**北京科技大学计算机与通信工程学院**

**代码阅读报告**

**课程名称**：计算机组成原理课程设计

**学生姓名**： 刘梓瑄

**专业**： 计算机科学与技术

**班级**： 计172班

**学号**： 41724039

**指导教师**： 刘宏岚 张磊

**报告成绩**：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**实验地点**： 逸夫楼304

**实验时间**：2019年11月16日----2019年12月20日

**北京科技大学实验报告**

学院：计通学院 专业：计算机科学与技术 班级：计172班

姓名：刘梓瑄 学号：41724039 实验日期：2019年12月15日

**一、课设目的与要求**

* 学会处理器的设计方法：单周期或多周期或流水线。
* 掌握处理器设计过程中指令扩展的方法。
* 能够运用现代工具独立实现一个完整的处理器。
* 了解处理器功能测试的方法：仿真测试及FPGA测试。
* 计算机系统观的建立，对所设计的处理器在整个计算机系统中的位置有所了解。

**二、实验设备（环境）及要求**

龙芯实验箱一体化实验平台。

OS：Win10 64位

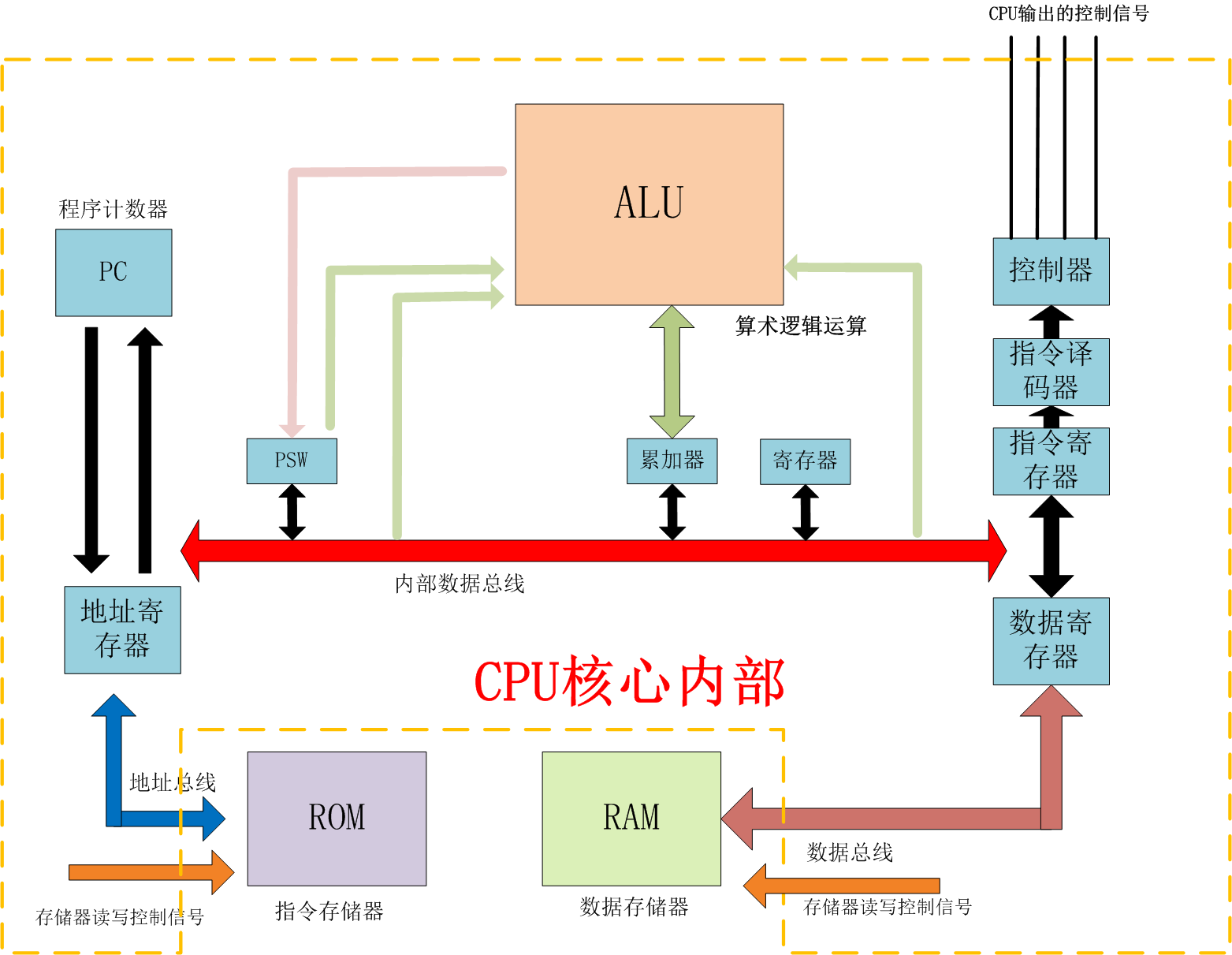
Software：Vivado2018.3开发工具

VirtualBox虚拟机+Ubuntu16.04.6

**三、设计过程与结果分析**

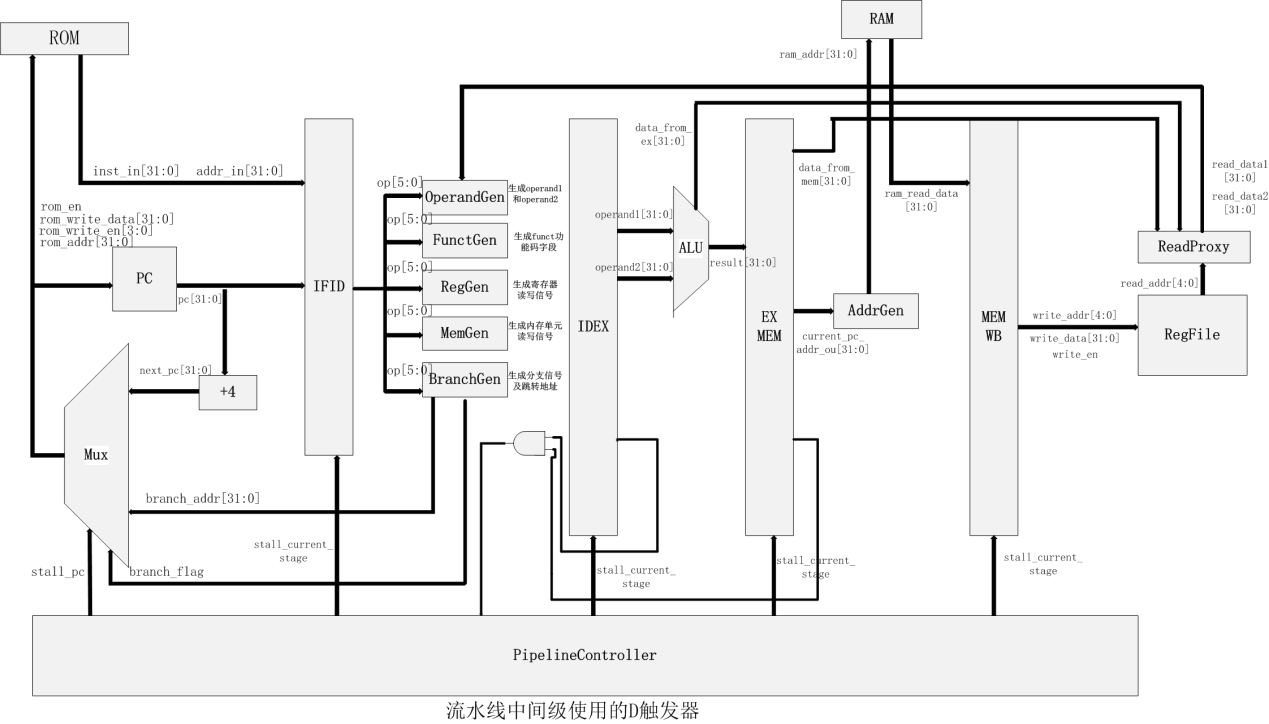
**1TinyMIPS总体结构框图**（自己动手重新画图，推荐画图工具：gliffy或draw.io或者其它）

所使用画图工具：Microsoft Visio 2017

****

**2数据通路图**（自己动手重新画图，需要将每个信号的位数在图上有所显示）

所使用画图工具：Microsoft Visio 2017

****

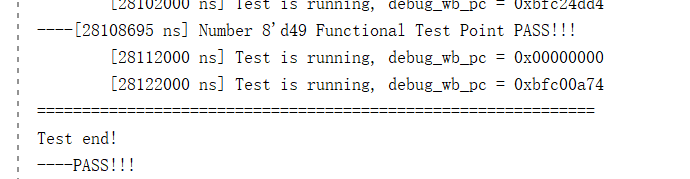
如数据通路图所示，现对TinyMIPS微处理器做如下说明：

TinyMIPS是一款32位的微处理，支持五级流水线结构，分别对应为五个功能模块，分别是：IF（取指）、ID（译码）、EX（执行）、MEM（访存）、WB(写回）。而这五个功能模块也对应在TinyMIPS中一条指令执行的完整操作所包含的五个步骤：

1. IF级负责从存储器（内存或缓存，在TinyMIPS中是只读存储器ROM）中取出指令。
2. ID级负责对指令进行译码操作，并完成从寄存器堆中读取数据，生成执行指令所需要的操作数和众控制信号。
3. EX级负责根据ID级的译码结果执行对应的ALU（算术逻辑运算）操作。
4. MEM级负责处理可能产生访存请求的指令，并向存储器（内存或缓存）发送控制信号。
5. WB级负责将指令的执行结果写回寄存器堆。

**3．MIPS指令集的设计过程**

原本的TinyMIPS微处理中只支持22条指令：ADDU，ADDIU，SUBU，SLT，SLTU，AND，LUI，OR，XOR，SLL，SLLV，SRAV，SRLV，BEQ，BNE，JAL，JALR，LB，LBU，LW，SB，SW。它们足以支持一个简单，且功能完备的CPU的运行。在经过一个月的学习之后，我按学长公布的CDE功能测试环境（即CDE49）进行了指令扩展操作，扩展了17条指令（NOR，SRA，SRL，SLTI，SLTIU，ANDI，ORI，XORI，BGEZ，BGTZ，BLEZ，BLTZ，J，JR，LH，LHU，SH）。俗话说“纸上得来终觉浅，绝知此事要躬行”前后花费两天时间，也得到了学长不少的指导帮助，我成功完成了这17条指令的扩展操作，最终成功通过了CDE环境中的49个功能测试点，才正真理解了一个完整CPU的设计架构和一条指令的完整执行流程与思路，也算是始于此处，开始对计算机硬件设计产生兴趣，通过49个功能测试点的截图如下图所示：



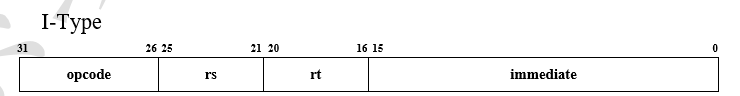
在接下来的部分中我会对全部17条指令的实现过程做一个详细的解释说明。

在开始之前，我先解释一下自己对MIP指令集的理解。

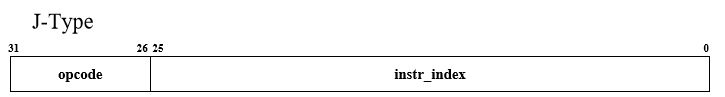
1. TinyMIPS微处理器可以处理的数据格式有以下几种：
2. 比特（bit,b）
3. 字节（Byte,8bits,B）
4. 半字（HalfWord,16bits,H）
5. 字（Word,32bits,W）
6. TinyMips微处理器内部提供了32个通用寄存器，r0~r31,r0寄存器具有特殊含义：即它存储的值永远为0。（对r0做赋值操作永远也不能改变它的值）寄存器的定义在路径： CDE49\_2\soft\func\include中进行了宏定义。
7. TinyMIPS采用的是小端存储模式，即第0字节永远是最低有效字节，第0比特永远是最低有效比特。
8. 指令格式：首先需要明确的是——所有的指令长度均为32比特

所有指令可概括为以下三种类型：

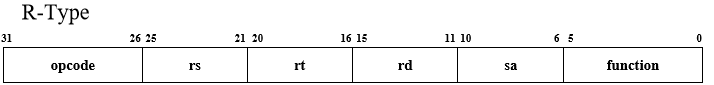
1. 立即数型（I-Type）



1. 跳转型（J—Type）

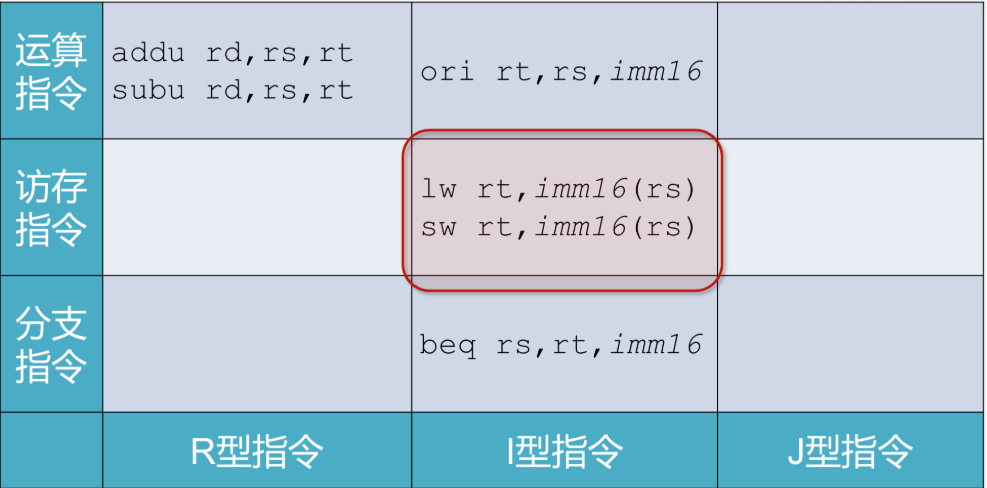


1. 寄存器型（R—Type）



1. 按指令执行的不同维度对指令进行分类

因为对于MIPS这样的指令系统来说，其运算指令是不能直接访问存储器的。因此，我们需要设计单独的访存指令，用来完成寄存器和存储器之间的数据传输过程。用一个简单的图对不同维度的指令进行说明：



这张图非常清晰的说明了，一条MIPS指令同时具有两个维度，第一个维度（运算、访存、分支）决定了指令的性质，它决定当前指令执行的是具体哪一种操作，程序计数器PC的值是否需要修改，是否要进行存储器的读写操作，在EX级要执行什么逻辑运算等等。而第二个维度（R型，I型，J型）指明的则是指令格式如何确定执行这条指令的所需操作数的来源等信息。

我在指令过程中也会以这两个维度为讨论基础，详细说明每一条指令的实现过程。

注意：需要特别说明的是，除了在CDE中进行全部49个功能测试点的测试外，我在调试过程中编写了core模块的仿真测试文件ID\_tb.V用于在拓展完每一条指令后初步确认当前指令的功能是否得到了正确实现，因为共用了一个core模块，我把代码在这里统一列出，避免报告中出现太多的冗余代码。此外，我没有使用虚拟机做为反汇编工具，每一条指令的机器代码均来自我的手动反汇编过程。

ID\_tb.V文件内容如下：

`timescale 1ns / 1ps

module ID\_tb();

reg clk, rst;

integer tick;

initial begin

clk=0;

rst=1;

#7 rst=0;

end

always begin

#5 clk=~clk;

end

wire rom\_en;

wire[`MEM\_SEL\_BUS] rom\_write\_en;

wire[`ADDR\_BUS] rom\_addr;

wire[`DATA\_BUS] rom\_read\_data, rom\_write\_data;

wire ram\_en;

wire[`MEM\_SEL\_BUS] ram\_write\_en;

wire[`ADDR\_BUS] ram\_addr;

wire[`DATA\_BUS] ram\_read\_data, ram\_write\_data;

wire debug\_reg\_write\_en;

wire[`REG\_ADDR\_BUS] debug\_reg\_write\_addr;

wire[`DATA\_BUS] debug\_reg\_write\_data;

wire[`ADDR\_BUS] debug\_pc\_addr;

Core core(

.clk (clk),

.rst (rst),

.stall (0),

.rom\_en (rom\_en),

.rom\_write\_en (rom\_write\_en),

.rom\_addr (rom\_addr),

.rom\_read\_data (rom\_read\_data),

.rom\_write\_data (rom\_write\_data),

.ram\_en (ram\_en),

.ram\_write\_en (ram\_write\_en),

.ram\_addr (ram\_addr),

.ram\_read\_data (ram\_read\_data),

.ram\_write\_data (ram\_write\_data),

.debug\_reg\_write\_en (debug\_reg\_write\_en),

.debug\_reg\_write\_addr (debug\_reg\_write\_addr),

.debug\_reg\_write\_data (debug\_reg\_write\_data),

.debug\_pc\_addr (debug\_pc\_addr)

);

RAM ram(

.clk (clk),

.ram\_en (ram\_en),

.ram\_write\_en (ram\_write\_en),

.ram\_addr (ram\_addr),

.ram\_write\_data (ram\_write\_data),

.ram\_read\_data (ram\_read\_data)

);

ROM rom(

.clk (clk),

.rom\_en (rom\_en),

.rom\_write\_en (rom\_write\_en),

.rom\_addr (rom\_addr),

.rom\_write\_data (rom\_write\_data),

.rom\_read\_data (rom\_read\_data)

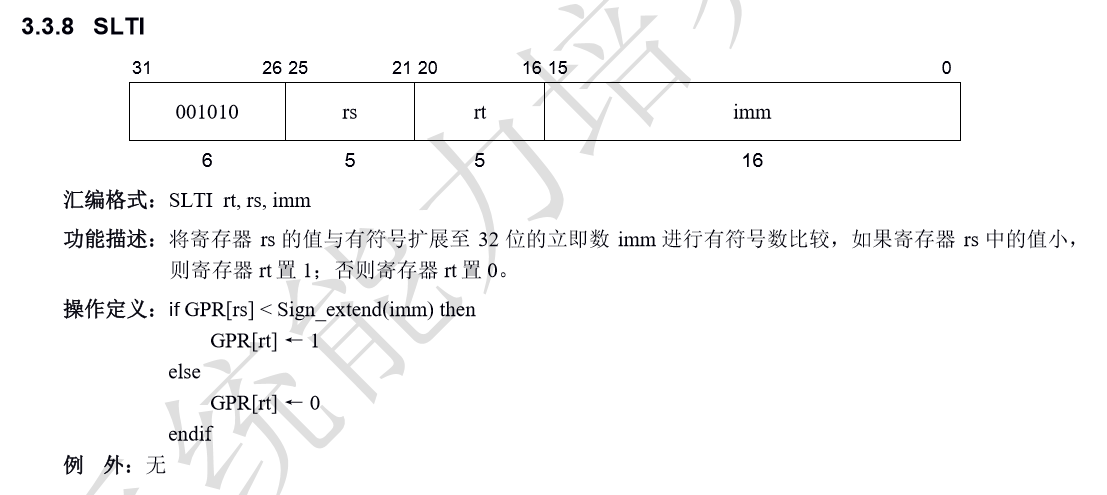
);

endmodule

**4．17条扩展指令的实现过程**

**4.1 SLTI**

1）指令格式



I型指令，opcode段为001010，逻辑运算类指令。

1. 操作数1来自rs寄存器。
2. 操作数2来自imm立即数有符号扩展至32位操作数。
3. 运算结果需写回rt寄存器。

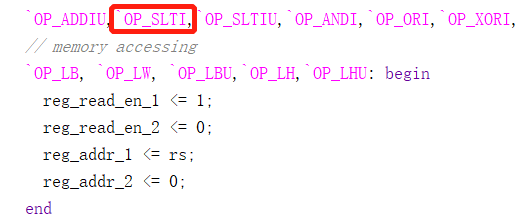
2）实现代码：

a.在opcode.V中添加OP\_SLTI字段

C:\Users\Admin\AppData\Local\Temp\1576487797(1).png//对应SLTI指令的opcode字段，用于唯一识别该条指令。

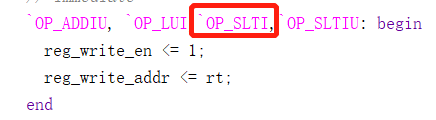
b.在RegGen.V中实现寄存器堆的读写定义。

//操作数1来自读出寄存器rs的值



//1号寄存器读使能有效，且1号寄存器的地址是rs寄存器

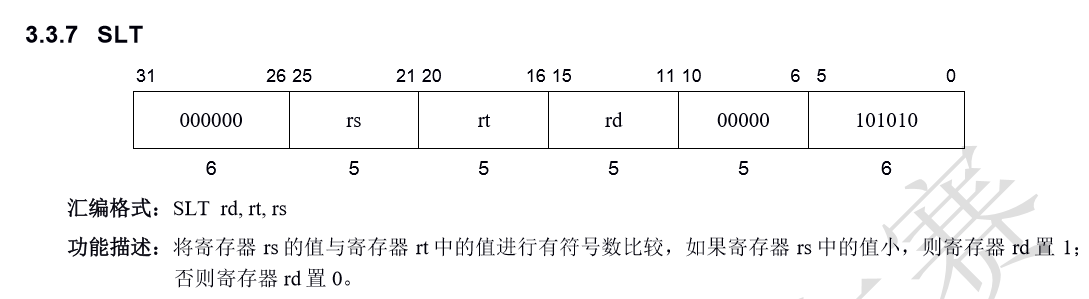
//最后得到的运算结果需要写回rt寄存器



//寄存器写使能有效，写回的寄存器地址为rt寄存器。

1. 在functGen.V中定义操作码的生成情况

C:\Users\Admin\AppData\Local\Temp\1576488201(1).png

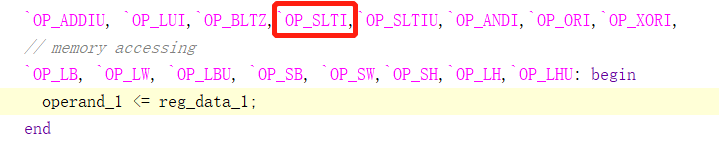


//SLT指令也是对两个操作数进行有符号比较，其指令格式中funct字段为//101010（在funct.V宏定义中已经实现）

//现把这两种相同的逻辑运算操作归并为一类

1. 在operandGen.V中完成两个操作数的生成操作

操作数1来自从rs寄存器中直接读出的数

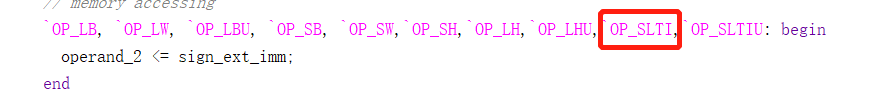


操作数2来自对立即数imm做符号扩展，扩展至32位数

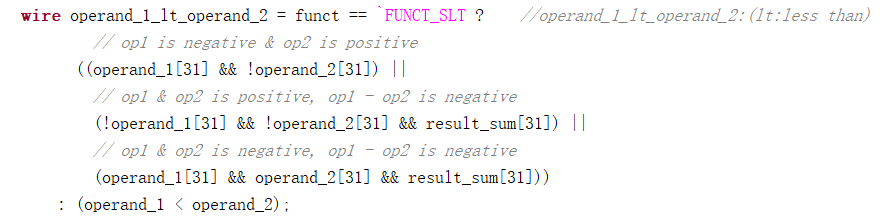
在最初的CPU版本中已经实现了操作数的符号扩展功能，如下图所示：

C:\Users\Admin\AppData\Local\Temp\1576491184(1).png

//实现在16位立即数的高位补上该立即数的符号



1. 在EX.V中确定该指令的执行情况:



C:\Users\Admin\AppData\Local\Temp\1576491314(1).png

//实现的是有符号数比较，operand1<operand2时，result结果由31个0和结果1拼接成32位结果数。

3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0xFFFF //将8号寄存器赋值-1（以补码形式表示）

SLTI $9,$8,0x5678 //将8号寄存器中的值与立即数5678H进行比较，结果回送回9号寄存器

b.对照指令格式将这两条指令翻译成机器代码的格式

001001 00000 01000 1111 1111 1111 1111

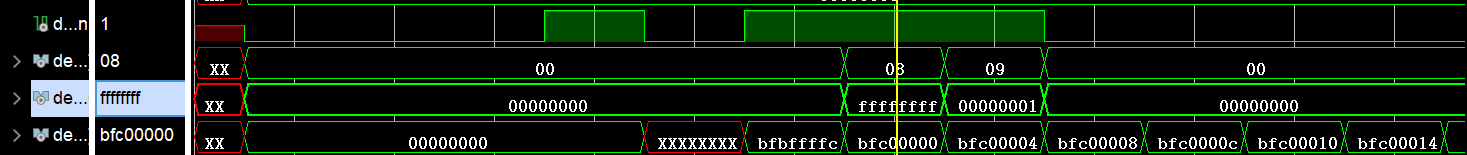
001010 01000 01001 0101 0110 0111 1000

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

24 08 FF FF

29 09 56 78

d.仿真波形及分析：



此刻将立即数-1H送往8号寄存器，符合设计要求。

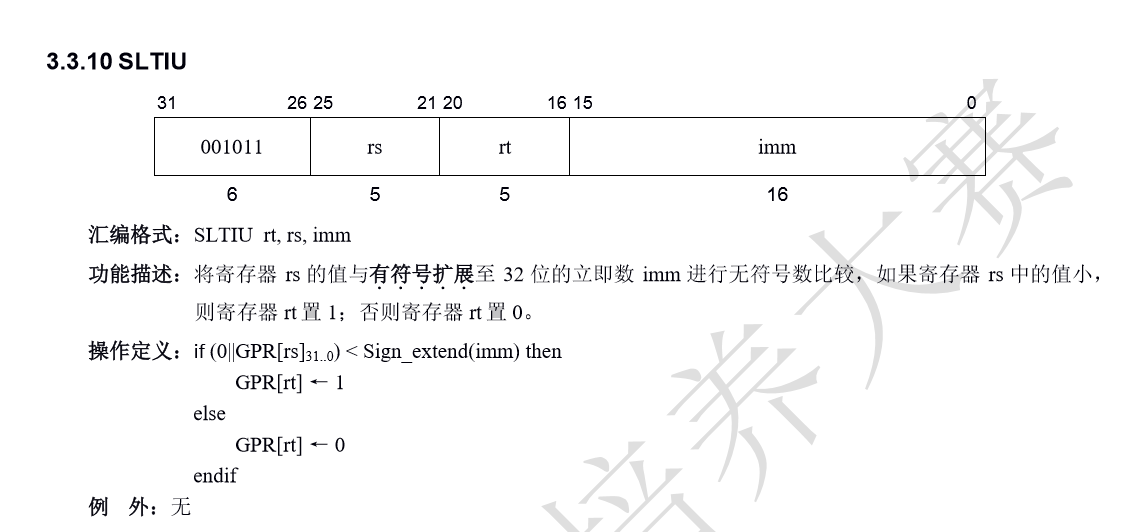


下一时刻比较操作已完成，（因-1H<5678H，做的是有符号比较），回送数值应为1。如图易知，9号寄存器被置1，符合设计要求。

综上所述，该条指令功能设计正确。

**4.2 SLTIU**

1）指令格式



I型指令，opcode段为001011，逻辑运算类指令。

a.操作数1来自rs寄存器

b.操作数2来自imm立即数有符号扩展至32位操作数。

c.运算结果需写回rt寄存器。

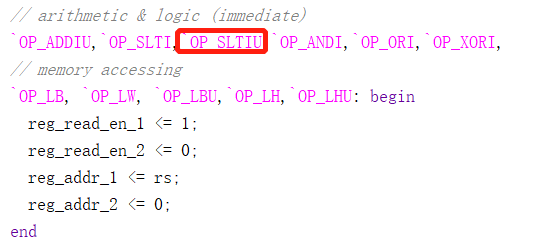
2）实现代码

a.在opcode.V中添加OP\_SLTIU字段

C:\Users\Admin\AppData\Local\Temp\1576493106(1).png//对应SLTIU指令的opcode字段

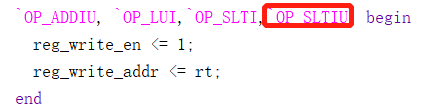
b.在RegGen.V中实现寄存器堆的读写定义。

操作数1来自读出寄存器rs的值



//1号寄存器读使能有效，且1号寄存器的地址是rs寄存器

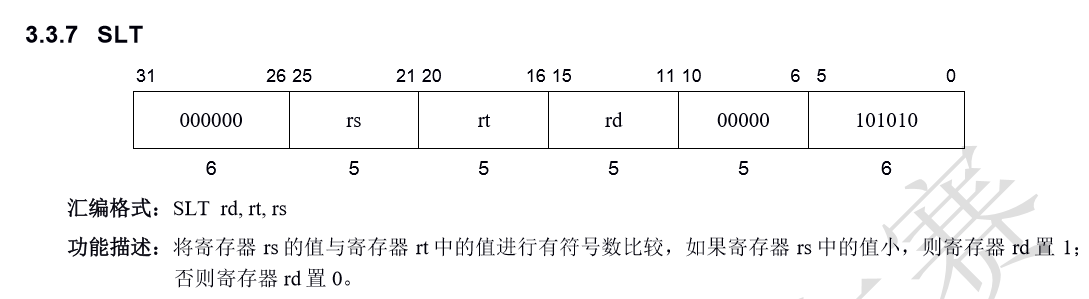
//最后得到的运算结果需要写回rt寄存器



//寄存器写使能有效，写回的寄存器地址为rt寄存器。

c.在functGen.V中定义操作码的生成情况

C:\Users\Admin\AppData\Local\Temp\1576493212(1).png

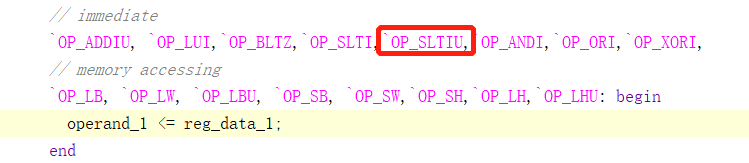


//SLT指令也是两个操作数进行有大小比较，其指令格式中funct字段为//101010（在funct.V宏定义中已经实现）

//现把这两种相同的逻辑运算操作归并为一类

d.在operandGen.V中完成两个操作数的生成操作

操作数1来自从rs寄存器中直接读出的数

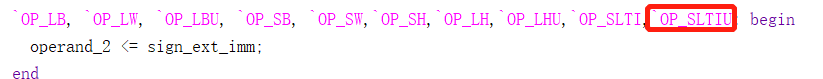


操作数2来自对立即数imm做符号扩展，扩展至32位数

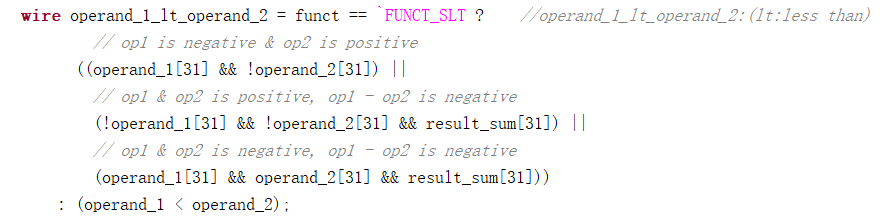
在最初的CPU版本中已经实现了操作数的符号扩展功能，如下图所示：

C:\Users\Admin\AppData\Local\Temp\1576491184(1).png

//实现在16位立即数的高位补上该立即数的符号



e.在EX.V中确定该指令的执行情况



由此图已知，这一比较赋值操作同时覆盖了有符号比较和无符号比较，故而不需添加其他操作。

C:\Users\Admin\AppData\Local\Temp\1576491314(1).png

//实现的是无符号数比较，operand1<operand2时，result结果由31个0和结果1拼接成32位结果数。

3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0x1234 //将8号寄存器赋值1234H

SLTIU $9,$8,0x5678 //将8号寄存器中的值与立即数5678H进行无符号数比较，结果回送回9号寄存器

b.对照指令格式将这两条指令翻译成机器代码的格式

001001 00000 01000 0001 0010 0011 0100

001011 01000 01001 0101 0110 0111 1000

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

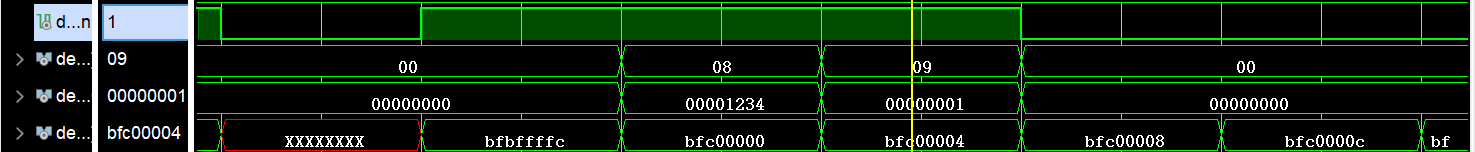
24 08 12 34

2D 09 56 78

d.仿真波形及分析：



此刻将立即数1234H送往8号寄存器，符合设计要求。

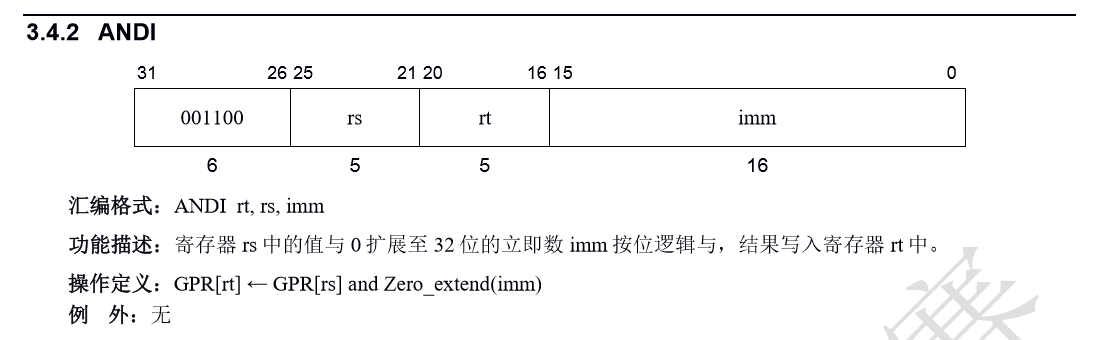


下一时刻比较操作已完成，（因1234H<5678H，做的是无符号数比较），回送数值应为1。如图易知，9号寄存器被置1，符合设计要求。

综上所述，该条指令功能设计正确。

**4.3 ANDI**

1）指令格式



I型指令，opcode段为001100，逻辑运算类指令。

a.操作数1来自rs寄存器

b.操作数2来自imm立即数零扩展至32位操作数。

c.运算结果需写回rt寄存器。

2）实现代码

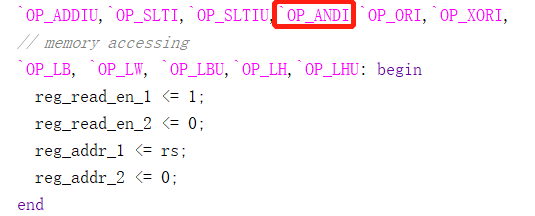
a.在opcode.V中添加OP\_SLTI字段

C:\Users\Admin\AppData\Local\Temp\1576495568(1).png

//对应ANDI指令的opcode字段，用于唯一识别该条指令。

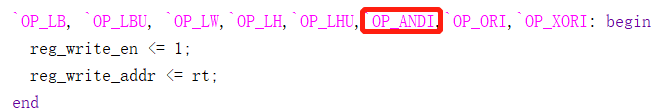
b.在RegGen.V中实现寄存器堆的读写定义。

操作数1来自读出寄存器rs的值



//1号寄存器读使能有效，且1号寄存器的地址是rs寄存器

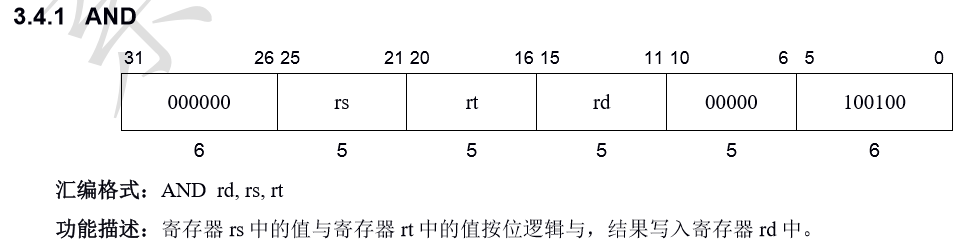
//最后得到的运算结果需要写回rt寄存器



//寄存器写使能有效, 写回的寄存器地址为rt寄存器。

c.在functGen.V中定义操作码的生成情况

C:\Users\Admin\AppData\Local\Temp\1576496231(1).png

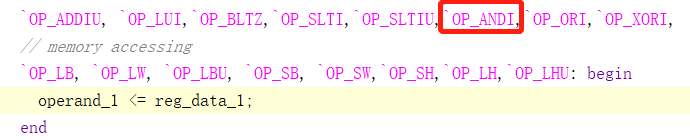


//AND指令也是两个操作数进行按位逻辑与操作，其指令格式中funct字为//100100（在funct.V宏定义中已经实现）

//现把这两种相同的逻辑运算操作归并为一类

d.在operandGen.V中完成两个操作数的生成操作

//操作数1来自从rs寄存器中直接读出的数

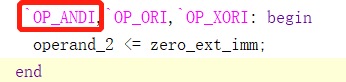


操作数2来自对立即数imm做零扩展，扩展至32位数

//在最初的CPU版本中还未实现了操作数的符号扩展功能，我自己编写了零扩展的情况，代码如下图所示：

C:\Users\Admin\AppData\Local\Temp\1576496384(1).png

//实现在16位立即数的高位补上零，将16位立即数扩展至32位的操作数



e.在EX.V中确定该指令的执行情况

C:\Users\Admin\AppData\Local\Temp\1576496507(1).png

//实现的是两操作数按位做逻辑与操作，逻辑运算符为：&

3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0x1234 //将8号寄存器赋值1234H

ANDI $9,$8,0x5678 //将8号寄存器中的值与立即数5678H进行按位逻辑与操作，结果回送回9号寄存器

b.对照指令格式将这两条指令翻译成机器代码的格式

001001 00000 01000 0001 0010 0011 0100

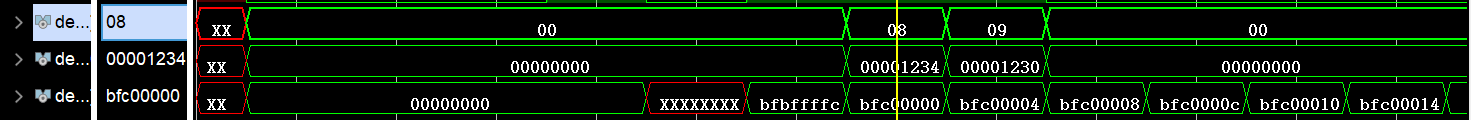
001100 01000 01001 0101 0110 0111 1000

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

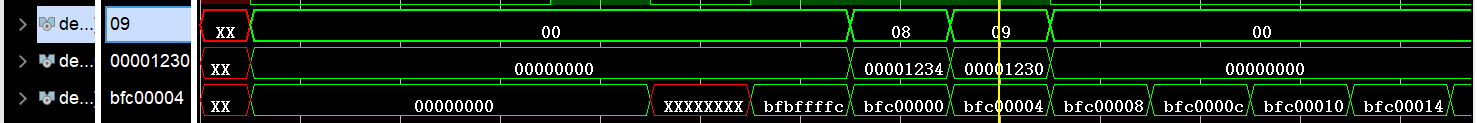
24 08 12 34

31 09 56 78

d.仿真波形及分析：



此刻将立即数1234H送往8号寄存器，符合设计要求。

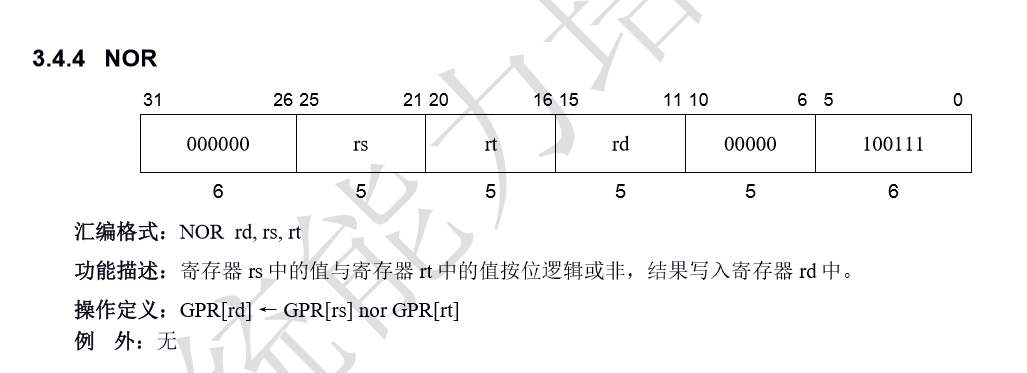


下一时刻按位与操作已完成，（因1234H与5678H做按位与操作后得到的结果应为1230H），回送数值应为1230H。如图易知，9号寄存器被置1230H，符合设计要求。

综上所述，该条指令功能设计正确。

**4.4 NOR**

1）指令格式



R型指令，opcode段为000000，funct段为100111，逻辑运算类指令。

a.操作数1来自rs寄存器。

b.操作数2来自rt寄存器。

c.运算结果需写回rd寄存器。

2）实现代码

a.在funct.V中添加OP\_NOR字段

C:\Users\Admin\AppData\Local\Temp\1576497272(1).png

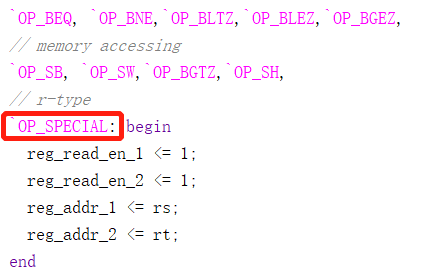
//对应ANDI指令的funct字段，用于唯一识别该条R型指令。

b.在RegGen.V中实现寄存器堆的读写定义。

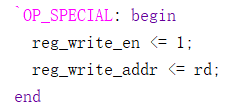
操作数1来自读出寄存器rs，操作数2来自读寄存器rt的值。

//1，2号寄存器读使能均有效，且1号寄存器的地址是rs寄存器，2号寄存器//的地址是rt寄存器

//最后得到的运算结果需要写回rd寄存器(opcode是000000的，均要归为//OP\_SPECIAL一类)



//寄存器写使能有效, 写回的寄存器地址为rd寄存器。



c.在functGen.V中定义操作码的生成情况

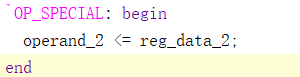
C:\Users\Admin\AppData\Local\Temp\1576498181(1).png

//funct字段直接由指令输入的funct\_in部分决定

d.在operandGen.V中完成两个操作数的生成操作

R型指令两个操作数均来自两个寄存器，把它们归在OP\_SPECIAL字段进行统一读操作与写操作，如下图所示





e.在EX.V中确定该指令的执行情况

C:\Users\Admin\AppData\Local\Temp\1576498495(1).png

//实现的是两操作数按位逻辑或操作后对结果取反，逻辑运算表达式如图所示。

3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0x1234 //将8号寄存器赋值1234H

ADDIU $9,$9,0x5678 //将8号寄存器赋值5678H

NOR $10,$8,$9 //将8号寄存器与9号寄存器中的值做NOR操作结果回送10号寄存器

b.对照指令格式将这两条指令翻译成机器代码的格式

001001 00000 01000 0001 0010 0011 0100

001001 00000 01001 0101 0110 0111 1000

000000 01000 01001 01010 00000 100111

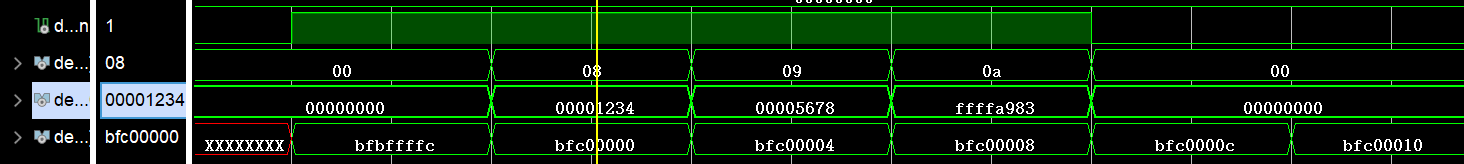
c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

24 08 12 34

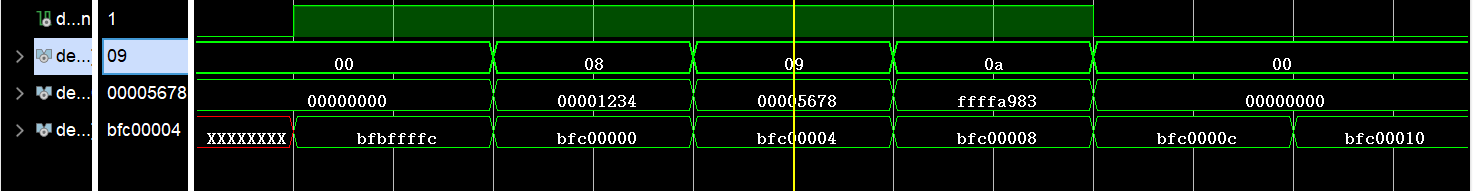
24 09 56 78

01 09 50 27

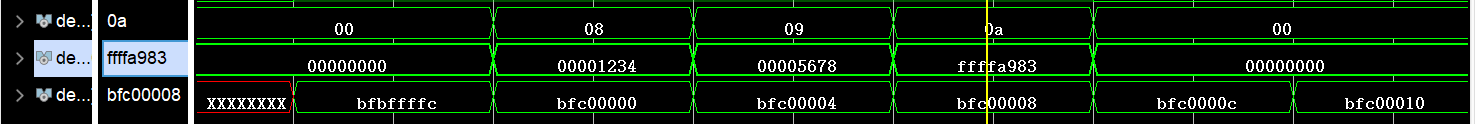
d.仿真波形及分析：



此刻将立即数1234H送往8号寄存器，符合设计要求。



下一时刻将立即数5678H送往9号寄存器，符合设计要求。

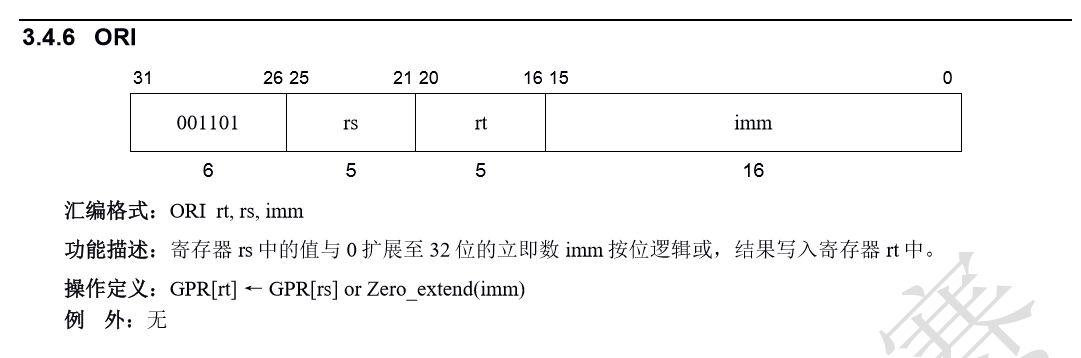


下一时刻NOR操作已完成，（因1234H与5678H做NOR操作:按位或操作后再按位取反后得到的结果应为ffffa983H。如图易知，10号寄存器被置ffffa983H，符合设计要求。

综上所述，该条指令功能设计正确。

**4.5 ORI**

1）指令格式



I型指令，opcode段为001101，逻辑运算类指令。

a.操作数1来自rs寄存器。

b.操作数2来自imm立即数零扩展至32位操作数。

c.运算结果需写回rt寄存器。

2）实现代码

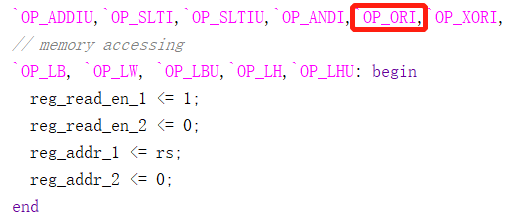
a.在opcode.V中添加OP\_ORI字段

C:\Users\Admin\AppData\Local\Temp\1576499171(1).png

//对应ORI指令的opcode字段，用于唯一识别该条指令。

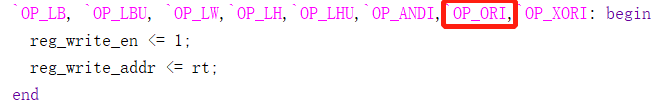
b.在RegGen.V中实现寄存器堆的读写定义。

操作数1来自读出寄存器rs的值



//1号寄存器读使能有效，且1号寄存器的地址是rs寄存器

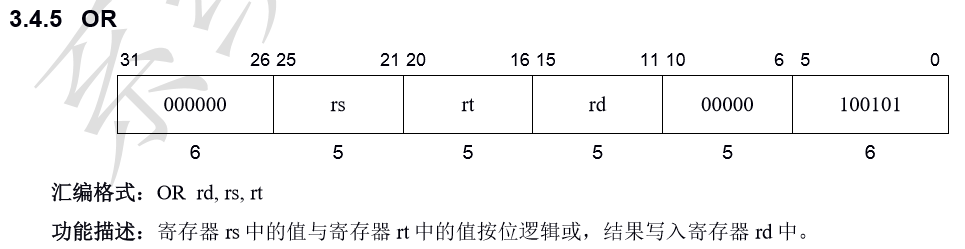
//最后得到的运算结果需要写回rt寄存器



//寄存器写使能有效, 写回的寄存器地址为rt寄存器。

c.在functGen.V中定义操作码的生成情况

C:\Users\Admin\AppData\Local\Temp\1576499399(1).png

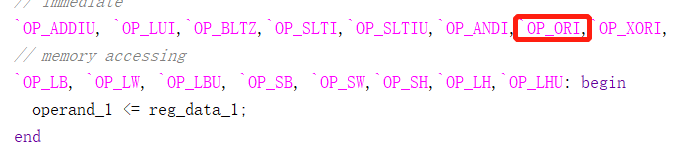


//OR指令也是两个操作数进行按位逻辑或操作，其指令格式中funct字为//100101（在funct.V宏定义中已经实现）

//现把这两种相同的逻辑运算操作归并为一类

d.在operandGen.V中完成两个操作数的生成操作

操作数1来自从rs寄存器中直接读出的数

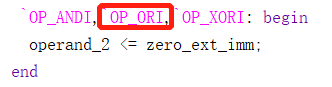


操作数2来自对立即数imm做零扩展，扩展至32位数

零扩展的代码如下图所示：

C:\Users\Admin\AppData\Local\Temp\1576496384(1).png

//实现在16位立即数的高位补上零，将16位立即数扩展至32位的操作数



e.在EX.V中确定该指令的执行情况

C:\Users\Admin\AppData\Local\Temp\1576499558(1).png

//实现的是两操作数按位做逻辑或操作，逻辑运算符为：|

3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0x1234 //将8号寄存器赋值1234H

ORI $9,$8,0x5678 //将8号寄存器中的值与立即数5678H进行按位逻辑或操作，结果回送回9号寄存器

b.对照指令格式将这两条指令翻译成机器代码的格式

001001 00000 01000 0001 0010 0011 0100

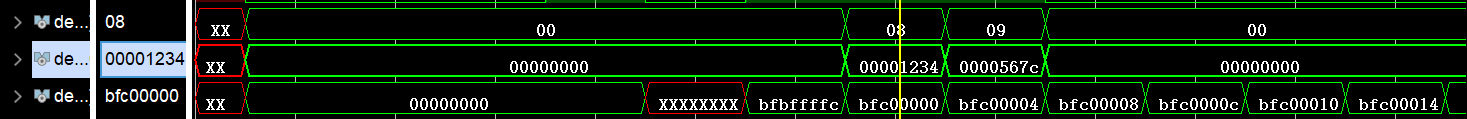
001101 01000 01001 0101 0110 0111 1000

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

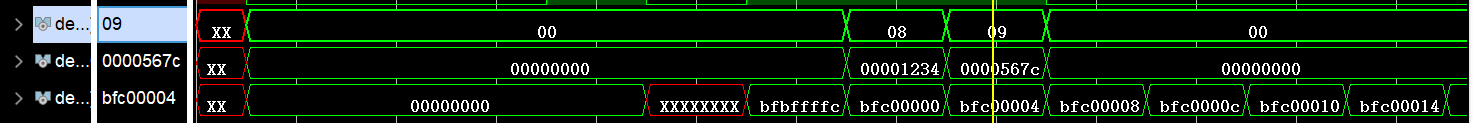
24 08 12 34

35 09 56 78

d.仿真波形及分析：



此刻将立即数1234H送往8号寄存器，符合设计要求。

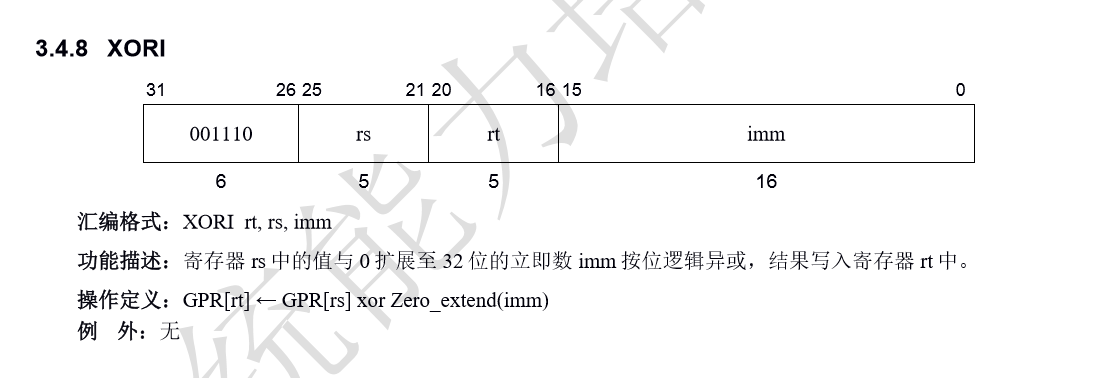


下一时刻按位或操作已完成，（因1234H与5678H做按位或操作后得到的结果应为567CH），回送数值应为567CH。如图易知，9号寄存器被置567CH，符合设计要求。

综上所述，该条指令功能设计正确。

**4.6 XORI**

1）指令格式



I型指令，opcode段为001110，逻辑运算类指令。

a.操作数1来自rs寄存器

b.操作数2来自imm立即数零扩展至32位操作数。

c.运算结果需写回rt寄存器。

2）实现代码

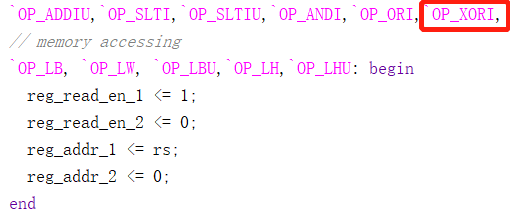
a.在opcode.V中添加OP\_XORI字段

C:\Users\Admin\AppData\Local\Temp\1576499887(1).png

//对应XORI指令的opcode字段，用于唯一识别该条指令。

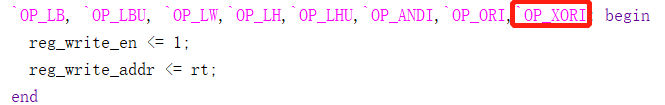
b.在RegGen.V中实现寄存器堆的读写定义。

操作数1来自读出寄存器rs的值



//1号寄存器读使能有效，且1号寄存器的地址是rs寄存器

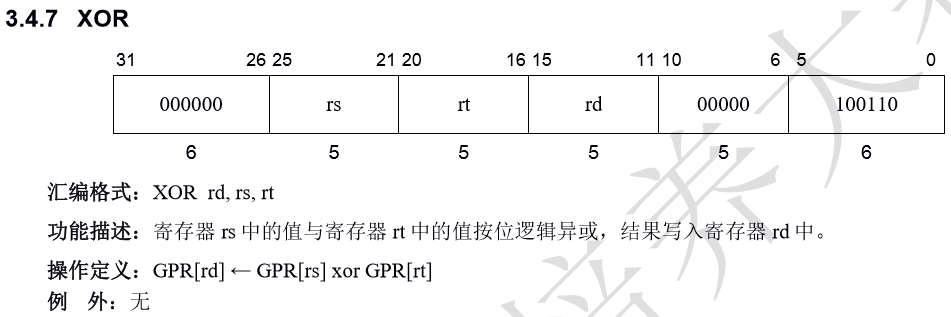
//最后得到的运算结果需要写回rt寄存器



//寄存器写使能有效, 写回的寄存器地址为rt寄存器。

c.在functGen.V中定义操作码的生成情况

C:\Users\Admin\AppData\Local\Temp\1576499993(1).png

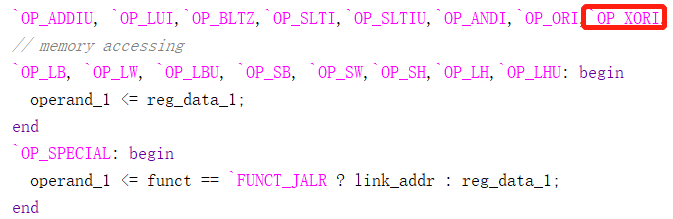


//XOR指令也是两个操作数进行按位逻辑异或操作，其指令格式中funct字//为100110（在funct.V宏定义中已经实现）

//现把这两种相同的逻辑运算操作归并为一类

d.在operandGen.V中完成两个操作数的生成操作

操作数1来自从rs寄存器中直接读出的数

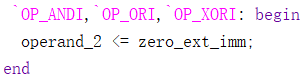


操作数2来自对立即数imm做零扩展，扩展至32位数

零扩展的代码如下图所示：

C:\Users\Admin\AppData\Local\Temp\1576496384(1).png

//实现在16位立即数的高位补上零，将16位立即数扩展至32位的操作数



e.在EX.V中确定该指令的执行情况

C:\Users\Admin\AppData\Local\Temp\1576500130(1).png

//实现的是两操作数按位做逻辑异或操作，逻辑运算符为：^

3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0x1234 //将8号寄存器赋值1234H

XORI $9,$8,0x5678 //将8号寄存器中的值与立即数5678H进行按位逻辑异或操作，结果回送回9号寄存器

b.对照指令格式将这两条指令翻译成机器代码的格式

001001 00000 01000 0001 0010 0011 0100

001110 01000 01001 0101 0110 0111 1000

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

24 08 12 34

39 09 56 78

d.仿真波形及分析：



此刻将立即数1234H送往8号寄存器，符合设计要求。

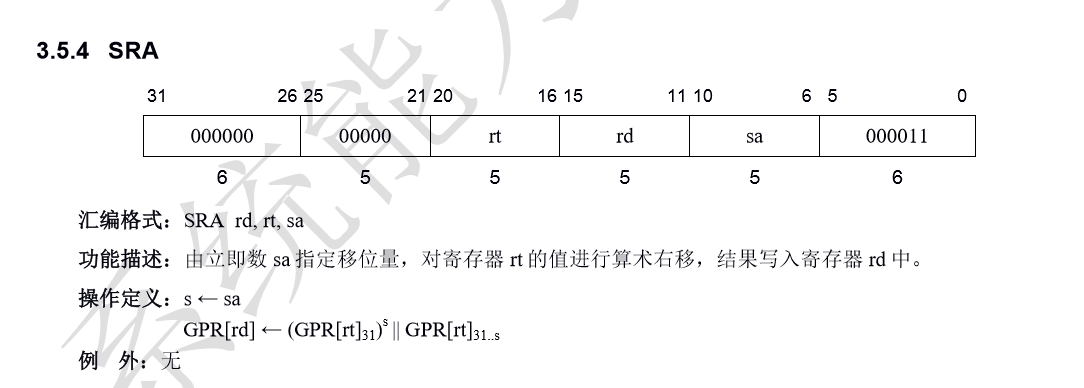


下一时刻按位或操作已完成，（因1234H与5678H做按位异或操作后得到的结果应为444CH），回送数值应为444CH。如图易知，9号寄存器被置444CH，符合设计要求。

综上所述，该条指令功能设计正确。

**4.7 SRA**

1）指令格式



R型指令，opcode段为000000，funct段为000011，逻辑运算类指令。

a.偏移量来自立即数sa。

b.操作数2来自rt寄存器。

c.运算结果需写回rd寄存器。

2）实现代码

a.在funct.V中添加OP\_SRA字段

C:\Users\Admin\AppData\Local\Temp\1576501279(1).png

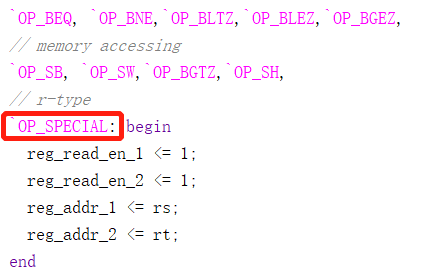
//对应SRA指令的funct字段，用于唯一识别该条R型指令。

b.在RegGen.V中实现寄存器堆的读写定义。

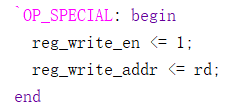
操作数1来自读出寄存器rs，操作数2来自读寄存器rt的值。

//1，2号寄存器读使能均有效,最后得到的运算结果需要写回rd寄存器

//opcode是000000的，均要归为OP\_SPECIAL一类)



//寄存器写使能有效, 写回的寄存器地址为rd寄存器。



c.在functGen.V中定义操作码的生成情况

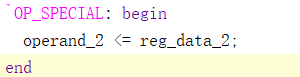
C:\Users\Admin\AppData\Local\Temp\1576498181(1).png

//funct字段直接由指令输入的funct\_in部分决定

d.在operandGen.V中完成两个操作数的生成操作

R型指令两个操作数均来自两个寄存器，把它们归在OP\_SPECIAL字段进行统一读操作与写操作，如下图所示





e.在EX.V中确定该指令的执行情况

C:\Users\Admin\AppData\Local\Temp\1576501307(1).png

//实现的是操作数2按指定值执行算术右移（高位补操作数2的符号位）.

3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0x8123H //将8号寄存器赋值8123H

SRA $9,$8,0004 //将8号寄存器内数算术右移四位，结果回送至9号寄存器。

b.对照指令格式将这两条指令翻译成机器代码的格式

001001 00000 01000 1000 0001 0010 0011

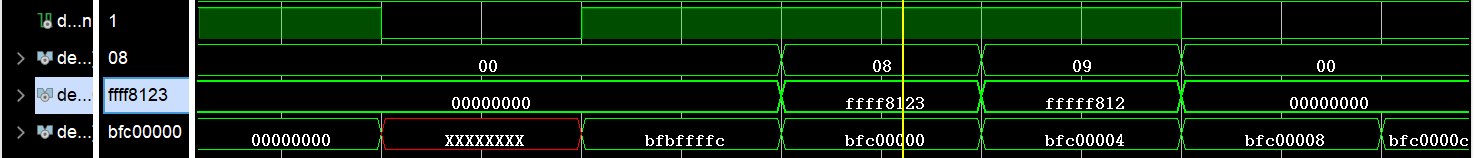
000000 00000 01000 01001 00100 000011

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

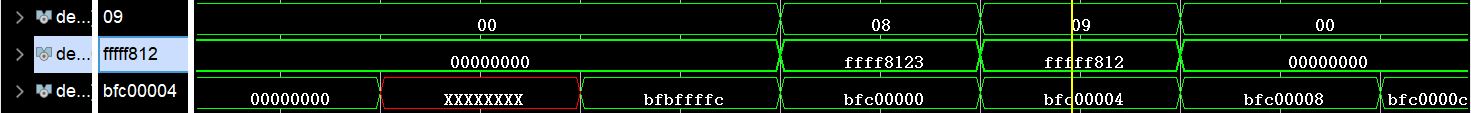
24 08 81 23

00 08 49 03

d.仿真波形及分析：



此刻将立即数8123H送往8号寄存器，符合设计要求。

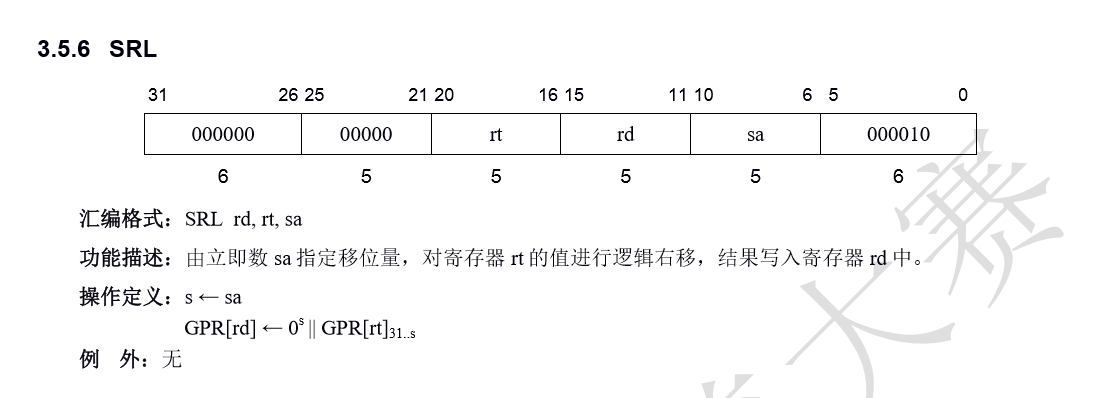


下一时刻逻辑右移操作已完成，（8123H逻辑4位后高位补1，得到的结果应为FFFFF812H）如图易知，9寄存器被置FFFFF812H，符合设计要求。

综上所述，该条指令功能设计正确。

**4.8 SRL**

1）指令格式



R型指令，opcode段为000000，funct段为000010，逻辑运算类指令。

a.偏移量来自立即数sa。

b.操作数2来自rt寄存器。

c.运算结果需写回rd寄存器。

2）实现代码

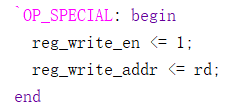
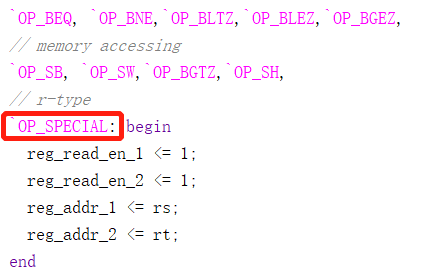
a.在funct.V中添加OP\_SRL字段

C:\Users\Admin\AppData\Local\Temp\1576500547(1).png

//对应SRL指令的funct字段，用于唯一识别该条R型指令。

b.在RegGen.V中实现寄存器堆的读写定义。

//操作数1来自读出寄存器rs，操作数2来自读寄存器rt的值。

//1，2号寄存器读使能均有效，最后得到的运算结果需要写回rd寄存器

c.在functGen.V中定义操作码的生成情况

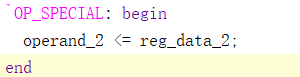
C:\Users\Admin\AppData\Local\Temp\1576498181(1).png

//funct字段直接由指令输入的funct\_in部分决定

d.在operandGen.V中完成两个操作数的生成操作

R型指令两个操作数均来自两个寄存器，把它们归在OP\_SPECIAL字段进行统一读操作与写操作，如下图所示





e.在EX.V中确定该指令的执行情况

C:\Users\Admin\AppData\Local\Temp\1576500623(1).png

//实现的是操作数2按指定值执行算术右移（高位补0）操作。

3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0x1234 //将8号寄存器赋值1234H

SRL $9,$8,0004 //将8号寄存器内数算术右移四位，结果回送至9号寄存器。

b.对照指令格式将这两条指令翻译成机器代码的格式

001001 00000 01000 0001 0010 0011 0100

000000 00000 01000 01001 00100 000010

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

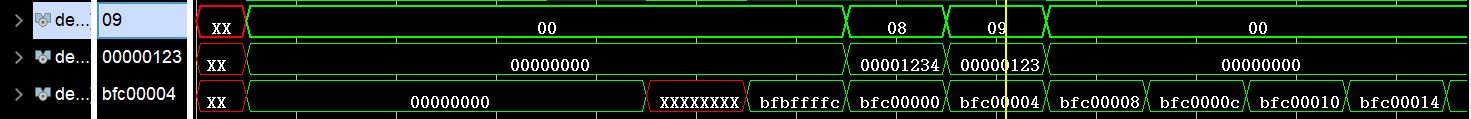
24 08 12 34

00 08 49 02

d.仿真波形及分析：



此刻将立即数1234H送往8号寄存器，符合设计要求。

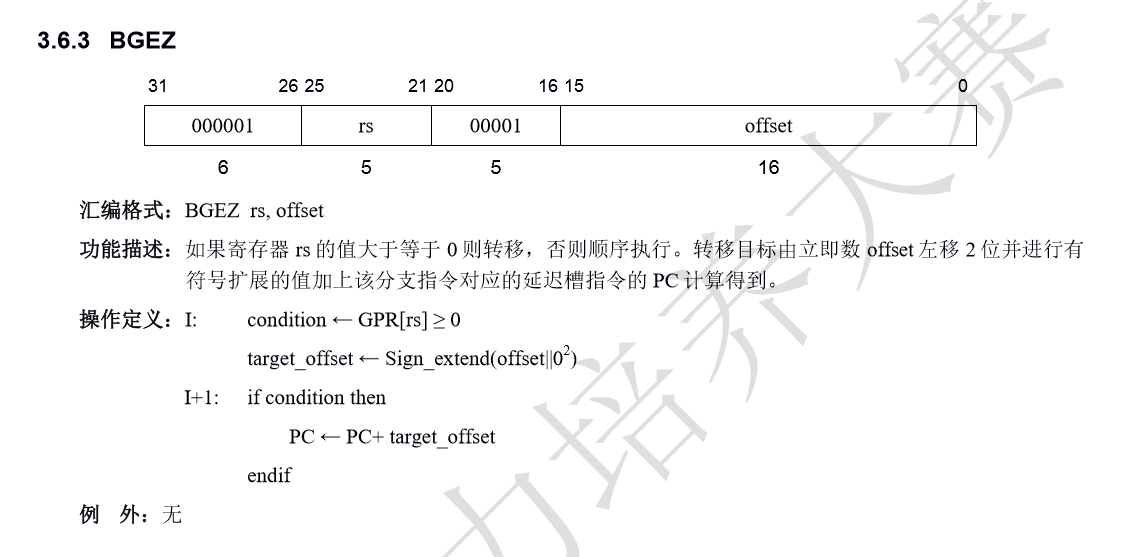


下一时刻逻辑右移操作已完成，（1234H逻辑右移4位后高位补0，得到的结果应为0123H）如图易知，9寄存器被置0123H，符合设计要求。

综上所述，该条指令功能设计正确。

**4.9 BGEZ**

1）指令格式



I型指令，opcode段为000001，跳转类指令。

a.操作数1来自rs寄存器

b.转移目标由立即数offset 左移两位并进行有符号拓展的值加上该分支指令对应的延迟槽指令的PC计算得到。

c.运算结果无需写回。

2）实现代码

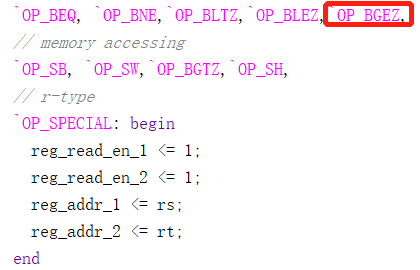
a.在opcode.V中添加OP\_BGEZ字段

C:\Users\Admin\AppData\Local\Temp\1576502081(1).png

//对应BGEZ指令的opcode字段，用于唯一识别该条指令。

b. 在RegGen中添加寄存器读使能

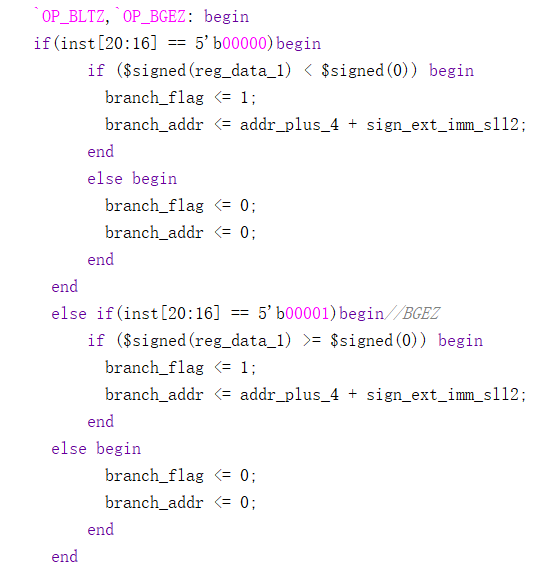
操作数1来自读出寄存器rs的值（注意rs和rt都需要读，虽然rt字段固定其不做为操作数但后来需要rt对应字段的值00001做一个与BLTZ指令的区分）



//1，2号寄存器读使能均有效，且1号寄存器的地址是rs寄存器，2号寄存器的地址是rt寄存器

//最后得到的运算结果无需写回

c.在BranchGen.V中编辑跳转条件（注意与BLTZ指令归并为一类进行处理。因为它们的opcode字段是相同的，需要按指令字段的第16到20位做一个区分）



3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0x1234 //给8号寄存器赋值1234H

BGEZ $8,label //8号寄存器值大于0则进行跳转

nop //延迟槽空指令

ADDU $10,$8,&8 //未成功跳转时执行该指令

label:

ADDU $10,$10,$10 //若成功跳转，10号寄存器值应为0

b.对照指令格式将这两条指令翻译成机器代码的格式

0010 0100 0000 1000 0001 0010 0011 0100

0000 0101 0000 0001 0000 0000 0000 0010

0000 0000 0000 0000 0000 0000 0000 0000

000000 01000 01000 01010 00000 100001

000000 01010 01010 01010 00000 100001

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

24 08 12 34

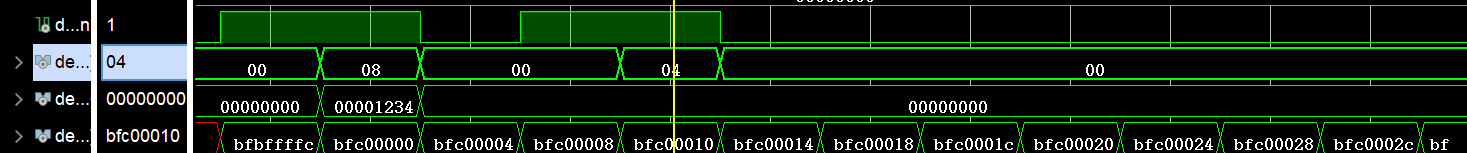
05 01 00 02

00 00 00 00

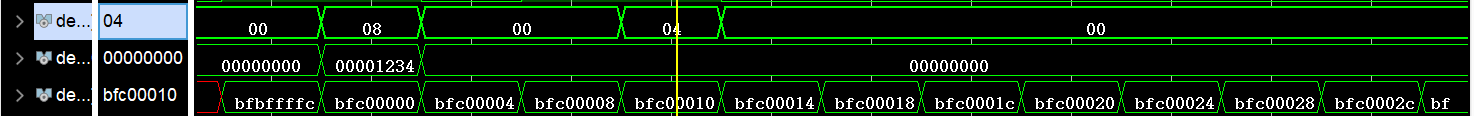
01 08 50 21

01 4a 50 21

d.仿真波形及分析：



此刻将立即数1234H送往8号寄存器，符合设计要求。

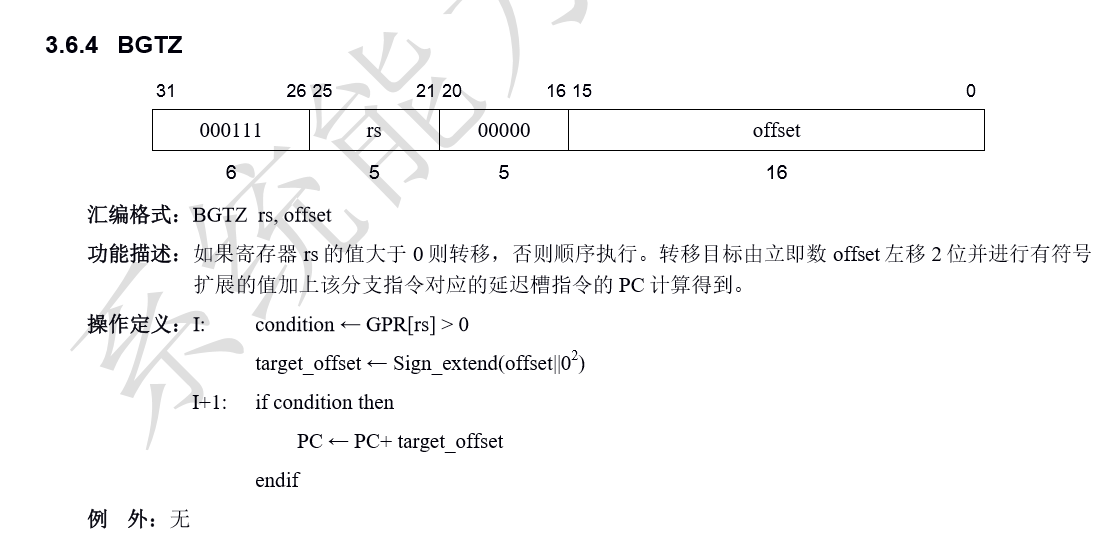


延迟槽指令（nop）执行之后发生跳转，（因1234H大于0，符合跳转条件）程序计数器值PC由bfc00008变为bfc00010，证明发生跳转（PC差值不再是4，且没有顺序执行对10号寄存器的赋值操作）符合设计要求。

综上所述，该条指令功能设计正确。

**4.10 BGTZ**

1）指令格式



I型指令，opcode段为000111，跳转类指令。

a.操作数1来自rs寄存器

b.转移目标由立即数offset 左移两位并进行有符号拓展的值加上该分支指令对应的延迟槽指令的PC计算得到。

c.运算结果无需写回。

2）实现代码

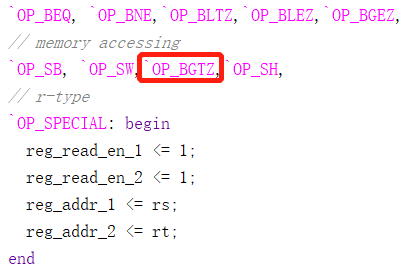
a.在opcode.V中添加OP\_BGTZ字段

C:\Users\Admin\AppData\Local\Temp\1576503924(1).png

//对应BGTZ指令的opcode字段，用于唯一识别该条指令。

b. 在RegGen中添加寄存器读使能

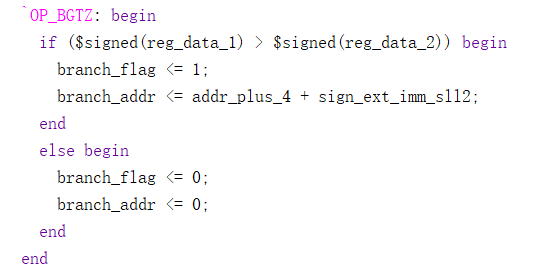
操作数1来自读出寄存器rs的值（注意rs和rt都需要读，另一个操作数将0读入reg\_data\_2中，可直接用于之后的比较操作。）



//1，2号寄存器读使能均有效，且1号寄存器的地址是rs寄存器，2号寄存器的地址是rt寄存器

//最后得到的运算结果无需写回

c.在BranchGen.V中编辑跳转条件（做的是有符号比较）



3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0x1234 //给8号寄存器赋值1234H

BGTZ $8,label //8号寄存器值大于0则进行跳转

nop //延迟槽空指令

ADDU $10,$8,&8 //未成功跳转时执行该指令

label:

ADDU $10,$10,$10 //若成功跳转，10号寄存器值应为0

b.对照指令格式将这两条指令翻译成机器代码的格式

0010 0100 0000 1000 0001 0010 0011 0100

0001 1101 0000 0000 0000 0000 0000 0002

0000 0000 0000 0000 0000 0000 0000 0000

000000 01000 01000 01010 00000 100001

000000 01010 01010 01010 00000 100001

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

24 08 12 34

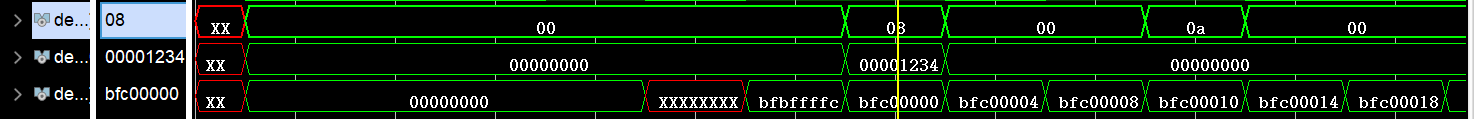
1d 00 00 02

00 00 00 00

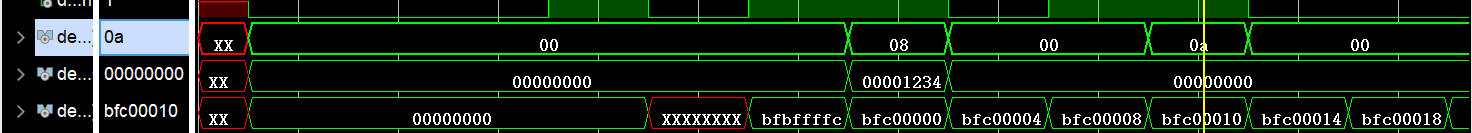
01 08 50 21

01 4a 50 21

d.仿真波形及分析：



此刻将立即数1234H送往8号寄存器，符合设计要求。

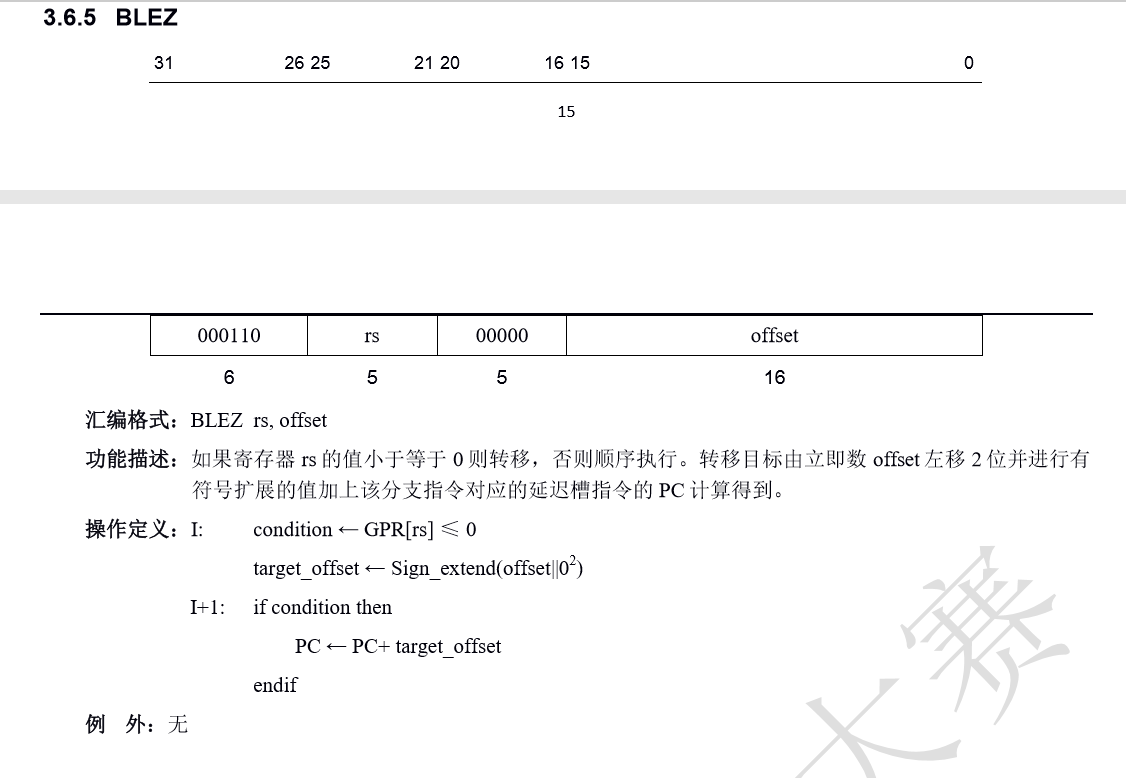


延迟槽指令（nop）执行之后发生跳转，（因1234H大于0，符合跳转条件）程序计数器值PC由bfc00008变为bfc00010，并将结果0送回10号寄存器，证明发生跳转（PC差值不再是4，且没有顺序执行对10号寄存器的赋值操作）符合设计要求。

综上所述，该条指令功能设计正确。

**4.11 BLEZ**

1）指令格式



I型指令，opcode段为000110，跳转类指令。

a.操作数1来自rs寄存器

b.转移目标由立即数offset 左移两位并进行有符号拓展的值加上该分支指令对应的延迟槽指令的PC计算得到。

c.运算结果无需写回。

2）实现代码

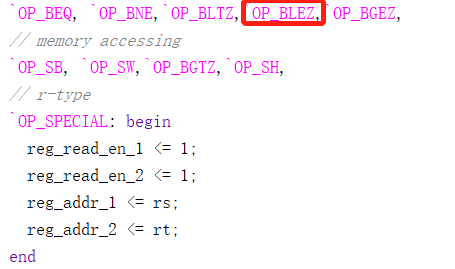
a.在opcode.V中添加OP\_BLEZ字段

C:\Users\Admin\AppData\Local\Temp\1576504399(1).png

//对应BLEZ指令的opcode字段，用于唯一识别该条指令。

b. 在RegGen中添加寄存器读使能。

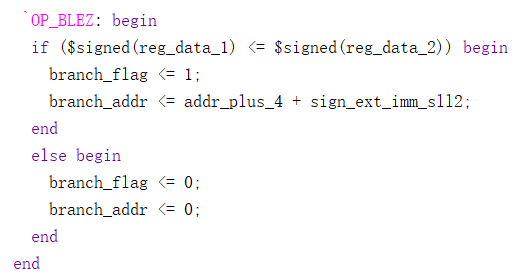
操作数1来自读出寄存器rs的值（注意rs和rt都需要读，另一个操作数将0读入reg\_data\_2中，可直接用于之后的比较操作）



//1，2号寄存器读使能均有效，且1号寄存器的地址是rs寄存器，2号寄存器的地址是rt寄存器

//最后得到的运算结果无需写回

c.在BranchGen.V中编辑跳转条件（做的是有符号比较）



3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0xFFFF //给8号寄存器赋值-1H（以补码形式表示）

BLEZ $8,label //8号寄存器值小于等于0则进行跳转

nop //延迟槽空指令

ADDU $10,$8,&8 //未成功跳转时执行该指令

label:

ADDU $10,$10,$10 //若成功跳转，10号寄存器值应为0

b.对照指令格式将这两条指令翻译成机器代码的格式

0010 0100 0000 1000 1111 1111 1111 1111

000110 01000 00000 0000 0000 0000 0010

0000 0000 0000 0000 0000 0000 0000 0000

000000 01000 01000 01010 00000 100001

000000 01010 01010 01010 00000 100001

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

24 08 FF FF

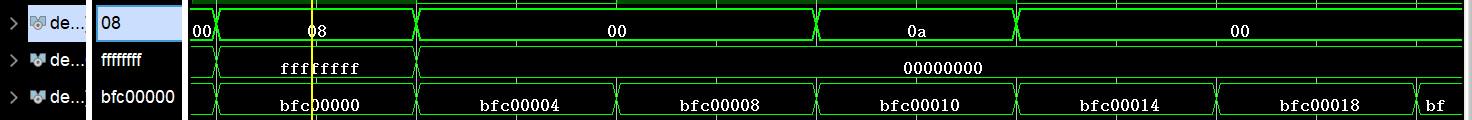
19 00 00 02

00 00 00 00

01 08 50 21

01 4a 50 21

d.仿真波形及分析：



此刻将立即数FFFFH送往8号寄存器，符合设计要求。

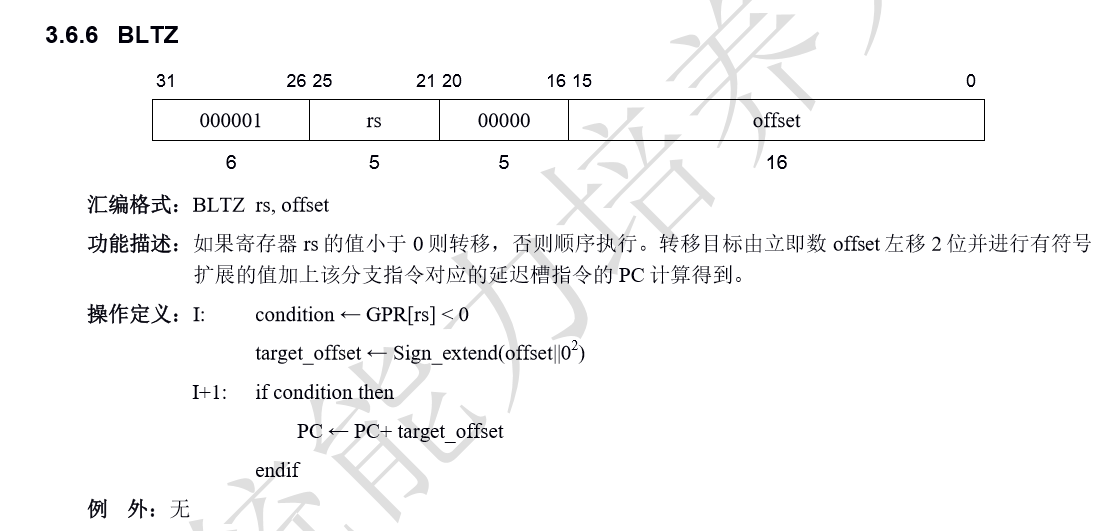


延迟槽指令（nop）执行之后发生跳转，（因-1H小于0，符合跳转条件）程序计数器值PC由bfc00008变为bfc00010，并将结果0送回10号寄存器，证明发生跳转（PC差值不再是4，且没有顺序执行对10号寄存器的赋值操作）符合设计要求。

综上所述，该条指令功能设计正确。

**4.12 BLTZ**

1）指令格式



I型指令，opcode段为000001，跳转类指令。

a.操作数1来自rs寄存器

b.转移目标由立即数offset 左移两位并进行有符号拓展的值加上该分支指令对应的延迟槽指令的PC计算得到。

c.运算结果无需写回。

2）实现代码

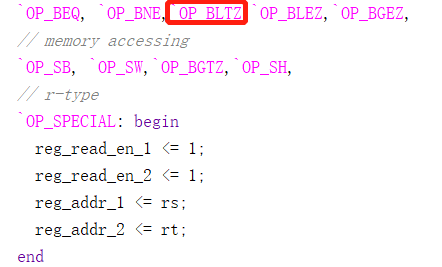
a.在opcode.V中添加OP\_BLTZ字段

C:\Users\Admin\AppData\Local\Temp\1576504900(1).png

//对应BGTZ指令的opcode字段，用于唯一识别该条指令。

b. 在RegGen中添加寄存器读使能

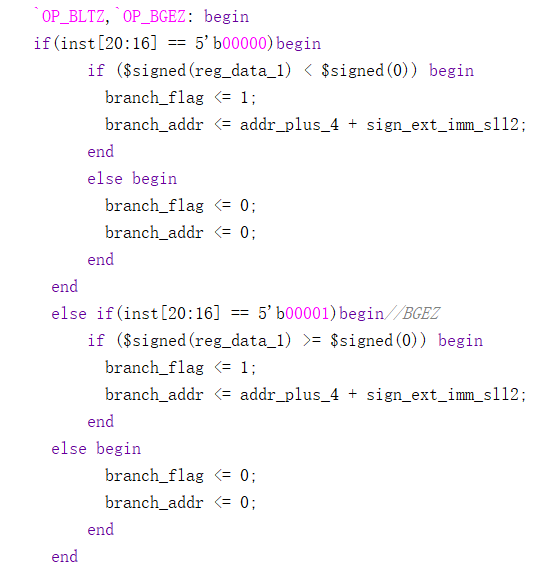
操作数1来自读出寄存器rs的值（注意rs和rt都需要读，第16位-20位需要用于与BGEZ指令做区别，因为这两条指令opcode字段的值一致）



//1，2号寄存器读使能均有效，且1号寄存器的地址是rs寄存器，2号寄存器的地址是rt寄存器

//最后得到的运算结果无需写回

c.在BranchGen.V中编辑跳转条件（做的是有符号比较）



3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0xFFFF //给8号寄存器赋值-1H（以补码形式表示）

BLTZ $8,label //8号寄存器值小于0则进行跳转

nop //延迟槽空指令

ADDU $10,$8,&8 //未成功跳转时执行该指令

label:

ADDU $10,$10,$10 //若成功跳转，10号寄存器值应为0

b.对照指令格式将这两条指令翻译成机器代码的格式

0010 0100 0000 1000 1111 1111 1111 1111

000001 01000 00000 0000 0000 0000 0010

0000 0000 0000 0000 0000 0000 0000 0000

000000 01000 01000 01010 00000 100001

000000 01010 01010 01010 00000 100001

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

24 08 FF FF

05 00 00 02

00 00 00 00

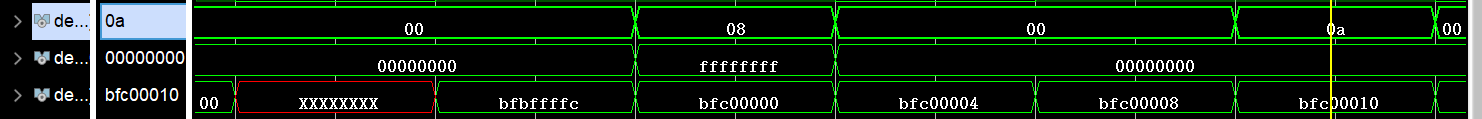
01 08 50 21

01 4a 50 21

d.仿真波形及分析：



此刻将立即数FFFFH送往8号寄存器，符合设计要求。

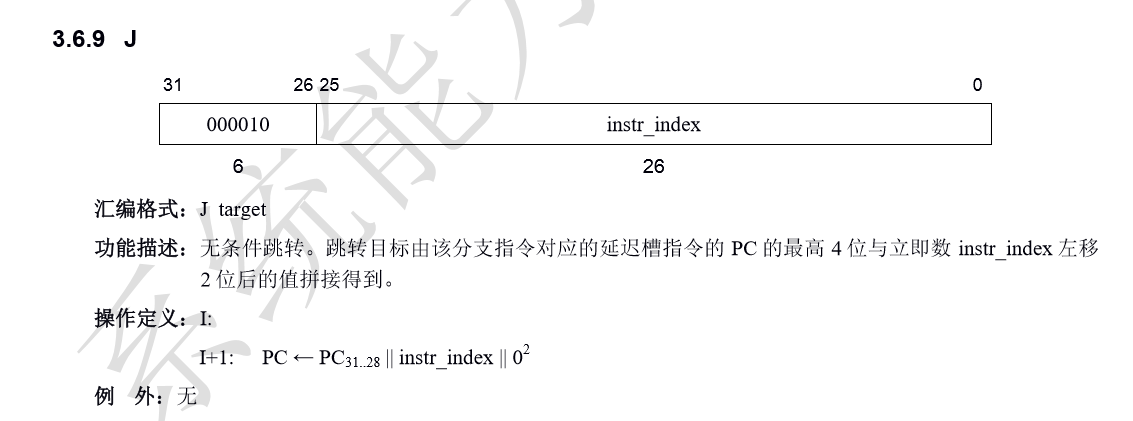


延迟槽指令（nop）执行之后发生跳转，（因-1H小于0，符合跳转条件）程序计数器值PC由bfc00008变为bfc00010，并将结果0送回10号寄存器，证明发生跳转（PC差值不再是4，且没有顺序执行对10号寄存器的赋值操作）符合设计要求。

综上所述，该条指令功能设计正确。

**4.13 J**

1）指令格式



I型指令，opcode段为000010，跳转类指令。

a.无条件跳转

b.转移目标由该分支指令对应的延迟槽指令的PC的最高4位与立即数instr\_index左移2位后的值拼接后得到。

c.运算结果无需写回。

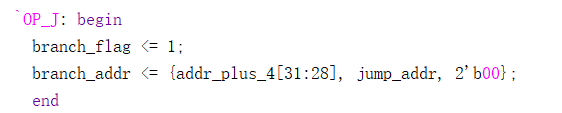
2）实现代码

a.在opcode.V中添加OP\_J字段

C:\Users\Admin\AppData\Local\Temp\1576505297(1).png

//对应J指令的opcode字段，用于唯一识别该条指令。

b.在BranchGen.V中编辑跳转条件（无条件跳转）



3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0xFFFF //给8号寄存器赋值-1H（以补码形式表示）

J //无条件跳转

nop //延迟槽空指令

b.对照指令格式将这两条指令翻译成机器代码的格式

0010 0100 0000 1000 1111 1111 1111 1111

000010 000000 0000 0000 0000 0000 0001

0000 0000 0000 0000 0000 0000 0000 0000

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

24 08 FF FF

08 00 00 01

00 00 00 00

d.仿真波形及分析：



此刻将立即数FFFFH送往8号寄存器，符合设计要求。

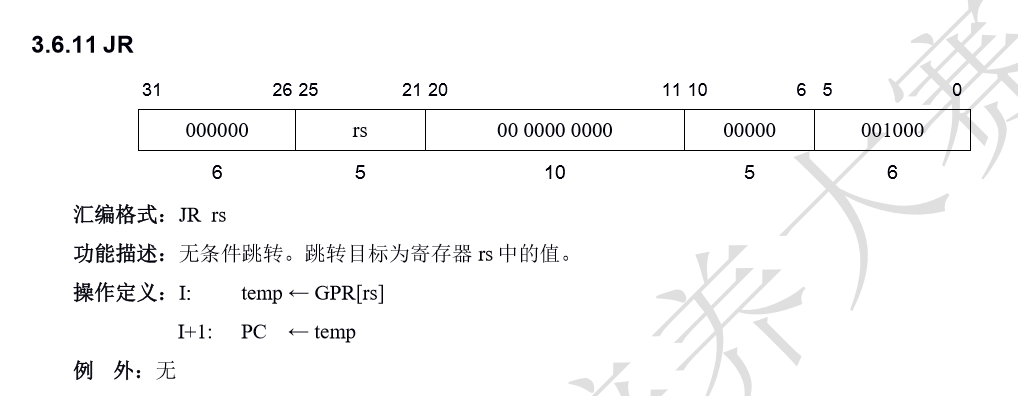


延迟槽指令（nop）执行之后发生跳转，程序计数器值PC由bfc00008变为bfc00004，证明发生跳转（PC值发生了很大变化，不是执行简单的加4操作）符合设计要求。

综上所述，该条指令功能设计正确。

**4.14 JR**

1）指令格式



R型指令，opcode段为000000，跳转类指令。

a.无条件跳转

b.转移目标为寄存器rs中的值。

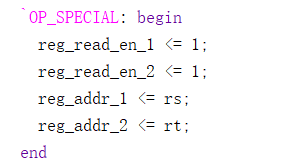
c.运算结果无需写回。

2）实现代码（

a.在funct.V中添加OP\_JR字段，用于唯一识别该条指令

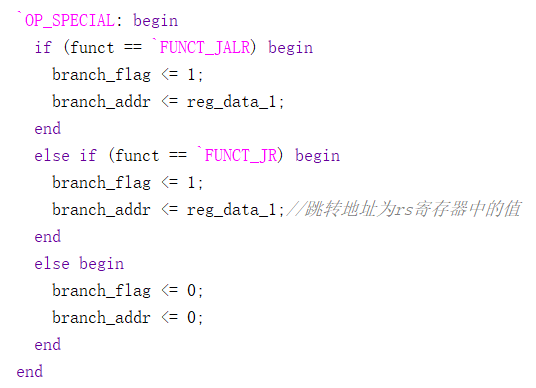
C:\Users\Admin\AppData\Local\Temp\1576673084(1).png

b.在RegGen.V中添加寄存器读写信号，因为跳转目标来自从寄存器rs中读出的数。



已通过OP\_SPECIAL字段进行了统一读写信号控制。

c.在BranchGen.V中编辑跳转条件（无条件跳转）需要通过funct字段与JALR指令做区分。



//跳转目标来自于rs寄存器中的值

3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0x1s234 //给8号寄存器赋值1234H

JR //无条件跳转

nop //延迟槽空指令

b.对照指令格式将这两条指令翻译成机器代码的格式

0010 0100 0000 1000 0001 0010 0011 0100

000000 01000 00 0000 0000 00000 001000

0000 0000 0000 0000 0000 0000 0000 0000

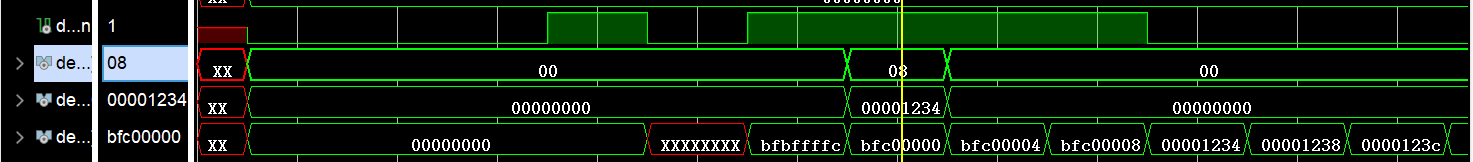
c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

24 08 12 34

01 00 00 08

00 00 00 00

d.仿真波形及分析：



此刻将立即数1234H送往8号寄存器，符合设计要求。

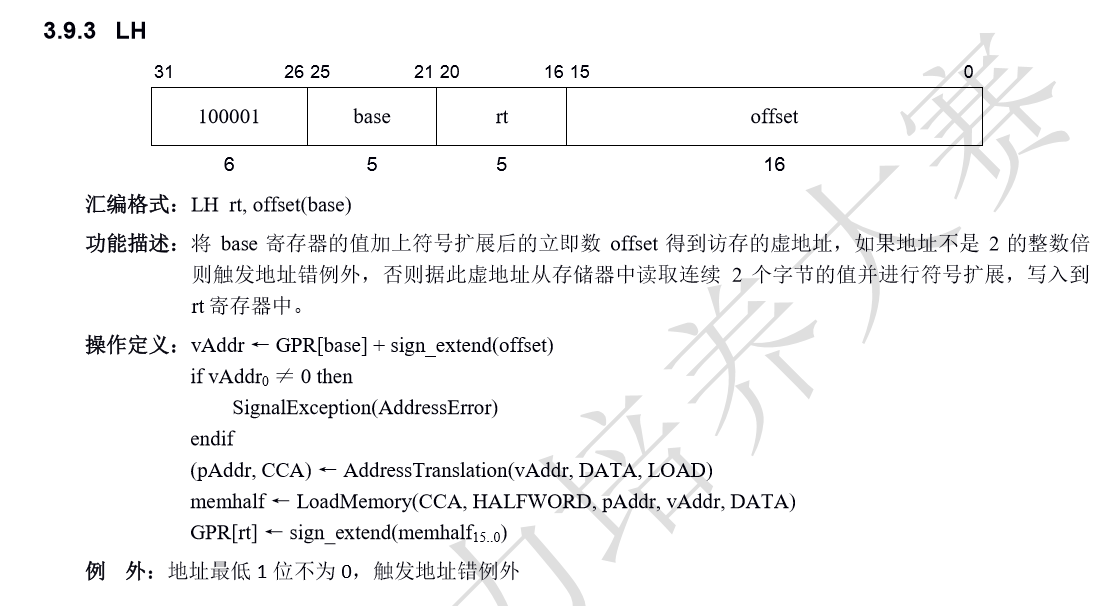


延迟槽指令（nop）执行之后发生跳转，程序计数器值PC由bfc00008变为b00048d0。该值是由该分支指令对应的跳转地址是rs寄存器中的值，证明发生跳转（跳转到了目标地址）符合设计要求。

综上所述，该条指令功能设计正确。

**4.15 LH**

1）指令格式



I型指令，opcode段为100001，访存类指令。

a.Load类指令用于从内存中取数。

b.该条指令的访存地址来自base寄存器的值和立即数offset运算后得到的访存虚地址。

c.运算结果需写回rt寄存器堆。

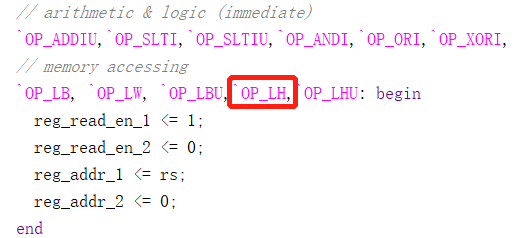
2）实现代码

a.在opcode.V中添加OP\_LH字段

C:\Users\Admin\AppData\Local\Temp\1576673532(1).png//对应LH指令的opcode字段，用于唯一识别该条指令

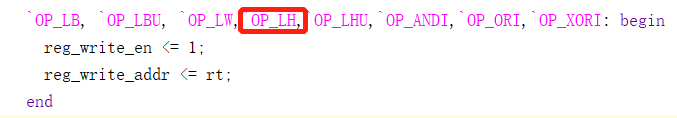
b.在RegGen.V中实现寄存器堆的读写定义。

//base基址值来自读出寄存器rs的值

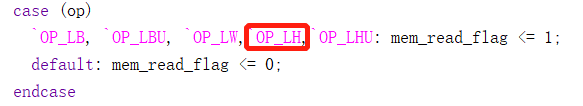


//1号寄存器读使能有效，且1号寄存器的地址是rs寄存器

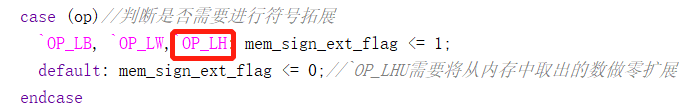
最后读出的结果要写回rt寄存器，修改寄存器写信号



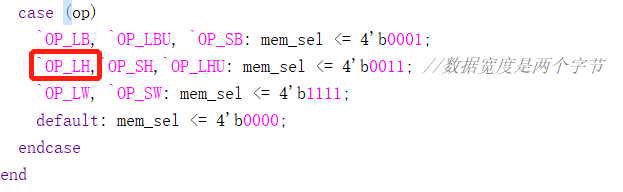
c.在MemGen.V中添加mem\_read\_flag信号，这是存储器读信号；



修改mem\_sign\_ext\_flag信号。这里完成的是将从内存中读出的数做零扩展或符号扩展。LH指令中需要对从内存中读出的数做符号扩展：



修改字节使能信号，定义访问时的数据宽度，便于以正确的形式访问内存单元：



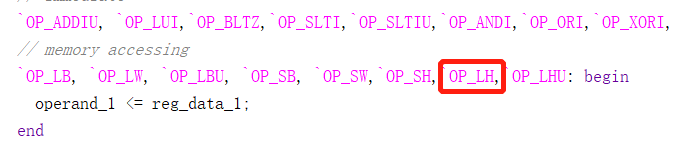
//进行半字访问时mem\_sel信号为：4’b0011。

d.在FunctGen.V中需要编辑LU具有的加法功能，以便通过base和offset的值计算所访问的内存单元。

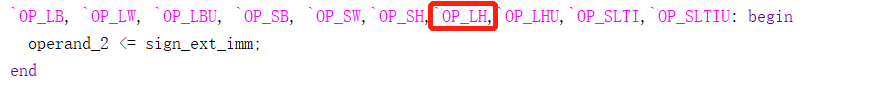


e. 在Opreand.V中生成需要的两个操作数，offset字段可以看做是立即数（immideate）只需要判断是符号扩展还是零扩展，LH指令需要对立即数做符号扩展

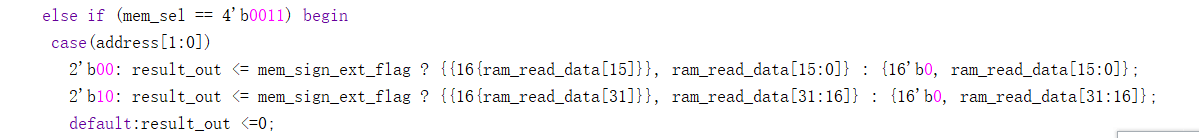
//生成的操作数1来自从寄存器中直接读出的数：



//生成的操作数2来自立即数部分做符号拓展读出的值



f.在写回级WB.V修改写回的信号：



3）仿真测试a.汇编测试指令格式如下：

ADDIU $8,$0,0xbfc0 //将8号寄存器赋值bfc0H

ADDIU $9,$0,0x3333 //将9号寄存器赋值3333H

SW $9,0x1234 //将9号寄存器低半字存入内存单元

LH $10,0x1234 //取内存单元中的值，结果写回10号寄存器

b.对照指令格式将这两条指令翻译成机器代码的格式

001001 00000 01000 0001 0010 0011 0100

001001 00000 01001 0011 0011 0011 0011

101011 01000 01001 0001 0010 0011 0100

100001 01000 01010 0001 0010 0011 0100

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

24 08 12 34

24 09 33 33

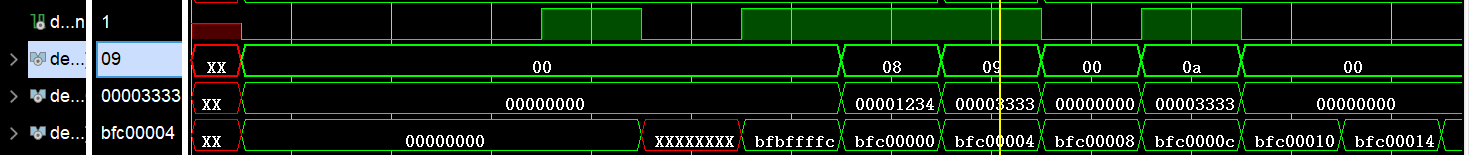
ad 09 12 34

85 0a 12 34

d.仿真波形及分析：



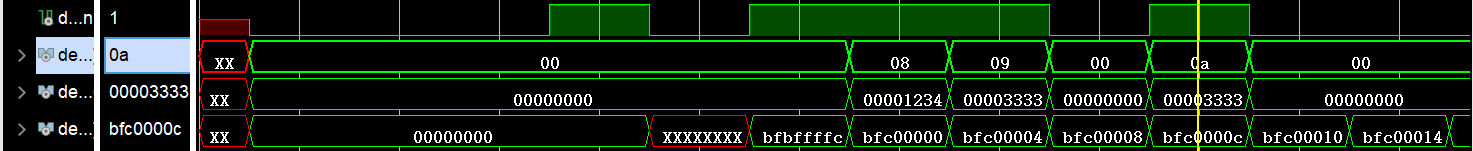
此刻将立即数1234H送往8号寄存器，符合设计要求。



下一时刻将立即数3333H送往9号寄存器，符合设计要求。



下一时刻执行store指令，将9号寄存器中的立即数3333H存入指定的内存单元（bfc01234）

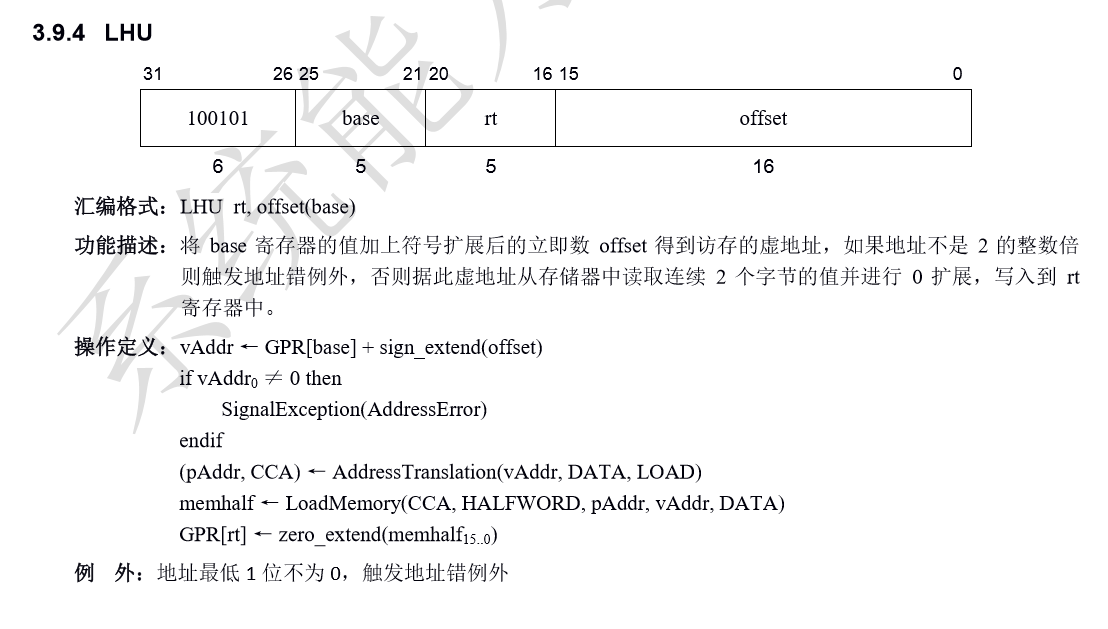


下一时刻执行LH指令，从指定的内存单元（bfc01234）中读取半字，结果写回10号寄存器，如图所示，该功能实现正确。

综上所述，该条指令功能设计正确。

**4.16 LHU**

1）指令格式



I型指令，opcode段为100101，访存类指令。

a.Load类指令用于从内存中取数。

b.该条指令的访存地址来自base寄存器的值和立即数offset运算后得到的访存虚地址。

c.运算结果需写回rt寄存器。

2）实现代码

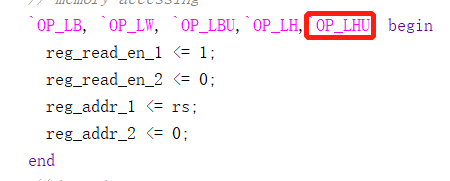
a.在opcode.V中添加OP\_LHU字段

C:\Users\Admin\AppData\Local\Temp\1576675651(1).png

//对应LHU指令的opcode字段，用于唯一识别该条指令

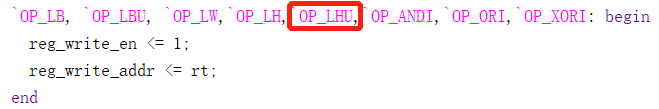
b.在RegGen.V中实现寄存器堆的读写定义。

base基址值来自读出寄存器rs的值



//1号寄存器读使能有效，且1号寄存器的地址是rs寄存器

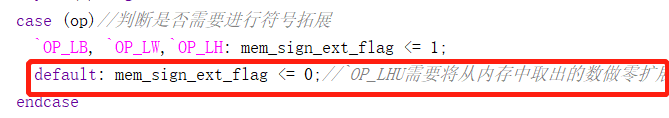
最后读出的结果要写回rt寄存器，修改寄存器写信号



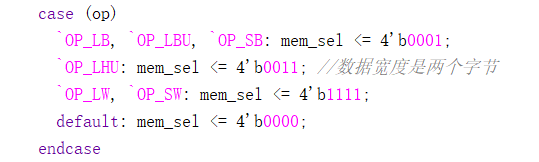
c.在MemGen.V中添加mem\_read\_flag信号，这是存储器读信号；

C:\Users\Admin\AppData\Local\Temp\1576371188(1).png

修改mem\_sign\_ext\_flag信号。这里完成的是将从内存中读出的数做零扩展或符号扩展。LHU指令中需要对从内存中读出的数做零扩展：



修改字节使能信号，定义访问时的数据宽度，便于以正确的形式访问内存单元：



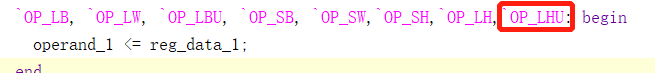
进行半字访问时mem\_sel信号为：4’b0011。

d.在FunctGen.V中需要编辑LHU具有的加法功能，以便通过base和offset的值计算所访问的内存单元。

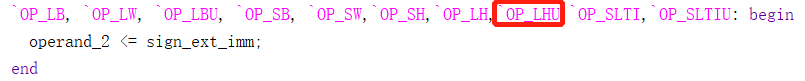


e. 在Opreand.V中生成需要的两个操作数，offset字段可以看做是立即数（immideate）只需要判断是符号扩展还是零扩展，LH指令需要对立即数做符号扩展

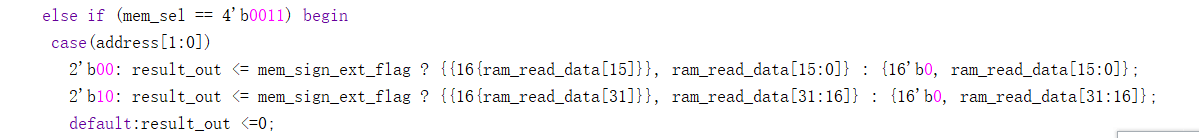
//生成的操作数1来自从寄存器中直接读出的数：



//生成的操作数2来自立即数部分做符号拓展读出的值



f.在写回级WB.V修改写回的信号：



3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0xbfc0 //将8号寄存器赋值bfc0H

ADDIU $9,$0,0x3333 //将9号寄存器赋值3333H

SW $9,0x1234 //将9号寄存器低半字存入内存单元

LHU $10,0x1234 //取内存单元中的值，结果写回10号寄存器

b.对照指令格式将这两条指令翻译成机器代码的格式

001001 00000 01000 0001 0010 0011 0100

001001 00000 01001 0011 0011 0011 0011

101011 01000 01001 0001 0010 0011 0100

100101 01000 01010 0001 0010 0011 0100

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

24 08 12 34

24 09 33 33

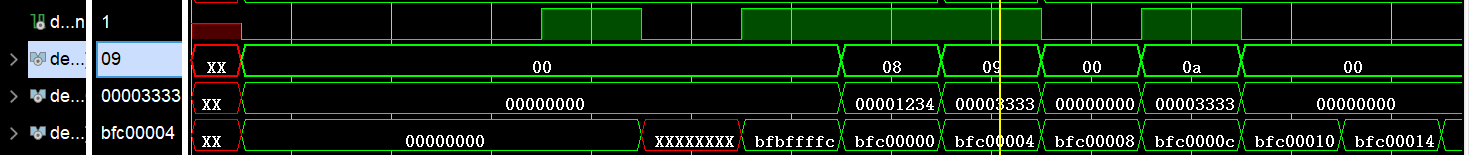
ad 09 12 34s

95 0a 12 34

d.仿真波形及分析：



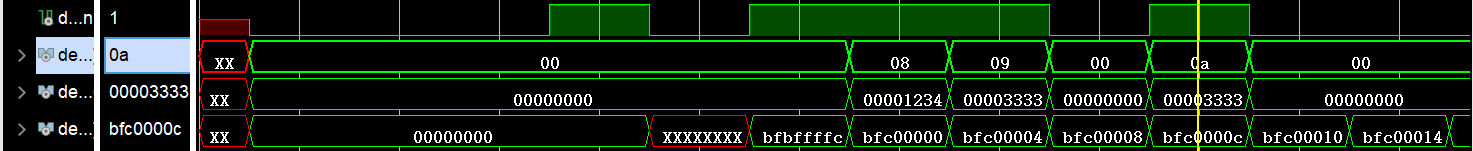
此刻将立即数1234H送往8号寄存器，符合设计要求。



下一时刻将立即数3333H送往9号寄存器，符合设计要求。



下一时刻执行store指令，将9号寄存器中的立即数3333H存入指定的内存单元（bfc01234）

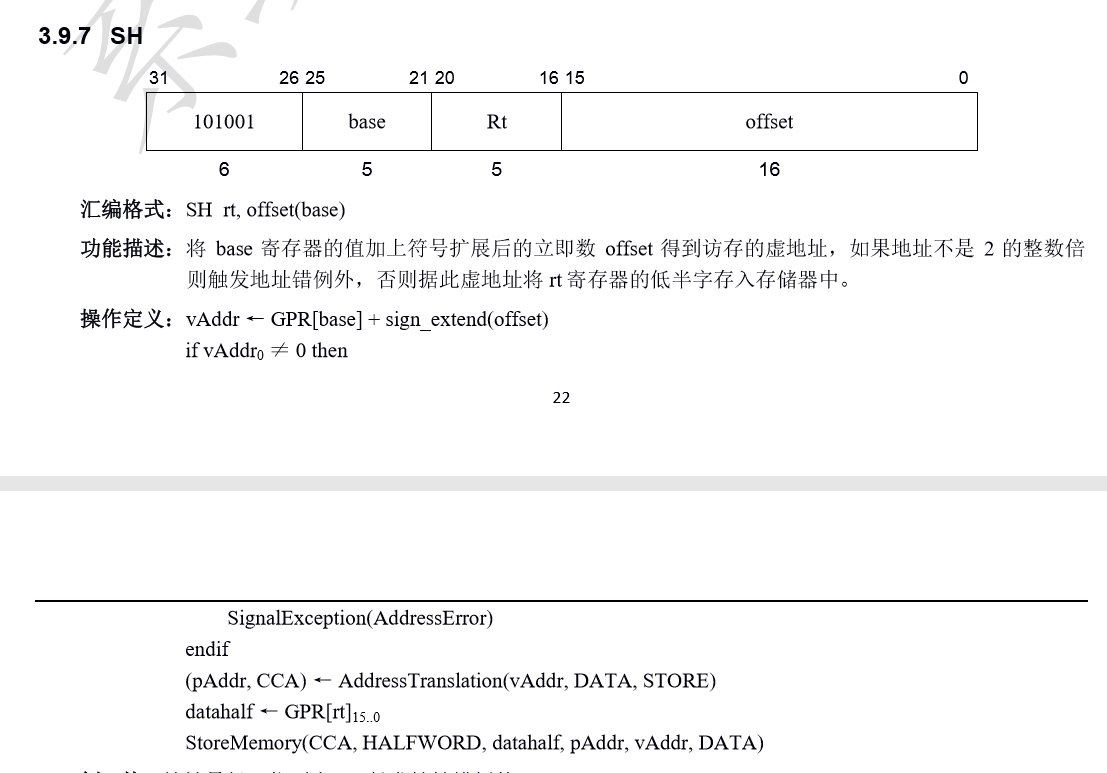


下一时刻执行LH指令，从指定的内存单元（bfc01234）中读取半字，结果写回10号寄存器，如图所示，该功能实现正确。

综上所述，该条指令功能设计正确。

**4.17 SH**

1）指令格式



I型指令，opcode段为101001，访存类指令。

a.store类指令用于向内存中的数。

b.该条指令的访存地址来自base寄存器的值和立即数offset运算后得到的访存虚地址。

c.写入内存的值来自rt寄存器。

2）实现代码

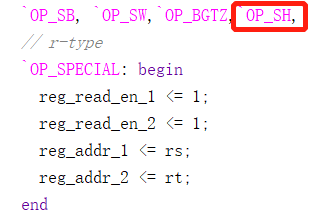
a.在opcode.V中添加OP\_SH字段

C:\Users\Admin\AppData\Local\Temp\1576676528(1).png

//对应SH指令的opcode字段，用于唯一识别该条指令

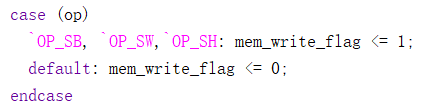
b.在RegGen.V中实现寄存器堆的读写定义。

//base基址值来自读出寄存器rs的值



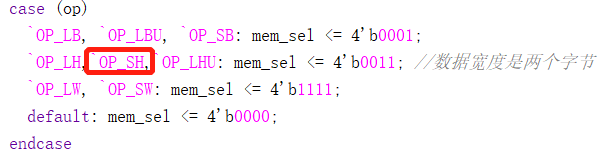
//1\2号寄存器读使能均有效，存入内存的数需要从rt寄存器中读取

c.在MemGen.V中添加mem\_write\_flag信号，这是存储器写信号；



修改字节使能信号，定义访问时的数据宽度，便于以正确的形式访问内存单元：

进行半字访问时mem\_sel信号为：4’b0011。

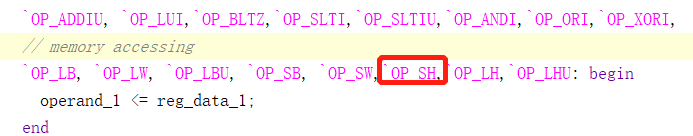


d.在FunctGen.V中需要编辑SH具有的加法功能，以便通过base和offset的值计算所访问的内存单元。

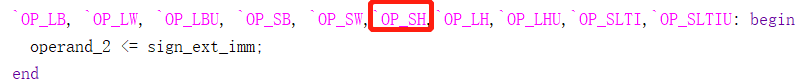


e. 在Opreand.V中生成需要的两个操作数，offset字段可以看做是立即数（immideate）只需要判断是符号扩展还是零扩展，LH指令需要对立即数做符号扩展

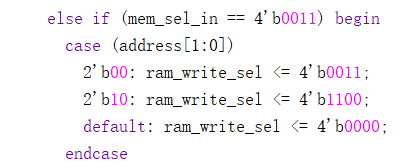
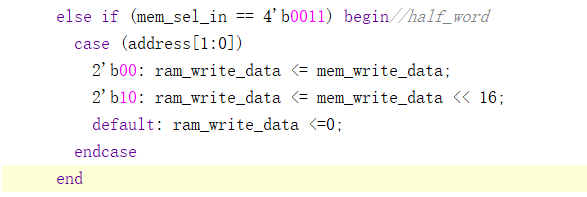
//生成的操作数1来自从寄存器中直接读出的数：



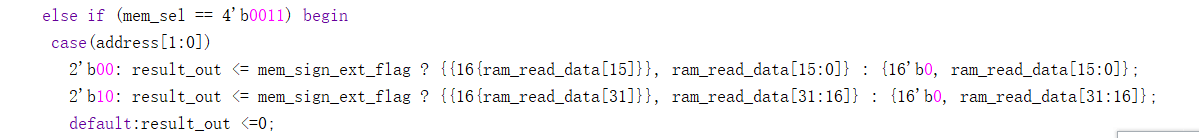
//生成的操作数2来自立即数部分做符号拓展读出的值



f. 在MEM.V一级编辑访问两个字节的操作，注意半字读写时只能访问偶地址，所以只取（00）或（10），再把相应的半字位置1。还要修改ram\_write\_data因为要将要写的数据与写使能信号对应起来必须要对要写的数据做移位操作。

g.在写回级WB.V修改写回的信号：



3）仿真测试

a.汇编测试指令格式如下：

ADDIU $8,$0,0xbfc0 //将8号寄存器赋值bfc0H

ADDIU $9,$0,0x3333 //将9号寄存器赋值3333H

SH $9,0x1234 //将9号寄存器低半字存入内存单元

LHU $10,0x1234 //取内存单元中的值，结果写回10号寄存器

b.对照指令格式将这两条指令翻译成机器代码的格式

001001 00000 01000 0001 0010 0011 0100

001001 00000 01001 0011 0011 0011 0011

101001 01000 01001 0001 0010 0011 0100

100101 01000 01010 0001 0010 0011 0100

c.二进制码太过冗长，每4位对应一个16进制数，保存至inst\_rom.bin文件，文件的存放路径是： mycpu\_prj1\mycpu.sim\sim\_1\behav\xsim\,可供CPU仿真模块读取并执行。

24 08 12 34

24 09 33 33

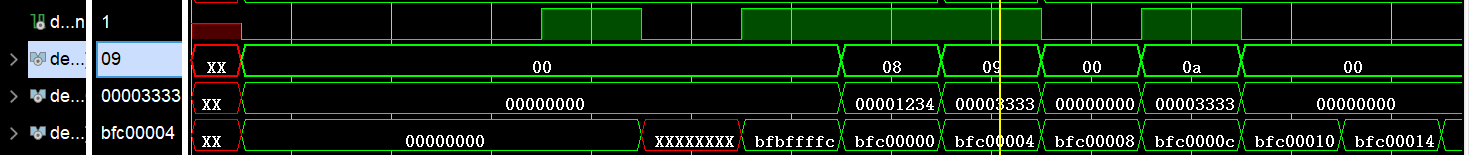
a5 09 12 34

95 0a 12 34

d.仿真波形及分析：



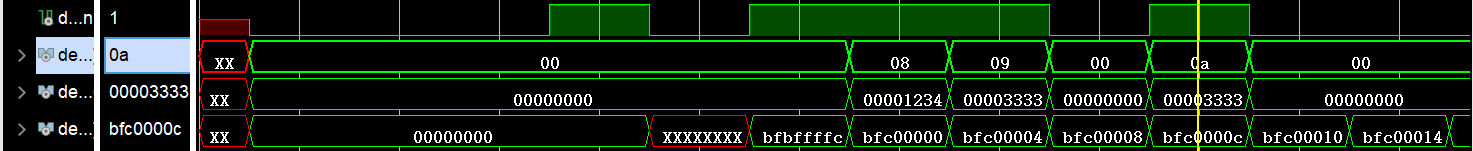
此刻将立即数1234H送往8号寄存器，符合设计要求。



下一时刻将立即数3333H送往9号寄存器，符合设计要求。



下一时刻执行SH指令，将9号寄存器中的立即数3333H存入指定的内存单元（bfc01234）



下一时刻执行LH指令，从指定的内存单元（bfc01234）中读取半字，结果写回10号寄存器，如图所示，该功能实现正确。

综上所述，该条指令功能设计正确。

**四：结论（讨论）**

1. **结论**

在学习课程设计的这一阶段中，得益于张磊老师和邢其正学长的帮助，我在课程所提供的TinyMIPS微处理的基础上理解、学习并掌握了设计流水线CPU的基本方法和必需的基础结构模块。第一次接触硬件编程是在一年前的数字逻辑课上，当时只浅显的知道如何使用vivado来实现硬件逻辑，但并未真正理解设计的意义和其具有的功能，后来在上学期接触了计算机组成原理这一门计算机学科基础课程，使用vivado做了一个简单的单周期CPU，它可以支持一些简单指令的执行。再加之理论课，我才更好的掌握了计算机设计架构的知识，但学习程度还是只浮于表面，不够深入。这次的课程设计基于我已经学习过的计算机组成原理，操作系统，汇编语言等等计算机方面的基础课程。再加上实验指导书的详细指导、老师与学长的精彩讲解，我自己不断摸索，在学习、理解、并完成指令扩展的操作中，我真正掌握了设计一套流水线式微处理器的设计方法，这对于我理解：用户对于计算机的每一步操作，计算机是如何从高级语言翻译成汇编语言，直至翻译成计算机可以理解的机器语言（简单的二进制代码）并最终执行提供了良好的一次实践机会。对于我掌握CPU的总体结构设计提供了很大帮助。

1. **讨论**

实验中的开始，是我对计算机组成原理相关知识的回顾与复盘过程，计组是上学期的课程内容，我对部分知识已经有了遗忘现象，于是我在最初的学习过程中首先选择复习有关计算机组成原理的相关知识，重新整体把握CPU的设计架构。

第二阶段，是要读懂学长提供的TinyMIPS代码，我从工程中的汇编语言部分开始，结合实验指导书上册中提供的项目索引目录，结合五级流水线数据通路图，通过完成TinyMIPS处理器的仿真测试和FPGA测试，理解了流水线数据通路和控制逻辑的基本设计原理。理解了CPU流水线设计中将指令信息逐渐传递这一设计关键，为我之后的自主设计过程奠定了坚实的理论基础。

下一阶段，是指令执行过程的学习阶段。结合学长在课堂上所讲解的内容，对已经在工程中实现的22条简单指令做了执行过程分析，编写分模块仿真验证，并测试指令功能。阅读A02文档MIPS指令集规范文档，学习不同指令类型对指令执行方式的影响，基本理解了流水线这种处理结构是如何加快指令的执行速度的。

最后一阶段是指令自设计阶段，从指令格式文档中选择了17条跨越不同指令维度，具有不同功能的指令进行了实现。在设计过程中，基于已有的TinyMIPS功能，主要需要自己扩展的部分是五级流水线中的ID（译码级）和EX（执行级），需根据指令格式修改：寄存器读写信号、访存读写信号、指令生成功能、操作数生成功能和指令需要具体执行的逻辑运算操作等部分。

自扩展过程中，由于我对知识学习的不扎实，曾遇到不小的问题，例如遗漏部分访问信号、没有编写出正确的指令运算格式，寻址内存字长指定不正确等问题。得益于邢其正学长的讲解和答疑，这些曾困扰我的问题最终均被一一修正，我也成功通过了全部49个功能测试点，进一步完善了TinyMIPS可以提供的功能。

1. **改进方向**

针对我现在的学习成果，我已经对TinyMIPS中一条基本汇编指令的执行过程有了较好的把握，但对于该微处理器的其他分模块功能与如何实现，我尚未完全理解透彻。今后，我会继续加深自己对计算机底层硬件逻辑的学习与掌握，深入学习汇编编程，理解流水线CPU中各类相关问题的产生机理和解决方式。

**北京科技大学实验报告**

学院：计通学院 专业：计算机科学与技术 班级：计172班

姓名：刘梓瑄 学号：41724039 实验日期：2019年12月15日

**五、教师评审**

|  |  |
| --- | --- |
| **教师评语** | **实验成绩** |
| （虽然课设主要侧重于验证问题，但是建议各位老师从解决“工程技术问题”，特别是“复杂工程问题”的角度去评审学生课设过程及代码阅读报告，主要内容包括提出问题、分析问题、解决问题及验证问题。**要有较详细的评审意见**。）  签名：  日期： |  |