end if;
end;
```

经过上述修改即可解决目前所能预料到的数据相关，在后期添加加载存储指令的时候还需要考虑更多数据相关的问题，届时还需要修改此处的 opdata_select 过程。

数据通路相应的也有如下变化：



主要是在执行阶段选择 ALU 的操作数的时候要考虑回写阶段、访存阶段，所以 MUX 的输入增加了，参考上图中蓝色、红色的连线。

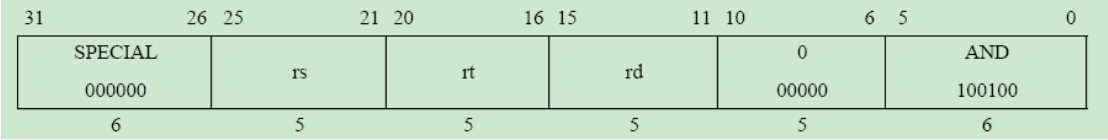
3.2 实现其余的逻辑操作指令

在第二天中实现了 ORI 指令，今天将实现剩余的七条逻辑操作指令，如下表所示：

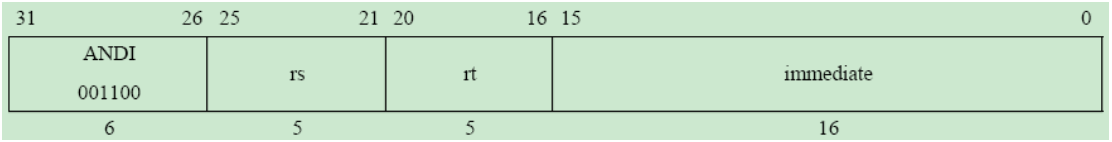
指令	用法	作用	说明
and	and rd,rs,rt	rd <- rs and rt	rs 寄存器的值逻辑“与”rt 寄存器的值，结果存入寄存器 rd
andi	andi rt,rs,immediate	rt <- rs and immediate	指令中的 16 位立即数进行零扩展为 32 位，然后逻辑“与”rs 寄存器的值，结果存入 rt 寄存器中
or	or rd,rs,rt	rd <- rs or rt	寄存器 rs 的值逻辑“或”寄存器 rt 的值，结果存入寄存器 rd
lui	lui rt,immediate	rt <- immediate || 0^{16}	指令中的立即数左移 16 位，结果存入寄存器 rt
xor	xor rd,rs,rt	rd <- rs xor rt	寄存器 rs 的值逻辑“异或”寄存器 rt 的值，结果存入寄存器 rd
xori	xori rt,rs,immediate	rt <- rs xor immediate	指令中的 16 位立即数零扩展为 32 位后，逻辑“异或”寄存器 rs 的值，结果存入寄存器 rt
nor	nor rd,rs,rt	rd <- rs nor rt	寄存器 rs 的值逻辑“或非”寄存器 rt 的值，结果存入寄存器 rd

上述各指令格式如下：

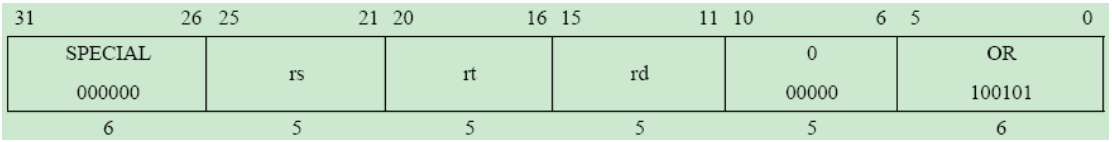
AND



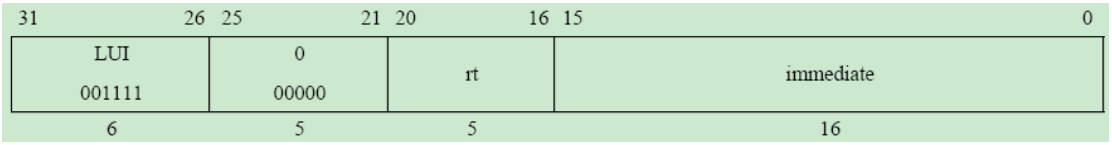
ANDI



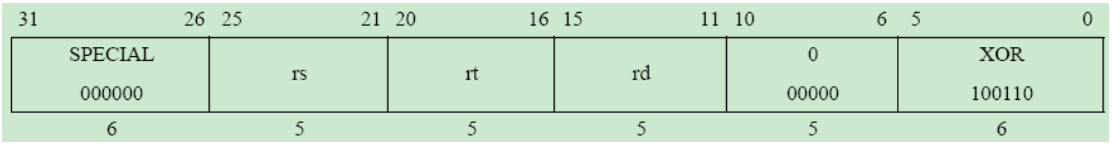
OR



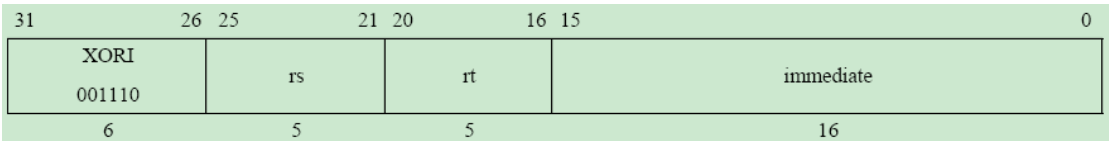
LUI



XOR



XORI



为了实现上述指令，只需要修改第二天中 iu.vhd 的 logic_op、inst_decode 两个过程，其中在 inst_decode 添加对上述指令的译码过程，如下（主要是蓝色代码部分）：

```
procedure inst_decode(inst : word;
    wreg : out std_logic;
    rdo : out std_logic_vector(4 downto 0);
    aluop  : out std_logic_vector(7 downto 0);
    alusel : out std_logic_vector(2 downto 0);
    rfe1, rfe2 : out std_logic; rfa1, rfa2 : out rfatype;
    imm : out word;
    new_cnt : out std_logic_vector(1 downto 0);
    inst_valid : out std_logic) is
    variable op : std_logic_vector(5 downto 0);
    variable op2 : std_logic_vector(4 downto 0);
    variable op3 : std_logic_vector(5 downto 0);
    variable op4 : std_logic_vector(4 downto 0);
  begin
```

```
op    := inst(31 downto 26);
op2   := inst(10 downto 6);
op3   := inst(5 downto 0);
op4   := inst(20 downto 16);
aluop := EXE_NOP_OP;
alusel := EXE_RES_NOP;
wreg := '0';
rdo := inst(15 downto 11);

rfe1 := '0'; rfe2 := '0';
rfa1 := inst(25 downto 21);
rfa2 := inst(20 downto 16);
imm := (others=>'0');
 new_cnt := "00";
 inst_valid := '0';
case op is
   when EXE_SPECIAL_INST =>
     case op2 is
        when "00000" =>
           case op3 is
             when EXE_OR   => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                      aluop := EXE_OR_OP;   alusel := EXE_RES_LOGIC;
                      inst_valid := '1';
              when EXE_AND  => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                      aluop := EXE_AND_OP;  alusel := EXE_RES_LOGIC;
                      inst_valid := '1';
               when EXE_XOR  => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                     aluop := EXE_XOR_OP;  alusel := EXE_RES_LOGIC;
                     inst_valid := '1';
              when EXE_NOR  => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                     aluop := EXE_NOR_OP;  alusel := EXE_RES_LOGIC;
                     inst_valid := '1';
               when others =>
           end case;
        when others =>
     end case;
   when EXE_ORI  => rfe1 := '1'; wreg :='1'; rdo := inst(20 downto 16);
                aluop := EXE_OR_OP;  alusel := EXE_RES_LOGIC;
                inst_valid := '1';
                imm(15 downto 0) := inst(15 downto 0);

   when EXE_ANDI => rfe1 := '1'; wreg :='1'; rdo := inst(20 downto 16);
                aluop := EXE_AND_OP; alusel := EXE_RES_LOGIC;
```

```
                        inst_valid := '1';
                        imm(15 downto 0) := inst(15 downto 0);

          when EXE_XORI => rfe1 := '1'; wreg :='1'; rdo := inst(20 downto 16);
                        aluop := EXE_XOR_OP; alusel := EXE_RES_LOGIC;
                        inst_valid := '1';
                        imm(15 downto 0) := inst(15 downto 0);

          when EXE_LUI =>  rfe1 := '1'; wreg :='1'; rdo := inst(20 downto 16);
                        aluop := EXE_OR_OP;  alusel := EXE_RES_LOGIC;
                        inst_valid := '1';
                        imm(31 downto 16) := inst(15 downto 0);
                        imm(15 downto 0) := (others => '0');
          when others =>
        end case;
end;
```

其中 rfe1 表示是否需要读取寄存器 1，rfe2 表示是否需要读取寄存器 2，wreg 表示是否有要写的目的寄存器，rdo 表示要写的目的寄存器地址，aluop 表示 ALU 要进行的算术操作类型，alusel 表示 ALU 算术操作的结果选择，inst_valid 表示指令是否有效，imm 是指令运行可能需要的立即数的值。

logic_op 过程修改如下（主要是添加了蓝色代码部分），其中的 r.e.aluop 就来自上面译码阶段的 inst_decode 过程：

```
procedure logic_op(r : registers; aluin1, aluin2: word;
  logicres : out word) is
    variable logicout : word;
   begin
      logicout := (others => '0');
       case r.e.aluop is
          when EXE_OR_OP  => logicout := aluin1 or aluin2;
          when EXE_AND_OP => logicout := aluin1 and aluin2;
          when EXE_NOR_OP => logicout := aluin1 nor aluin2;
          when EXE_XOR_OP => logicout := aluin1 xor aluin2;
          when others => logicout := (others => '-');
       end case;
       logicres := logicout;
end;
```

3.3 测试例程

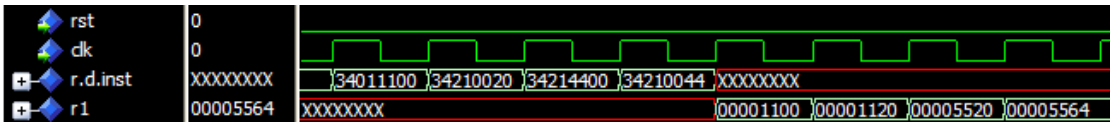
共有两个测试例程，Day3_1 用来测试数据相关的解决效果，位于 asm_test/Day3_1 目录下，Day3_2 用来测试其余逻辑操作指令是否实现正确。位于 asm_test/Day3_2 目录下。OpenMIPS 的代码参考 10_Days_make_OpenMIPS/Day3。

Day3_1：

程序如下：

```
    .org 0x0
    .global _start
    .set noat
_start:
   ori $1,$0,0x1100       # r1 = r0 | 0x1100 = 0x1100
   ori $1,$1,0x0020       # r1 = r1 | 0x0020 = 0x1120
   ori $1,$1,0x4400       # r1 = r1 | 0x4400 = 0x5520
   ori $1,$1,0x0044       # r1 = r1 | 0x0044 = 0x5564
```

程序执行过程中，r1 的值应该依次变为 0x1100、0x1120、0x5520、0x5564，ModelSim 仿真效果如下，可知 OpenMIPS 正确解决了数据相关的问题：

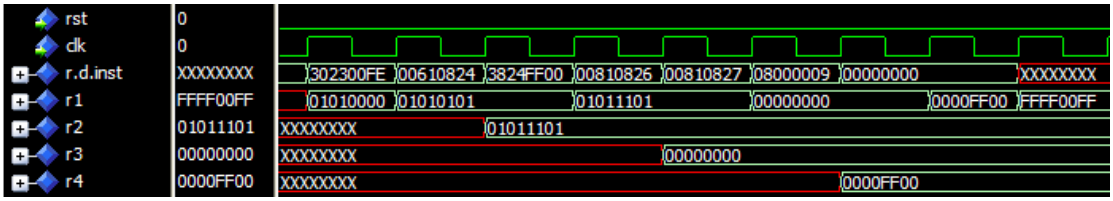


Day3_2：

程序如下：

```
    .org 0x0
    .global _start
    .set noat
_start:
   lui  $1,0x0101
   ori  $1,$1,0x0101
   ori  $2,$1,0x1100       # r2 = r1 | 0x1100 = 0x01011101
   or   $1,$1,$2           # r1 = r1 | r2 = 0x01011101
   andi $3,$1,0x00fe       # r3 = r1 & 0x00fe = 0x00000000
   and  $1,$3,$1           # r1 = r3 & r1 = 0x00000000
   xori $4,$1,0xff00       # r4 = r1 ^ 0xff00 = 0x0000ff00
   xor  $1,$4,$1           # r1 = r4 ^ r1 = 0x0000ff00
   nor  $1,$4,$1           # r1 = r4 ~^ r1 = 0xffff00ff   nor is "not or"
```

程序执行过程中寄存器 r1、r2、r3、r4 的变化分别参考程序注释，ModelSim 仿真效果如下，可知 OpenMIPS 正确实现了其余的逻辑操作指令。



仿真的步骤可以参考第二天中的说明。

第四天

主要内容

（1）实现移位操作指令——SLL、SLLV、SRA、SRAV、SRL、SRLV
（2）实现乘法、除法之外的所有算术操作指令——ADD、ADDI、ADDIU、ADDU、CLO、CLZ、SLT、SLTI、SLTIU、SLTU、SUB、SUBU

4.1 实现移位操作指令

今天将实现移位操作指令，所有的移位操作指令如下表所示：

指令	用法	作用	说明
sll	sll rd,rt,sa	rd <- rt << sa	将 rt 寄存器的值逻辑左移 sa 位，结果存入寄存器 rd
sllv	sllv rd,rt,rs	rd <- rt << rs	将 rt 寄存器的值逻辑左移 rs(5,0)位，结果存入寄存器 rd
sra	sra rd,rt,sa	rd <- rt >> sa (arithmetic)	将寄存器 rt 的值算术右移 sa 位，结果存入寄存器 rd
srav	srav rd,rt,rs	rd <- rt >> rs (arithmetic)	将寄存器 rt 的值算术右移 rs(5,0)位，结果存入寄存器 rd
srl	srl rd,rt,sa	rd <- rt >> sa	将寄存器 rt 的值逻辑右移 sa 位，结果存入寄存器 rd
srlv	srlv rd,rt,rs	rd <- rt >> rs	将寄存器 rt 的值逻辑右移 rs(5,0)位，结果存入寄存器 rd

要在第三天的基础上实现移位指令，首先需要修改流水线的译码阶段的 inst_decode 过程，添加如下代码（注意蓝色标注部分），实现对移位指令的译码：

```
procedure inst_decode(inst : word;
    wreg : out std_logic;
    rdo : out std_logic_vector(4 downto 0);
    aluop  : out std_logic_vector(7 downto 0);
    alusel : out std_logic_vector(2 downto 0);
    rfe1, rfe2 : out std_logic; rfa1, rfa2 : out rfatype;
    imm : out word;
    new_cnt : out std_logic_vector(1 downto 0);
    inst_valid : out std_logic) is
    variable op : std_logic_vector(5 downto 0);
    variable op2 : std_logic_vector(4 downto 0);
    variable op3 : std_logic_vector(5 downto 0);
```

```
  variable op4 : std_logic_vector(4 downto 0);
begin

     op    := inst(31 downto 26);
     op2   := inst(10 downto 6);
     op3   := inst(5 downto 0);
     op4   := inst(20 downto 16);
     aluop := EXE_NOP_OP;
     alusel := EXE_RES_NOP;
     wreg := '0';
     rdo := inst(15 downto 11);

     rfe1 := '0'; rfe2 := '0';
     rfa1 := inst(25 downto 21);
     rfa2 := inst(20 downto 16);
     imm := (others=>'0');
      new_cnt := "00";
      inst_valid := '0';
     case op is
        when EXE_SPECIAL_INST =>
          case op2 is
             when "00000" =>
                case op3 is
                   ……
                    when EXE_SLLV => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                              aluop := EXE_SLL_OP;  alusel := EXE_RES_SHIFT;
                              inst_valid := '1';
                    when EXE_SRLV => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                              aluop := EXE_SRL_OP;  alusel := EXE_RES_SHIFT;
                              inst_valid := '1';
                    when EXE_SRAV => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                              aluop := EXE_SRA_OP;  alusel := EXE_RES_SHIFT;
                              inst_valid := '1';
                    when others =>
                end case;
             when others =>
          end case;
  ……

if(inst(31 downto 21) = "00000000000" ) then
        if(inst(5 downto 0) = EXE_SLL ) then
            imm(4 downto 0) := inst(10 downto 6);
            rfe2 := '1';
            wreg :='1';
```

```
                rdo := inst(15 downto 11);
                aluop := EXE_SLL_OP;
                alusel := EXE_RES_SHIFT;
                inst_valid := '1';
            elsif (inst(5 downto 0) = EXE_SRL ) then
                imm(4 downto 0) := inst(10 downto 6);
                rfe2 := '1';
                wreg :='1';
                rdo := inst(15 downto 11);
                aluop := EXE_SRL_OP;
                alusel := EXE_RES_SHIFT;
                inst_valid := '1';
            elsif (inst(5 downto 0) = EXE_SRA ) then
                imm(4 downto 0) := inst(10 downto 6);
                rfe2 := '1';
                wreg :='1';
                rdo := inst(15 downto 11);
                aluop := EXE_SRA_OP;
                alusel := EXE_RES_SHIFT;
                inst_valid := '1';
            end if;
        end if;

end;
```

其中各个输出信号的含义在第三天已经解释过。

接着需要在流水线的执行阶段，依据移位指令的运算符、操作数，进行移位运算，然后依据指令选择移位运算结果或者逻辑操作结果作为最终的运算结果，准备写入 Regfile，如下（其中蓝色部分是新添加或者修改的）：

```
   --调用过程 logic_op 进行逻辑运算，结果存储在 ex_logic_res 中
   logic_op(r, ex_opdata1, ex_opdata2, ex_logic_res);

   --调用过程 shift_op 进行移位运算，结果存储在 ex_shift_res 中
   shift_op(r, ex_opdata1, ex_opdata2, ex_shift_res);

   --调用过程 alu_select，依据操作类型，选择对应的运算结果存储到 ex_result 中
   alu_select(r, ex_logic_res, ex_shift_res, ex_result);

   --将最终的运算结果传递到访存阶段
    v.m.result := ex_result;
```

在上述代码中调用了两个过程，一个是 shift_op，用来实现移位操作，一个是 alu_select，用来实现运算结果的选择，shift_op 的代码不再给出，童鞋们可以参考 10_Days_make_OpenMIPS/Day4_1/iu.vhd 文件，alu_select 的代码如下：

```
procedure alu_select(r : registers; logicout, shiftout: word;
                     res: out word) is
```

```
    variable aluresult : word;
  begin

     aluresult := (others => '0');
      case r.e.alusel is
         when EXE_RES_LOGIC => aluresult := logicout;
         when EXE_RES_SHIFT => aluresult := shiftout;
         when others => aluresult := zero32;
      end case;
      res := aluresult;
end;
```

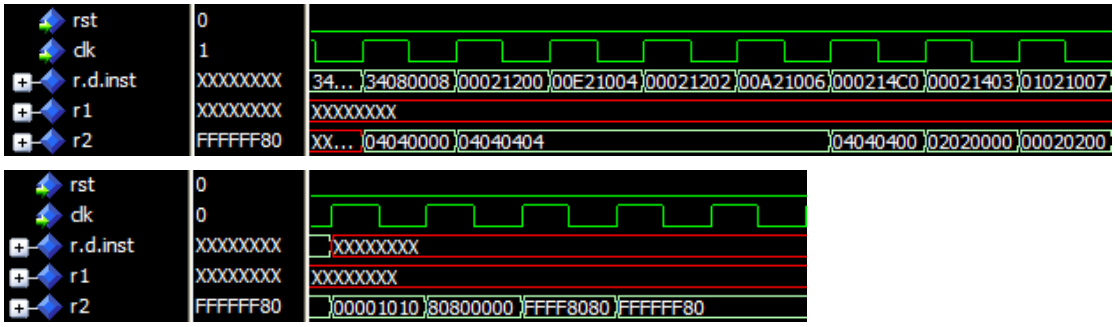
在后续过程中，随着算术指令、移动操作指令的实现，会逐步充实 **alu_select** 过程。

4.2 移位操作指令测试

测试程序如下，在 asm_test/Day4_1 文件夹下。

```
  .org 0x0
  .set noat
  .global _start
_start:
  lui   $2,0x0404
  ori   $2,$2,0x0404
  ori   $7,$0,0x7
  ori   $5,$0,0x5
  ori   $8,$0,0x8
  sll   $2,$2,8    # r2 = 0x40404040 sll 8  = 0x04040400
  sllv  $2,$2,$7   # r2 = 0x04040400 sll 7  = 0x02020000
  srl   $2,$2,8    # r2 = 0x02020000 srl 8  = 0x00020200
  srlv  $2,$2,$5   # r2 = 0x00020200 srl 5  = 0x00001010
  sll   $2,$2,19   # r2 = 0x00001010 sll 19 = 0x80800000
  sra   $2,$2,16   # r2 = 0x80800000 sra 16 = 0xffff8080
  srav  $2,$2,$8   # r2 = 0xffff8080 sra 8  = 0xffffff80
```

该程序执行过程中，寄存器 r2 的变化应该如注释中所列。在 ModelSim 中仿真运行，得到如下仿真结果：



从仿真结果可知，OpenMIPS 正确实现了移位操作指令。OpenMIPS 的代码参考 10_Days_make_OpenMIPS/Day4_1。

4.3 实现乘法、除法之外的所有算术操作指令

今天还要实现乘法、除法之外的所有算术操作指令，如下表所示：

指令	用法	作用	说明
add	add rd,rs,rt	rd <- rs + rt	寄存器 rs 的值与寄存器 rt 的值相加，如果没有溢出，那么结果存储在寄存器 rd 中，如果有溢出，那么不修改寄存器 rd，并且产生异常
addi	addi rt,rs,immediate	rt <- rs + immediate	指令中的 16 位立即数进行符号扩展为 32 位，然后与寄存器 rs 的值相加，如果没有溢出，那么结果存储在寄存器 rt 中，如果有溢出，那么不修改寄存器 rt，并且产生异常
addiu	addiu rt,rs,immediate	rt <- rs + immediate	指令中的 16 位立即数进行符号扩展为 32 位，然后与寄存器 rs 的值相加，结果存储到寄存器 rt 中，即使溢出，也不产生异常
addu	add rd,rs,rt	rd <- rs + rt	寄存器 rs 的值与寄存器 rt 的值相加，结果存储在寄存器 rd 中，即使溢出，也不产生异常
clo	clo rd,rs	rd <- count_leading_ones rs	从最高位开始数寄存器 rs 中 1 的个数，直到遇到第一个 0，如果 rs 中所有位都为 1，那么将 32 存储到 rd 中
clz	clz rd,rs	rd <- count_leading_zeros rs	从最高位开始数寄存器 rs 中 0 的个数，直到遇到第一个 1，如果 rs 中所有位都为 0，那么将 32 存储到 rd 中
slt	slt rd,rs,rt	rd <- (rs < rt)	寄存器 rs 的值与寄存器 rt 的值进行有符号数的比较，如果 rs 小，那么将 1 存储到寄存器 rd 中，反之，将 0 存储到寄存器 rd 中
slti	slti rt,rs,immediate	rt <- (rs < immediate)	指令中的 16 位立即数进行符号扩展为 32 位，与寄存器 rs 的值进行有符号数的比较，如果 rs 小，那么将 1 存储到寄存器 rt 中，反之，将 0 存储到寄存器 rt 中
sltiu	sltiu rt,rs,immediate	rt <- (rs < immediate)	指令中的 16 位立即数进行符号扩展为 32 位，与寄存器 rs 的值进行无符号数的比较，如果 rs 小，那么将 1 存储到寄存器 rt 中，反之，将 0 存储到寄存器 rt 中
sltu	sltu rd,rs,rt	rd <- (rs < rt)	寄存器 rs 的值与寄存器 rt 的值进行无符号数的比较，如果 rs 小，那么将 1 存储到寄存器 rd 中，反之，将 0 存储到寄存器 rd 中
sub	sub rd,rs,rt	rd <- rs - rt	寄存器 rs 的值与寄存器 rt 的值相减，如

			果没有溢出，那么结果存储在寄存器 rd 中，如果有溢出，那么不修改寄存器 rd，并且产生异常
subu	sub rd,rs,rt	rd <- rs - rt	寄存器 rs 的值与寄存器 rt 的值相减，结果存储在寄存器 rd 中，即使溢出也不产生异常

从上表可以知道有一些指令可能会产生溢出，为了简化，今天先不考虑溢出时，异常处理的情况。

上表中的指令大致可以分为三小类：加减运算、比较、count 类，如下：

- 加减运算：add、addi、addiu、addu、sub、subu
- 比较：slt、slti、sltiu、sltu
- count 类：clo、clz

添加思路与 4.1 节添加移位指令的思路一致，在过程 inst_decode 中增加对上述算术指令的译码，另外，增加过程 arithmetic_op 专门处理算术运算，然后在流水线的执行阶段调用该过程，最后，修改 alu_select 过程。

译码阶段的 inst_decode 过程修改如下，注意其中蓝色标注的部分：

```
procedure inst_decode(inst : word;
    wreg : out std_logic;
    rdo : out std_logic_vector(4 downto 0);
    aluop  : out std_logic_vector(7 downto 0);
    alusel : out std_logic_vector(2 downto 0);
    rfe1, rfe2 : out std_logic; rfa1, rfa2 : out rfatype;
    imm : out word;
    new_cnt : out std_logic_vector(1 downto 0);
    inst_valid : out std_logic) is
    variable op : std_logic_vector(5 downto 0);
    variable op2 : std_logic_vector(4 downto 0);
    variable op3 : std_logic_vector(5 downto 0);
    variable op4 : std_logic_vector(4 downto 0);
  begin

      op    := inst(31 downto 26);
      op2   := inst(10 downto 6);
      op3   := inst(5 downto 0);
      op4   := inst(20 downto 16);
      aluop := EXE_NOP_OP;
      alusel := EXE_RES_NOP;
      wreg := '0';
      rdo := inst(15 downto 11);

      rfe1 := '0'; rfe2 := '0';
      rfa1 := inst(25 downto 21);
      rfa2 := inst(20 downto 16);
```

```
        imm := (others=>'0');
        new_cnt := "00";
        inst_valid := '0';
        case op is
           when EXE_SPECIAL_INST =>
             case op2 is
                when "00000" =>
                   case op3 is
                       ……
                       when EXE_SLT  => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                            aluop := EXE_SLT_OP;  alusel := EXE_RES_ARITHMETIC;
                            inst_valid := '1';
                       when EXE_SLTU => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                            aluop := EXE_SLTU_OP; alusel := EXE_RES_ARITHMETIC;
                            inst_valid := '1';
                       when EXE_ADD  => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                            aluop := EXE_ADD_OP;  alusel := EXE_RES_ARITHMETIC;
                            inst_valid := '1';
                       when EXE_ADDU => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                            aluop := EXE_ADDU_OP; alusel := EXE_RES_ARITHMETIC;
                            inst_valid := '1';
                       when EXE_SUB  => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                            aluop := EXE_SUB_OP;  alusel := EXE_RES_ARITHMETIC;
                            inst_valid := '1';
                       when EXE_SUBU => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                            aluop := EXE_SUBU_OP; alusel := EXE_RES_ARITHMETIC;
                            inst_valid := '1';
                       when others =>
                   end case;
                when others =>
             end case;
           ……
           when EXE_SLTI => rfe1 := '1'; wreg :='1'; rdo := inst(20 downto 16);
                        aluop := EXE_SLT_OP; alusel := EXE_RES_ARITHMETIC;
                        inst_valid := '1';
                        imm(15 downto 0) := inst(15 downto 0);
                        imm(31 downto 16) := (others => inst(15));
         when EXE_SLTIU => rfe1 := '1'; wreg :='1'; rdo := inst(20 downto 16);
                         aluop := EXE_SLTU_OP; alusel := EXE_RES_ARITHMETIC;
                         inst_valid := '1';
                         imm(15 downto 0) := inst(15 downto 0);
                         imm(31 downto 16) := (others => inst(15));
         when EXE_ADDI => rfe1 := '1'; wreg :='1'; rdo := inst(20 downto 16);
                          aluop := EXE_ADDI_OP; alusel := EXE_RES_ARITHMETIC;
```

```
                        inst_valid := '1';
                        imm(15 downto 0) := inst(15 downto 0);
                         imm(31 downto 16) := (others => inst(15));
          when EXE_ADDIU => rfe1 := '1'; wreg :='1'; rdo := inst(20 downto 16);
                        aluop := EXE_ADDIU_OP; alusel := EXE_RES_ARITHMETIC;
                        inst_valid := '1';
                        imm(15 downto 0) := inst(15 downto 0);
                        imm(31 downto 16) := (others => inst(15));
          when EXE_SPECIAL2_INST =>
             case op3 is
               when EXE_CLZ   => rfe1 := '1'; rfe2 := '0'; wreg :='1';
                         aluop := EXE_CLZ_OP;   alusel := EXE_RES_ARITHMETIC;
                         inst_valid := '1';
               when EXE_CLO   => rfe1 := '1'; rfe2 := '0'; wreg :='1';
                         aluop := EXE_CLO_OP;   alusel := EXE_RES_ARITHMETIC;
                         inst_valid := '1';
               when others =>
             end case;
          when others =>
       end case;
      ……

end;
```

主要是依据指令设置 Regfile 模块的读写信号，以及 ALU 的运算类型。在流水线执行阶段添加的过程 arithmetic_op 如下，其中依据算术操作指令类型进行算术运算：

```
procedure arithmetic_op(r : registers; aluin1, aluin2: word;
  arithmeticres : out word; overflow : out boolean) is
    variable aluin2_mux,result : word;
    variable comp : std_logic;
    variable comp_a,comp_b,compareres : word;
    variable countres : word;
   begin
      overflow := false;
      countres := (others => '0');

    --如果是指令 clz，那么调用函数 find_first_one 计算操作数 aluin1 从高位
    --开始的第一个 1 之前的 0 的个数，反之将 aluin1 取反，再调用函数 find_first_one
    --实际效果就是计算操作数 aluin1 从高位开始的第一个 0 之前的 1 的个数
      if r.e.aluop = EXE_CLZ_OP then
         countres := find_first_one(aluin1 );
      else
         countres := find_first_one(not aluin1);
      end if;
```

```
--如果是符号数比较，那么设置 comp 为 1，反之，设置 comp 为 0
if r.e.aluop = EXE_SLT_OP then
  comp := '1';            --是有符号数比较
else
  comp := '0';            --是无符号数比较
end if;

--以下进行比较运算
comp_a := (aluin1(31) xor comp) & aluin1(30 downto 0);
comp_b := (aluin2(31) xor comp) & aluin2(30 downto 0);

if comp_a < comp_b then
  compareres := (others => '0');
  compareres(0) := '1';
else
  compareres := (others => '0');
end if;

  --如果是减法，那么将操作数 aluin2 取反加 1，将减法转化为加法
  if(r.e.aluop = EXE_SUB_OP or r.e.aluop = EXE_SUBU_OP) then
    aluin2_mux := (not aluin2) + 1;
  else
    aluin2_mux := aluin2;
  end if;

  --result 是减法、加法运算结果
  result := aluin1 + aluin2_mux;

  --依据具体的算术指令，选择上面几个运算结果中的一个，作为最终算术运算结果
     case r.e.aluop is
        when EXE_ADD_OP | EXE_ADDI_OP =>   --add、addi 指令要考虑溢出的情况
            if( (aluin1(31)='1' and aluin2_mux(31)='1' and
                 result(31)='0') or
                (aluin1(31)='0' and aluin2_mux(31)='0' and
                 result(31)='1')) then
               overflow := true;
            end if;
        when EXE_SUB_OP =>           --sub 指令要考虑溢出的情况
            if( aluin1(31)='0' and aluin2_mux(31)='1' and result(31)='1' )
              then
               overflow := true;
            end if;
        --比较指令的结果是 compareres
        when EXE_SLT_OP | EXE_SLTU_OP => result := compareres;
```

```
            --clo、clz 指令的结果是 countres
            when EXE_CLZ_OP | EXE_CLO_OP => result := countres;
            when others  => null;
         end case;

        --算术运算的最终结果存放在 arithmeticres 中
        arithmeticres := result;

end;
```

修改流水线执行阶段如下：

```
  --调用过程 logic_op 进行逻辑运算，结果存储在 ex_logic_res 中
  logic_op(r, ex_opdata1, ex_opdata2, ex_logic_res);

  --调用过程 shift_op 进行移位运算，结果存储在 ex_shift_res 中
  shift_op(r, ex_opdata1, ex_opdata2, ex_shift_res);

  --调用过程 arithmetic_op 进行算术运算，结果存储在 ex_arithmetic_op 中
  arithmetic_op(r, ex_opdata1, ex_opdata2, ex_arithmetic_res, overflow);

  --对于 add\addi\sub 指令，如果运算有溢出，那么不修改目的寄存器
  if(overflow = true) then
    v.m.wreg := '0';
  end if;

  --调用过程 alu_select，依据操作类型，选择对应的运算结果存储到 ex_result 中
  alu_select(r, ex_logic_res, ex_shift_res, ex_arithmetic_res, ex_result);

  --将最终的运算结果传递到访存阶段
   v.m.result := ex_result;
```

主要修改的地方是在其中调用了过程 arithmetic_op，注意：如果 overflow 为 true，那么不修改目的寄存器，所以需要设置 v.w.wreg 为 0。

最后需要修改 alu_select 过程，其中将算术运算的结果也作为执行结果的选择之一，如下：

```
procedure alu_select(r : registers; logicout, shiftout, arithmeticres: word;
                     res: out word) is
    variable aluresult : word;
  begin

     aluresult := (others => '0');
      case r.e.alusel is
         when EXE_RES_LOGIC => aluresult := logicout;
         when EXE_RES_SHIFT => aluresult := shiftout;
         when EXE_RES_ARITHMETIC => aluresult := arithmeticres;
```

```
            when others => aluresult := zero32;
        end case;
        res := aluresult;
 end;
```

4.4 乘法、除法外的所有算术操作指令测试

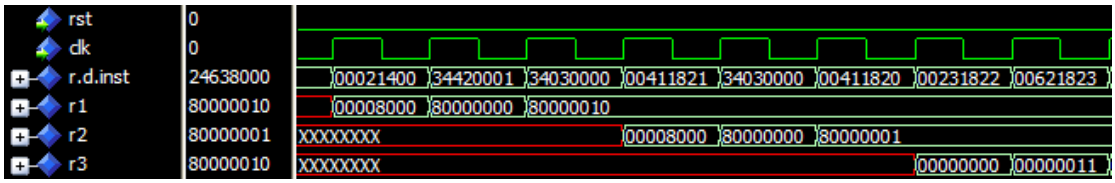
测试程序如下，分为三部分，分别测试上面分出的三类指令（加减运算、比较、count 类），在 asm_test/Day4_2 目录下：

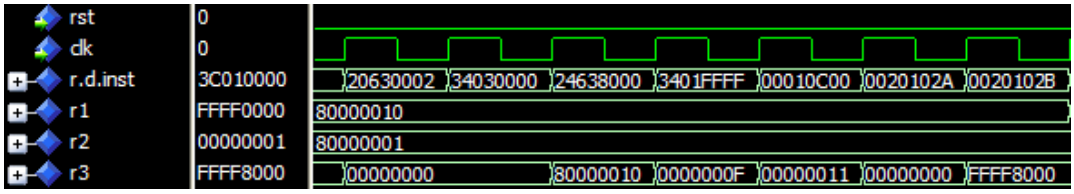
```
  .org 0x0
  .set noat
  .global _start
_start:

  ######### add\addi\addiu\addu\sub\subu ##########

  ori  $1,$0,0x8000          # r1 = 0x8000
  sll  $1,$1,16             # r1 = 0x80000000
  ori  $1,$1,0x0010          # r1 = 0x80000010

  ori  $2,$0,0x8000          # r2 = 0x8000
  sll  $2,$2,16             # r2 = 0x80000000
  ori  $2,$2,0x0001          # r2 = 0x80000001

  ori  $3,$0,0x0000          # r3 = 0x00000000
  addu $3,$2,$1             # r3 = 0x00000011
  ori  $3,$0,0x0000          # r3 = 0x00000000
  add  $3,$2,$1             # overflow,r3 keep 0x00000000

  sub   $3,$1,$3            # r3 = 0x80000010
  subu  $3,$3,$2            # r3 = 0xF

  addi $3,$3,2              # r3 = 0x11
  ori  $3,$0,0x0000          # r3 = 0x00000000
  addiu $3,$3,0x8000         # r3 = 0xffff8000

  #########    slt\sltu\slti\sltiu    ##########

  or   $1,$0,0xffff          # r1 = 0xffff
  sll  $1,$1,16             # r1 = 0xffff0000
  slt  $2,$1,$0             # r2 = 1
  sltu $2,$1,$0              # r2 = 0
  slti $2,$1,0x8000          # r2 = 1
```

```
sltiu $2,$1,0x8000        # r2 = 1

#########        clo\clz        ##########

lui $1,0x0000         # r1 = 0x00000000
clo $2,$1            # r2 = 0x00000000
clz $2,$1            # r2 = 0x00000020

lui $1,0xffff         # r1 = 0xffff0000
ori $1,$1,0xffff      # r1 = 0xffffffff
clz $2,$1            # r2 = 0x00000000
clo $2,$1            # r2 = 0x00000020

lui $1,0xa100         # r1 = 0xa1000000
clz $2,$1            # r2 = 0x00000000
clo $2,$1            # r2 = 0x00000001

lui $1,0x1100         # r1 = 0x11000000
clz $2,$1            # r2 = 0x00000003
clo $2,$1            # r2 = 0x00000000
```

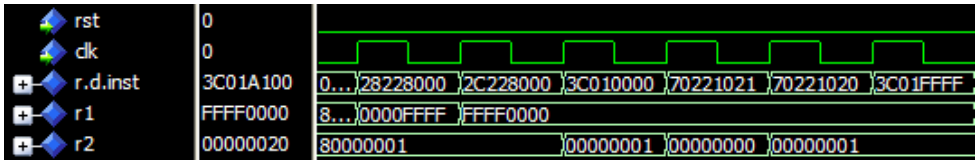
如果 OpenMIPS 实现正确，那么 r1、r2、r3 的变化应该与程序的注释一致。使用 ModelSim 仿真得到如下结果，观察 r1、r2、r3 的变化可知，OpenMIPS 正确实现了乘法、除法之外的所有算术操作指令。OpenMIPS 的代码参考 10_Days_make_OpenMIPS/Day4_2。

add、addi、addiu、addu、sub、subu 指令的仿真结果

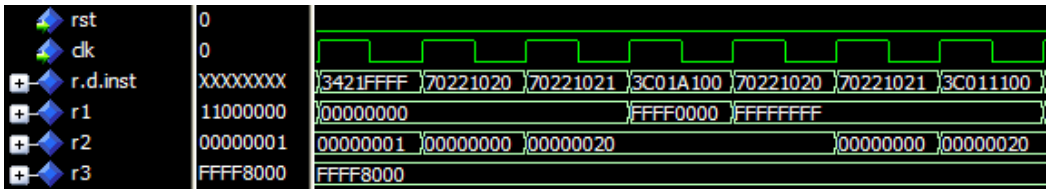


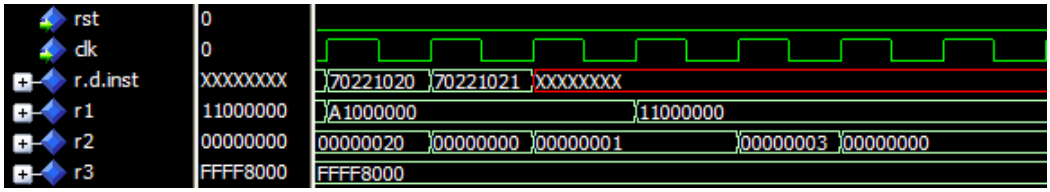


slt、slti、sltu、sltiu 指令的仿真结果



clo、clz 指令的仿真结果


rst
clk
r.d.inst
r1
r2
r3
0
0
XXXXXXXX
11000000
00000001
FFFF8000
3421FFFF
70221020
70221021
3C01A100
70221020
70221021
3C011100
00000000
FFFF0000
FFFFFFFF
00000001
00000000
00000020
00000000
00000020
FFFF8000


rst
clk
r.d.inst
r1
r2
r3
0
0
XXXXXXXX
11000000
00000000
FFFF8000
70221020
70221021
XXXXXXXX
A1000000
11000000
00000020
00000000
00000001
00000003
00000000
FFFF8000

第五天

主要内容

（1）实现乘法指令——MADD、MADDU、MSUB、MSUBU、MUL、MULT、MULTU
（2）实现除法指令——DIV、DIVU

5.1 实现乘法指令

今天要实现的是乘法、除法指令，涉及到多周期指令的执行问题，还需要新增加一对寄存器 HI、LO，用来存储乘法、除法的结果。乘法指令如下表所示：

指令	用法	作用	说明
madd	madd rs,rt	(hi,lo) <- (hi,lo) + rs * rt	寄存器 rs 的值与寄存器 rt 的值作为符号数相乘，累加到(hi,lo)
maddu	maddu rs,rt	(hi,lo) <- (hi,lo) + rs * rt	寄存器 rs 的值与寄存器 rt 的值作为无符号数相乘，累加到(hi,lo)
msub	madd rs,rt	(hi,lo) <- (hi,lo) - rs * rt	寄存器 rs 的值与寄存器 rt 的值作为符号数相乘，累减到(hi,lo)
msubu	maddu rs,rt	(hi,lo) <- (hi,lo) - rs * rt	寄存器 rs 的值与寄存器 rt 的值作为无符号数相乘，累减到(hi,lo)
mul	mul rd,rs,rt	rd <- rs * rt	寄存器 rs 的值与寄存器 rt 的值作为符号数相乘，结果的低 32bit 存储到寄存器 rd
mult	mult rs,rt	(hi,lo) <- rs * rt	寄存器 rs 的值与寄存器 rt 的值作为符号数相乘，结果的低 32bit 存储到寄存器 lo，高 32bit 存储到寄存器 hi
multu	multu rs,rt	(hi,lo) <- rs * rt	寄存器 rs 的值与寄存器 rt 的值作为无符号数相乘，结果的低 32bit 存储到寄存器 lo，高 32bit 存储到寄存器 hi

上表中与乘法有关的指令可以分为两小类：

- 简单乘法：mul、mult、multu
- 乘法与加法、减法复合运算：madd、maddu、msub、msubu

如果采用 FPGA 提供的乘法器，那么可以直接使用乘法操作符实现简单乘法，这样简单乘法就可以在一个时钟周期内完成，但是对于乘法与加法、减法复合运算，不仅要进行乘法，还需要进行一步加法或者减法，所以在 OpenMIPS 的设计中，上面第二小类指令使用两个时钟周期完成，第一个时钟周期进行乘法，第二个时钟周期进行加法、减法，也就是多周期指令。童鞋们还记得前几天在流水线译码阶段调用的 inst_decode 过程，其有一个输出 cnt

信号，该信号就表示指令是否是多周期的，前几天一直没有使用该信号，始终设置为 0，因为之前实现的都是单周期指令，今天就需要使用该信号了。

好了，我们分两步实现本小节的目标，第一步是实现简单乘法，不考虑多周期的问题，第二步是实现乘法与加法、减法复合运算，需要考虑多周期的问题。

5.1.1 简单乘法指令的实现

（1）因为 mult、multu 指令需要将乘法结果存储到寄存器 HI、LO 中，所以首先需要添加 HI、LO 寄存器，有以下步骤：

修改 stdlib.vhd 中访存阶段、回写阶段的寄存器，添加对 HI、LO 寄存器的写信号，如下：

```
--访存阶段的寄存器
type memory_reg_type is record
  waddr : rfatype;        --要写入的目的寄存器
  wreg  : std_logic;      --是否要写入目的寄存器
  result : word;          --要写入目的寄存器的值
  whilo : std_logic;      --是否要写 hi、lo 寄存器
  hilo : std_logic_vector(63 downto 0);  --要写入 hi、lo 寄存器的值
end record;

--回写阶段的寄存器
type write_reg_type is record
  result : word;          --要写入目的寄存器的值
  waddr  : rfatype;       --要写入目的寄存器
  wreg   : std_logic;     --是否要写入目的寄存器
  whilo  : std_logic;     --是否要写 hi、lo 寄存器
  hilo   : std_logic_vector(63 downto 0);  --要写入 hi、lo 寄存器的值
end record;
```

修改 iu.vhd，添加 HI、LO 寄存器，如下：

```
architecture rtl of iu is
  ……
  signal HI,LO : word;
  ……
```

（2）修改流水线译码阶段调用的 inst_decode 过程，添加对 mul、mult、multu 指令的译码过程，如下：

```
  procedure inst_decode(……) is
      ……
    begin

        op    := inst(31 downto 26);
        op2   := inst(10 downto 6);
        op3   := inst(5 downto 0);
```

```
        op4   := inst(20 downto 16);
        aluop := EXE_NOP_OP;
        alusel := EXE_RES_NOP;
        wreg := '0';
        rdo := inst(15 downto 11);

        rfe1 := '0'; rfe2 := '0';
        rfa1 := inst(25 downto 21);
        rfa2 := inst(20 downto 16);
        imm := (others=>'0');
         new_cnt := "00";
         inst_valid := '0';
        case op is
           when EXE_SPECIAL_INST =>
             case op2 is
                when "00000" =>
                   case op3 is
                     ……
                     when EXE_MULT => rfe1 := '1'; rfe2 := '1'; wreg :='0';
                                aluop := EXE_MULT_OP;  inst_valid := '1';
                     when EXE_MULTU=> rfe1 := '1'; rfe2 := '1'; wreg :='0';
                                aluop := EXE_MULTU_OP;  inst_valid := '1';
                       when others =>
                   end case;
                when others =>
             end case;
           ……
          when EXE_MUL => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                aluop := EXE_MUL_OP;   alusel := EXE_RES_MUL; inst_valid := '1';
                    ……
end;
```

注意的是这三条指令只有 mul 指令需要写目的寄存器，所以设置 wreg 为 1，其余两条指令都不需要写目的寄存器（需要修改 HI、LO 寄存器），所以设置 wreg 为 0。

（3）修改流水线的执行阶段，如下，主要是在其中调用新的过程 mul_op：

```
  --调用过程 logic_op 进行逻辑运算，结果存储在 ex_logic_res 中
  logic_op(r, ex_opdata1, ex_opdata2, ex_logic_res);

  --调用过程 shift_op 进行移位运算，结果存储在 ex_shift_res 中
  shift_op(r, ex_opdata1, ex_opdata2, ex_shift_res);

  --调用过程 arithmetic_op 进行算术运算，结果存储在 ex_arithmetic_op 中
  arithmetic_op(r, ex_opdata1, ex_opdata2, ex_arithmetic_res, overflow);
```

```
--对于 add\addi\sub 指令，如果运算有溢出，那么不修改目的寄存器
if(overflow = true) then
  v.m.wreg := '0';
end if;

--调用过程 mul_op 进行乘法运算，结果存储在 ex_mul_res 中
mul_op(r, ex_opdata1, ex_opdata2, ex_mul_res);

--调用过程 alu_select，依据操作类型，选择对应的运算结果存储到 ex_result 中
alu_select(r, ex_logic_res, ex_shift_res, ex_arithmetic_res,
           ex_mul_res, ex_result);

--将最终的运算结果传递到访存阶段
v.m.result := ex_result;

--依据乘法结果，给出新的 hilo、whilo 的值
 set_new_hilo(r, v, ex_mul_res, v.m.hilo, v.m.whilo );
```

其中调用过程 mul_op 进行指定的乘法运算，结果存储在 ex_mul_res 中，这是一个 64bit 的变量，该变量要传递到 alu_select 中，参与最终运算结果的选择。此外，对于 mult、multu 指令还要修改 HI、LO 寄存器，通过调用过程 set_new_hilo 实现此目的。具体代码如下：

```
procedure mul_op(r : registers; aluin1, aluin2: word;
  mulres : out std_logic_vector(63 downto 0)) is
    variable mulout : std_logic_vector(63 downto 0);
    variable opdata1, opdata2 : word;
  begin
     mulout := (others => '0');
     if((r.e.aluop = EXE_MUL_OP or r.e.aluop = EXE_MULT_OP )
        and aluin1(31) = '1') then
       opdata1 := (not aluin1) + 1;    --将有符号数转化为无符号数进行运算
     else
       opdata1 := aluin1;
     end if;

     if((r.e.aluop = EXE_MUL_OP or r.e.aluop = EXE_MULT_OP )
        and aluin2(31) = '1') then
       opdata2 := (not aluin2) + 1;    --将有符号数转化为无符号数进行运算
     else
       opdata2 := aluin2;
   end if;

   mulout := opdata1 * opdata2;

      case r.e.aluop is
         when EXE_MUL_OP | EXE_MULT_OP =>
```

```
                --有符号乘法，且积为负，那么对乘法结果取其补码
                if((aluin1(31) xor aluin2(31)) = '1') then
                  mulout := (not mulout) + 1;
                end if;
             when others =>
          end case;
          mulres := mulout;
end;

procedure alu_select(r : registers; logicout, shiftout, arithmeticres: word;
                     mulres: std_logic_vector(63 downto 0); res: out word) is
    variable aluresult : word;
  begin

      aluresult := (others => '0');
        case r.e.alusel is
           when EXE_RES_LOGIC => aluresult := logicout;
           when EXE_RES_SHIFT => aluresult := shiftout;
           when EXE_RES_ARITHMETIC => aluresult := arithmeticres;
           when EXE_RES_MUL => aluresult := mulres(31 downto 0);
           when others => aluresult := zero32;
        end case;
        res := aluresult;
end;

 procedure set_new_hilo(r, v: registers;
                        mul_res: in std_logic_vector(63 downto 0);
                        hilo: out std_logic_vector(63 downto 0);
                        whilo: out std_logic ) is
begin
  hilo := (others => '0');
  whilo := '0';

  --mult\multu 指令将乘法结果存放在 HI、LO 寄存器，所以设置 whilo 为 1，hilo 为乘法结果
  if((r.e.aluop = EXE_MULT_OP or r.e.aluop = EXE_MULTU_OP)) then
    hilo := mul_res;
    whilo := '1';
   else
    hilo := (others => '0');
    whilo := '0';
  end if;
end;
```

（4）修改流水线的访存阶段，主要是在其中增加对 HI、LO 寄存器的修改信息，如下：

```
    --将 HI、LO 的写信号传递到回写阶段
    v.w.whilo  := r.m.whilo;
    v.w.hilo   := r.m.hilo;
```

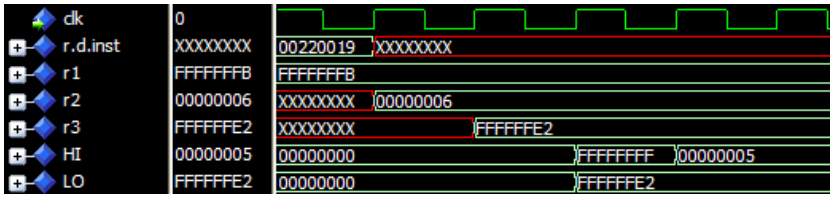
（5）修改流水线的时序逻辑，添加如下代码，实现最终对 HI、LO 寄存器的修改：

```
reg : process (clk)
begin
  if rising_edge(clk) then
      if(rst = '1') then
          ……
     else
         ……
         if(r.w.whilo = '1') then
           HI <= r.w.hilo(63 downto 32);
           LO <= r.w.hilo(31 downto 0);
         end if;
    end if;
  end if;
end process;
```

测试程序如下（参考 asm_test/Day5_1）：

```
  .org 0x0
  .set noat
  .global _start
_start:
  ori  $1,$0,0xffff
  sll  $1,$1,16
  ori  $1,$1,0xfffb          # r1 = -5
  ori  $2,$0,6               # r2 = 6
  mul  $3,$1,$2              # r3 = -30 = 0xfffffffe2

  mult $1,$2                 # hi = 0xffffffff
                             # lo = 0xfffffffe2

  multu $1,$2                # hi = 0x5
                             # lo = 0xfffffffe2
```

首先调用 mul 指令，结果存储在 r3 寄存器中，然后调用 mult、multu 指令，结果存储在 HI、LO 寄存器中，预期的执行结果参考代码注释，ModelSim 仿真结果如下图所示，从中可知 OpenMIPS 正确实现了 mul、mult、multu 指令。OpenMIPS 对应的代码位于 10_Days_make_OpenMIPS/Day5_1 目录下。



5.1.2 乘法与加法、减法复合运算指令的实现

本小节将实现乘法与加法、减法复合运算指令：madd、maddu、msub、msubu。这四条指令决定采用两个周期实现，第一个时钟周期进行乘法运算，第二个时钟周期进行加法、减法运算。实际就是要求流水线暂停一个时钟周期，在 OpenMIPS 的实现中，最终效果是取指、译码阶段暂停一个时钟周期，执行、访存、回写阶段正常向前流转。

（1）修改流水线译码阶段的 inst_decode 过程，添加对 madd、maddu、msub、msubu 指令的译码，如下：

```
procedure inst_decode(……) is
    ……
  begin

       op    := inst(31 downto 26);
       op2   := inst(10 downto 6);
       op3   := inst(5 downto 0);
       op4   := inst(20 downto 16);
       aluop := EXE_NOP_OP;
       alusel := EXE_RES_NOP;
       wreg := '0';
       rdo := inst(15 downto 11);

       rfe1 := '0'; rfe2 := '0';
       rfa1 := inst(25 downto 21);
       rfa2 := inst(20 downto 16);
       imm := (others=>'0');
  new_cnt := "00";
  inst_valid := '0';
       case op is
          when EXE_SPECIAL_INST =>
            case op2 is
               when "00000" =>
                  case op3 is
                    ……
                  end case;
               when others =>
            end case;
          ……
               when EXE_MADD  => rfe1 := '1'; rfe2 := '1'; wreg :='0';
```

```
                              aluop := EXE_MADD_OP;  alusel := EXE_RES_MUL;
                              new_cnt := "01"; inst_valid := '1';
                    when EXE_MADDU => rfe1 := '1'; rfe2 := '1'; wreg :='0';
                              aluop := EXE_MADDU_OP; alusel := EXE_RES_MUL;
                              new_cnt := "01"; inst_valid := '1';
                    when EXE_MSUB  => rfe1 := '1'; rfe2 := '1'; wreg :='0';
                              aluop := EXE_MSUB_OP;  alusel := EXE_RES_MUL;
                              new_cnt := "01"; inst_valid := '1';
                    when EXE_MSUBU => rfe1 := '1'; rfe2 := '1'; wreg :='0';
                              aluop := EXE_MSUBU_OP; alusel := EXE_RES_MUL;
                              new_cnt := "01"; inst_valid := '1';
                        ……
end;
```

注意其中设置 cnt 为 01，表示是多周期指令，同时设置 wreg 为 0 表示这四条指令都不需要写通用寄存器。

（2）在流水线执行阶段调用函数 get_new_hilo，得到最新的 HI、LO 寄存器的值，如下：

```
   --madd、maddu、msub、msubu 指令需要与 HI、LO 寄存器的值运算，此处调用
   --函数 get_new_hilo 得到最新的 HI、Lo 寄存器的值
    newhilo := get_new_hilo(r,v,HI,LO);
```

其中 get_new_hilo 函数定义如下：

```
 function get_new_hilo(r,v: registers; hi,lo: word) return std_logic_vector is
 begin
   if(r.m.whilo = '1' ) then  --如果访存阶段要写 HI、LO
     return r.m.hilo;
   elsif(r.w.whilo = '1' ) then  --如果回写阶段要写 HI、LO
     return r.w.hilo;
   else
     return hi & lo;           --否则，直接返回 HI、LO 寄存器的值
   end if;
 end;
```

（3）修改流水线执行阶段最后调用的 set_new_hilo 过程，如下：

```
 procedure set_new_hilo(r, v: registers;
           mul_res: in std_logic_vector(63 downto 0);
           new_hilo : in std_logic_vector(63 downto 0);
           hilo_temp: out std_logic_vector(63 downto 0);
           hilo: out std_logic_vector(63 downto 0); whilo: out std_logic;
           new_cnt : out std_logic_vector(1 downto 0) ) is
   variable temp : std_logic_vector(63 downto 0);
 begin
  hilo := new_hilo;
```

```
  whilo := '0';
  new_cnt := r.e.cnt;
  hilo_temp := new_hilo;    --目前最新的 HI、LO 寄存器的值

  --mult\multu 指令将乘法结果存放在 HI、LO 寄存器，所以设置 whilo 为 1，hilo 为乘法结果
  if((r.e.aluop = EXE_MULT_OP or r.e.aluop = EXE_MULTU_OP)) then
    hilo := mul_res;
    whilo := '1';
  elsif(r.e.aluop = EXE_MADD_OP or r.e.aluop = EXE_MADDU_OP) then
    if( r.e.cnt = "01" ) then  --MADD、MADDU 是多周期指令，cnt 为 1 表示是第一个周期
                               --此时执行乘法
      hilo_temp := mul_res;
      new_cnt := "00";         --设置 new_cnt 为 00，表示下一个时钟周期是多周期指令的
                               --执行阶段最后一个时钟周期
    else
      hilo := new_hilo + r.e.hilo;  --cnt 为 00，表示是第二个周期，此时执行加法
                                    --r.e.hilo 中存储的上一个周期计算出来的乘法结果
      whilo := '1';                 --第二个周期 MADD、MADDU 指令会修改 HI、LO 的值
    end if;
  elsif(r.e.aluop = EXE_MSUB_OP or r.e.aluop = EXE_MSUBU_OP) then
    temp := (not r.e.hilo) +1;  --求补码，将减法转化为加法，r.e.hilo 是上一个周期
                                --计算出来的乘法结果
    if( r.e.cnt = "01" ) then --MSUB、MSUBU 是多周期指令，cnt 为 1 表示是第一个周期
                              --此时执行乘法
      new_cnt := "00";
      hilo_temp := mul_res;   --设置 new_cnt 为 00，表示下一个时钟周期是多周期指令的
                              --执行阶段最后一个时钟周期
    else
      hilo := new_hilo + temp;   --cnt 为 00，表示是第二个周期，此时执行减法
      whilo := '1';              --第二个周期 MSUB、MSUBU 指令会修改 HI、LO 的值
    end if;
   else
    hilo := (others => '0');
    whilo := '0';
  end if;
end;
```

对于 madd、maddu 指令而言，当 cnt 为 1 时表示是第一个执行周期，此时将乘法结果赋值给 hilo_temp，并且设置新的 cnt 为 0，当 cnt 为 0 时表示是第二个执行周期，此时使用 hilo 的值加上一个周期计算出来的乘积（已经保存到了 r.e.hilo 中），并在这个周期设置 whilo 为 1，表示写 HI、LO 寄存器，写入的值就是刚刚计算出来的和。

对于 msub、msubu 指令而言，当 cnt 为 1 时表示是第一个执行周期，此时将乘法结果赋值给 hilo_temp，并且设置新的 cnt 为 0，当 cnt 为 0 时表示是第二个执行周期，此时使用 hilo 的值减去上一个周期计算出来的乘积（已经保存到了 r.e.hilo 中），并在这个周期设置 whilo 为 1，表示写 HI、LO 寄存器，写入的值就是刚刚计算出来的差。

（4）在流水线的执行阶段设置是否需要暂停执行阶段，如下：

```
if(r.e.cnt /= "00") then     --在 inst_decode 过程中依据指令设置 r.e.cnt 的值
    --对于多周期指令，设置该值为"1"，表示流水线因为多周期指令而暂停
    ex_stall_for_multicycle_inst := '1';
    v.m.whilo := '0';
    v.m.wreg := '0';
  else
    ex_stall_for_multicycle_inst := '0';
end if;
```

如果 cnt 不等于 0，那么设置 ex_stall_for_multicycle_inst 为 1，表示流水线由于多周期指令而暂停。

（5）修改取指阶段，如下：

```
if (rst = '1') then
  v.f.pc := "00000000000000000000000000000000";
elsif(ex_stall_for_multicycle_inst = '1') then
  --如果正处于执行阶段的是多周期指令，那么不改变 pc 的值
  v.f.pc := r.f.pc;
else
  --下一条指令是当前读取指令地址加 4
  v.f.pc := r.f.pc(31 downto 2) + 1;
end if;
```

如果 ex_stall_for_multicycle_inst 为 1，那么不改变 pc 的值。

（6）修改 rin 的值，不再是简单的等于 v，而是有所变化。如果正处于执行阶段的指令是多周期指令，那么保存该指令的操作数，访存、回写阶段的指令可以继续向前流转，同时保存执行阶段指令计算出的 HI、LO 的值，以及 cnt 的值，因为，对多周期指令而言，每执行一个周期，cnt 的值减 1，直到 cnt 为 0，此时表示当前是多周期指令的最后一个周期，

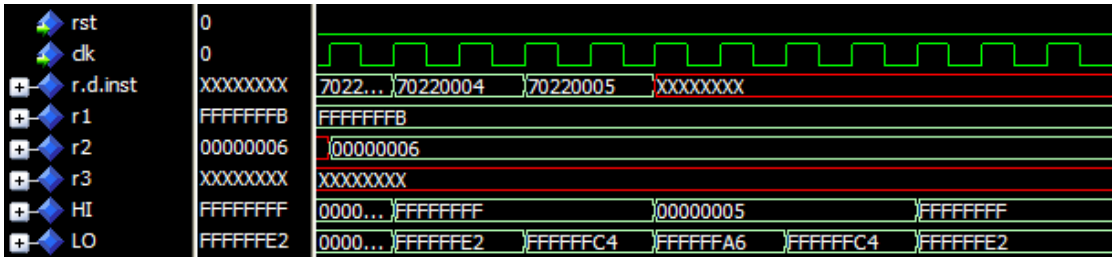
```
if(ex_stall_for_multicycle_inst = '1') then
  rin.e.reg1 <= ex_opdata1;
  rin.e.reg2 <= ex_opdata2;
  rin <= r;
  rin.e.cnt <= cnt_temp;
  rin.e.hilo <= hilo_temp;
  rin.m <= v.m;                --访存阶段继续向前流转
  rin.w <= v.w;                --回写阶段继续向前流转
else
  rin <= v;
end if;
```

上面的 hilo_temp 保存在 rin.e.hilo 中，将在时序逻辑中赋值给 r.e.hilo，对指令 madd、maddu、msub、msubu 而言，该值就是在第一个时钟周期中计算出来的乘积。

测试程序如下（参考 asm_test/Day5_2）：

```
   .org 0x0
   .set noat
   .global _start
_start:
   ori  $1,$0,0xffff
   sll  $1,$1,16
   ori  $1,$1,0xfffb          # r1 = -5
   ori  $2,$0,6               # r2 = 6

   mult $1,$2                # hi = 0xffffffff
                             # lo = 0xfffffffe2

   madd $1,$2                # hi = 0xffffffff
                             # lo = 0xfffffffc4

   maddu $1,$2               # hi = 0x5
                             # lo = 0xfffffffa6

   msub $1,$2                # hi = 0x5
                             # lo = 0xfffffffc4

   msubu $1,$2               # hi = 0xffffffff
                             # lo = 0xfffffffe2
```

预期执行效果如程序中注释所示，ModelSim 仿真如下，从仿真结果可知，OpenMIPS 正确实现了 madd、maddu、msub、msubu 指令，而且都是两个周期，如下，OpenMIPS 的代码位于 10_Days_make_OpenMIPS/Day5_2 目录下。



5.2 实现除法指令

OpenMIPS 支持的除法指令有两条：div、divu，其说明如下表所示：

指令	用法	作用	说明
div	div rs,rt	(hi,lo) <- rs / rt	寄存器 rs 的值与寄存器 rt 的值作为符号数相除，商存储到寄存器 lo，余数存储到寄存器 hi
divu	multu rs,rt	(hi,lo) <- rs / rt	寄存器 rs 的值与寄存器 rt 的值作为无符

			号数相除，商存储到寄存器 lo，余数存储到寄存器 hi

没有硬件模块直接实现除法，所以此处采用试商法实现除法，对于 32 位的除法，需要至少 32 个时钟周期才能得到除法结果。

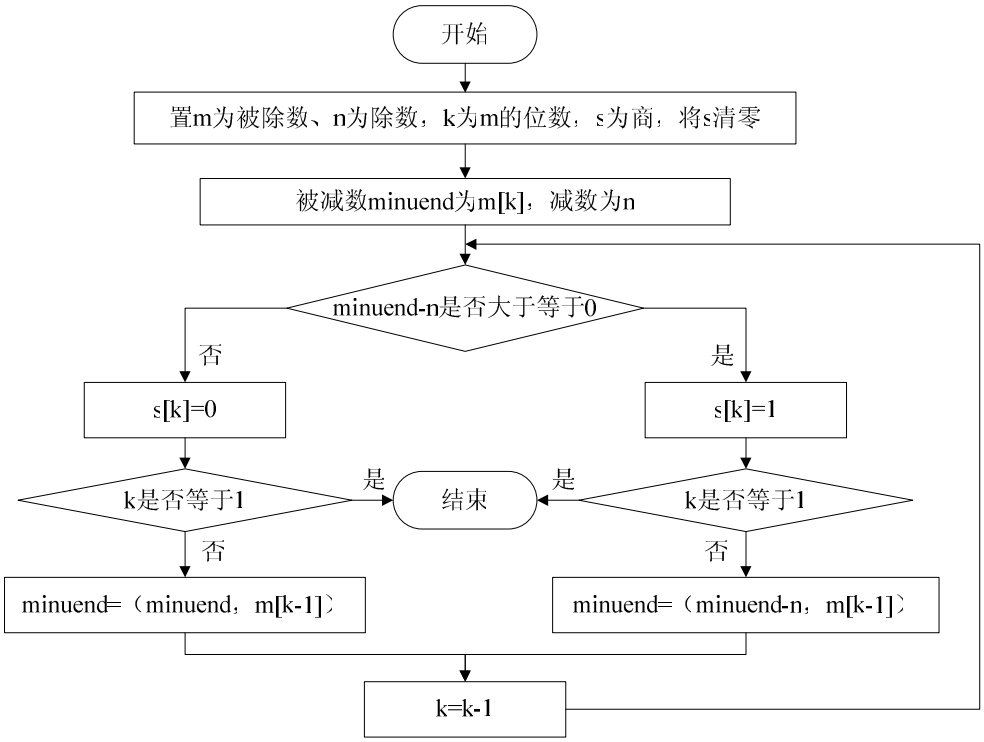
设被除数是 m，除数是 n，商保存在 s 中，被除数的位数是 k，其计算步骤如下（为了便于说明，在此处所有数据的最低位称之为第 1 位，而不称为第 0 位）：

（1）取出被除数的最高位 m[k]，使用被除数的最高位减去除数 n，如果结果大于等于 0，则商的 s[k]为 1，反之为 0。

（2）如果上一步得出的结果是 0，表示当前的被减数小于除数，则取出被除数 m[k-1]，与当前被减数组合为下一轮的被减数；如果上一步得出的结果是 1，表示当前的被减数大于除数，则利用第 2 步中减法的结果与被除数剩下的值的最高位 m[k-1]组合为下一轮的被减数。k 等于 k-1。

（3）新的被减数减去除数，如果结果大于等于 0，则商的 s[k]为 1，否则 s[k]为 0，后面的步骤重复 2-3，直到 k 等于 1。

上述步骤可以使用下图描述。

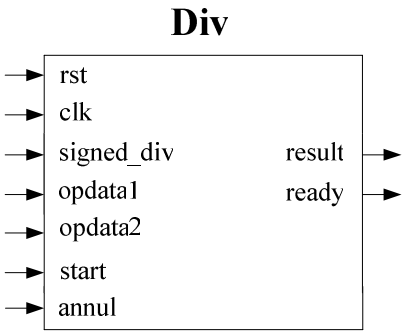


以 1101 除以 0010 为例， 采用试商法时的计算步骤如下表所示。

步骤		minuend	minuend-n	k	s	说明
置初值				4	0000	置被除数 m 为 1101，除数 n 为 0010，k 为 4，同时 s 清零
第 0 步	开始时	1	小于 0	4	0000	(1-0010) < 0 , s[4]=0
	结束时	11		3		新的 minuend=（minuend，m[k-1]）
第 1 步	开始时	11	大于 0	3	0100	(11-0010) > 0, s[3]=1

	结束时	10		2		新的 minuend=（11-0010，m[k-1]）=10
第 2 步	开始时	10	等于 0	2	0110	(10-0010) = 0, s[2]=1
	结束时	01		1		新的 minuend=（10-0010，m[k-1]）=01
第 3 步	开始时	01	小于 0	1	0110	(01-0010) < 0, s[1]=0
结束					0110	最终得到商为 0110

OpenMIPS 单独写了一个文件 div.vhd，其中实现除法运算，其接口如下图所示：



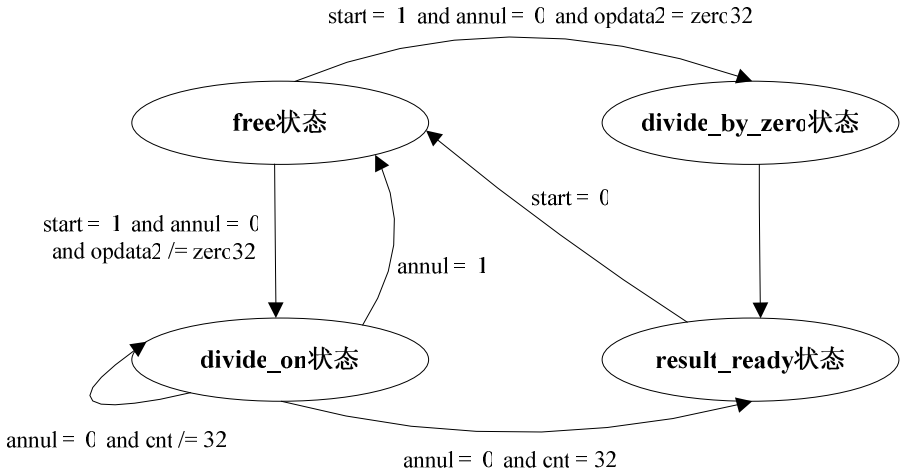
div 模块接口含义如下表所示：

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号，高电平有效
2	clk	1	输入	时钟信号
3	signed_div	1	输入	是否是有符号除法，为 1 表示是有符号除法
4	opdata1	32	输入	被除数
5	opdata2	32	输入	除数
6	start	1	输入	是否开始除法运算，为 1 表示开始除法运算
7	annul	1	输入	是否取消除法运算，为 1 表示取消除法运算
8	result	64	输出	除法输出结果，高 32 位是余数，低 32 位是商
9	ready	1	输出	除法结果是否得出，为 1 表示除法结果已经算出

div 模块主要部分是一个状态机，共有四个状态，如下：

- free：除法模块空闲
- divide_by_zero：除数是 0
- divide_on：除法运算进行过程中
- result_ready：除法运算结束

其状态转变如下图所示：



复位的时候 div 模块处于 free 状态，当输入 start 为 1，且 annul 为 0 时，表示除法操作开始：

- 如果除数 opdata2 为 0，那么进入 divide_by_zero 状态，直接给出除法结果，这里设置为-1，余数也为-1，然后进入 result_ready 状态，并通知 iu 模块除法运算结果得到，iu 模块设置 start 为 0，除法运算结束
- 如果除数 opdata2 不为 0，那么就进入 divide_on 状态，使用试商法，经过 32 个时钟周期，得出除法结果，然后进入 result_ready 状态，同样的通知 iu 模块除法运算结果得到，iu 模块设置 start 为 0，除法运算结束

从上面的表述可知 div 模块要与 iu 模块交互，所以需要修改 iu 模块的接口如下：

之前的 iu 模块接口

```
entity iu is
  port (
    clk   : in  std_logic;
    rst   : in  std_logic;
    imem_addr : out word;
    imem_data : in  word;
    dmem_addr : out word;
    dmem_we : out std_logic;
    dmem_wdata : out word;
    dmem_rdata : in word;
    dmem_sel  : out std_logic_vector(3 downto 0);
    rf_o   : out iregfile_in_type;
    rf_i   : in  iregfile_out_type
    );
end;
```

修改后的 iu 模块接口

```
entity iu is
  port (
    clk   : in  std_logic;
    rst   : in  std_logic;
```

```
    imem_addr : out word;
    imem_data : in  word;
    dmem_addr : out word;
    dmem_we : out std_logic;
    dmem_wdata : out word;
    dmem_rdata : in word;
    dmem_sel  : out std_logic_vector(3 downto 0);
    rf_o   : out iregfile_in_type;
    rf_i   : in  iregfile_out_type;
    signed_div : out std_logic;
    start_div : out std_logic;
    annul_div : out std_logic;
    div_result : in std_logic_vector(63 downto 0);
    div_result_ready : in std_logic;
    div_opdata1,div_opdata2 : out word
    );
end;
```

增加了与除法有关的信号输入输出，其含义可以参考 div 模块的接口说明，应该是十分清楚地，童鞋们一看即明白。div 模块的内容不再在教程中列出，可以参考 10_Days_make_OpenMIPS/Day5_3/div.vhd 文件。下面介绍一下为了实现除法指令而对流水线做的一些修改。

（1）修改流水线译码阶段的 inst_decode 过程，添加对指令 div、divu 译码的过程，如下：

```
  procedure inst_decode(……) is
      ……
    begin

        op    := inst(31 downto 26);
        op2   := inst(10 downto 6);
        op3   := inst(5 downto 0);
        op4   := inst(20 downto 16);
        aluop := EXE_NOP_OP;
        alusel := EXE_RES_NOP;
        wreg := '0';
        rdo := inst(15 downto 11);

        rfe1 := '0'; rfe2 := '0';
        rfa1 := inst(25 downto 21);
        rfa2 := inst(20 downto 16);
        imm := (others=>'0');
         new_cnt := "00";
         inst_valid := '0';
        case op is
```

```
          when EXE_SPECIAL_INST =>
            case op2 is
               when "00000" =>
                  case op3 is
                    ……
                    when EXE_DIV  => rfe1 := '1'; rfe2 := '1'; wreg :='0';
                                     aluop := EXE_DIV_OP;  new_cnt :="01";
                                     inst_valid := '1';
                    when EXE_DIVU => rfe1 := '1'; rfe2 := '1'; wreg :='0';
                                     aluop := EXE_DIVU_OP; new_cnt :="01";
                                     inst_valid := '1';
                     when others =>
                  end case;
               when others =>
            end case;
          ……
                   ……
end;
```

div、divu 两条除法指令都不需要写通用寄存器，所以设置 wreg 为 0，但是都是多周期指令，所以设置 new_cnt 为 01。

（2）在流水线的执行阶段会调用函数 set_new_hilo 函数，此处需要修改 set_new_hilo 函数，其中设置 iu 模块与除法有关的信号，如下：

```
--依据指令，设置 HI、LO 寄存器的值
procedure set_new_hilo(r, v: registers;
           mul_res: in std_logic_vector(63 downto 0);
           new_hilo : in std_logic_vector(63 downto 0);
           hilo_temp: out std_logic_vector(63 downto 0);
           hilo: out std_logic_vector(63 downto 0); whilo: out std_logic;
           new_cnt : out std_logic_vector(1 downto 0);
           start_div, signed_div, annul_div: out std_logic ) is
   variable temp : std_logic_vector(63 downto 0);
begin
  hilo := new_hilo;
  whilo := '0';
  new_cnt := r.e.cnt;
  hilo_temp := new_hilo;    --目前最新的 HI、LO 寄存器的值
  start_div := '0';
  annul_div := '0';
  signed_div := '0';

    ……
   elsif( v.e.aluop = EXE_DIV_OP ) then  --除法也是多周期指令
     --当 iu 模块的输入 div_result_ready 为 1 时，表示除法结束
```

```
      if(div_result_ready = '1') then
         new_cnt := "00";       --设置 new_cnt 为 00，表示下一个时钟周期是多周期指令的
                                --执行阶段最后一个时钟周期
         hilo := div_result;    --将除法结果写入 HI、LO 寄存器
         whilo := '1';
         start_div := '0';      --通知除法模块进入 free 状态
      else
        new_cnt := "01";
        whilo := '0';
        start_div := '1';       --通知除法模块开始除法运算
        signed_div := '1';      --div 指令进行的是有符号除法
      end if;
    elsif( v.e.aluop = EXE_DIVU_OP ) then
      if( div_result_ready = '1') then
         new_cnt := "00";
         hilo := div_result;
         whilo := '1';
         start_div := '0';
      else
        new_cnt := "01";
        whilo := '0';
        start_div := '1';
        signed_div := '0';       --divu 指令进行的是无符号除法
      end if;
    else
     hilo := new_hilo;
     whilo := '0';
   end if;
 end;
```

相应还需要修改流水线执行阶段调用 set_new_hilo 过程的代码，修改如下：

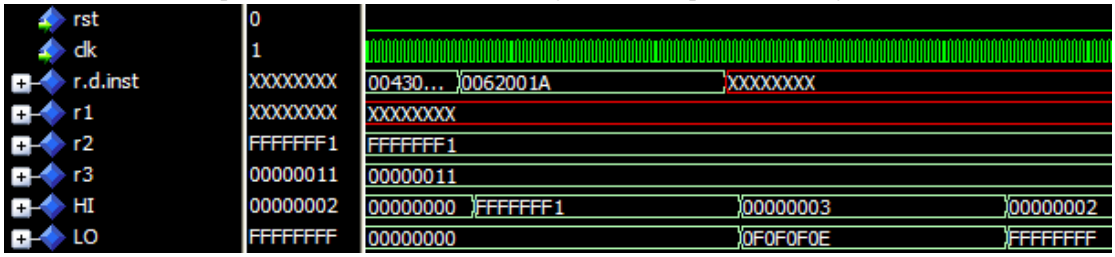
```
   set_new_hilo(r, v, ex_mul_res, newhilo, hilo_temp, v.m.hilo, v.m.whilo,
                cnt_temp, start_div_temp, signed_div_temp, annul_div_temp );

   annul_div <= annul_div_temp;
   start_div <= start_div_temp;
   signed_div <= signed_div_temp;
```

测试程序如下（参考 asm_test/Day5_3）：

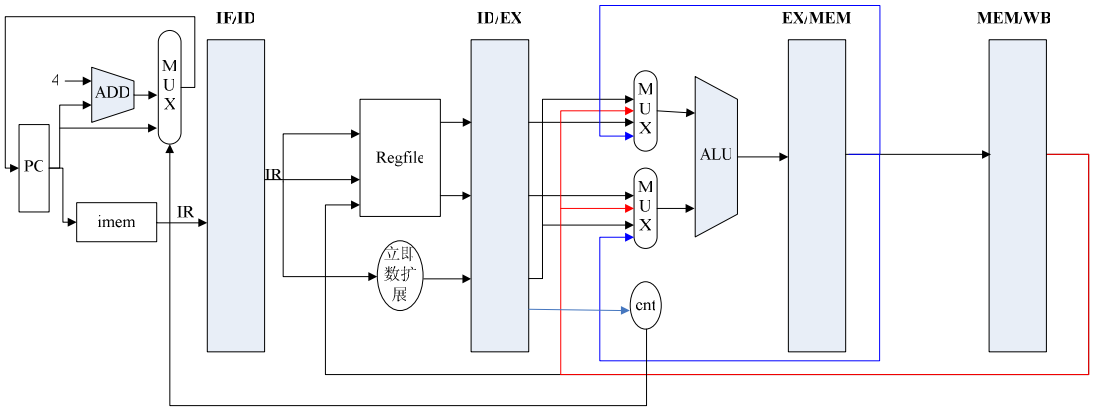
```
  .org 0x0
  .global _start
_start:
  ori  $2,$0,0xffff
  sll  $2,$2,16
  ori  $2,$2,0xfff1         # r2 = -15
```

```
ori  $3,$0,0x11           # r3 = 17

div $zero,$2,$3           # hi = 0xffffffff1
                          # lo = 0x0
divu $zero,$2,$3          # hi = 0x00000003
                          # lo = 0x0f0f0f0e
div  $zero,$3,$2          # hi = 2
                          # lo = 0xffffffff
```

预期执行效果如代码注释所列，ModelSim 仿真效果如下，从中可知 OpenMIPS 正确实现了除法指令。OpenMIPS 的代码位于 10_Days_make_OpenMIPS/Day5_3 目录下。



实现乘法、除法指令后的 OpenMIPS 数据流图如下，主要是 cnt 的值参与到了 pc 值的选择中，当 cnt 的值不为 00 时，pc 值保持不变，这样可以实现多周期指令。



第六天

主要内容

（1）实现移动操作指令——MFHI、MFLO、MOVN、MOVZ、MTHI、MTLO
（2）实现控制指令——NOP、SSNOP

6.1 实现移动操作指令

昨天可能比较痛苦吧，因为为了实现乘法、除法类指令，需要考虑多周期指令执行的情况，还需要增加除法模块，总之，昨天修改的内容很多，童鞋们都比较辛苦，今天就放松一下，实现比较简单的移动操作指令、控制指令，都是单周期指令。本小节介绍移动操作指令的实现。

OpenMIPS 支持的移动操作指令如下表所示：

指令	用法	作用	说明
mfhi	mfhi rd	rd <- hi	把 HI 寄存器的内容复制到寄存器 rd
mflo	mflo rd	rd <- lo	把 LO 寄存器的内容复制到寄存器 rd
movn	movn rd,rs,rt	if rt /= 0 then rd <- rs	如果寄存器 rt 的值不为 0，那么将寄存器 rs 的值复制到寄存器 rd
movz	movz rd,rs,rt	if rt = 0 then rd <- rs	如果寄存器 rt 的值为 0，那么将寄存器 rs 的值复制到寄存器 rd
mthi	mthi rs	hi <- rs	把寄存器 rs 的值复制到寄存器 HI
mtlo	mtlo rs	lo <- rs	把寄存器 rs 的值复制到寄存器 LO

为了实现移动操作指令，需要增加、修改的代码如下：

（1）修改流水线译码阶段调用的过程 inst_decode，在其中增加对移动操作类指令的译码，如下：

```
procedure inst_decode(……) is
    ……
  begin

      op    := inst(31 downto 26);
      op2   := inst(10 downto 6);
      op3   := inst(5 downto 0);
      op4   := inst(20 downto 16);
      aluop := EXE_NOP_OP;
      alusel := EXE_RES_NOP;
      wreg := '0';
      rdo := inst(15 downto 11);
```

```
        rfe1 := '0'; rfe2 := '0';
        rfa1 := inst(25 downto 21);
        rfa2 := inst(20 downto 16);
        imm := (others=>'0');
         new_cnt := "00";
         inst_valid := '0';
        case op is
           when EXE_SPECIAL_INST =>
             case op2 is
                when "00000" =>
                   case op3 is
                     ……
                     when EXE_MFHI => rfe1 := '0'; rfe2 := '0'; wreg :='1';
                               aluop := EXE_MFHI_OP; alusel := EXE_RES_MOVE;
                               inst_valid := '1';
                     when EXE_MTHI => rfe1 := '1'; rfe2 := '0'; wreg :='0';
                               aluop := EXE_MTHI_OP; inst_valid := '1';
                     when EXE_MFLO => rfe1 := '0'; rfe2 := '0'; wreg :='1';
                               aluop := EXE_MFLO_OP; alusel := EXE_RES_MOVE;
                               inst_valid := '1';
                     when EXE_MTLO => rfe1 := '1'; rfe2 := '0'; wreg :='0';
                               aluop := EXE_MTLO_OP; inst_valid := '1';
                     when EXE_MOVZ => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                               aluop := EXE_MOVZ_OP; alusel := EXE_RES_MOVE;
                               inst_valid := '1';
                     when EXE_MOVN => rfe1 := '1'; rfe2 := '1'; wreg :='1';
                               aluop := EXE_MOVN_OP; alusel := EXE_RES_MOVE;
                               inst_valid := '1';
                      when others =>
                   end case;
                when others =>
             end case;
           ……

end;
```

（2）在流水线执行阶段添加过程 move_op，其中执行移动操作类指令 movn、movz、mfhi、mflo，计算结果 ex_move_res 参与过程 alu_select 选择最终结果。如下：

```
  --调用过程 logic_op 进行逻辑运算，结果存储在 ex_logic_res 中
  logic_op(r, ex_opdata1, ex_opdata2, ex_logic_res);

  --调用过程 shift_op 进行移位运算，结果存储在 ex_shift_res 中
  shift_op(r, ex_opdata1, ex_opdata2, ex_shift_res);
```

```
  --调用过程 arithmetic_op 进行算术运算，结果存储在 ex_arithmetic_op 中
  arithmetic_op(r, ex_opdata1, ex_opdata2, ex_arithmetic_res, overflow);

  --对于 add\addi\sub 指令，如果运算有溢出，那么不修改目的寄存器
  if(overflow = true) then
    v.m.wreg := '0';
  end if;

  --调用过程 mul_op 进行乘法运算，结果存储在 ex_mul_res 中
  mul_op(r, ex_opdata1, ex_opdata2, ex_mul_res);

  --调用过程 move_op 进行移动操作类指令的执行，结果存储在 ex_move_res 中
  move_op(r, ex_opdata1, ex_opdata2, newhilo, ex_move_res, notmove);

  --对于 movn\movz 指令，如果条件不满足，那么 notmove 为 true，表示不修改目的寄存器
  --所以设置 v.m.wreg 为 0
  if(notmove = true) then
    v.m.wreg := '0';
  end if;

  --调用过程 alu_select，依据操作类型，选择对应的运算结果存储到 ex_result 中
  alu_select(r, ex_logic_res, ex_shift_res, ex_arithmetic_res, ex_move_res,
             ex_mul_res, ex_result);
```

其中 move_op 过程定义如下：

```
 procedure move_op(r : registers; aluin1, aluin2: word;
                newhilo : std_logic_vector(63 downto 0);
                moveres : out word; notmove : out boolean) is
    variable moveout,zeros : word;
   begin
      moveout := (others => '0');
      zeros := (others => '0');
      notmove := false;
       case r.e.aluop is
          when EXE_MFHI_OP  => moveout := newhilo(63 downto 32);
          when EXE_MFLO_OP => moveout := newhilo(31 downto 0);
          when EXE_MOVN_OP =>
               --movn 指令是条件寄存器不等于 0 的时候修改寄存器的值
               if aluin2 /= zeros then
                  moveout := aluin1;
               else
                  notmove := true;
               end if;
```

```
          when EXE_MOVZ_OP =>
               --movz 指令是条件寄存器等于 0 的时候修改寄存器的值
               if aluin2 = zeros then
                 moveout := aluin1;
               else
                 notmove := true;
               end if;
          when others => moveout := (others => '-');
       end case;
       moveres := moveout;
end;
```

（3）修改流水线执行阶段调用的过程 set_new_hilo，以实现指令 mthi、mthi，这两个指令分别修改 HI、LO 寄存器，如下：

```
--依据指令，设置 HI、LO 寄存器的值
procedure set_new_hilo(r, v: registers; aluin1, aluin2: word;
           mul_res: in std_logic_vector(63 downto 0);
           new_hilo : in std_logic_vector(63 downto 0);
           hilo_temp: out std_logic_vector(63 downto 0);
           hilo: out std_logic_vector(63 downto 0); whilo: out std_logic;
           new_cnt : out std_logic_vector(1 downto 0);
           start_div, signed_div, annul_div: out std_logic ) is
  variable temp : std_logic_vector(63 downto 0);
begin
  hilo := new_hilo;
  whilo := '0';
  new_cnt := r.e.cnt;
  hilo_temp := new_hilo;    --目前最新的 HI、LO 寄存器的值
  start_div := '0';
  annul_div := '0';
  signed_div := '0';

  ……
  elsif(r.e.aluop = EXE_MTHI_OP) then   --是 mthi 指令，需要写 HI 寄存器
     hilo(63 downto 32) := aluin1;
     hilo(31 downto 0) := new_hilo(31 downto 0); --LO 寄存器保持不变
     whilo := '1';
  elsif (v.e.aluop = EXE_MTLO_OP) then  --是 mtlo 指令，需要些 LO 寄存器
     hilo(31 downto 0) := aluin1;
     hilo(63 downto 32) := new_hilo(63 downto 32); --HI 寄存器保持不变
     whilo := '1';
   ……
   else
    hilo := new_hilo;
```

```
    whilo := '0';
  end if;
end;
```

经过以上修改即可实现所有的移动操作指令：movz、movn、mfhi、mflo、mthi、mtlo。

6.2 实现控制指令

MIPS32 指令手册中将 nop、ssnop 两条指令归类为控制指令（Instruction Control Instructions），对于 OpenMIPS 而言，这两条指令的作用是相同的，都是空指令，什么都不做，所以要实现这两条指令其实很简单，只需要修改流水线译码过程调用的 inst_decode 函数即可，在其中增加对 nop、ssnop 的译码过程，如下：

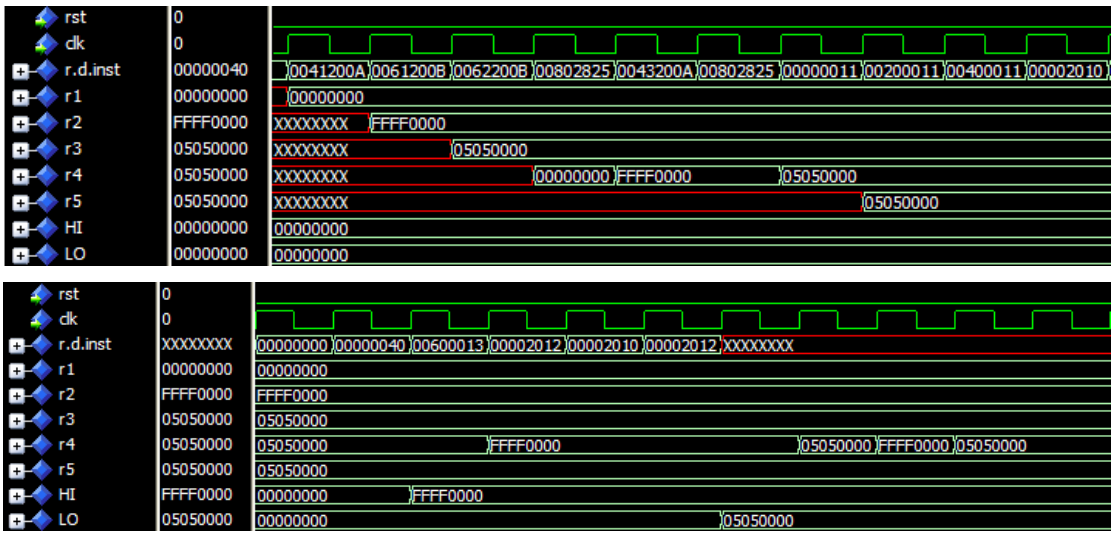
```
procedure inst_decode(……) is
    ……
  begin

       op    := inst(31 downto 26);
       op2   := inst(10 downto 6);
       op3   := inst(5 downto 0);
       op4   := inst(20 downto 16);
       aluop := EXE_NOP_OP;
       alusel := EXE_RES_NOP;
       wreg := '0';
       rdo := inst(15 downto 11);

       rfe1 := '0'; rfe2 := '0';
       rfa1 := inst(25 downto 21);
       rfa2 := inst(20 downto 16);
       imm := (others=>'0');
   new_cnt := "00";
   inst_valid := '0';
   ……
       if(inst = zero32 or inst = SSNOP) then  --指令 nop 的二进制编码等于 zero32
         rfe1 := '0'; rfe2 := '0';
         wreg :='0';
         aluop := EXE_NOP_OP;
         alusel := EXE_RES_NOP;
         inst_valid := '1';
       end if;
end;
```

测试程序如下（位于 asm_test/Day6 文件夹下）：

```
 .org 0x0
 .set noat
```

```
   .global _start
_start:
   lui $1,0x0000         # r1 = 0x00000000
   lui $2,0xffff         # r2 = 0xffff0000
   lui $3,0x0505         # r3 = 0x05050000
   lui $4,0x0000         # r4 = 0x00000000

   movz $4,$2,$1         # r4 = 0xffff0000
   movn $4,$3,$1         # r4 = 0xffff0000
   movn $4,$3,$2         # r4 = 0x05050000
   or   $5,$4,$0        # r5 = 0x05050000
   movz $4,$2,$3         # r4 = 0x05050000
   or   $5,$4,$0        # r5 = 0x05050000

   mthi $0              # hi = 0x00000000
   mthi $1              # hi = 0x00000000
   mthi $2              # hi = 0xffff0000
   mfhi $4              # r4 = 0xffff0000
   nop
   ssnop
   mtlo $3              # lo = 0x05050000
   mflo $4              # r4 = 0x05050000
   mfhi $4              # r4 = 0xffff0000
   mflo $4              # r4 = 0x05050000
```

其执行过程应该如注释中所列，在 ModelSim 中仿真执行，得到如下仿真结果，从中可知 OpenMIPS 正确实现了移动操作类指令、控制指令。OpenMIPS 的代码位于 10_days_make_OpenMIPS/Day6 文件夹下。



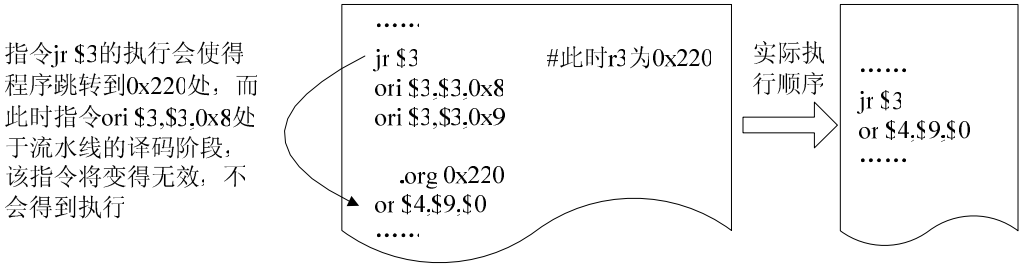
第七天

主要内容

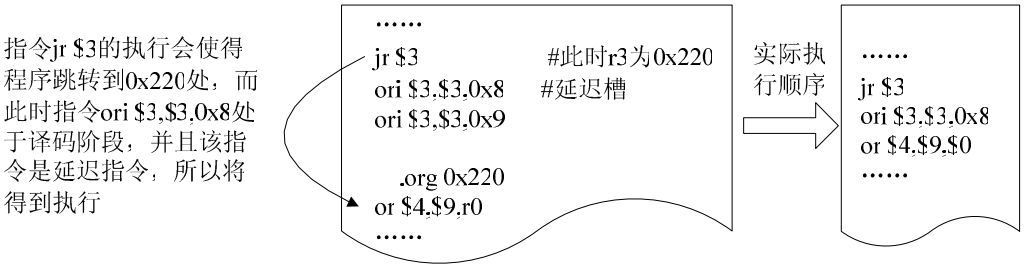
（1）实现跳转指令——J、JAL、JALR、JR
（2）实现分支指令——B、BAL、BEQ、BGEZ、BGEZAL、BGTZ、BLEZ、BLTZ、BLTZAL、BNE

7.1 实现跳转指令

在介绍跳转指令实现之前，先介绍一下延迟槽的概念。流水线有三种相关：数据相关、结构相关、控制相关。其中控制相关指流水线中的转移指令或者其他需要改写 PC 的指令造成的相关。这些指令改写了 PC 值，所以下一条执行的指令可能会发生变化，如果下一条执行的指令地址不是当前指令地址加 4，那么流水线处于译码阶段的指令就无效，需要重新取指。如下图所示。



从取到转移目标地址的指令，到该指令进入执行阶段至少需要两个时钟周期，也就是处理器至少浪费了两个时钟周期，为了减少损失，规定转移指令后面的指令位置为“延迟槽”，延迟槽中的指令被称为“延迟指令”（也可称之为“延迟槽指令”）。延迟指令总是被执行，与转移发生与否没有关系。引入延迟槽后的指令执行顺序如图 6.2 所示。在 OpenMIPS 处理器的设计中就使用了延迟槽技术。在后面分析测试程序时需要注意这一点。



OpenMIPS 支持的转移指令可以分为两类：跳转指令、分支指令，本小节介绍跳转指令

的实现，共有四条指令，如下表所示：

指令	用法	作用	说明
j	j target	pc <- pc(31,28) || target || ‘00’	跳转到新的地址，新的地址的低 28 位是指令低 26 位左移 2 位形成的，新的地址的高 4 位是延迟槽指令地址的高 4 位，跳转前需要先执行延迟槽中的指令
jal	jal target	r31 <- pc + 8 pc <- pc(31,28) || target || ‘00’	将当前指令后的第 2 条指令地址存入链接寄存器 r31，然后跳转到新的地址，新的地址的低 28 位是指令低 26 位左移 2 位形成的，新的地址的高 4 位是延迟槽指令地址的高 4 位，跳转前需要先执行延迟槽中的指令
jalr	jalr rs jalr rd,rs	rd <- pc + 8 pc <- rs	将当前指令后的第 2 条指令地址存入寄存器 rd（如果没有指定 rd，那么默认是存入寄存器 r31），然后跳转到新的地址，新的地址就是寄存器 rs 的值，跳转前需要先执行延迟槽中的指令
jr	jr rs	pc <- rs	跳转到新的地址，新的地址就是寄存器 rs 的值，跳转前需要先执行延迟槽中的指令

为了实现跳转指令需要对 OpenMIPS 的流水线作如下修改：

（1）修改译码阶段调用的 inst_decode 过程，添加对 j、jal、jalr、jr 指令的译码，如下：

```
procedure inst_decode(inst : word;
  wreg : out std_logic;
    rdo : out std_logic_vector(4 downto 0);
    aluop  : out std_logic_vector(7 downto 0);
    alusel : out std_logic_vector(2 downto 0);
    rfe1, rfe2 : out std_logic; rfa1, rfa2 : out rfatype;
  imm : out word;
  new_cnt : out std_logic_vector(1 downto 0);
  inst_valid : out std_logic) is
    variable op : std_logic_vector(5 downto 0);
    variable op2 : std_logic_vector(4 downto 0);
    variable op3 : std_logic_vector(5 downto 0);
    variable op4 : std_logic_vector(4 downto 0);
   begin

        op    := inst(31 downto 26);
        op2   := inst(10 downto 6);
        op3   := inst(5 downto 0);
        op4   := inst(20 downto 16);
        aluop := EXE_NOP_OP;
        alusel := EXE_RES_NOP;
        wreg := '0';
        rdo := inst(15 downto 11);
```

```
        rfe1 := '0'; rfe2 := '0';
        rfa1 := inst(25 downto 21);
        rfa2 := inst(20 downto 16);
        imm := (others=>'0');
    new_cnt := "00";
    inst_valid := '0';
        case op is
           when EXE_SPECIAL_INST =>
             case op2 is
                when "00000" =>
                   case op3 is
                     ……
                     when EXE_JALR => rfe1 := '1'; rfe2 := '0'; wreg :='1';
                          aluop := EXE_JALR_OP; alusel := EXE_RES_JUMP_BRANCH;
                          inst_valid := '1';
                     when EXE_JR   => rfe1 := '1'; rfe2 := '0'; wreg :='0';
                          aluop := EXE_JR_OP; alusel := EXE_RES_JUMP_BRANCH;
                          inst_valid := '1';
                     when others =>
                   end case;
                when others =>
             end case;
           ……
         when EXE_J    => rfe1 := '0';rfe2 := '0'; wreg :='0'; aluop := EXE_J_OP;
                          alusel := EXE_RES_JUMP_BRANCH; inst_valid := '1';
         when EXE_JAL  => rfe1 := '0';rfe2 := '0'; wreg :='1'; aluop := EXE_JAL_OP;
                          alusel := EXE_RES_JUMP_BRANCH; rdo := "11111";
                          inst_valid := '1';
           ……
```

注意的是：指令 jal、jalr 都是需要将返回地址写到特定寄存器，所以 wreg 为 1，其中 jal 指令默认将返回地址写到链接寄存器 r31，所以 rdo 等于”11111”，指令 j、jr 都不需要保存返回地址，所以 wreg 为 0。

（2）在流水线的执行阶段添加过程 jump_branch_op，其中判断是否要转移，以及转移目标地址，要保存的返回地址，如下：

```
--调用过程 logic_op 进行逻辑运算，结果存储在 ex_logic_res 中
logic_op(r, ex_opdata1, ex_opdata2, ex_logic_res);

--调用过程 shift_op 进行移位运算，结果存储在 ex_shift_res 中
shift_op(r, ex_opdata1, ex_opdata2, ex_shift_res);

--调用过程 arithmetic_op 进行算术运算，结果存储在 ex_arithmetic_op 中
arithmetic_op(r, ex_opdata1, ex_opdata2, ex_arithmetic_res, overflow);
```

```
  --对于 add\addi\sub 指令，如果运算有溢出，那么不修改目的寄存器
  if(overflow = true) then
    v.m.wreg := '0';
  end if;

  --调用过程 mul_op 进行乘法运算，结果存储在 ex_mul_res 中
  mul_op(r, ex_opdata1, ex_opdata2, ex_mul_res);

  --调用过程 move_op 进行移动操作类指令的执行，结果存储在 ex_move_res 中
  move_op(r, ex_opdata1, ex_opdata2, newhilo, ex_move_res, notmove);

  --对于 movn\movz 指令，如果条件不满足，那么 notmove 为 true，表示不修改目的寄存器
  --所以设置 v.m.wreg 为 0
  if(notmove = true) then
    v.m.wreg := '0';
  end if;

  --调用过程 jump_branch_op，判断是否需要转移，转移的目标地址，要保存的返回地址的值
  jump_branch_op(r, ex_opdata1, ex_opdata2, jump_branch_res,
                 ex_jump_target, jump_branch_true, v.e.dslot);

  --调用过程 alu_select，依据操作类型，选择对应的运算结果存储到 ex_result 中
  alu_select(r, ex_logic_res, ex_shift_res, ex_arithmetic_res, ex_move_res,
             ex_mul_res, jump_branch_res, ex_result);
```

其中 jump_branch_op 的定义如下，注意其中对目标地址的计算方法：

```
 procedure jump_branch_op(r: registers;
                          aluin1, aluin2: word;
                          jump_branch_res, jump_target: out word;
                          jump_branch_true: out std_logic;
                          next_is_dslot: out std_logic) is
  variable temp:pctype;
 begin
  temp := r.e.pc + 2;
  jump_target := (others => '0');
  jump_branch_true := '0';
  next_is_dslot := '0';

  if(r.e.aluop = EXE_JAL_OP or r.e.aluop = EXE_JALR_OP) then
    jump_branch_res := temp & "00";
  end if;

  case r.e.aluop is
```

```
    when EXE_JALR_OP | EXE_JR_OP =>
         jump_target := aluin1;      --jr、jalr 指令的目标地址就是寄存器 rs 的值
         jump_branch_true := '1';
         next_is_dslot := '1';
    when EXE_J_OP | EXE_JAL_OP =>    --j、jal 指令的目标地址计算方法稍微复杂一点
         jump_target := temp(31 downto 28) & r.e.inst(25 downto 0) & "00";
         jump_branch_true := '1';
         next_is_dslot := '1';
    when others =>
  end case;
end;
```

（3）修改执行阶段的过程 alu_select，将 jump_branch_op 过程的结果也作为一个输入参数，参与最终运算结果的选择，如下：

```
procedure alu_select(r : registers; logicout, shiftout, arithmeticres,
                  moveres: word; mulres: std_logic_vector(63 downto 0);
                  jump_branch_res: word; res: out word) is
    variable aluresult : word;
  begin

      aluresult := (others => '0');
       case r.e.alusel is
          when EXE_RES_LOGIC => aluresult := logicout;
          when EXE_RES_SHIFT => aluresult := shiftout;
          when EXE_RES_ARITHMETIC => aluresult := arithmeticres;
          when EXE_RES_MUL => aluresult := mulres(31 downto 0);
          when EXE_RES_MOVE => aluresult := moveres;
          when EXE_RES_JUMP_BRANCH => aluresult := jump_branch_res;
          when others => aluresult := zero32;
       end case;
       res := aluresult;
end;
```

（4）修改流水线的取指阶段，当 jump_branch_op 的输出 jump_branch_true 为 1 时，修改 pc 的值，使其等于转移目标地址，如下：

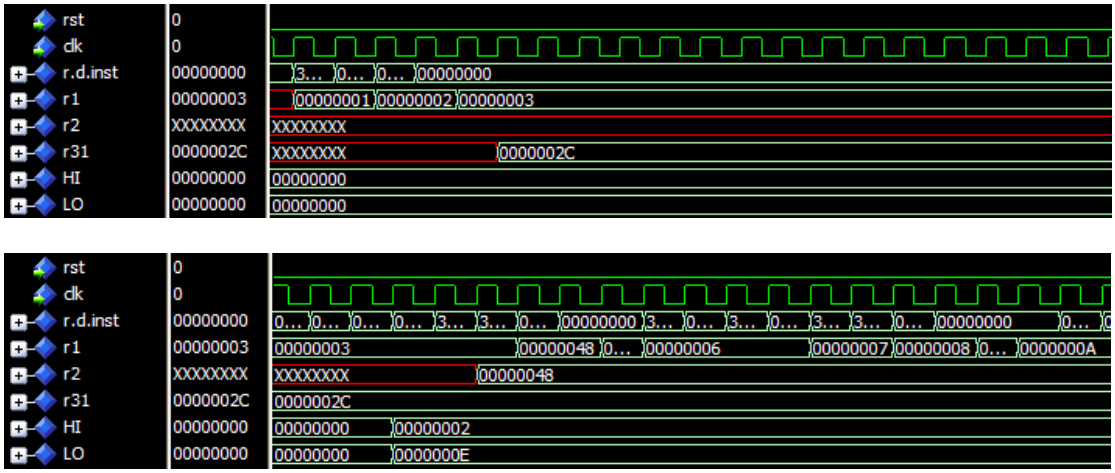
```
  if (rst = '1') then
    v.f.pc := "00000000000000000000000000000000";
  elsif(ex_stall_for_multicycle_inst = '1') then
    --如果正处于执行阶段的是多周期指令，那么不改变 pc 的值
    v.f.pc := r.f.pc;
  elsif(jump_branch_true = '1') then
    --执行阶段是转移指令，设置新的 pc 值，同时设置取指阶段的指令为 NOP
    v.f.pc := ex_jump_target(31 downto 2);
    v.d.inst := zero32;
```

```
  else
    --下一条指令是当前读取指令地址加 4
    v.f.pc := r.f.pc(31 downto 2) + 1;
  end if;
```

测试程序如下，位于 asm_test/Day7_1 目录下：

```
   .org 0x0
   .set noat
   .set noreorder
   .set nomacro
   .global _start
_start:
1  ori  $1,$0,0x0001   # r1 = 0x1
2  j    0x20
3  ori  $1,$0,0x0002   # r1 = 0x2
   ori  $1,$0,0x1111
   ori  $1,$0,0x1100

   .org 0x20
4  ori  $1,$0,0x0003   # r1 = 0x3
5  jal  0x40
6  div  $zero,$31,$1   # $31 = 0x2c, $1 = 0x3
                       # HI = 0x2, LO = 0xe
9  ori  $1,$0,0x0005   # r1 = 0x5
10 ori  $1,$0,0x0006   # r1 = 0x6
11 j    0x60
12 nop

   .org 0x40

7  jalr $2,$31
8  or   $1,$2,$0        # r1 = 0x48
16 ori  $1,$0,0x0009    # r1 = 0x9
17 ori  $1,$0,0x000a    # r1 = 0xa
18 j 0x80
19 nop

   .org 0x60
13 ori  $1,$0,0x0007    # r1 = 0x7
14 jr   $2
15 ori  $1,$0,0x0008    # r1 = 0x8
   ori  $1,$0,0x1111
   ori  $1,$0,0x1100

   .org 0x80
   nop

_loop:
20 j _loop
21 nop
```

左边的一列数字显示的指令实际执行顺序，预期效果如程序中注释所示，ModelSim 仿真如下图所示，从仿真结果可知，OpenMIPS 正确实现了 j、jal、jalr、jr 这四条跳转指令。OpenMIPS 的代码位于 10_Days_make_OpenMIPS/Day7_1 目录下。



7.2 实现分支指令

分支指令与跳转指令的区别有两点：首先分支指令是相对转移，其次大部分的分支指令都需要首先判断是否满足分支条件，然后才决定是否跳转到分支目标。OpenMIPS 支持的分支指令有 b、bal、beq、bgez、bgezal、bgtz、blez、bltz、bltzal、bne，其中 b 指令是 beq 指令的特例，bal 指令也是 bgezal 指令的特例，所以不用特意考虑 b、bal 指令。指令说明如下表所示：

指令	用法	作用	说明
beq	beq rs,rt,offset	if rs = rt then branch	如果寄存器 rs 的值等于寄存器 rt 的值，那么转移到目标地址，目标地址等于 16 位的 offset 左移 2 位，并且符号扩展为 32 位，然后与延迟槽指令地址相加的和
bgez	bgez rs,offset	if rs >= 0 then branch	如果寄存器 rs 的值大于等于 0，那么转移到目标地址，目标地址等于 16 位的 offset 左移 2 位，并且符号扩展为 32 位，然后与延迟槽指令地址相加的和
bgezal	bgezal rs,offset	if rs >= 0 then branch	如果寄存器 rs 的值大于等于 0，那么转移到目标地址，目标地址等于 16 位的 offset 左移 2 位，并且符号扩展为 32 位，然后与延迟槽指令地址相加的和，最后将本指令地址加 8 保存到链接寄存器 r31 中
bgtz	bgtz rs,offset	if rs > 0 then branch	如果寄存器 rs 的值大于 0，那么转移到目标地址，目标地址等于 16 位的 offset 左移 2 位，并且符号扩展为 32 位，然后与延迟槽指令地址相加的和
blez	blez rs,offset	if rs <= 0 then branch	如果寄存器 rs 的值小于等于 0，那么转移到目标地址，目标地址等于 16 位的 offset 左移 2 位，并且符号扩展为 32 位，然后与延迟槽指令地址相加的和

bltz	bltz rs,offset	if rs < 0 then branch	如果寄存器 rs 的值小于 0，那么转移到目标地址，目标地址等于 16 位的 offset 左移 2 位，并且符号扩展为 32 位，然后与延迟槽指令地址相加的和
bltzal	bltzal rs,offset	if rs < 0 then branch	如果寄存器 rs 的值小于 0，那么转移到目标地址，目标地址等于 16 位的 offset 左移 2 位，并且符号扩展为 32 位，然后与延迟槽指令地址相加的和，最后将本指令地址加 8 保存到链接寄存器 r31 中
bne	bne rs,rt,offset	if rs /= rt then branch	如果寄存器 rs 的值不等于寄存器 rt 的值，那么转移到目标地址，目标地址等于 16 位的 offset 左移 2 位，并且符号扩展为 32 位，然后与延迟槽指令地址相加的和

上述指令也很好理解，以“al”结尾的指令表示要写链接寄存器 r31，指令中间的“lt”表示小于，“le”表示小于等于，“gt” 表示大于，“ge”表示大于等于。

分支指令相比跳转指令的区别就在于多了一点判断语句，同时对转移目标地址的计算方法不同，所以实现起来并不复杂，流水线中要修改的地方如下：

（1）修改 inst_decode 过程，其中添加对分支指令的译码，如下：

```
procedure inst_decode(inst : word;
  wreg : out std_logic;
    rdo : out std_logic_vector(4 downto 0);
    aluop  : out std_logic_vector(7 downto 0);
    alusel : out std_logic_vector(2 downto 0);
    rfe1, rfe2 : out std_logic; rfa1, rfa2 : out rfatype;
  imm : out word;
  new_cnt : out std_logic_vector(1 downto 0);
  inst_valid : out std_logic) is
    variable op : std_logic_vector(5 downto 0);
    variable op2 : std_logic_vector(4 downto 0);
    variable op3 : std_logic_vector(5 downto 0);
    variable op4 : std_logic_vector(4 downto 0);
   begin

        op    := inst(31 downto 26);
        op2   := inst(10 downto 6);
        op3   := inst(5 downto 0);
        op4   := inst(20 downto 16);
        aluop := EXE_NOP_OP;
        alusel := EXE_RES_NOP;
        wreg := '0';
        rdo := inst(15 downto 11);

        rfe1 := '0'; rfe2 := '0';
```

```
        rfa1 := inst(25 downto 21);
        rfa2 := inst(20 downto 16);
        imm := (others=>'0');
         new_cnt := "00";
         inst_valid := '0';
        case op is
        when EXE_BEQ  => rfe1 := '1';rfe2 := '1'; wreg :='0';
                         aluop := EXE_BEQ_OP;
                         alusel := EXE_RES_JUMP_BRANCH; inst_valid := '1';
         when EXE_BGTZ => rfe1 := '1';rfe2 := '0'; wreg :='0';
                         aluop := EXE_BGTZ_OP;
                         alusel := EXE_RES_JUMP_BRANCH; inst_valid := '1';
         when EXE_BLEZ => rfe1 := '1';rfe2 := '0'; wreg :='0';
                          aluop := EXE_BLEZ_OP;
                          alusel := EXE_RES_JUMP_BRANCH; inst_valid := '1';
         when EXE_BNE  => rfe1 := '1';rfe2 := '1'; wreg :='0';
                          aluop := EXE_BNE_OP;
                          alusel := EXE_RES_JUMP_BRANCH; inst_valid := '1';

         when EXE_REGIMM_INST =>
            case op4 is
              when EXE_BGEZ   => rfe1 := '1';rfe2 := '0'; wreg :='0';
                          aluop := EXE_BGEZ_OP;
                          alusel := EXE_RES_JUMP_BRANCH; inst_valid := '1';
              when EXE_BGEZAL => rfe1 := '1';rfe2 := '0'; wreg :='1';
                          aluop := EXE_BGEZAL_OP;
                          alusel := EXE_RES_JUMP_BRANCH; rdo := "11111";
                          inst_valid := '1';
              when EXE_BLTZ   => rfe1 := '1';rfe2 := '0'; wreg :='0';
                          aluop := EXE_BLTZ_OP;
                          alusel := EXE_RES_JUMP_BRANCH; inst_valid := '1';
              when EXE_BLTZAL => rfe1 := '1';rfe2 := '0'; wreg :='1';
                           aluop := EXE_BLTZAL_OP;
                           alusel := EXE_RES_JUMP_BRANCH;
                           rdo := "11111"; inst_valid := '1';
           ……
```

（2）修改 jump_branch_op 过程，其中添加对分支指令的分析，主要是判断是否满足跳转条件：

```
 procedure jump_branch_op(r: registers;
                      aluin1, aluin2: word;
                      jump_branch_res, jump_target: out word;
                      jump_branch_true: out std_logic;
                      next_is_dslot: out std_logic) is
```

```
  variable temp:pctype;
  variable is_zero, less_than_zero: std_logic;
  variable sign_ext : std_logic_vector(13 downto 0);
  variable jump_branch_address: word;
begin
  temp := r.e.pc + 2;           --当前指令的后面第二条指令地址
  jump_branch_res := (others => '0');
  jump_target := (others => '0');
  sign_ext := (others => r.e.inst(15));
  jump_branch_address := (r.d.pc & "00") +
                         (sign_ext & r.e.inst(15 downto 0) & "00");
  jump_branch_true := '0';
  next_is_dslot := '0';
  if(aluin1 = zero32) then
    is_zero := '1';
  else
    is_zero := '0';
  end if;

  if(aluin1(31) = '1') then  --依据源操作数的最高位判断是否小于0
     less_than_zero := '1';
  else
     less_than_zero := '0';
  end if;

  if(r.e.aluop = EXE_JAL_OP or r.e.aluop = EXE_JALR_OP or
     r.e.aluop = EXE_BGEZAL_OP or r.e.aluop = EXE_BLTZAL_OP) then
    jump_branch_res := temp & "00";   --该值要写入链接寄存器r31
  end if;

  case r.e.aluop is
    ……
    when EXE_BEQ_OP =>
         if(aluin1 = aluin2) then    --源操作数相等时转移
            jump_target := jump_branch_address;
            jump_branch_true := '1';
            next_is_dslot := '1';
         end if;
    when EXE_BGEZ_OP | EXE_BGEZAL_OP => --源操作数大于等于0时转移
         if(less_than_zero = '0') then
            jump_target := jump_branch_address;
            jump_branch_true := '1';
            next_is_dslot := '1';
         end if;
```

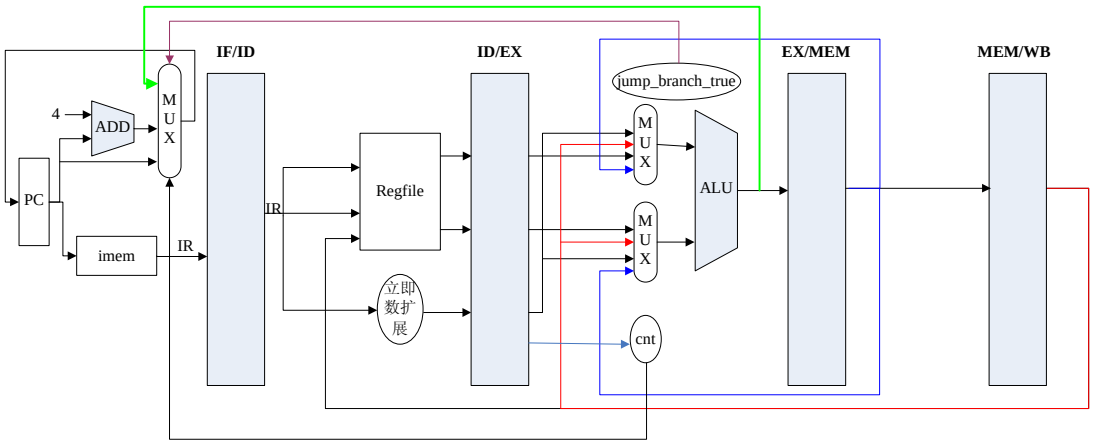
```
    when EXE_BGTZ_OP =>                 --源操作数大于0时转移
         if(less_than_zero = '0' and is_zero = '0') then
           jump_target := jump_branch_address;
           jump_branch_true := '1';
           next_is_dslot := '1';
         end if;
    when EXE_BLEZ_OP =>                 --源操作数小于等于0时转移
         if(less_than_zero = '1' and is_zero = '1') then
           jump_target := jump_branch_address;
           jump_branch_true := '1';
           next_is_dslot := '1';
         end if;
    when EXE_BLTZ_OP | EXE_BLTZAL_OP =>  --源操作数小于0时转移
         if(less_than_zero = '1') then
           jump_target := jump_branch_address;
           jump_branch_true := '1';
           next_is_dslot := '1';
         end if;
    when EXE_BNE_OP =>                   --源操作数不相等时转移
         if(aluin1 /= aluin2) then
           jump_target := jump_branch_address;
           jump_branch_true := '1';
           next_is_dslot := '1';
         end if;
    when others =>
  end case;
end;
```

测试例程如下，代码位于 asm_test/Day7_2 目录下：

```
   .org 0x0
   .set noat
   .set noreorder
   .set nomacro
   .global _start
_start:
1     ori  $3,$0,0x8000
2     sll  $3,16              # r3 = 0x80000000
3     ori  $1,$0,0x0001       # r1 = 0x1
4     b    s1
5     ori  $1,$0,0x0002       # r1 = 0x2
1:
      ori  $1,$0,0x1111
      ori  $1,$0,0x1100

      .org 0x20
s1:
6     ori  $1,$0,0x0003       # r1 = 0x3
7     bal  s2
8     div  $zero,$31,$1       # r31 = 0x2c, r1 =0x3
                              # HI = 0x2, LO = 0xe
      ori  $1,$0,0x1100
      ori  $1,$0,0x1111
      bne  $1,$0,s3
      nop
      ori  $1,$0,0x1100
      ori  $1,$0,0x1111

      .org 0x50
s2:
9     ori  $1,$0,0x0004       # r1 = 0x4
10    beq  $2,$2,s3
11    or   $1,$31,$0          # $1 = 0x2c
      ori  $1,$0,0x1111
      ori  $1,$0,0x1100
2:
15    ori  $1,$0,0x0007       # r1 = 0x7
16    ori  $1,$0,0x0008       # r1 = 0x8
17    bgtz $1,s4
18    ori  $1,$0,0x0009       # r1 = 0x9
      ori  $1,$0,0x1111
      ori  $1,$0,0x1100

      .org 0x80
s3:
12    ori  $1,$0,0x0005       # r1 = 0x5
13    BGEZ $1,2b
14    ori  $1,$0,0x0006       # r1 = 0x6
      ori  $1,$0,0x1111
      ori  $1,$0,0x1100

      .org 0x100
s4:
19    ori  $1,$0,0x000a       # r1 = 0xa
20    BGEZAL $3,s3
21    or   $1,$0,$31          # r1 = 0x10c
22    ori  $1,$0,0x000b       # r1 = 0xb
23    ori  $1,$0,0x000c       # r1 = 0xc
24    ori  $1,$0,0x000d       # r1 = 0xd
25    ori  $1,$0,0x000e       # r1 = 0xe
26    bltz $3,s5
27    ori  $1,$0,0x000f       # r1 = 0xf
      ori  $1,$0,0x1100

      .org 0x130
s5:
28    ori  $1,$0,0x0010       # r1 = 0x10
29    blez $1,2b
30    ori  $1,$0,0x0011       # r1 = 0x11
31    ori  $1,$0,0x0012       # r1 = 0x12
32    ori  $1,$0,0x0013       # r1 = 0x13
33    bltzal $3,s6
34    or   $1,$0,$31          # r1 = 0x14c
      ori  $1,$0,0x1100

      .org 0x160
s6:
35    ori $1,$0,0x0014        # r1 = 0x14
36    nop

_loop:
37    j _loop
38    nop
```

左边一列显示的是指令执行顺序，程序中的注释是预期执行效果，ModelSim 仿真如下图所示，从中可知 OpenMIPS 正确实现了分支指令。OpenMIPS 的代码位于 10_Days_make_OpenMIPS/Day7_2 目录下。

添加跳转、分支指令后的数据流图如下所示，主要是将执行阶段的结果也参与到了 PC 值的选择：



第八天

主要内容

（1）实现加载类指令——LB、LBU、LH、LHU、LL、LW、LWL、LWR
（2）实现存储类指令——SB、SC、SH、SW、SWL、SWR

8.1 实现加载、存储类指令

今天要实现的是加载、存储类指令，这里就要使用到之前一直没有使用的 dmem 模块了，由于这两类指令很相似，就不再分开介绍实现步骤了，统一介绍。

OpenMIPS 支持的加载、存储类指令如下表所示：

指令	用法	作用	说明
lb	lb rt,offset(base)	rt <- memory[offset + base]	从内存的 offset+base 处加载一个字节，符号扩展至 32 位，保存到寄存器 rt
lbu	lbu rt,offset(base)	rt <- memory[offset + base]	从内存的 offset+base 处加载一个字节，零扩展至 32 位，保存到寄存器 rt
lh	lh rt,offset(base)	rt <- memory[offset + base]	从内存的 offset+base 处加载一个半字，符号扩展至 32 位，保存到寄存器 rt
lhu	lhu rt,offset(base)	rt <- memory[offset + base]	从内存的 offset+base 处加载一个半字，零扩展至 32 位，保存到寄存器 rt
ll	ll rt,offset(base)	rt <- memory[offset + base]	从内存的 offset+base 处加载一个字，保存到寄存器 rt，是原子的“读改写”操作中的读操作
lw	lw rt,offset(base)	rt <- memory[offset + base]	从内存的 offset+base 处加载一个字，保存到寄存器 rt
lwl	lwl rt,offset(base)	rt <- rt merge memory[offset + base]	非对齐地址加载指令
lwr	lwr rt,offset(base)	rt <- rt merge memory[offset + base]	非对齐地址加载指令
sb	sb rt,offset(base)	memory[offset + base] <- rt	将 rt 的最低字节存储到内存中指定地址
sc	sc rt,offset(base)	if automatic_update then memory[offset + base] <- rt, rt <- 1 else rt <- 0	向内存的 offset+base 存储一个字，是原子的“读改写”操作中的写操作，只有当满足原子操作的时候，才写内存指定地址，同时设置 rt 为 1，反之不修改内存指定地址，同时设置 rt 为 0
sh	sh	memory[offset + base]	将 rt 的低两个字节存储到内存中指定地址

	rt,offset(base)	<- rt	
sw	sw rt,offset(base)	memory[offset + base] <- rt	将 rt 存储到内存中指定地址
swr	swr rt,offset(base)	memory[offset + base] <- rt	非对齐地址存储指令
swl	swl rt,offset(base)	memory[offset + base] <- rt	非对齐地址存储指令

在实现的过程中，有以下几点需要注意：

（1）加载指令要在访存阶段的最后才能获取到新的数据，所以要修改数据相关的处理方法。

（2）加载指令在取到数据后需要依据加载指令类型、目标地址进行对齐操作。

（3）存储指令需要依据存储指令类型、目标地址设置 iu 模块的输出信号 sel 的值，该值会输出到 dmem 模块。

（4）要仔细体会 LWL、LWR、SWL、SWR 指令的效果

（5）SC、LL 指令的实现比较特别

以下分别说明为了实现上述几点，所作的修改：

（1）加载指令要在访存阶段的最后才能获取到新的数据，所以要修改数据相关的处理方法

在前面发生的数据相关都是在执行阶段的一开始就检测并解决的，但是引入加载指令后，加载指令会在访存阶段的最后取到数据，这样，如果下一条指令与访存指令存在数据相关，那么之前的方法就无法解决这个问题。参考《计算机体系结构——量化研究方法（第 5 版）》的 467 页的“需要停顿的数据冒险”一小节。

解决上述问题的基本思路是：如果加载指令与下一条指令存在数据相关，那么暂停下一条指令的执行，直到加载指令进入回写阶段，再继续执行。为此，需要修改流水线执行阶段调用的数据相关处理函数 opdata_select，如下，添加其中蓝色代码

```
procedure opdata_select(r,v: registers; opdata1 : out word;
                        opdata2: out word; ex_stall : out std_logic) is
variable ex_stall1, ex_stall2 : std_logic;
begin
  if r.e.rfe1='0' then
      opdata1 := r.e.imm;                          --源操作数是立即数
  elsif (r.m.waddr = r.e.rfa1 and r.m.wreg = '1'
         and r.e.rfe1 = '1' and (r.m.load_op="00")) then
    opdata1 := r.m.result;      --与上一条指令存在数据相关，上一条指令不是加载指令
  elsif (r.m.waddr = r.e.rfa1 and r.m.wreg = '1'
         and r.e.rfe1 = '1' and (r.m.load_op/="00")) then
    ex_stall1 := '1';           --与上一条指令存在数据相关，上一条指令是加载指令
    opdata1 := zero32;
  elsif (r.m.waddr = r.e.rfa1 and r.m.wreg = '1' and r.e.rfe1 = '1') then
    opdata1 := r.m.result;                  --与上一条指令存在数据相关
  elsif (r.w.waddr = r.e.rfa1 and r.w.wreg = '1' and r.e.rfe1 = '1') then
    opdata1 := r.w.result;                  --与上上一条指令存在数据相关
```

```
    else
      opdata1 := r.e.reg1;                    --不存在数据相关
    end if;

    if r.e.rfe2='0' then
        opdata2 := r.e.imm;                          --源操作数是立即数
    elsif (r.m.waddr = r.e.rfa2 and r.m.wreg = '1'
           and r.e.rfe2 = '1' and (r.m.load_op="00")) then
      opdata2 := r.m.result;      --与上一条指令存在数据相关，上一条指令不是加载指令
    elsif (r.m.waddr = r.e.rfa2 and r.m.wreg = '1'
           and r.e.rfe2 = '1' and (r.m.load_op/="00")) then
      ex_stall2 := '1';
      opdata2 := zero32;          --与上一条指令存在数据相关，上一条指令是加载指令
    elsif (r.m.waddr = r.e.rfa2 and r.m.wreg = '1' and r.e.rfe2 = '1' ) then
      opdata2 := r.m.result;                    --与上一条指令存在数据相关
    elsif (r.w.waddr = r.e.rfa2 and r.w.wreg = '1' and r.e.rfe2 = '1' ) then
      opdata2 := r.w.result;                    --与上上一条指令存在数据相关
    else
      opdata2 := r.e.reg2;                    --不存在数据相关
    end if;

    if(ex_stall1 = '1' or ex_stall2 = '1') then
      ex_stall := '1'; --如果与上一条加载指令存在数据相关，那么设置 ex_stall 为 1
    else
      ex_stall := '0';
    end if;

  end;
```

相应的调用改为如下，如果加载指令与下一条指令存在数据相关，那么设置 ex_stall_for_new_RF 为 1,：

```
    --操作数选择，是立即数还是寄存器的值，要考虑加载指令与下一条指令可能存在的数据相关问题，
    --如果存在这种数据相关，那么设置 ex_stall_for_new_RF 为 1
    opdata_select(r, v, ex_opdata1, ex_opdata2, ex_stall_for_new_RF);
```

修改取指阶段如下，如果 ex_stall_for_new_RF 为 1，那么暂停取指，同时流水线的译码、执行阶段也暂停，访存、回写阶段继续向前流转。

```
    if (rst = '1') then
      v.f.pc := "00000000000000000000000000000000";
    elsif(ex_stall_for_multicycle_inst = '1' or ex_stall_for_new_RF = '1') then
      --如果正处于执行阶段的是多周期指令或者与上一条加载指令存在数据相关，那么不改变 pc 的值
      v.f.pc := r.f.pc;
    elsif(jump_branch_true = '1') then
      --执行阶段是转移指令，设置新的 pc 值，同时设置取指阶段的指令为 NOP
      v.f.pc := ex_jump_target(31 downto 2);
      v.d.inst := zero32;
```

```
    else
      --下一条指令是当前读取指令地址加 4
      v.f.pc := r.f.pc(31 downto 2) + 1;
     end if;

    --如果正处于执行阶段的指令与上一条加载指令存在数据相关，那么保存该指
    --令的操作数，访存、回写阶段的指令可以继续向前流转
    if(ex_stall_for_new_RF = '1') then
      rin.e.reg1 <= ex_opdata1;
      rin.e.reg2 <= ex_opdata2;
      rin <= r;
      rin.m <= v.m;
      rin.w <= v.w;
    elsif(ex_stall_for_multicycle_inst = '1') then
    --如果正处于执行阶段的多周期指令，那么保存该指令的操作数，访存、
    --回写阶段的指令可以继续向前流转，同时保存执行阶段指令计算出的 HI、LO
    --的值，以及 cnt 的值，对多周期指令而言，每执行一个周期，cnt 的值减 1，
    --cnt 为 0 表示当前是多周期指令的最后一个周期
      rin.e.reg1 <= ex_opdata1;
      rin.e.reg2 <= ex_opdata2;
      rin <= r;
      rin.e.cnt <= cnt_temp;
      rin.e.hilo <= hilo_temp;
      rin.m <= v.m;                --访存阶段继续向前流转
      rin.w <= v.w;                --回写阶段继续向前流转
    else
      rin <= v;
    end if;
```

（2）加载指令在取到数据后需要依据加载指令类型、目标地址进行对齐操作

修改流水线的回写阶段，添加函数 load_align，如下：

```
--load_op 不为 00，表示是加载指令，需要依据指令类型、目标地址，
--对 dmem 的返回值 dmem_rdata 进行修改对齐
  if(r.m.load_op /= "00") then
    v.w.result := load_align(r.m.aluop, r.m.sel, r.m.load_op,
                             v.w.LLbit, dmem_rdata, r.m.opdata2);
  end if;
```

其中函数 load_align 定义如下：

```
  --依据具体的加载指令，调整加载到的数据，作为最终要写入目的寄存器的数据
  function load_align(aluop : std_logic_vector(7 downto 0);
                  sel: std_logic_vector(3 downto 0);
                  load_op: std_logic_vector(1 downto 0); oldLLbit : std_logic;
                  data: word; opdata2: word) return std_logic_vector is
```

```
      variable data_sign_ext,result : word;
begin
       data_sign_ext(31 downto 0) := (others => '0');
       if(load_op = "11") then       --是 SC 指令，将 LLbit 的值写入目的寄存器
         result := data_sign_ext(30 downto 0) & oldLLbit;
       elsif(aluop = EXE_LWL_OP) then  --MIPS32 指令集中的 LWL 允许非对齐加载
         case sel is
           when "1000" => result := data;
           when "0100" => result := data(23 downto 0) & opdata2(7 downto 0);
           when "0010" => result := data(15 downto 0) & opdata2(15 downto 0);
           when "0001" => result := data(7 downto 0) & opdata2(23 downto 0);
           when others =>
         end case;
       elsif(aluop = EXE_LWR_OP) then  --MIPS32 指令集中的 LWR 允许非对齐加载
         case sel is
           when "1000" => result := opdata2(31 downto 8) & data(31 downto 24);
           when "0100" => result := opdata2(31 downto 16) & data(31 downto 16);
           when "0010" => result := opdata2(31 downto 24) & data(31 downto 8);
           when "0001" => result := data;
           when others =>
         end case;
       else
         case sel is
           when "1000" =>             --加载字节，sel 为 1000
             if(load_op = "01") then  --符号扩展加载
               data_sign_ext(23 downto 0 ) := (others => data(31));
             end if;
             result := data_sign_ext(23 downto 0) & data(31 downto 24);
           when "0100" =>             --加载字节，sel 为 0100
             if(load_op = "01") then
               data_sign_ext(23 downto 0 ) := (others => data(23));
             end if;
             result := data_sign_ext(23 downto 0) & data(23 downto 16);
           when "0010" =>             --加载字节，sel 为 0010
             if(load_op = "01") then
               data_sign_ext(23 downto 0 ) := (others => data(15));
             end if;
             result := data_sign_ext(23 downto 0) & data(15 downto 8);
           when "0001" =>             --加载字节，sel 为 0001
             if(load_op = "01") then
               data_sign_ext(23 downto 0 ) := (others => data(7));
             end if;
             result := data_sign_ext(23 downto 0) & data(7 downto 0);
           when "1100" =>             --加载字节，sel 为 1100
```

```
            if(load_op = "01") then
              data_sign_ext(15 downto 0 ) := (others => data(31));
            end if;
            result := data_sign_ext(15 downto 0) & data(31 downto 16);
          when "0011" =>              --加载字节，sel 为 0011
            if(load_op = "01") then
              data_sign_ext(15 downto 0 ) := (others => data(15));
            end if;
            result := data_sign_ext(15 downto 0) & data(15 downto 0);
          when others =>              --加载字
            result := data(31 downto 0);
        end case;
      end if;
      return result;
  end;
```

（3）存储指令需要依据存储指令类型、目标地址设置 sel 的值

在流水线执行阶段添加函数 load_store_op，其中依据存储操作类型设置 sel 的值，如下：

```
  --调用过程 load_store_op，在其中依据加载存储操作类型，设置 dmem 模块的输入信号
  load_store_op(r.e.inst, r.e.aluop, ex_opdata1, ex_opdata2,
           v.m.we, v.m.sel, v.m.addr, v.m.wdata, v.m.load_op, v.m.store_op);
```

其中 load_store_op 的定义如下：

```
  procedure load_store_op(inst : word; aluop : std_logic_vector(7 downto 0);
            aluin1, aluin2: word;
            we_o : out std_logic;
            sel_o : out std_logic_vector(3 downto 0);
            addr_o,data_o : out word;
            load_op: out std_logic_vector(1 downto 0);
            store_op : out std_logic) is
    variable temp,imm,temp1: word;
   begin
      addr_o := (others => '0');
      we_o := '0';
      imm(15 downto 0) := inst(15 downto 0);
      imm(31 downto 16) := (others => inst(15));  --注意此处是符号扩展
      temp := aluin1 + imm;

      --字节加载、存储指令，要依据目标地址设置 sel 的值，注意 OpenMIPS 工作在大端模式
      if(aluop = EXE_LB_OP or aluop = EXE_SB_OP or aluop = EXE_LBU_OP
         or aluop = EXE_LWL_OP or aluop = EXE_LWR_OP) then
        temp1 := aluin2(7 downto 0) & aluin2(7 downto 0) &
                 aluin2(7 downto 0) & aluin2(7 downto 0);
      case temp(1 downto 0) is
```

```
      when "00" => sel_o := "1000";
      when "01" => sel_o := "0100";
      when "10" => sel_o := "0010";
      when "11" => sel_o := "0001";
      when others => sel_o := "1111";
    end case;
  --半字加载、存储指令，要依据目标地址设置 sel 的值，注意 OpenMIPS 工作在大端模式
  elsif(aluop = EXE_LH_OP or aluop = EXE_LHU_OP or aluop = EXE_SH_OP) then
    temp1 := aluin2(15 downto 0) & aluin2(15 downto 0);
    case temp(1 downto 0) is
      when "00" => sel_o := "1100";
      when "10" => sel_o := "0011";
      when others => sel_o := "1111";
    end case;
  --MIPS32 指令集允许非对齐存储，对应的就是指令 SWL、SWR
  elsif(aluop = EXE_SWL_OP) then
    temp1 := aluin2;
    case temp(1 downto 0) is
      when "00" => sel_o := "1111";
      when "01" => sel_o := "0111";
      when "10" => sel_o := "0011";
      when "11" => sel_o := "0001";
      when others => sel_o := "1111";
    end case;
  elsif(aluop = EXE_SWR_OP) then
    case temp(1 downto 0) is
      when "00" => sel_o := "1000";
                   temp1 := aluin2(7 downto 0) & zero32(23 downto 0);
      when "01" => sel_o := "1100";
                   temp1 := aluin2(15 downto 0) & zero32(15 downto 0);
      when "10" => sel_o := "1110";
                   temp1 := aluin2(23 downto 0) & zero32(7 downto 0);
      when "11" => sel_o := "1111"; temp1 := aluin2(31 downto 0);
      when others => sel_o := "1111"; temp1 := aluin2(31 downto 0);
    end case;
  else
    temp1 := aluin2;
    sel_o := "1111";
  end if;
     data_o := (others => '0');
     load_op := "00";
     store_op := '0';
     if(aluop = EXE_LB_OP or aluop = EXE_LH_OP or aluop = EXE_LW_OP
        or aluop = EXE_LL_OP or aluop = EXE_LWL_OP or aluop = EXE_LWR_OP) then
```

```
        addr_o := temp;
        we_o := '0';
        load_op := "01";  --有符号加载
      elsif(aluop = EXE_LBU_OP or aluop = EXE_LHU_OP) then
        addr_o := temp;
        we_o := '0';
        load_op := "10";   --无符号加载
      elsif(aluop = EXE_SB_OP or aluop = EXE_SH_OP or
            aluop = EXE_SW_OP or aluop = EXE_SWL_OP) then
        addr_o := temp;
        we_o := '1';
        store_op := '1';   --store_op 为 1 表示是存储操作
        data_o := temp1;   --要存储的数据
      elsif(aluop = EXE_SWR_OP) then
        addr_o := temp;
        we_o := '1';
        store_op := '1';   --store_op 为 1 表示是存储操作
        data_o := temp1;   --要存储的数据
      elsif(aluop = EXE_SC_OP) then
        addr_o := temp;
        we_o := '1';
        store_op := '1';   --store_op 为 1 表示是存储操作
        load_op := "11";   --SC 指令专用的 load_op，因为 SC 指令既有存储操作，
                           --也有修改目的寄存器的操作
        data_o := aluin2;  --要存储的数据
      end if;
end;
```

（4）要仔细体会 LWL、LWR、SWL、SWR 指令的效果

LWL、LWR、SWL、SWR 这四条指令是 MIPS32 中比较特别的指令，用于非对齐地址的加载、存储，比如：正常情况下，字的加载指令 lw，要求目标地址的最低两位为 00，否则会产生对齐异常，使用 LWL、LWR 指令可以不用关心目标地址最低两位的值，其执行效果可以参考 MIPS32 指令集手册（位于 doc 目录下），其中有详细的说明，并且有例子，童鞋们可以自己体会理解。OpenMIPS 实现了 LWL、LWR、SWL、SWR 这四条指令，其实现步骤在上面已经涉及，此处不再重复。

（5）SC、LL 指令的实现比较特别

这两条指令也是 MIPS32 指令手册中比较特别的指令，用来实现原子“读-修改-写”操作，LL 指令读取内存中的数据，设置 LLbit 为 1，然后可以对该数据进行一定的操作，随后使用 SC 指令将该数据写回内存，并且设置 LLbit 为 0。

如果在 LL、SC 指令之间没有执行 eret 指令、没有其他处理器执行 LL\SC 指令（OpenMIPS 是单核处理器，所以这一点不用考虑），那么 SC 指令将寄存器 rt 的值存储回内存，同时设置 rt 为 1，表示原子操作成功，反之，SC 不修改内存，同时设置 rt 为 0，表示原子操作失败。为了实现 LL、SC 指令，OpenMIPS 做了如下修改：

修改流水线访存阶段，增加如下代码：

```
--对于 SC 指令，如果 LLbit 为 0，那么需要写入目的寄存器的值为 0，原子操作失败，
--反之为 1，原子操作成功
if(r.m.aluop = EXE_SC_OP ) then
  if(v.w.LLbit = '0') then
    v.w.result := zero32;       --写入目的寄存器的值为 0
    dmem_we <= '0';             --同时不再将数据写入 dmem
  else
    v.w.result := zero32(30 downto 0) & '1';  --写入目的寄存器的值为 1
  end if;
 end if;
```

修改流水线回写阶段，增加如下代码：

```
v.w.LLbit := r.w.LLbit;

--只有 SC 指令执行完毕后或者 eret 指令执行后，才修改 LLbit 的值为 0
if((r.w.aluop = EXE_SC_OP or r.w.aluop = EXE_ERET_OP)) then
  v.w.LLbit := '0';
end if;

--只有 LL 指令执行完毕后，才修改 LLbit 的值为 1
if((r.w.aluop = EXE_LL_OP)) then
  v.w.LLbit := '1';
 end if;
```

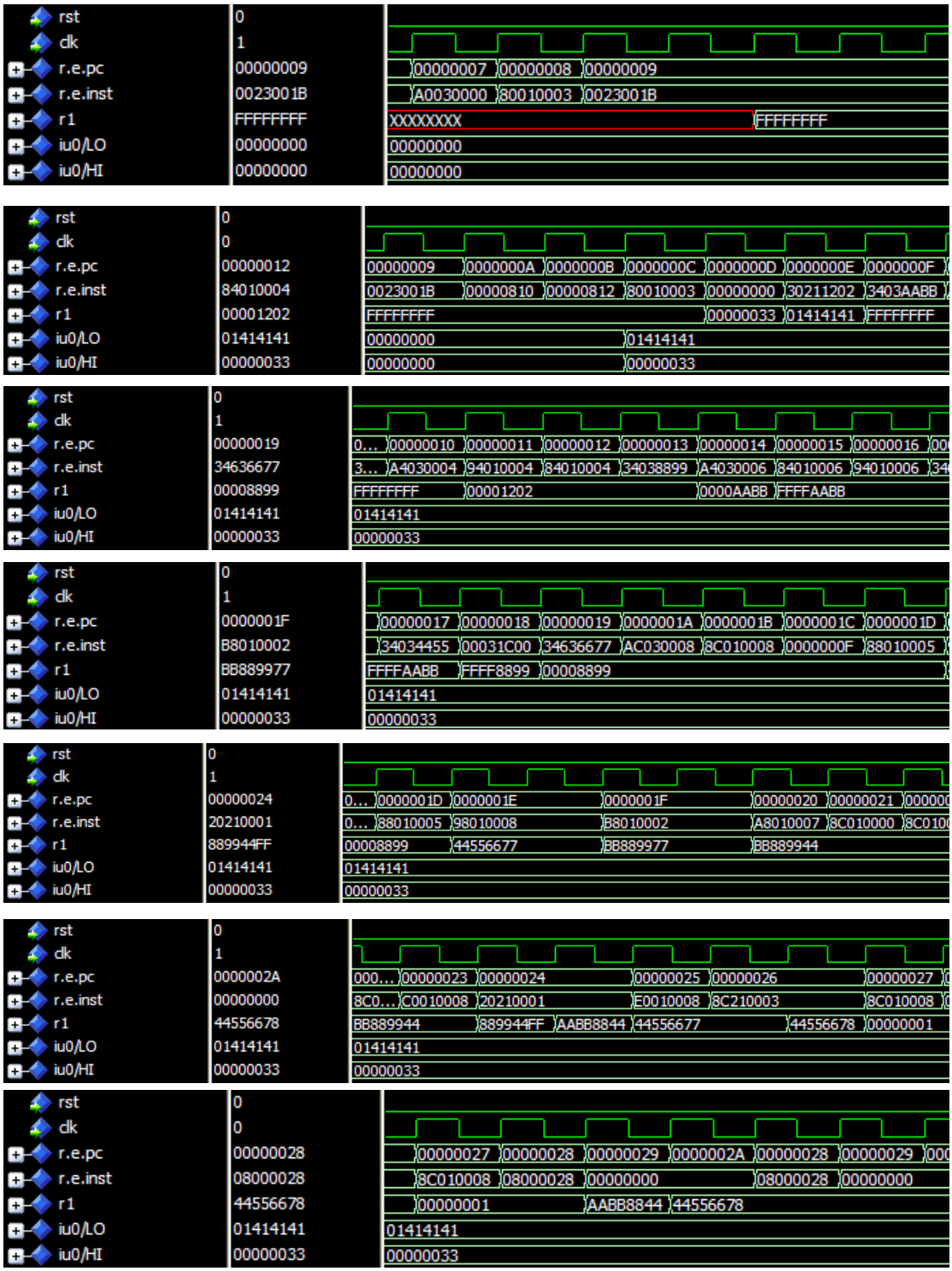
8.2 测试例程

测试程序 1 如下，代码位于 asm_test/Day8_1 目录下：

```
  .org 0x0
  .set noat
  .set noreorder
  .set nomacro
  .global _start
_start:
  ori  $3,$0,0xeeff
  sb   $3,0x3($0)      # [0x3] = 0xff
  srl  $3,$3,8
  sb   $3,0x2($0)      # [0x2] = 0xee
  ori  $3,$0,0xccdd
  sb   $3,0x1($0)      # [0x1] = 0xdd
  srl  $3,$3,8
  sb   $3,0x0($0)      # [0x0] = 0xcc
  lb   $1,0x3($0)      # r1 = 0xffffffff
  divu $zero,$1,$3
  mfhi $1             # r1 = 0x00000033
```

```
  mflo $1             # r1 = 0x01414141
  lb   $1,0x3($0)      # r1 = 0xffffffff
  nop
  andi $1,$1,0x1202    # r1 = 0x00001202

  ori  $3,$0,0xaabb
  sh   $3,0x4($0)      # [0x4] = 0xaa, [0x5] = 0xbb
  lhu  $1,0x4($0)      # r1 = 0x0000aabb
  lh   $1,0x4($0)      # r1 = 0xffffaabb

  ori  $3,$0,0x8899
  sh   $3,0x6($0)      # [0x6] = 0x88, [0x7] = 0x99
  lh   $1,0x6($0)      # r1 = 0xffff8899
  lhu  $1,0x6($0)      # r1 = 0x00008899

  ori  $3,$0,0x4455
  sll  $3,$3,0x10
  ori  $3,$3,0x6677
  sw   $3,0x8($0)      # [0x8] = 0x44, [0x9]= 0x55, [0xa]= 0x66, [0xb] = 0x77
  lw   $1,0x8($0)      # r1 = 0x44556677

  lwl  $1, 0x5($0)     # r1 = 0xbb889977
  lwr  $1, 0x8($0)     # r1 = 0xbb889944

  swr  $1, 0x2($0)     # [0x0] = 0x88, [0x1] = 0x99, [0x2]= 0x44, [0x3] = 0xff
  swl  $1, 0x7($0)     # [0x4] = 0xaa, [0x5] = 0xbb, [0x6] = 0x88, [0x7] = 0x44

  lw   $1, 0x0($0)     # r1 = 0x889944ff
  lw   $1, 0x4($0)     # r1 = 0xaabb8844

  ll   $1, 0x8($0)     # r1 = 0x44556677
  addi $1,$1,0x1       # r1 = 0x44556678
  sc   $1, 0x8($0)     # r1 = 0x00000001
  lw   $1, 0x3($1)     # r1 = 0xaabb8844
  lw   $1, 0x8($0)     # r1 = 0x44556678

_loop:
  j _loop
  nop
```

预期执行效果如程序中注释所示，ModelSim 仿真如下，从中可知 OpenMIPS 正确实现了加载存储类指令。OpenMIPS 的代码位于 10_Days_make_OpenMIPS/Day8 目录下。

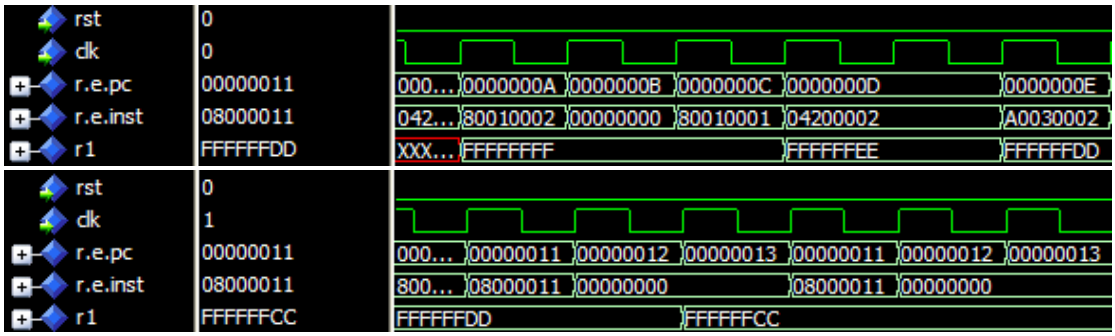


测试程序 2，目的是将加载、存储指令放在延迟槽中，测试是否能够正确实现，程序如下，代码位于 asm_test/Day8_2 目录下：

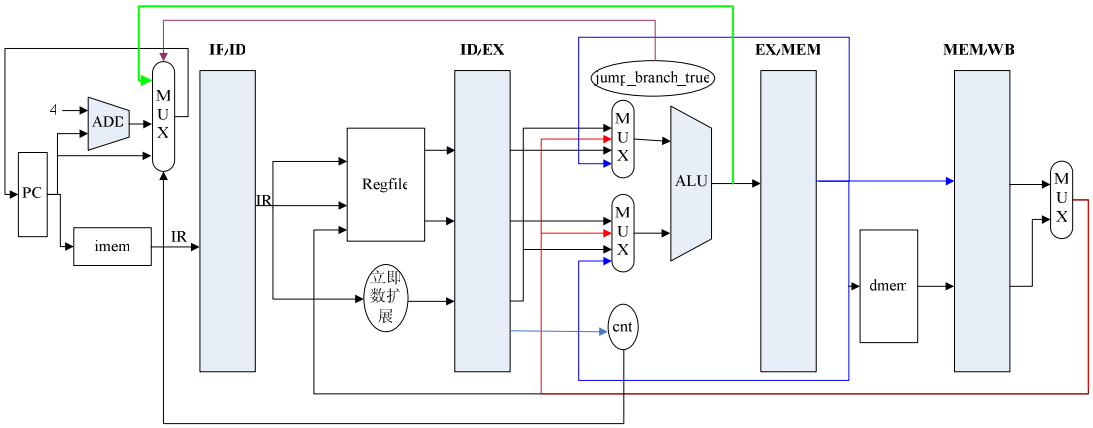
```
.org 0x0
.set noat
.set noreorder
.set nomacro
.global _start
```

```
_start:
   ori  $3,$0,0xeeff
   sb   $3,0x3($0)      # [0x3] = 0xff
   srl  $3,$3,8
   sb   $3,0x2($0)      # [0x2] = 0xee
   ori  $3,$0,0xccdd
   sb   $3,0x1($0)      # [0x1] = 0xdd
   srl  $3,$3,8
   sb   $3,0x0($0)      # [0x0] = 0xcc
   lb   $1,0x3($0)      # r1 = 0xffffffff
   bltz $1,s1
   lb   $1,0x2($0)      # r1 = 0xffffffee
   sll  $1,$1,0x10

s1:
   lb   $1,0x1($0)      # r1 = 0xffffffdd
   bltz $1,s2
   sb   $3,0x2($0)      # [0x2] = 0xcc
   or   $1,$1,$0

s2:
   lb   $1,0x2($0)      # r1 = 0xffffffcc

_loop:
   j _loop
   nop
```

ModelSim 仿真如下，观察 r1 的值，可知加载存储指令放在延迟槽中，也能正确执行。OpenMIPS 的代码位于 10_Days_make_OpenMIPS/Day8 目录下。



实现加载存储指令后的 OpenMIPS 数据流图如下，在流水线的访存阶段增加了 dmem 模块，同时在流水线回写阶段，增加了一个 2 选 1 的选择模块用于选择要写入 Regfile 的数据。


IF/ID
ID/EX
EX/MEM
MEM/WB
4
ADE
M
U
X
PC
imem
IR
IR
Regfile
立即数扩展
jump_branch_true
M
U
X
M
U
X
ALU
cnt
dmem
M
U
X

第九天

主要内容

（1）实现协处理器 CP0 的部分寄存器
（2）实现协处理器访问指令——MFC0、MTC0

9.1 实现协处理器 CP0 的部分寄存器

OpenMIPS 实现了 CP0 的部分寄存器，包括：count、compare、status、cause、EPC、config、PrId、BadVAddr、ErrorEPC，这些寄存器的定义可以参考 MIPS32 手册，或者《MIPS 体系结构与编程》、《MIPS 体系结构透视（See MIPS Run）》等书籍，也可以在 doc 目录下的 MIPS32_3_Privileged_Resource.pdf 文档中找到有关资料。

修改 stdlib.vhd，增加如下定义：

```
--根据 MIPS 标准，定义 CP0 寄存器的序号，比如 count 寄存器的序号是 9，就定义为 01001
Constant CP0_REG_COUNT    : std_logic_vector(4 downto 0) := "01001"; --可读写
constant CP0_REG_COMPARE  : std_logic_vector(4 downto 0) := "01011";--可读写
constant CP0_REG_STATUS   : std_logic_vector(4 downto 0) := "01100"; --可读写
constant CP0_REG_CAUSE    : std_logic_vector(4 downto 0) := "01101"; --只读
constant CP0_REG_EPC    : std_logic_vector(4 downto 0) := "01110";    --可读写
constant CP0_REG_PrId    : std_logic_vector(4 downto 0) := "01111";  --只读
constant CP0_REG_CONFIG    : std_logic_vector(4 downto 0) := "10000"; --只读

  type status_reg_type is record
   CU : std_logic_vector(3 downto 0);
   RP : std_logic;
   RE : std_logic;
   BEV : std_logic;
   TS : std_logic;
   SR : std_logic;
   NMI : std_logic;
   IM : std_logic_vector(7 downto 0);
   UM : std_logic;
   ERL : std_logic;
   EXL : std_logic;
   IE : std_logic;
 end record;

 type cause_reg_type is record
```

```
  BD : std_logic;
  CE : std_logic_vector(1 downto 0);
  IV : std_logic;
  WP : std_logic;
  IP : std_logic_vector(7 downto 0);
  ExcCode : std_logic_vector(4 downto  0);
end record;

type config_reg_type is record
  M : std_logic;
  ISP : std_logic;
  DSP : std_logic;
  MDU : std_logic;
  BE : std_logic;
  AT : std_logic_vector(1 downto 0);
  AR : std_logic_vector(2 downto 0);
  MT : std_logic_vector(2 downto 0);
end record;

type PrId_reg_type is record
  CompanyID : std_logic_vector(7 downto 0);
  ProcessorID : std_logic_vector(7 downto 0);
  Revision : std_logic_vector(7 downto 0);
end record;

subtype EPC_reg_type is std_logic_vector(31 downto 0);
subtype BadVAddr_reg_type is std_logic_vector(31 downto 0);
subtype ErrorEPC_reg_type is std_logic_vector(31 downto 0);
subtype count_reg_type is std_logic_vector(31 downto 0);
subtype compare_reg_type is std_logic_vector(31 downto 0);

type CP0_reg_type is record
   status : status_reg_type;     --CP0 中的 status 寄存器
   cause : cause_reg_type;       --CP0 中的 cause 寄存器
   config : config_reg_type;     --CP0 中的 config 寄存器
   PrId : PrId_reg_type;         --CP0 中的 PrId 寄存器
   EPC : EPC_reg_type;           --CP0 中的 EPC 寄存器
   BadVAddr : BadVAddr_reg_type; --CP0 中的 BadVAddr 寄存器
   ErrorEPC : ErrorEPC_reg_type; --CP0 中的 ErrorEPC 寄存器
   count : count_reg_type;       --CP0 中的 count 寄存器
   compare : compare_reg_type;   --CP0 中的 compare 寄存器
end record;
```

修改访存、回写阶段的结构体定义，增加协处理器 CP0，如下：

```
--访存阶段的寄存器
type memory_reg_type is record
  ……
  CP0: CP0_reg_type;        --最新的 CP0 寄存器的值
end record;

--回写阶段的寄存器
type write_reg_type is record
  ……
  CP0 : CP0_reg_type;       --CP0 寄存器的值
  CP0_t : CP0_reg_type;       --最新的 CP0 寄存器的值，下一个时钟周期会赋值给 CP0
end record;
```

9.2 实现协处理器访问指令

MIPS32 指令集中访问协处理器寄存器的指令有两条 MFC0、MTC0，OpenMIPS 实现了这两条指令。这两条指令说明如下：

指令	用法	作用	说明
mfc0	mfc0 rt,rd,sel	rt <- CPR[0,rd,sel]	将 rd、sel 指定的协处理器 CP0 中的寄存器读出，存储到 rt 中，其中如果没有指定 sel 的值，那么默认为 0
mtc0	mtc0 rt,rd,sel	CPR[0,rd,sel] <- rt	将 rt 的值存储到 rd、sel 指定的协处理器 CP0 中的寄存器，其中如果没有指定 sel 的值，那么默认为 0

为了实现这两条指令，需要对流水线作如下修改：

（1）流水线执行阶段修改如下：

```
  --调用过程 set_new_cp0_at_exstage，得到最新的 CP0 寄存器的值
  set_new_cp0_at_exstage(r, v, ex_opdata1, ex_opdata2, v.m.CP0);

  --调用过程 move_op 进行移动操作类指令的执行，结果存储在 ex_move_res 中
   move_op(r, v.m.CP0, ex_opdata1, ex_opdata2, newhilo, ex_move_res, notmove);
```

在 set_new_cp0_at_exstage 函数中设置新的 CP0 寄存器的值，其中就处理了 mtc0 指令，如下：

```
 procedure set_new_cp0_at_exstage(r,v: registers;
                   opdata1, opdata2: in word;
                   new_CP0: out CP0_reg_type) is
 begin
   --取得最新的 CP0 的值
   if(r.m.aluop = EXE_MTC0_OP) then
     new_CP0 := r.m.CP0;   --如果访存阶段的指令是 mtc0，那么 r.m.cp0 是最新的 CP0 的值
   else
```

```
    new_CP0 := v.w.CP0;   --反之，最新的 CP0 就是 v.w.CP0
  end if;

  if(r.e.aluop = EXE_MTC0_OP) then
     if(opdata2(31 downto 5) = "00000000000000000000000000000") then
     case opdata2(4 downto 0) is
               --mtc0 指令写 count 寄存器
               when CP0_REG_COUNT => new_CP0.count := opdata1;
               --mtc0 指令写 compare 寄存器
               when CP0_REG_COMPARE => new_CP0.compare := opdata1;
                                       new_CP0.count := zero32;
               --mtc0 指令写 EPC 寄存器
               when CP0_REG_EPC => new_CP0.EPC := opdata1;
               --mtc0 指令写 status 寄存器
               when CP0_REG_STATUS =>
                        new_CP0.status.CU := opdata1(31 downto 28);
                        new_CP0.status.RP := opdata1(27);
                        new_CP0.status.RE := opdata1(25);
                        new_CP0.status.BEV := opdata1(22);
                        new_CP0.status.TS := opdata1(21);
                        new_CP0.status.SR := opdata1(20);
                        new_CP0.status.NMI := opdata1(19);
                        new_CP0.status.IM := opdata1(15 downto 8);
                        new_CP0.status.UM := opdata1(4);
                        new_CP0.status.ERL := opdata1(2);
                        new_CP0.status.EXL := opdata1(1);
                        new_CP0.status.IE := opdata1(0);
               --mtc0 指令写 cause 寄存器
               when CP0_REG_CAUSE =>
                    new_CP0.cause.BD := opdata1(1);
                    new_CP0.cause.CE := opdata1(29 downto 28);
                    new_CP0.cause.IV := opdata1(23);
                    new_CP0.cause.WP := opdata1(22);
                    new_CP0.cause.IP := opdata1(15 downto 8);
                    new_CP0.cause.ExcCode := opdata1(6 downto 2);
               when others =>
           end case;
           end if;
    end if;
 end;
```

最新的 CP0 寄存器的值存储到 v.m.cp0 中，后者传递到 move_op 函数，在其中实现 mfc0 指令的处理，如下：

```
procedure move_op(r : registers; new_CP0 : CP0_reg_type;
              aluin1, aluin2: word;
```

```
                newhilo : std_logic_vector(63 downto 0);
                moveres : out word; notmove : out boolean) is
  variable moveout,zeros : word;
begin
   moveout := (others => '0');
   zeros := (others => '0');
   notmove := false;
    case r.e.aluop is
       ……
       when EXE_MFC0_OP =>
            if(aluin2(31 downto 5) = "00000000000000000000000000000") then
            case aluin2(4 downto 0) is
              when CP0_REG_COUNT => moveout := new_CP0.count;
              when CP0_REG_COMPARE => moveout := new_CP0.compare;
              when CP0_REG_EPC => moveout := new_CP0.EPC;
              when CP0_REG_STATUS =>
                     moveout := new_CP0.status.CU & new_CP0.status.RP &
                                '0' & new_CP0.status.RE &
                                "00" & new_CP0.status.BEV &
                                new_CP0.status.TS & new_CP0.status.SR &
                                new_CP0.status.NMI & "000" &
                                new_CP0.status.IM & "000" &
                                new_CP0.status.UM &
                                '0' & new_CP0.status.ERL &
                                new_CP0.status.EXL & new_CP0.status.IE;
              when CP0_REG_CAUSE =>
                     moveout := new_CP0.cause.BD & '0' &
                                new_CP0.cause.CE & "0000" &
                                new_CP0.cause.IV &
                                new_CP0.cause.WP & "000000" &
                                new_CP0.cause.IP & '0' &
                                new_CP0.cause.ExcCode & "00";
              when CP0_REG_PrId =>
                     moveout := "00000000" & new_CP0.PrId.CompanyID &
                                new_CP0.PrId.ProcessorID &
                                new_CP0.PrId.Revision;
              when CP0_REG_CONFIG =>
                     moveout := new_CP0.config.M & "000000" &
                                new_CP0.config.ISP & new_CP0.config.DSP &
                               "00" & new_CP0.config.MDU & "0000" &
                               new_CP0.config.BE & new_CP0.config.AT &
                               new_CP0.config.AR & new_CP0.config.MT &
                               "0000000";
              when others =>
```

```
                        moveout := (others => '0');
                end case;
                else
                  moveout := (others => '0');
                end if;
           when others => moveout := (others => '-');
        end case;
        moveres := moveout;
 end;
```

（2）流水线访存阶段增加如下代码，将新的 CP0 寄存器的值传递到回写阶段：

```
……
     v.w.CP0_t := r.m.CP0;
……
```

（3）流水线回写阶段增加如下代码，依据传递过来的值，设置新的 CP0 寄存器，同时判断是否发生时钟中断（当 compare 寄存器的值等于 count 寄存器的值的时候，会发生时钟中断，CPU 的输出 SI_TimerInt 为 1）。

```
……
   set_new_cp0_at_wbstage(r, v, v.w.CP0, timer_int);

   SI_TimerInt <= timer_int;
……
```

其中 set_new_cp0_at_wbstage 的定义如下：

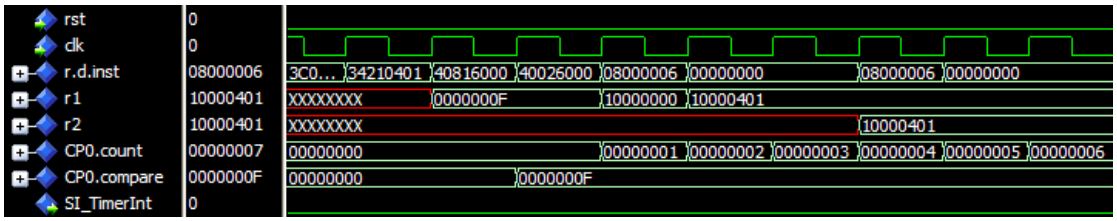
```
  procedure set_new_cp0_at_wbstage(r, v: registers;
                                   new_CP0: out CP0_reg_type;
                                   timer_int : out std_logic) is
  begin

    new_CP0 := r.w.CP0;

    --如果 count 寄存器的值等于 compare 寄存器的值，并且 compare 不为 0，那么输出时钟中断
    --同时，保持 count 寄存器的值不变，等待中断处理
    if((r.w.CP0.count = r.w.CP0.compare) and r.w.CP0.compare /= zero32) then
        timer_int := '1';
        new_CP0.count := r.w.CP0.count;
     --如果 compare 寄存器的值等于 0，那么 count 寄存器的值保持为 0，不计数，不产生中断
     elsif(r.w.CP0.compare = zero32) then
        new_CP0.count := zero32 ;
        timer_int := '0';
     --其余情况下，每个时钟周期 count 加 1
     else
        new_CP0.count := r.w.CP0.count + 1;
        timer_int := '0';
```

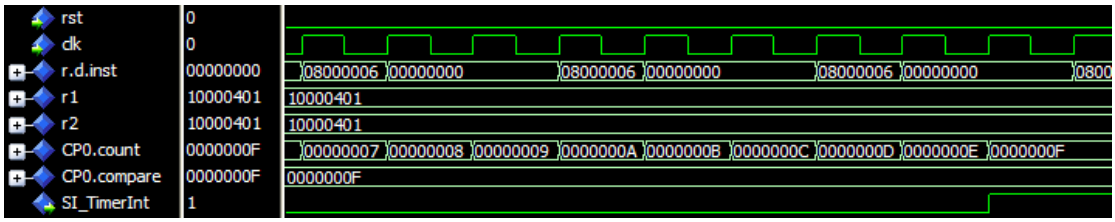
```
    end if;

    if(r.w.aluop = EXE_MTC0_OP ) then
       if(r.w.opdata2(31 downto 5) = "00000000000000000000000000000") then
       case r.w.opdata2(4 downto 0) is
            when CP0_REG_COUNT => new_CP0.count := r.w.CP0_t.count;
            --此处单独处理 compare 寄存器，因为，如果要写 compare 寄存器，
            --那么需要复位时钟中断，同时设置 count 为 0
            when CP0_REG_COMPARE =>  new_CP0.compare := r.w.opdata1;
                                     timer_int := '0'; new_CP0.count := zero32;
                  when others =>
                           new_CP0.EPC := r.w.CP0_t.EPC;
                           new_CP0.status := r.w.CP0_t.status;
                           new_CP0.cause := r.w.CP0_t.cause;
              end case;
              end if;
     end if;


  end;
```

测试程序如下，代码位于 asm_test/Day9 文件夹下。

```
  .org 0x0
  .set noat
  .set noreorder
  .set nomacro
  .global _start
_start:
  ori $1,$0,0xf
  mtc0 $1,$11,0x0  #写 compare 寄存器，开始计时
  lui $1,0x1000
  ori $1,$1,0x401
  mtc0 $1,$12,0x0  #将 0x401 写入 status 寄存器
  mfc0 $2,$12,0x0  #读 status 寄存器，r2=0x401

_loop:
  j _loop
  nop
```

程序首先写 compare 寄存器，使其值等于 0xf，这样 OpenMIPS 就开始计数，15 个时钟周期后 count 的值等于 compare 的值，会输出时钟中断，在程序的最后将 0x401 写入 status 寄存器，然后读出到 r2 中，ModelSim 仿真效果如下所示，从中可知 OpenMIPS 正确实现了 MIPS32 中的部分 CP0 寄存器，以及 CP0 寄存器访问指令。OpenMIPS 的代码位于 10_Days_make_OpenMIPS/Day9 目录下。


rst
clk
r.d.inst
r1
r2
CP0.count
CP0.compare
SI_TimerInt
0
0
08000006
10000401
10000401
00000007
0000000F
0
3C0...
34210401
40816000
40026000
08000006
00000000
08000006
00000000
XXXXXXXX
0000000F
10000000
10000401
XXXXXXXX
10000401
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000000
0000000F


rst
clk
r.d.inst
r1
r2
CP0.count
CP0.compare
SI_TimerInt
0
0
00000000
10000401
10000401
0000000F
0000000F
1
08000006
00000000
08000006
00000000
08000006
00000000
0800
10000401
10000401
00000007
00000008
00000009
0000000A
0000000B
0000000C
0000000D
0000000E
0000000F
0000000F

第十天

主要内容

（1）实现自陷指令——SYSCALL、TEQ、TEQI、TGE、TGEI、TGEIU、TGEU、TLT、TLTI、TLTIU、TLTU、TNE、TNEI
（2）实现中断处理
（3）实现异常返回指令——ERET

10.1 自陷指令

OpenMIPS 支持精确异常模型，当异常发生时，有关处理器的状态信息被存储到 CP0 的部分寄存器中，然后 OpenMIPS 处理器转移到事先定义好的一个地址，在那个地址中往往存储有异常处理例程，在其中进行异常处理，这个地址称为异常处理例程入口地址。

具体来说，当异常发生时，OpenMIPS 会进行如下操作：

（1）设置 EPC 寄存器：如果当前指令不在延迟槽中，那么当前指令（或下一条指令）地址被存储到 EPC 寄存器中，如果当前指令在延迟槽中，那么当前指令的上一条转移指令地址被存储到 EPC 寄存器中。

（2）设置 CASUE 寄存器：在其 ExeCode 字段存储异常原因代码，具体含义可以参考 MIPS32 说明，或者《MIPS 体系结构与编程》、《MIPS 体系结构透视》等书籍。

（3）设置 STATUS 寄存器：设置其 EXL 位为 1，表示处理器处于异常处理状态。

（4）转移到相应的异常处理例程继续执行。

当异常处理结束后，需要使用指令 eret 从异常返回，eret 指令会将 EPC 的值恢复到 PC，同时设置 STATUS 的 EXL 位为 0，处理器回到异常发生前的状态继续执行。

OpenMIPS 支持的异常类型及对应处理例程的入口地址如下表所示，其中无效指令、系统调用、自陷的处理例程入口地址都是 0x40，童鞋们可以自己修改，使得这三个异常的处理例程入口地址不一样。异常的优先级也是可以调整的，童鞋们只要理解了 OpenMIPS 的代码，这些都是可以灵活修改的。

异常类型及对应处理例程的入口地址

异常类型	优先级	处理例程地址	引起异常的条件
复位（Reset）	1	0x0	由软件或硬件复位引起
外部中断	2	0x20	OpenMIPS 支持 8 个外部中断
系统调用（System Call）	3	0x40	使用指令 syscall
无效指令	4	0x40	指令不是 OpenMIPS 支持的指令
自陷（Trap）	5	0x40	使用指令 teq、teqi、tge、tgei、tgeiu、tgeu、tlt、tlti、tltiu、tltu、tne、tnei 等

本小节介绍 syscall、自陷指令的实现。OpenMIPS 支持的 syscall、自陷指令的含义如下表所示。

指令	用法	作用	说明
syscall	syscall	引发系统调用异常	
teq	teq rs,rt	if rs = rt then trap	如果源操作数 rs 等于 rt，那么引发自陷异常
teqi	teqi rs,immediate	if rs = immediate then trap	指令中的 16 位立即数符号扩展为 32 位，然后与 rs 进行有符号数比较，如果相等，那么引发自陷异常
tge	tge rs,rt	if rs >= rt then trap	如果源操作数 rs 大于等于 rt(作为符号数比较)，那么引发自陷异常
tgei	tgei rs,immediate	if rs >= immediate then trap	指令中的 16 位立即数符号扩展为 32 位，然后与 rs 进行有符号数比较，如果 rs 大于等于立即数扩展的值，那么引发自陷异常
tgeiu	tgeiu rs,immediate	if rs >= immediate then trap	指令中的 16 位立即数符号扩展为 32 位，然后与 rs 进行无符号数比较，如果 rs 大于等于立即数扩展的值，那么引发自陷异常
tgeu	tgeu rs,rt	if rs >= rt then trap	如果源操作数 rs 大于等于 rt（作为无符号数比较），那么引发自陷异常
tlt	tlt rs,rt	if rs < rt then trap	如果源操作数 rs 小于 rt（作为符号数比较），那么引发自陷异常
tlti	tlti rs,immediate	if rs < immediate then trap	指令中的 16 位立即数符号扩展为 32 位，然后与 rs 进行有符号数比较，如果 rs 小于立即数扩展的值，那么引发自陷异常
tltiu	tltiu rs,immediate	if rs <immediate then trap	指令中的 16 位立即数符号扩展为 32 位，然后与 rs 进行无符号数比较，如果 rs 小于立即数扩展的值，那么引发自陷异常
tltu	tlt rs,rt	if rs < rt then trap	如果源操作数 rs 小于 rt（作为无符号数比较），那么引发自陷异常
tne	tne rs,rt	if rs /= rt then trap	如果源操作数 rs 不等于 rt，那么引发自陷异常
tnei	tnei rs,immediate	if rs /=immediate then trap	指令中的 16 位立即数符号扩展至 32 位，然后与 rs 比较，如果不相等，那么引发自陷异常

OpenMIPS 实现异常处理的基本思路是：为流水线除取指阶段外的每一个阶段添加一个 annul 信号，在流水线的回写阶段判断当前是否有异常发生，如果有异常发生，那么设置流水线每个阶段的 annul 信号为 1，表示取消各个阶段的执行，保存相应的值到 EPC、STATUS、CAUSE 等 CP0 寄存器，同时设置 PC 为异常处理入口地址，这样就实现了异常处理。具体修改如下：

（1）修改 stdlib.vhd 中流水线各个阶段寄存器的定义，添加 annul 信号：

```
--取指阶段的寄存器
type fetch_reg_type is record
  pc    : pctype;  --要读取的指令地址
end record;
```

```
--译码阶段的寄存器
type decode_reg_type is record
  ……
  annul  : std_logic;
end record;

--执行阶段的寄存器
type execute_reg_type is record
  ……
  annul  : std_logic;
end record;

--访存阶段的寄存器
type memory_reg_type is record
  ……
  annul  : std_logic;
end record;

--回写阶段的寄存器
type write_reg_type is record
  ……
  annul  : std_logic;
end record;

type registers is record
  f  : fetch_reg_type;
  d  : decode_reg_type;
  e  : execute_reg_type;
  m  : memory_reg_type;
  w  : write_reg_type;
end record;
```

（2）修改流水线译码阶段调用的 inst_deocde 函数，添加对自陷指令、syscall 指令的译码。

（3）修改流水线执行阶段，调用新增加的函数 trap_op，在其中判断是否满足自陷条件，如果满足自陷条件，那么其输出 exception_type 为“1000000”，该值会在回写阶段使用到。

```
procedure trap_op(r: registers; aluin1, aluin2: word;
          exception_type: out std_logic_vector(7 downto 0)) is
  variable signed_comp, a_gt_b, a_eq_b : std_logic;
  variable comp_a, comp_b : word;
begin

   if r.e.aluop = EXE_TGEIU_OP or r.e.aluop = EXE_TGEU_OP or
      r.e.aluop = EXE_TLTIU_OP or r.e.aluop = EXE_TLTU_OP then
```

```
      signed_comp := '0';            --是无符号数比较
    else
      signed_comp := '1';            --是有符号数比较
    end if;

    comp_a := (aluin1(31) xor signed_comp) & aluin1(30 downto 0);
    comp_b := (aluin2(31) xor signed_comp) & aluin2(30 downto 0);

    if comp_a > comp_b then
      a_gt_b := '1';
      a_eq_b := '0';
    elsif comp_a = comp_b then
      a_gt_b := '0';
      a_eq_b := '1';
    else
      a_gt_b := '0';
      a_eq_b := '0';
    end if;

    case r.e.aluop is
      when EXE_TEQI_OP | EXE_TEQ_OP =>
          if(a_eq_b = '1') then
            exception_type := EXCEPTION_TYPE_TRAP;  --满足自陷条件
          else
            exception_type := (others => '0');
          end if;
      when EXE_TGE_OP | EXE_TGEI_OP | EXE_TGEIU_OP | EXE_TGEU_OP =>
          if(a_gt_b = '1' or a_eq_b = '1') then
            exception_type := EXCEPTION_TYPE_TRAP;  --满足自陷条件
          else
            exception_type := (others => '0');
          end if;
      when EXE_TLT_OP | EXE_TLTI_OP | EXE_TLTIU_OP | EXE_TLTU_OP =>
          if(a_gt_b = '0' and a_eq_b = '0') then
            exception_type := EXCEPTION_TYPE_TRAP;  --满足自陷条件
          else
            exception_type := (others => '0');
          end if;
      when EXE_TNEI_OP | EXE_TNE_OP =>
          if(a_eq_b = '0') then
            exception_type := EXCEPTION_TYPE_TRAP;  --满足自陷条件
          else
            exception_type := (others => '0');
          end if;
```

```
        when others => exception_type := (others => '0');
      end case;
end;
```

（4）修改流水线回写阶段调用的 set_new_cp0_at_wbstage 函数，在其中进行异常判断及处理。

```
procedure set_new_cp0_at_wbstage(r, v: registers; new_CP0: out CP0_reg_type;
                int: in std_logic_vector(7 downto 0);
                exception_if_annul, exception_de_annul, exception_ex_annul,
                exception_mem_annul, exception_wb_annul: out std_logic;
                exception_target_pc: out word; timer_int : out std_logic) is
begin
  exception_if_annul := '0';
  exception_de_annul := r.d.annul;
  exception_ex_annul := r.e.annul;
  exception_mem_annul := r.m.annul;
  exception_wb_annul := r.w.annul;
  exception_target_pc := (others => '0');
  new_CP0 := r.w.CP0;
  ……
  ------------       处理 syscall 指令          ---------------
  if(r.w.aluop = EXE_SYSCALL_OP and r.w.annul = '0') then
     if(r.w.CP0.status.EXL = '0') then
       if(r.w.dslot = '1') then   --当前指令是否在延迟槽中
         new_CP0.EPC := r.w.pc & "00" - 4;
         new_CP0.cause.BD := '1';
       else                        --其余情况下设置 EPC 的值等于 PC
         new_CP0.EPC := r.w.pc & "00";
         new_CP0.cause.BD := '0';
       end if;
     end if;
     new_CP0.status.EXL := '1'; --设置 status 寄存器的 EXL 位为 1，
                                --表示处于异常处理中
     exception_if_annul := '1'; --取消取指阶段的指令
     exception_de_annul := '1'; --取消译码阶段的指令
     exception_ex_annul := '1'; --取消执行阶段的指令
     exception_mem_annul := '1'; --取消访存阶段的指令
     exception_wb_annul := '1';  --取消回写阶段的指令

     --此处设置 SYSCALL 的入口地址是 0x40
     exception_target_pc := "00000000000000000000000001000000";

     --设置 cause 寄存器的 ExcCode 位，存储异常代码
     new_CP0.cause.ExcCode := "01000";
     -------------        处理指令无效异常          --------------
```

```
    elsif(r.w.inst_valid = '0' and r.w.annul = '0') then
       if(r.w.CP0.status.EXL = '0') then
      if(r.w.dslot = '1') then
        new_CP0.EPC := r.w.pc & "00" - 4;
        new_CP0.cause.BD := '1';
      else
        new_CP0.EPC := r.w.pc & "00";
        new_CP0.cause.BD := '0';
      end if;
    end if;
    new_CP0.status.EXL := '1';
    exception_if_annul := '1';
    exception_de_annul := '1';
    exception_ex_annul := '1';
    exception_mem_annul := '1';
    exception_wb_annul := '1';

    --此处设置 invalid instruction 的入口地址是 0x40
    exception_target_pc := "00000000000000000000000001000000";
    new_CP0.cause.ExcCode := "01010";
  -------------          处理自陷指令异常          -------------
  elsif((r.w.exception_type and EXCEPTION_TYPE_TRAP) /= "00000000")
        and r.w.annul = '0') then
    if(r.w.CP0.status.EXL = '0') then
      if(r.w.dslot = '1') then
        new_CP0.EPC := r.w.pc & "00" - 4;
        new_CP0.cause.BD := '1';
      else
        new_CP0.EPC := r.w.pc & "00";
        new_CP0.cause.BD := '0';
      end if;
    end if;
    new_CP0.status.EXL := '1';
    exception_if_annul := '1';
    exception_de_annul := '1';
    exception_ex_annul := '1';
    exception_mem_annul := '1';
    exception_wb_annul := '1';

    --此处设置 Txx 指令的入口地址是 0x40
    exception_target_pc := "00000000000000000000000001000000";
    new_CP0.cause.ExcCode := "01101";
  ……
```

当判断异常发生后，设置流水线各个阶段的 annul 信号为 1，表示取消相应阶段的指令

执行。为此，需要修改流水线各个阶段，加入 annul 信号。要修改地方的比较多，本教程不再一一列出，童鞋们可以参考第十天的代码（位于 10_Days_make_OpenMIPS/Day10_1 目录下），也可以比较第十天与第九天的代码，从中体会 annul 的作用。

（5）修改取指阶段，当异常发生后，从异常处理地址取指，如下：

```
  if (rst = '1') then
    ……
  elsif(ex_stall_for_multicycle_inst = '1' or ex_stall_for_new_RF = '1') then
   ……
  elsif(exception_if_annul = '1') then
    v.f.pc := exception_target_pc(31 downto 2);
    v.d.annul := '1';
    v.d.inst := zero32;
  elsif(jump_branch_true = '1') then
    ……
  else
    ……
   end if;
```

10.2 实现中断处理

OpenMIPS 支持 8 个外部中断，外部中断响应要满足三个条件：

1、status 寄存器的中断使能位 IE 为 1

2、status 寄存器的对应中断掩码 IM 为 1

3、不处于异常处理过程中，也就是 status 寄存器的 EXL 位为 0

为了实现对外部中断的响应，需要修改 iu 模块的接口，添加中断输入，如下：

```
entity iu is
  port (
    ……
    int_i : in std_logic_vector(7 downto 0)
    );
end;
```

在每个时钟上升沿寄存中断信号，如下：

```
  reg : process (clk)
  begin
    if rising_edge(clk) then
        if(rst = '1') then
            ……
       else
          ......
           int_r <= int_i;
          ……
      end if;
    end if;
  end process;
```

在流水线回写阶段判断是否有外部中断发生，如果有，那么转移到中断处理例程入口地址，为此，需要修改回写阶段调用的 set_new_cp0_at_wbstage 函数，如下：

```
procedure set_new_cp0_at_wbstage(r, v: registers; new_CP0: out CP0_reg_type;
             int: in std_logic_vector(7 downto 0);
             exception_if_annul, exception_de_annul, exception_ex_annul,
             exception_mem_annul, exception_wb_annul: out std_logic;
             exception_target_pc: out word; timer_int : out std_logic) is
begin
  exception_if_annul := '0';
  exception_de_annul := r.d.annul;
  exception_ex_annul := r.e.annul;
  exception_mem_annul := r.m.annul;
  exception_wb_annul := r.w.annul;
  exception_target_pc := (others => '0');
  new_CP0 := r.w.CP0;
  ……
    --判断是否满足中断条件，具体的中断条件有：
    --1、status 寄存器的中断使能位 IE 为 1
    --2、status 寄存器的对应中断掩码 IM 为 1
    --3、不处于异常处理过程中，也就是 status 寄存器的 EXL 位为 0
    if((int_r and r.w.CP0.status.IM) /= "00000000") and
       r.w.CP0.status.EXL = '0' and r.w.CP0.status.IE = '1' and
       r.w.annul = '0') then
      if(r.w.dslot = '1') then           --当前指令是否在延迟槽中
        new_CP0.EPC := r.w.pc & "00" - 4;
        new_CP0.cause.BD := '1';
      --如果当前指令是存储指令，那么设置 EPC 为 PC+4，因为存储指令在访存阶段就已经
      --执行完毕，如果此处设置为 PC，那么当从异常返回时，会重复执行该存储指令
      elsif(r.w.store_op = '1') then
        new_CP0.EPC := r.w.pc & "00" + 4;
        new_CP0.cause.BD := '0';
      else                              --其余情况下设置 EPC 的值等于 PC
        new_CP0.EPC := r.w.pc & "00";
        new_CP0.cause.BD := '0';
      end if;
      new_CP0.status.EXL := '1';--设置 status 寄存器的 EXL 位为 1，表示处于异常处理中
      exception_if_annul := '1';          --取消取指阶段的指令
      exception_de_annul := '1';          --取消译码阶段的指令
      exception_ex_annul := '1';          --取消执行阶段的指令
      exception_mem_annul := '1';         --取消访存阶段的指令
      exception_wb_annul := '1';          --取消回写阶段的指令
      --此处设置中断的入口地址是 0x20
      exception_target_pc := "00000000000000000000000000100000";
      --设置 cause 寄存器的 ExcCode 位，存储异常代码
```

```
        new_CP0.cause.ExcCode := "00001";
        --设置 cause 寄存器的 IP 位，存储中断信号的值
        new_CP0.cause.IP := int_r;
……
```

10.3 实现异常返回指令——ERET

eret 指令的作用是从异常处理例程返回，其作用是将 EPC 的值写入 PC，同时设置 status 寄存器的 EXL 位为 0，从而可以响应新的异常。为了实现 eret 指令，只需要修改回写阶段的 set_new_cp0_at_wbstage 函数即可，添加如下代码：

```
  procedure set_new_cp0_at_wbstage(r, v: registers; new_CP0: out CP0_reg_type;
                int: in std_logic_vector(7 downto 0);
                exception_if_annul, exception_de_annul, exception_ex_annul,
                exception_mem_annul, exception_wb_annul: out std_logic;
                exception_target_pc: out word; timer_int : out std_logic) is
  begin
    exception_if_annul := '0';
    exception_de_annul := r.d.annul;
    exception_ex_annul := r.e.annul;
    exception_mem_annul := r.m.annul;
    exception_wb_annul := r.w.annul;
    exception_target_pc := (others => '0');
    new_CP0 := r.w.CP0;
    ……
      --处理 eret 返回指令
     elsif(r.w.aluop = EXE_ERET_OP and r.w.annul = '0') then
        exception_if_annul := '1';          --取消当前取指阶段的指令
        exception_de_annul := '1';          --取消当前译码阶段的指令
        exception_ex_annul := '1';          --取消当前执行阶段的指令
        exception_mem_annul := '1';         --取消当前访存阶段的指令
        exception_wb_annul := '1';          --取消当前回写阶段的指令
        exception_target_pc := r.w.CP0.EPC;    --返回到 EPC 寄存器保存的地址处
        new_CP0.status.EXL := '0';             --设置 status 寄存器的 EXL 位为 0
……
```

测试程序 1（位于 asm_test/Day10_1 目录下）

目的：测试 SYSCALL 指令是否正确实现

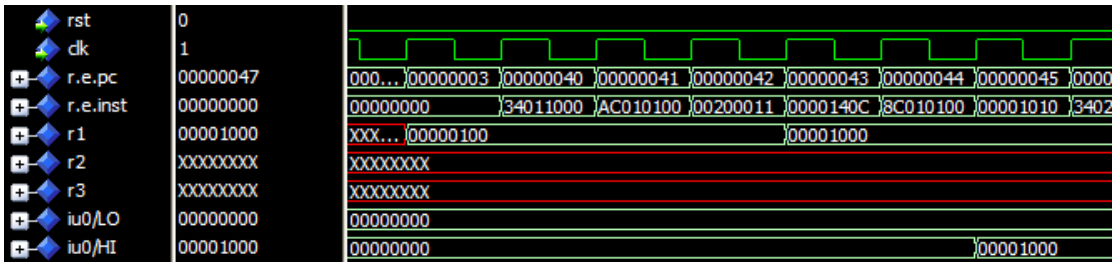
程序如下：

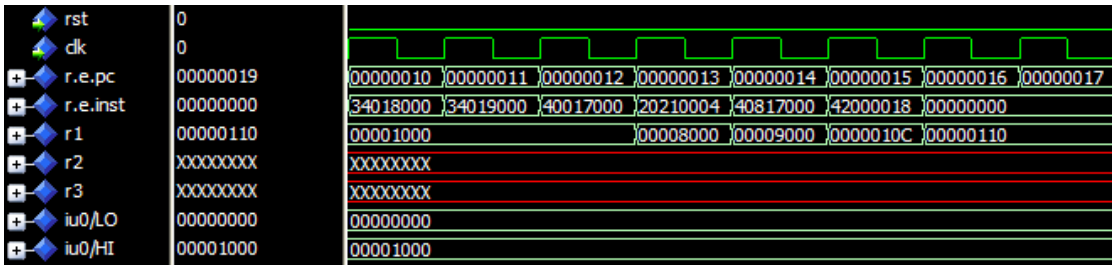
```
  .org 0x0
  .set noat
  .set noreorder
  .set nomacro
```

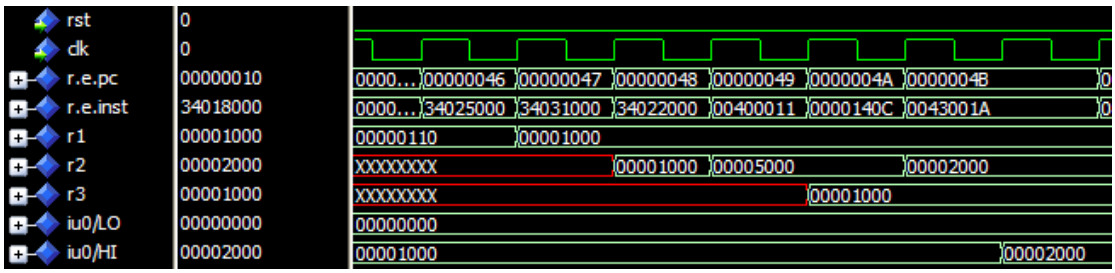
```
   .global _start
_start:
   ori $1,$0,0x100     # r1 = 0x100
   jr $1               # 跳转到 0x100 处
   nop

   .org 0x40           # syscall 系统调用处理例程入口地址
   ori $1,$0,0x8000    # r1 = 0x00008000
   ori $1,$0,0x9000    # r1 = 0x00009000
   mfc0 $1,$14,0x0     # 取得 EPC 的值，r1 = 0x0000010c
   addi $1,$1,0x4      # 将取到的 EPC 的值加 4，r1 = 0x00000110
   mtc0 $1,$14,0x0     # 将修改后的 EPC 的值保存回 EPC
   eret                # 异常返回
   nop

   .org 0x100
   ori $1,$0,0x1000    # r1 = 0x1000
   sw  $1, 0x0100($0)  # [0x100] = 0x00001000
   mthi $1           # HI = 0x00001000
   syscall 0x50
   lw  $1, 0x0100($0)  # r1 = 0x00001000
   mfhi $2           # r2 = 0x00001000
   ori $2,$0,0x5000    # r2 = 0x00005000

   ori $3,$0,0x1000    # r3 = 0x00001000
   ori $2,$0,0x2000    # r2 = 0x00002000
   mthi $2           # HI = 0x00002000
   syscall 0x50
   div $0,$2,$3       # HI = 0x0; Lo = 0x2
   j  _loop
   ori $2,$0,0x5000    # r2 = 0x00005000

_loop:
   j _loop
   nop
```

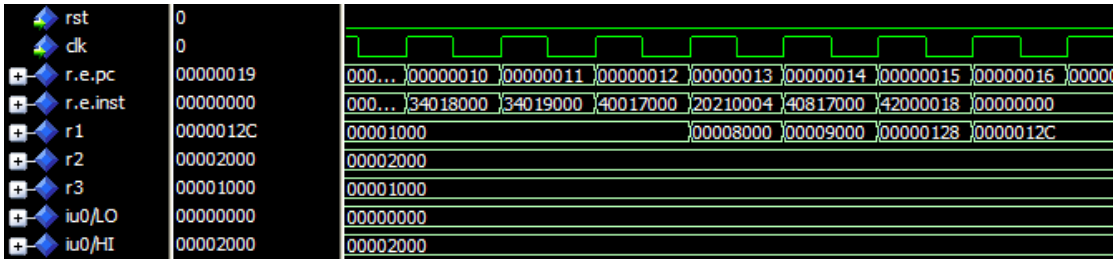
在 syscall 处理例程中将 EPC 的值加 4，是因为 syscall 系统调用会将其自身的地址保存到 EPC 寄存器中，如果直接使用 eret 返回，那么会又回到 syscall 指令处，从而导致再次执行系统调用指令 syscall，所以在 eret 指令之前，将 EPC 的值加 4，这样就可以返回 syscall 的下一条指令处继续执行。

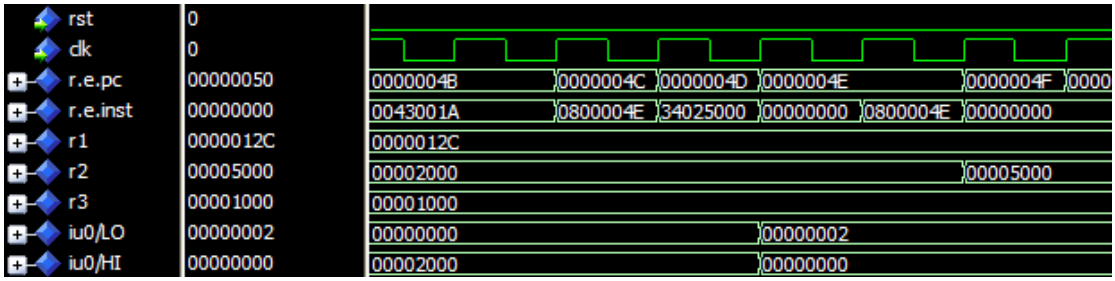
指令执行效果应该如程序中注释所示，ModelSim 仿真如下，观察 r1、r2、HI、LO 的变化可知，正确实现了 SYSCALL 指令。OpenMIPS 的代码位于 10_Days_make_OpenMIPS/Day10_3 目录下。











测试程序 2（位于 asm_test/Day10_2 文件夹下）

目的：测试 Trap Condition 指令是否正确实现

程序如下：

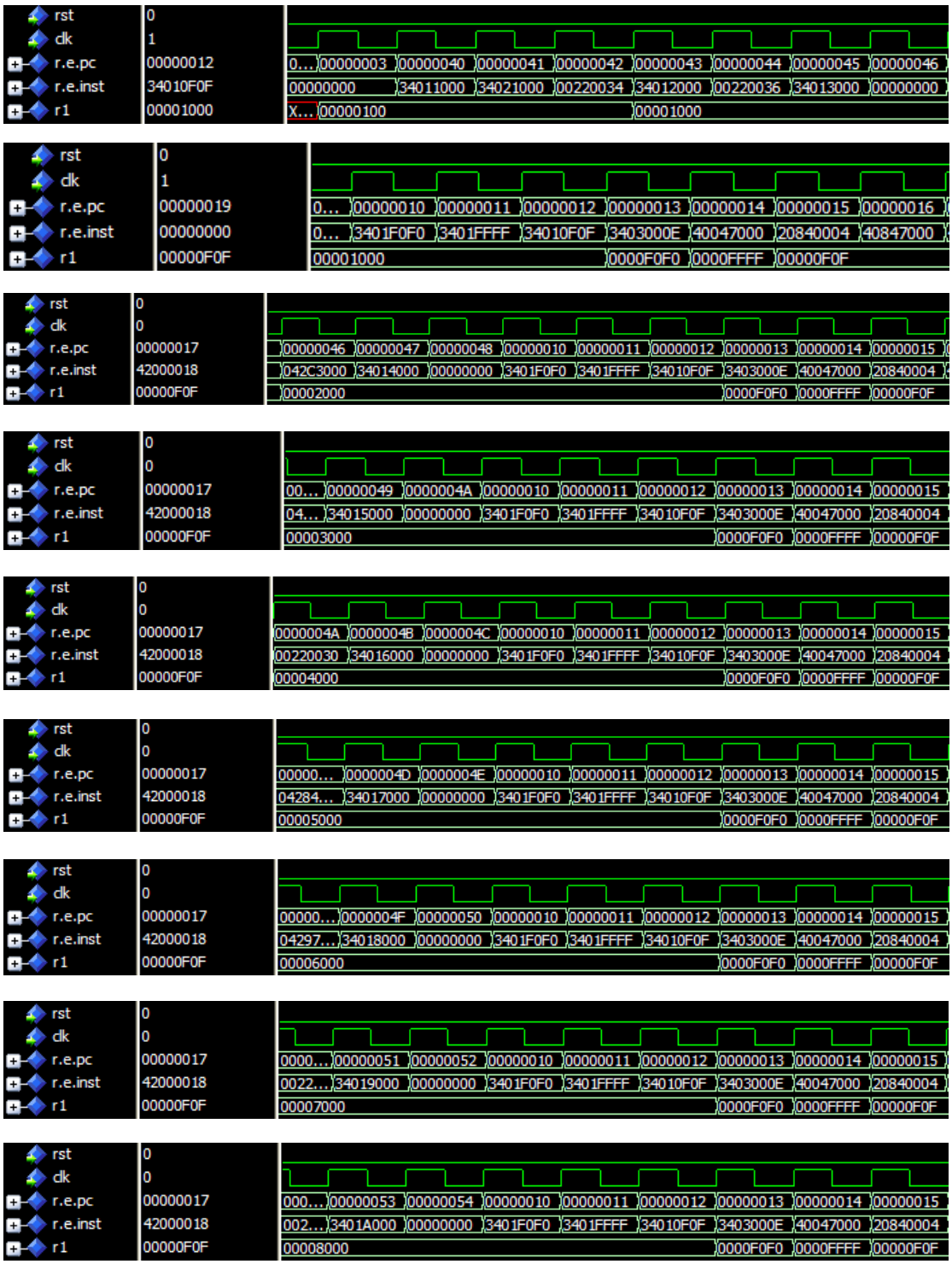
```
.org 0x0
.set noat
.set noreorder
.set nomacro
```

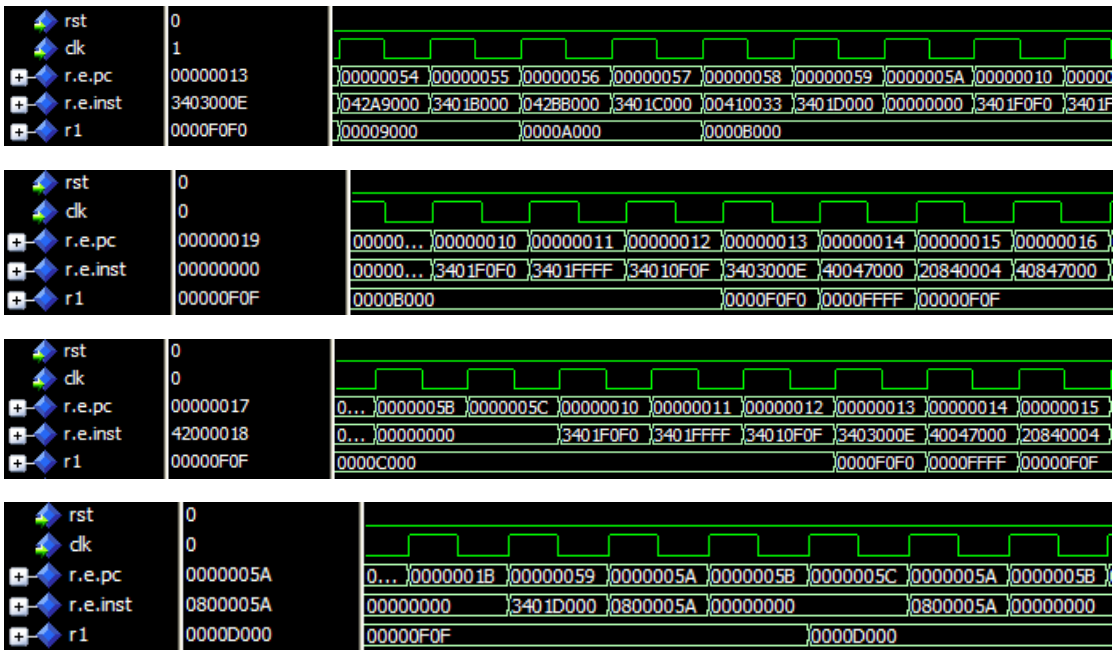
```
  .global _start
_start:
  ori $1,$0,0x100    # r1 = 0x100
  jr $1
  nop

  .org 0x40           # 自陷指令处理例程入口地址
  ori $1,$0,0xf0f0    # r1 = 0x0000f0f0
  ori $1,$0,0xffff    # r1 = 0x0000ffff
  ori $1,$0,0x0f0f    # r1 = 0x00000f0f
  ori $3,$0,0x0e     # r3 = 0x0000000e
  mfc0 $4,$14,0x0    # 取得 EPC 寄存器的值
  addi $4,$4,0x4     # 将取到的 EPC 寄存器的值加 4
  mtc0 $4,$14,0x0    # 将修改的后保存回 EPC 寄存器
  eret
  nop

  .org 0x100
  ori $1,$0,0x1000    # r1 = 0x00001000
  ori $2,$0,0x1000    # r2 = 0x00001000
  teq $1,$2          # trap happen
  ori $1,$0,0x2000    # r1 = 0x00002000
  tne $1,$2          # trap happen
  ori $1,$0,0x3000    # r1 = 0x00003000
  teqi $1,0x3000     # trap happen
  ori $1,$0,0x4000    # r1 = 0x00004000
  tnei $1,0x2000     # trap happen
  ori $1,$0,0x5000    # r1 = 0x00005000
  tge $1,$2          # trap happen
  ori $1,$0,0x6000    # r1 = 0x00006000
  tgei $1,0x4000     # trap happen
  ori $1,$0,0x7000    # r1 = 0x00007000
  tgeiu $1,0x7000    # trap happen
  ori $1,$0,0x8000    # r1 = 0x00008000
  tgeu $1,$2         # trap happen
  ori $1,$0,0x9000    # r1 = 0x00009000
  tlt $1,$2          # not trap
  ori $1,$0,0xa000    # r1 = 0x0000a000
  tlti $1,0x9000     # not trap
  ori $1,$0,0xb000    # r1 = 0x0000b000
  tltiu $1,0xb000    # trap happen ecause $1=0xb000 < 0xffffb000
  ori $1,$0,0xc000    # r1 = 0x0000c000
  tltu $2,$1         # trap happen
  ori $1,$0,0xd000    # r1 = 0x0000d000
```

```
_loop:
   j _loop
   nop
```

在自陷异常处理例程中将 EPC 的值加 4，是因为自陷异常会将其自身的地址保存到 EPC 寄存器中，如果直接使用 eret 返回，那么又会回到自陷指令处，从而导致再次执行自陷指令，所以在 eret 指令之前，将 EPC 的值加 4，这样就可以返回自陷指令的下一条指令处继续执行。

程序预期执行效果如程序中注释所示，ModelSim 仿真如下，观察 r1 的变化可知，OpenMIPS 正确实现了自陷指令。



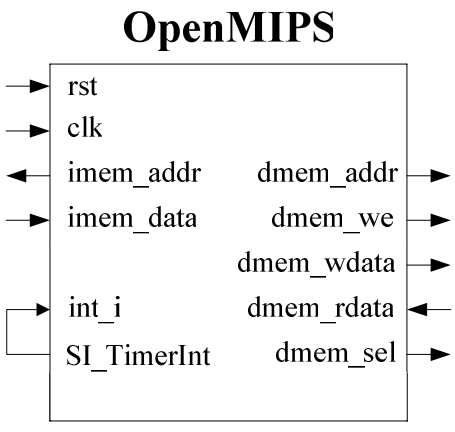


测试程序 3（位于 asm_test/Day10_3 文件夹下）

将 OpenMIPS 输出的时钟中断信号连接到其中断输入，从而产生中断信号，修改 OpenMIPS_min_sopc.vhd 如下：

```
……
interrupt <= "00000" & timerint & '0' & '0';

OpenMIPS0 : OpenMIPS port map (clk, rst, imem_addr, imem_data, dmem_addr,
                               dmem_we, dmem_wdata, dmem_rdata, dmem_sel,
                               interrupt, timerint);
……
```

实际效果如下所示：



测试程序如下：

```
.org 0x0
.set noat
```

```
   .set noreorder
   .set nomacro
   .global _start
_start:
   ori $1,$0,0x100     # r1 = 0x100
   jr $1              # 跳转到 0x100 处
   nop

   .org 0x20          # 中断处理例程入口
   addi $2,$2,0x1      # r2 加 1
   ori $1,$0,100
   mtc0 $1,$11,0x0     # 设置 compare 寄存器的值为 100
   eret               # 中断返回
   nop

   .org 0x100
   ori $2,$0,0x0
   ori $1,$0,100
   mtc0 $1,$11,0x0     # 设置 compare 寄存器的值为 100
   lui $1,0x1000
   ori $1,$1,0x401
   mtc0 $1,$12,0x0     # 存储 0x10000401 到 status 寄存器
                       # 作用是设置 IE 为 1，IM 为 00000100，从而使能时钟中断

_loop:
   j _loop
   nop
```

其中设置 compare 的值为 100，这样每经过 100 个时钟周期就会执行一次时钟中断处理例程，其中将 r2 寄存器的加 1。ModelSim 仿真如下，观察 r2 的变化，可知 OpenMIPS 正确实现了时钟中断。