**实验报告——单指令CPU电路设计**

**一、需求分析**

**第一部分：基本部件的设计**

第一部分重在理解各个基本部件的功能，经过查询MIPS指令集，总结如下：

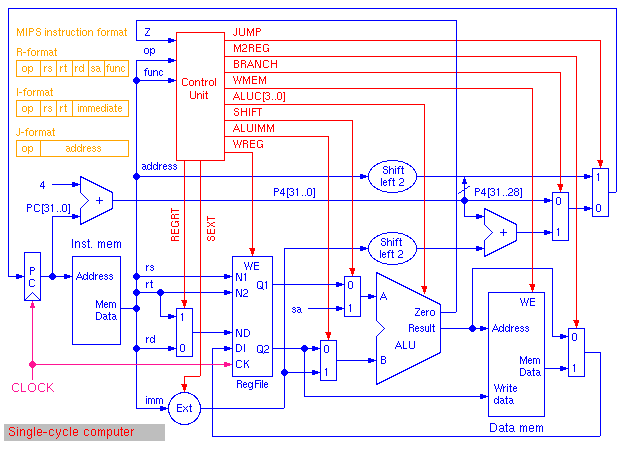
|  |  |  |  |
| --- | --- | --- | --- |
| **指令名称** | **指令格式** | **功能描述** | **汇编指令中的操作码** |
| add | ADD rd,rs,rt | GPR[rd] 🡨 GPR[rs] + GPR[rt] | 000000+100000 |
| sub | SUB rd,rs,rt | GPR[rd] 🡨 GPR[rs] - GPR[rt] | 000000+100010 |
| and | AND rd,rs,rt | GPR[rd] 🡨 GPR[rs] AND GPR[rt] | 000000+100100 |
| or | OR rd,rs,rt | GPR[rd] 🡨 GPR[rs] or GPR[rt] | 000000+100101 |
| slt | SLT rd,rs,rt | GPR[rd] 🡨 (GPR[rs] < GPR[rt]) | 000000+101010 |
| addi | ADDI rt,rs,immediate | GPR[rt]🡨 GPR[rs] + immediate | 001000 |
| andi | ANDI rt,rs,immediate | GPR[rt]🡨 GPR[rs] AND immediate | 001100 |
| ori | ORI rt,rs,immediate | GPR[rt]🡨 GPR[rs] or immediate | 001101 |
| slti | SLTI rt,rs,immediate | GPR[rt]🡨 (GPR[rs] < immediate) | 001010 |
| sw | SW rt,offset(base) | memory[GPR[base] + offset] 🡨  GPR[rt] | 101011 |
| lw | LW rt,offset(base) | GPR[rt]🡨 memory[GPR[base]+offset] | 100011 |
| j | Ins rt,rs,pos,size | GPR[rt]🡨  InsertField(GPR[rt],GPR[rs],msb,lsb]) | 000010 |
| nop | NOP | No operation | 000000+000000 |

**备注：**

1. GPR[r]表示通用寄存器r
2. Immediate表示立即数
3. memory[x]表示地址为x的内存空间
4. 具体的二进制指令具体格式没有在表中列出来

**第二部分：控制单元的设计**

第二部分主要是控制部分，如何控制各个单元完成具体的MIPS指令，时序如何组织。这一部分将会在实验设计中详细展开介绍。



**二、实验设计**

结合上课所讲的有关数字部件设计的内容，参考文档中给出的两个部分的任务分配，我将实现分为了datapath部分和controller部分，分别对应了文档中的两个部分的设计。

**第一部分:datapath**

**1.module mux2：**

1. **输入：**

**d0：**宽度为WIDTH，默认为8-bit

**d1：**宽度为WIDTH，默认为8-bit

**s：**表示选通控制位，为1时选择d1，为0时选择d0

1. **输出：**

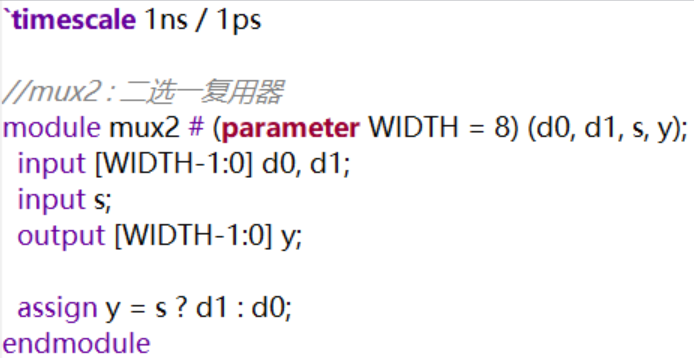
**y：**(宽度为WIDTH，默认为8-bit，选通器的输出结果)

1. **功能描述：**

参考需求文档中给出的简单的单指令CPU的电路图(如上图所示)，我们需要在regFile、ALU等多处用到二选一的选通器，所以在此实现作为工具module

1. **代码解析：**

代码实现较为简单，只需要一个条件表达式即可，代码如下：

****

**2.module alu：**

**a)输入：**

**src1：**表示第一个操作数，32-bit

**src2：**表示第二个操作数，32-bit

**alucont：**用于选择具体的operation，3-bit；根据文档需求，此处需要实现四个基本指令，add、sub、or、and以及beq，具体的alucont和operation的对应表如下：

|  |  |
| --- | --- |
| **alucont** | **operation** |
| 000 | src1 & src2 |
| 010 | src1 + src2 |
| 001 | src1 | src2 |
| 011 | src1 – src2 |
| 111 | src1 ＜ src2 ？ 1 : 0 |

**b)输出：**

**result：**表示两个操作数在alucont对应操作后结果，32-bit

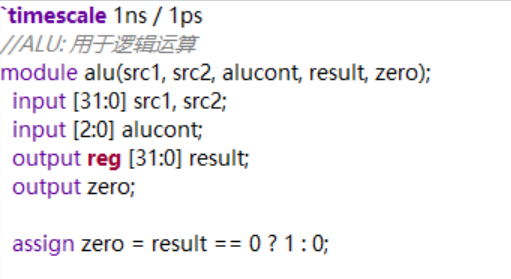
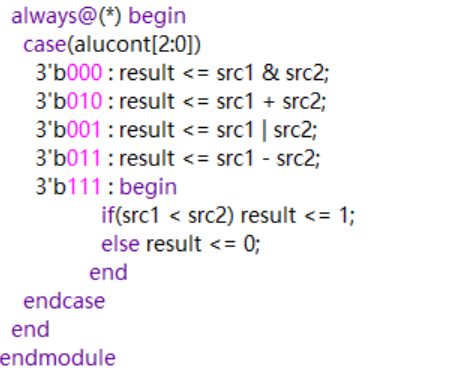
**zero：**表示result结果是否为0，1-bit，类似于汇编指令中的ZF

**c)功能：**

算术逻辑单元，主要是为了实现需求文档中的加减并或及比较操作

**d)关键代码解析：**

此处的实现比较简单，在我们的前几个LAB中设计过，也用到过，所以此处不再赘述，直接上代码：

//

**3.module sl2：**

**a)输入：**

**a：**移位操作的操作数，32-bit

**b)输出：**

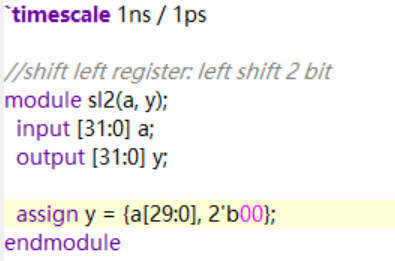
**y：**移位操作的结果，32-bit

**c)功能：**

将操作数a左移2位

**d)代码解析：**

将a的低30位赋给y的高30位，后面两位补0



**4.module extendNum：**

**a)输入：**

**imm：**要进行符号扩展的操作数，16-bit

**b)输出：**

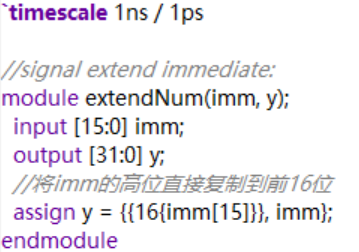
**y：**符号扩展之后的结果，32-bit

**c)功能：**

将操作数imm进行符号扩展，用于对

**d)代码：**

具体实现中，就是将y的最高16位用imm的符号位填充，也就是imm的最高位，低16位与imm相等，代码如下：



**5.module regFile：**

**a)输入：**

**clk：**时钟，1-bit

**regwriteEn：**写入使能控制位，1-bit

**regwriteaddr：**当enable有效时，在时钟上升沿写入寄存器的值，50bit

**regwritedata：**当enable有效时，在时钟上升沿写入寄存器的“地址”，32-bit

**rsaddr：**对应与第一部分指令的基本形式中的rs寄存器，5-bit

**rtaddr：**对应于第一部分指令的基本形式中的rt寄存器，5-bit

**b)输出：**

**rsdata：**rs寄存器中的值，32-bit

**rtdata：**rt寄存器中的值，32-bit

**c)功能：**

该模块实现了两部分的功能，统一起来都是对寄存器文件的操作，参考了前几次LAB中的寄存器文件的设计，写入控制位regwriteEn对应的就是前几次LAB中的WE：

①当enable有效时，在时钟上升沿将regwritedata写入regwriteaddr对应的寄存器中

②若rsaddr(或rtaddr)不为0，那么将对应的寄存器中的值读出来

**d)代码解析：**

因为此处也是前几次LAB中做过的内容，并且在此次实现中更为简单，所以此处具体的详细实现不再赘述，直接上代码：



**6.module PCPlus4：**

**a)输入：**

**a：**原始的PC寄存器中的值，32-bit

**b：**PC寄存器要加上的值，32-bit

**b)输出：**

**result：**加法之后，PC寄存器中的值，32-bit

**c)功能：**

这个模块主要是为了实现对PC寄存器中的值的操作，得到下一条指令

1. **代码解析：**

代码实现十分简单，result=a+b，此处不再贴代码截图

**7.module PC：**

**a)输入：**

**clk：**时钟，1-bit

**reset：**重置控制位，1-bit

**next：**下一条指令

**b)输出：**

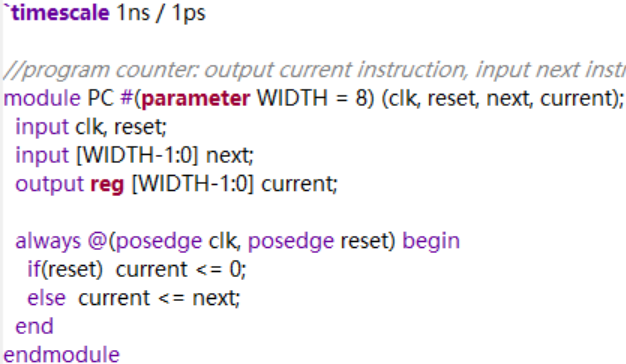
**current：**当前指令

**c)功能：**

输入下一条指令，在时钟周期上沿，根据reset控制位，得到当前指令

**d)代码解析：**

关键的部分在于，在时钟上升沿，如果reset=1，得到当前指令为0，否则为next指令



**8.module datapath：**

**a)功能：**

该module利用上面所实现的最基本的功能部件，实现datapath

**b)输入：**

**clk：**表示时钟，1-bit

**reset：**表示重置PC的控制位，1-bit

**memtoreg：**表示将内存数据写入寄存器的使能控制位，当该值有效时，内存中的值加载到寄存器中，1-bit

**alusrc：**表示当前指令是R类型还是I类型，1表示I类型，0表示r类型，1-bit

**regdst：**表示写入的是rt还是rd，R类型指令写入的是rd，对应该位为1，I类型指令写入的是rt，对应该位为0，1-bit

**regwrite：**表示是否要写入寄存器，1表示写入，1-bit

**jump：**表示当前指令是否为跳转指令，1表示跳转，1-bit

**alucontrol：**表示当前的操作码，对应表格已经在上面介绍module alu时给出，3-bit

**instr：**表示当前正在执行的指令，32-bit

**readdata：**表示从内存中读取的数值，在执行过程中可能会被加载到寄存器中，32-bit

**c)输出：**

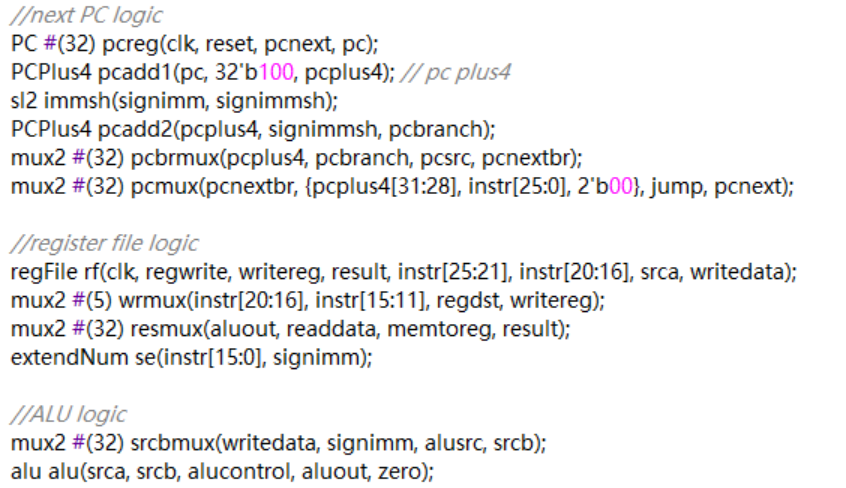
**zero：**表示alu的计算结果是否为0，1-bit

**pc：**表示下一条指令的地址

**aluout：**表示alu单元的计算结果

**writedata：**表示要写入的寄存器的地址

**d)整体逻辑：**

****

**第二部分：controller**

此处是有层次调用关系的，我们按照从下往上的顺序分析。

**1.module maindec**

**a)输入：**

**op：**表示当前指令的operation类型，6-bit

**b)输出：**

**memtoreg、memwrite、branch、alusrc、regdst、regwrite、jump、aluop**

**c)功能：**

根据op类型得到datapath中所需要的控制位

**d)代码解析：**

关键代码如下，实现很简单，不再过多赘述：



**2.module aludec**

**a)输入：**

**funct：**表示指令的低6位，当指令为R类型指令时，该6位可以标识不同的具体操作，6-bit

**aluop：**表示在指令最高6位的指令类型，当指令为非R类型的操作时，该6位可以直接用来区分不同的操作，6-bit

**b)输出：**

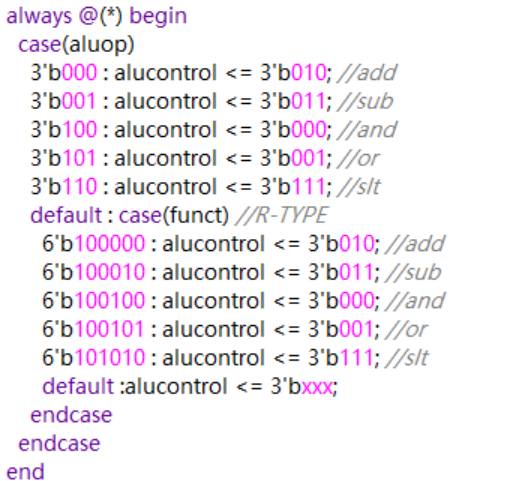
**alucontrol：**表示操作码，对应表在介绍module alu时已经列出来了，此处不再重复

**c)功能：**

根据指令中的必要信息(高六位加上必要时的低六位)得到操作类型对应的操作码

**d)代码解析：**

此处的实现较为简单，需要提一下的是对于R类型的指令，无论是ADD还是AND抑或是其他，所有的最高六位都为0，需要低六位来区分：

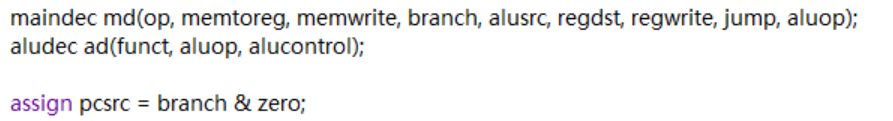


**3.module controller**

**a)功能：**

该模块整合了上面的两个模块，maindec和aludec，得到datapath中必要的控制位和ALU单元的操作类型

**d)代码解析：**



此处值得一提的是，只有branch为1且zero也为1时，pcsrc才会为1

**第三部分：top**

此部分将指令加载、数据加载与控制单元和datapath组成的mips组合起来，实现完整的简易CPU实现

**1.module mips**

整合了controller和datapath，得到一个CPU

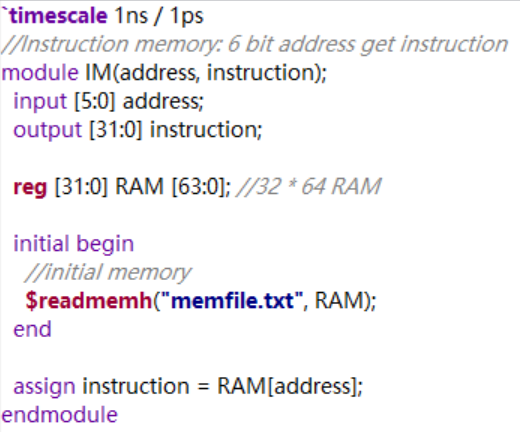
**2.module IM**

**a)功能：**

这个模拟的是指令从磁盘加载到内存的过程，完整的CPU需要从磁盘加载到内存中的指令

**b)代码解析：**

从文件中读取指令，放进RAM内存中



**3.module DM**

**a)功能：**

此处模拟的则是从内存加载数据和在时钟上升沿向内存写入数据

**b)代码解析：**

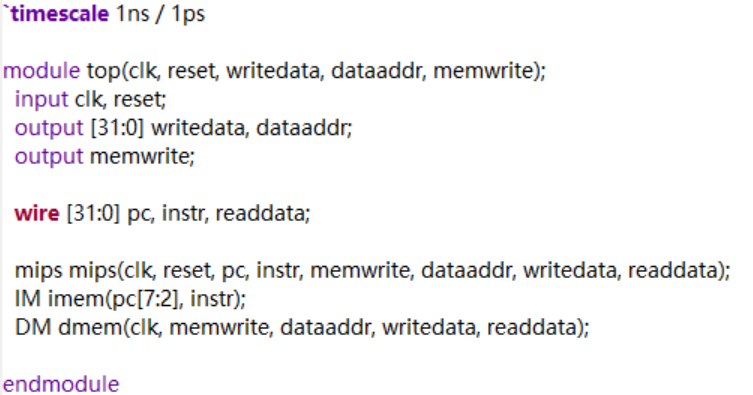


**4.module top**

**a)功能：**

整合了CPU-mips指令处理部分和数据内存、指令内存三部分

**b)代码解析：**

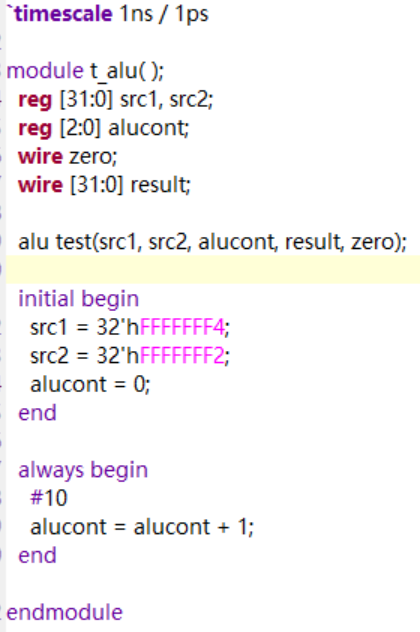


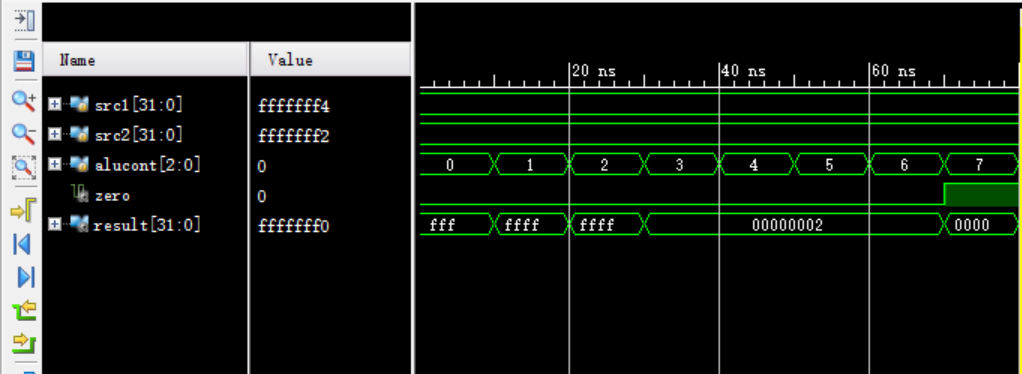
**三、仿真波形**

**1.module alu**

两个操作数，0xfffffff4和0xfffffff2，初始的alucont(操作码)为0，之后每过10个时钟周期加一

**代码和仿真图：**

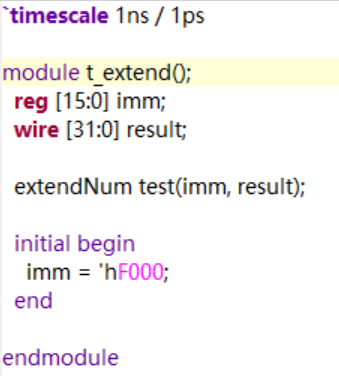


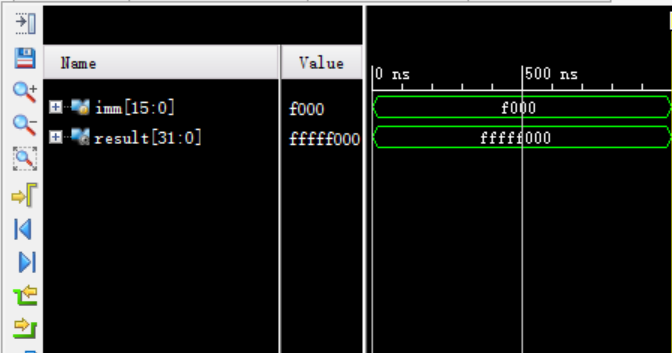


**2.module extendNum**

对0xF000进行扩展：

**代码和仿真图：**

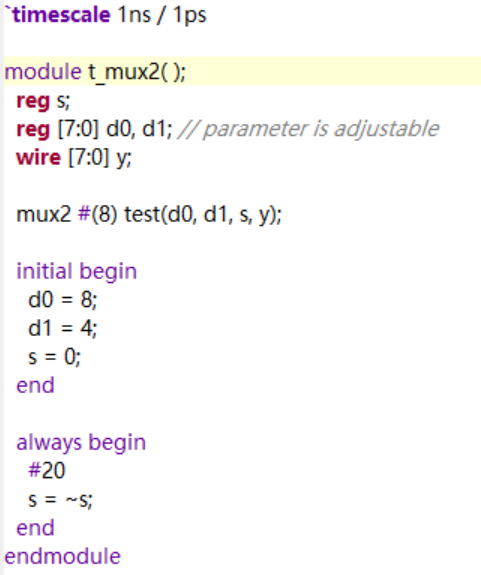


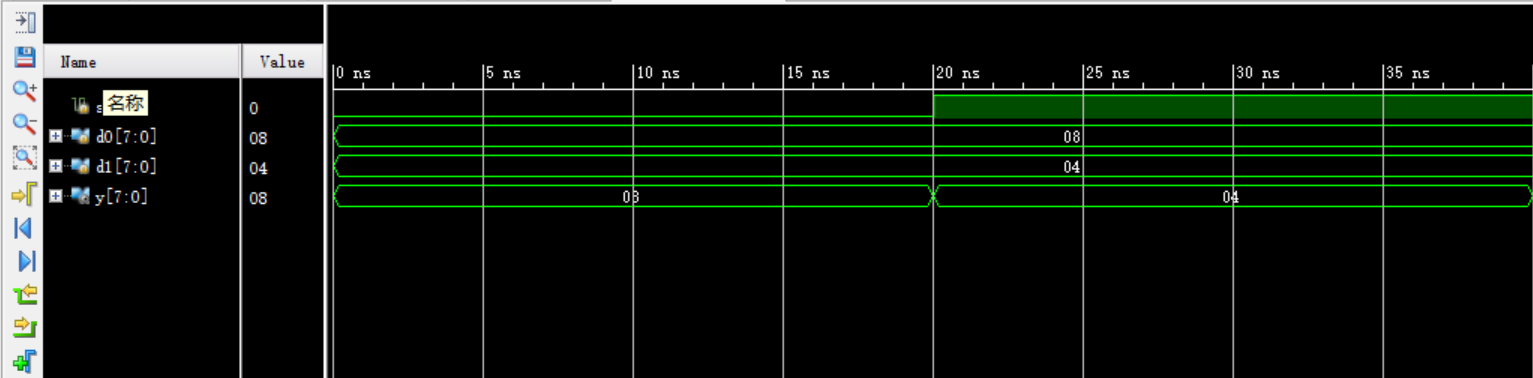


**3.module mux2**

d0=8，d1=4，s的初始值为0，每过20个时钟周期，对s取反

**代码和仿真图：**

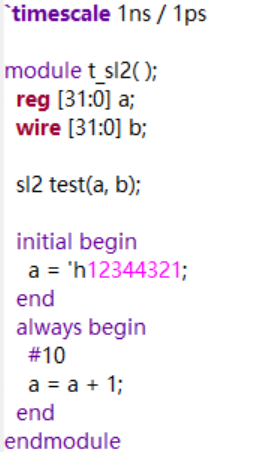


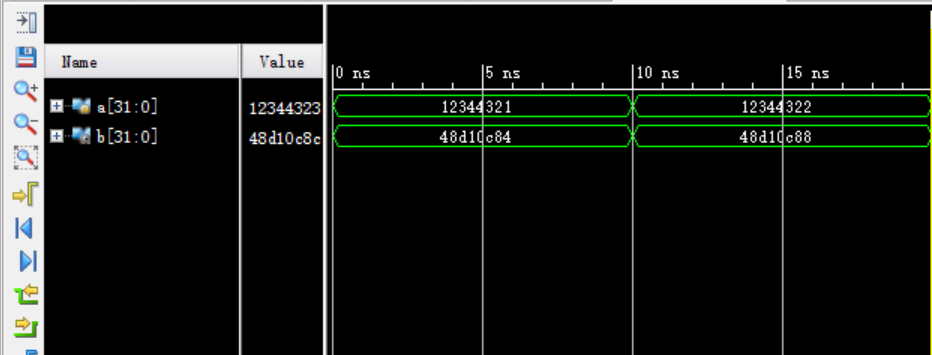


**4.module sl2**

设置a初始值为0x12344321，每过10个时钟周期对其加一，看左移两位生成的b随a的变化

**代码和仿真图：**

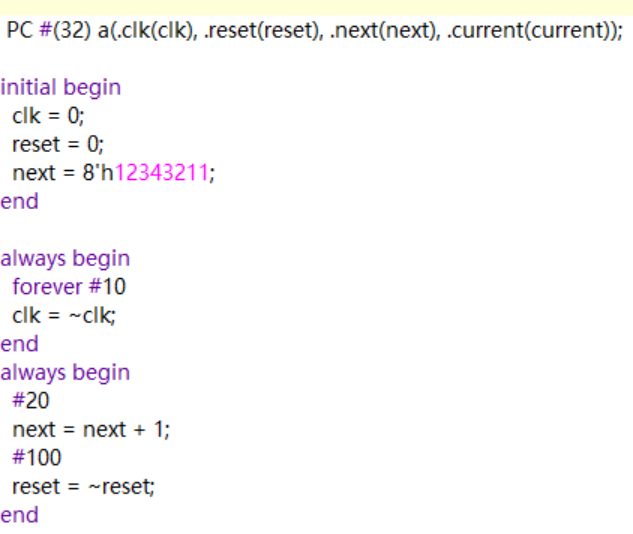


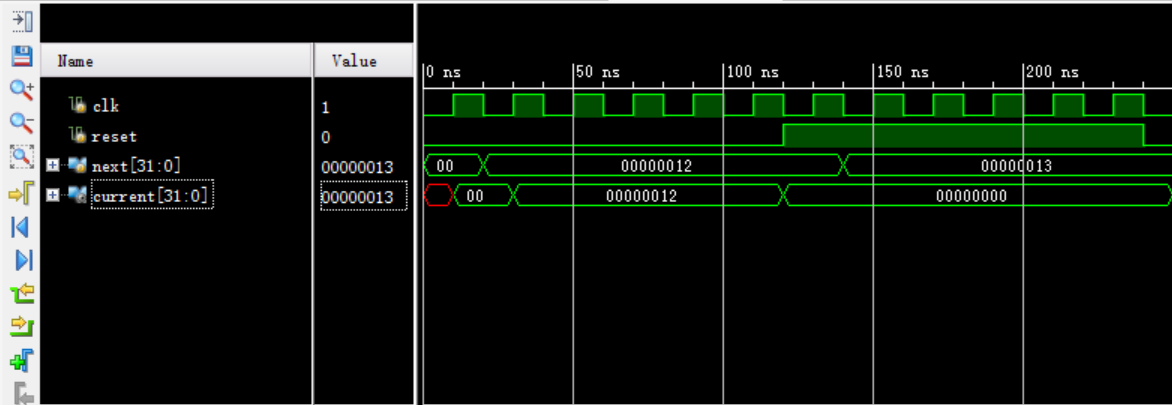


**5.module PC**

初始状态下，clk=0，reset=0，next=0x12343211，每过10个时钟周期，对clk取反，每过20个时钟周期，对next加一，100个时钟周期对reset取反

**代码和仿真图：**

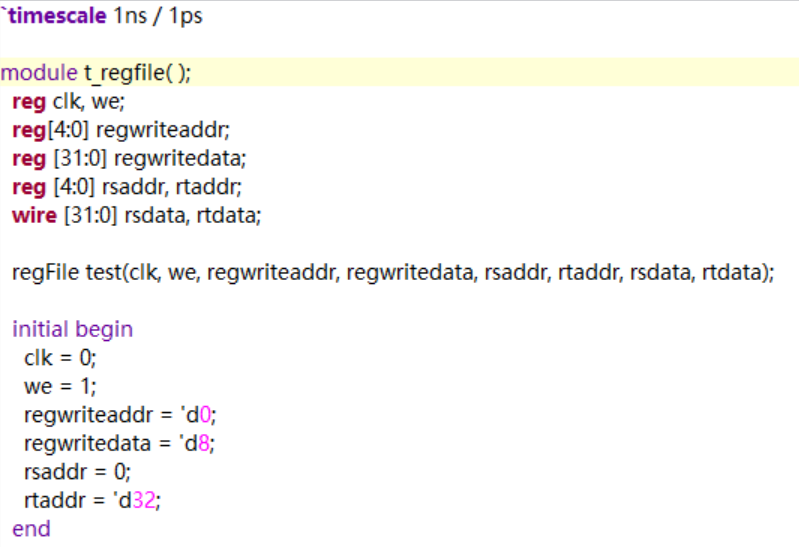




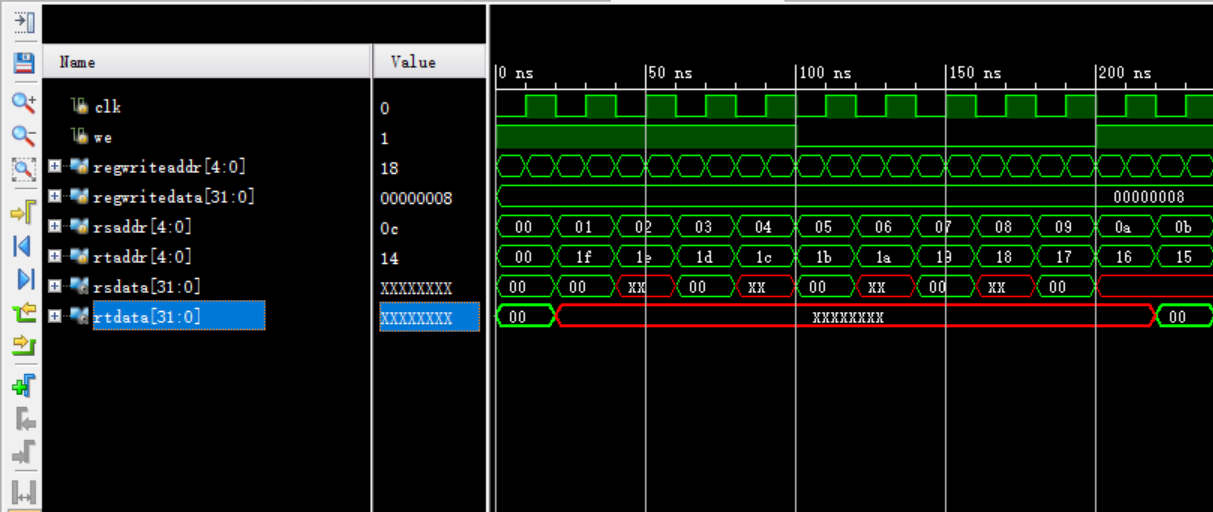
**6.module regFile**

通过对参数的周期性变化，测试了可写、可读的不同组合情况。

**代码和仿真图：**



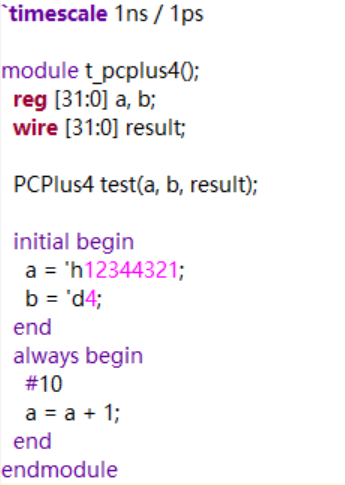


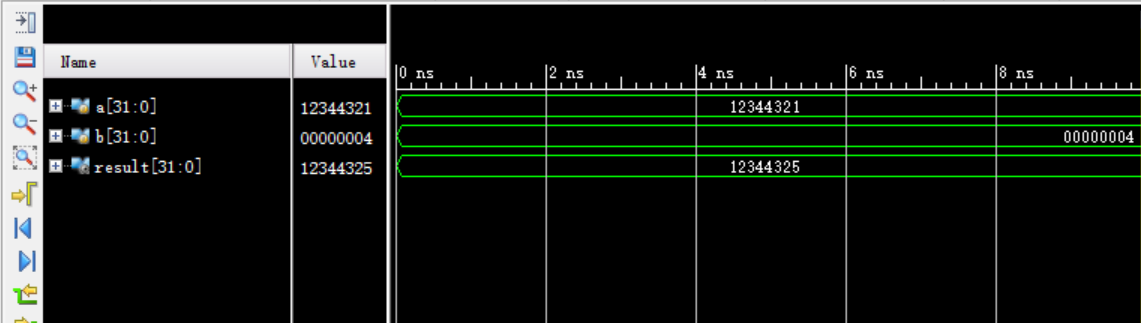


**7.module PCPlus4**

对a和b赋初值，每过10个时钟周期对a加一，查看result的变化

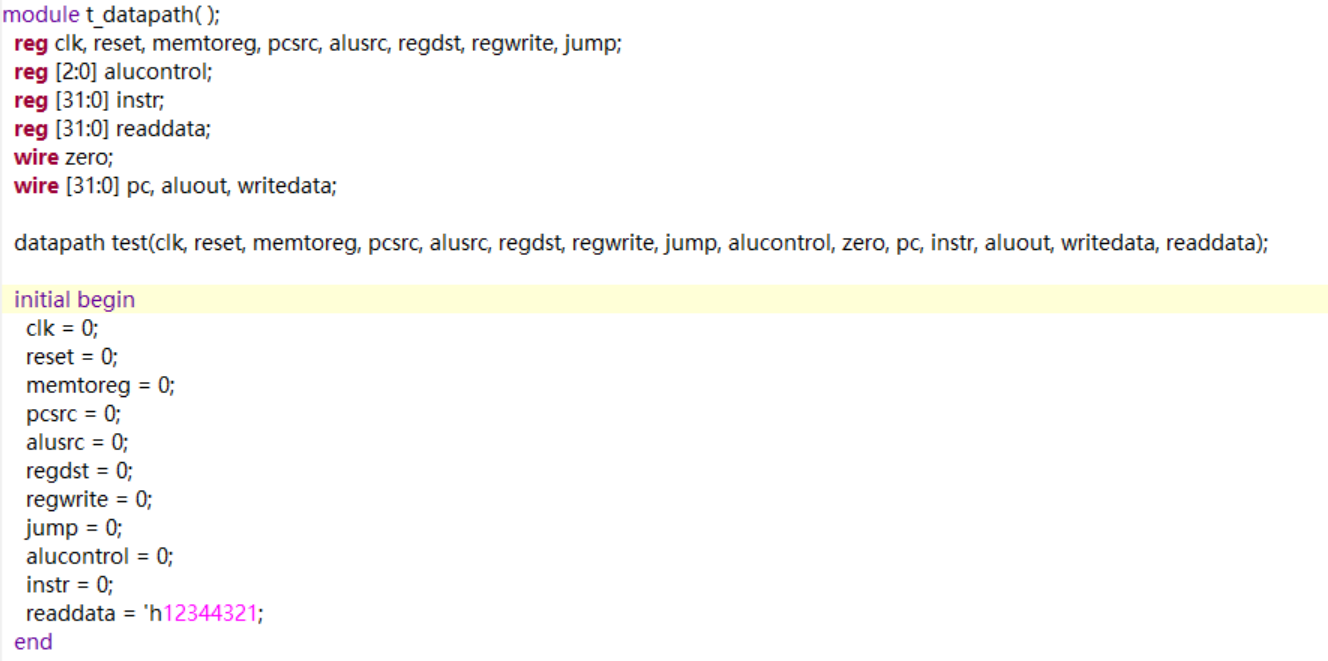
**代码和仿真图：**



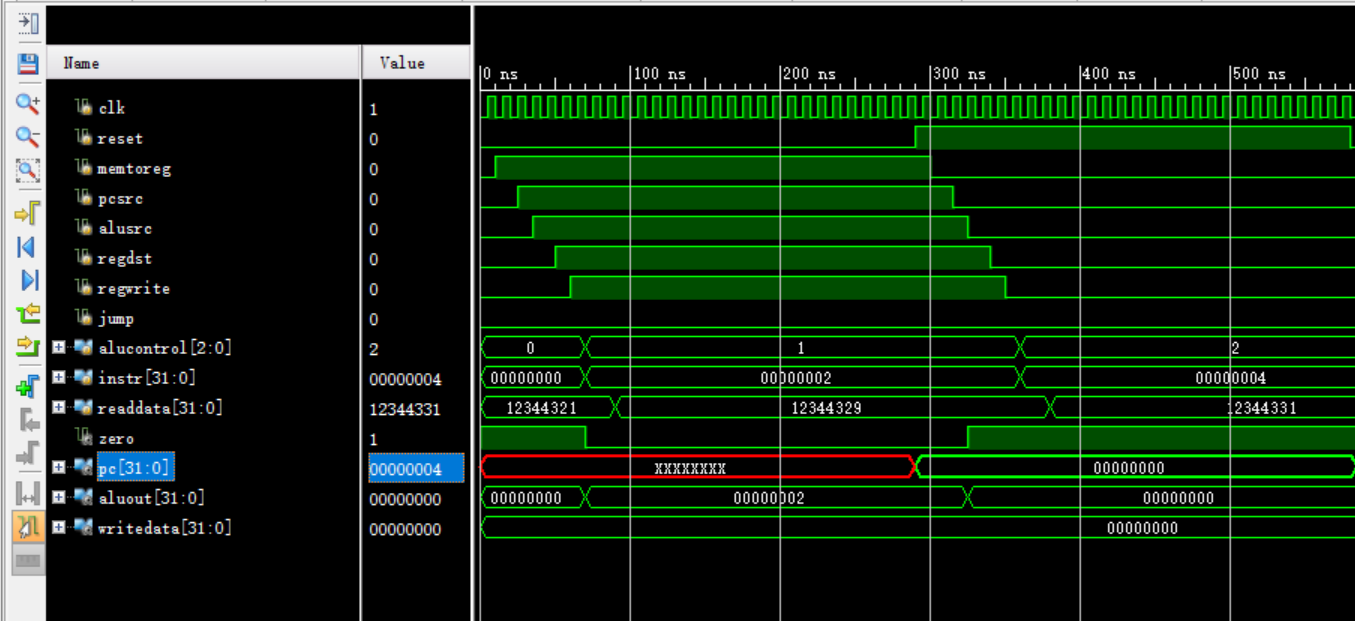


**8.module datapath**

datapath的输入主要是一些使能标志，在测试中，经过特定的时间对这些位取反，具体的测试用例在代码中很清晰，不再赘述：

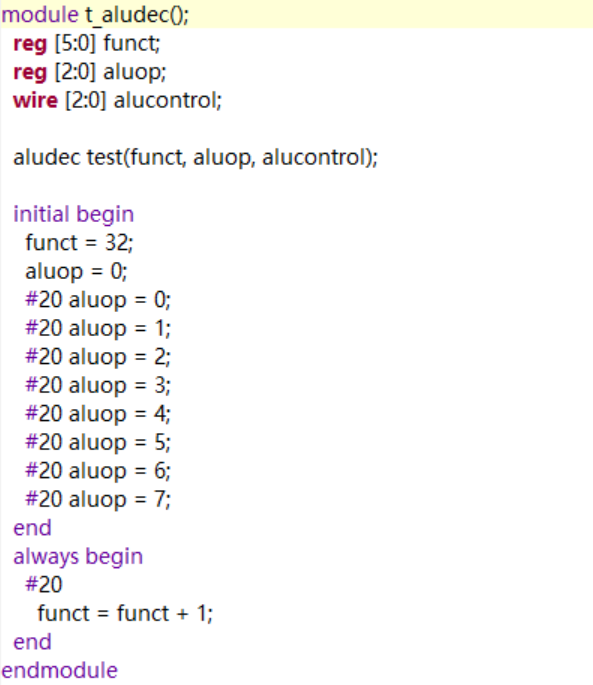


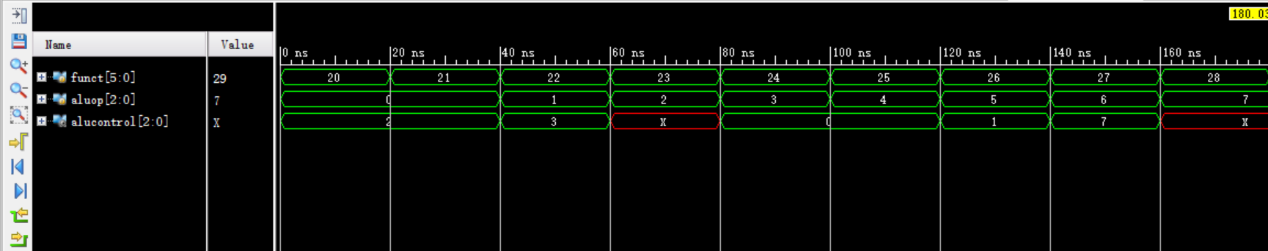




**9.module aludec**

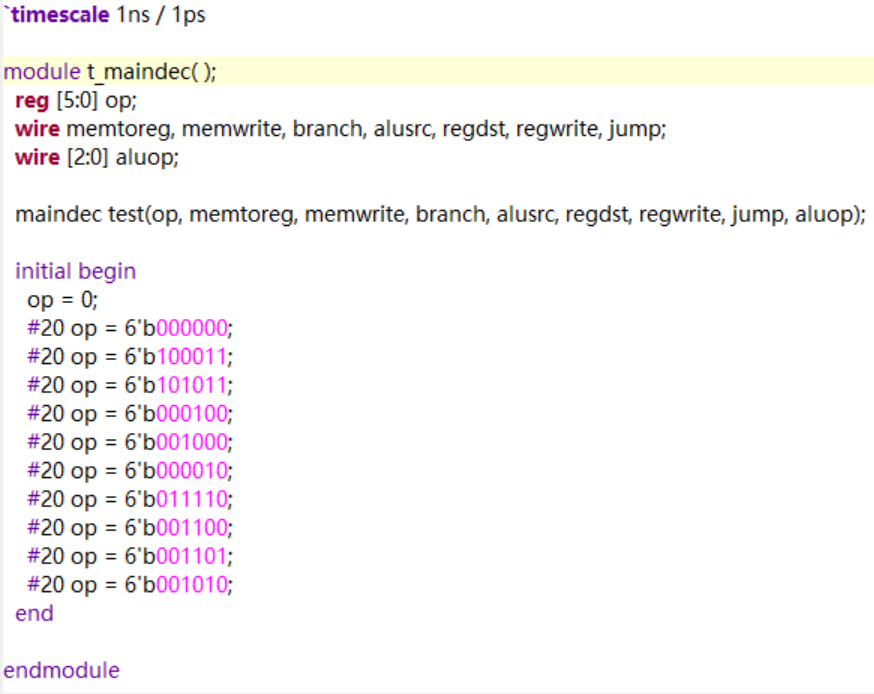
测试aluop为0-7的对应的不同alucontrol：

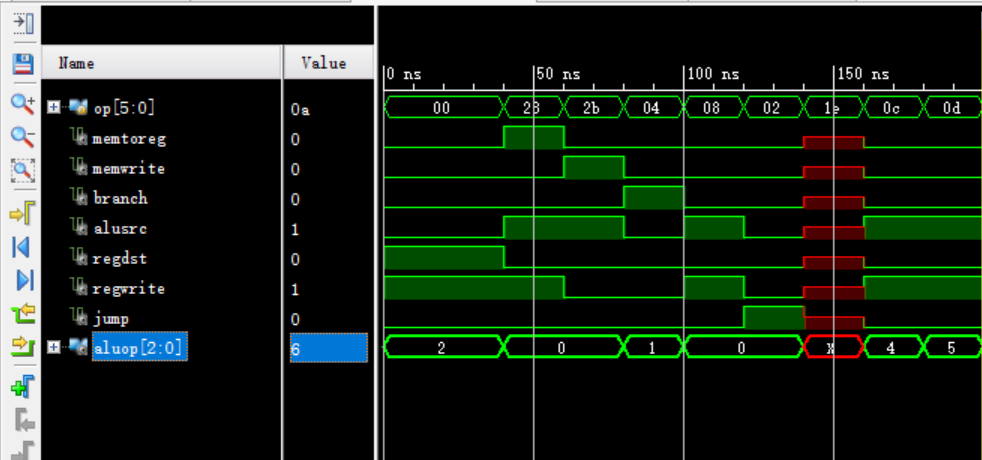




**10.module maindec**

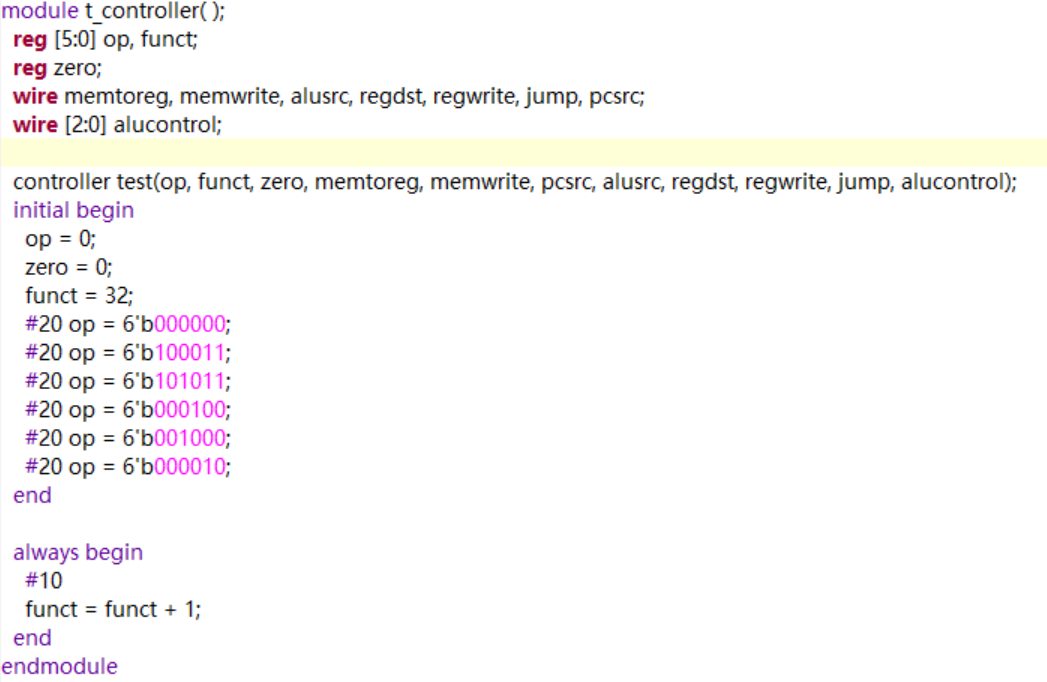
测试需求文档中不同的op对应的控制位输出：

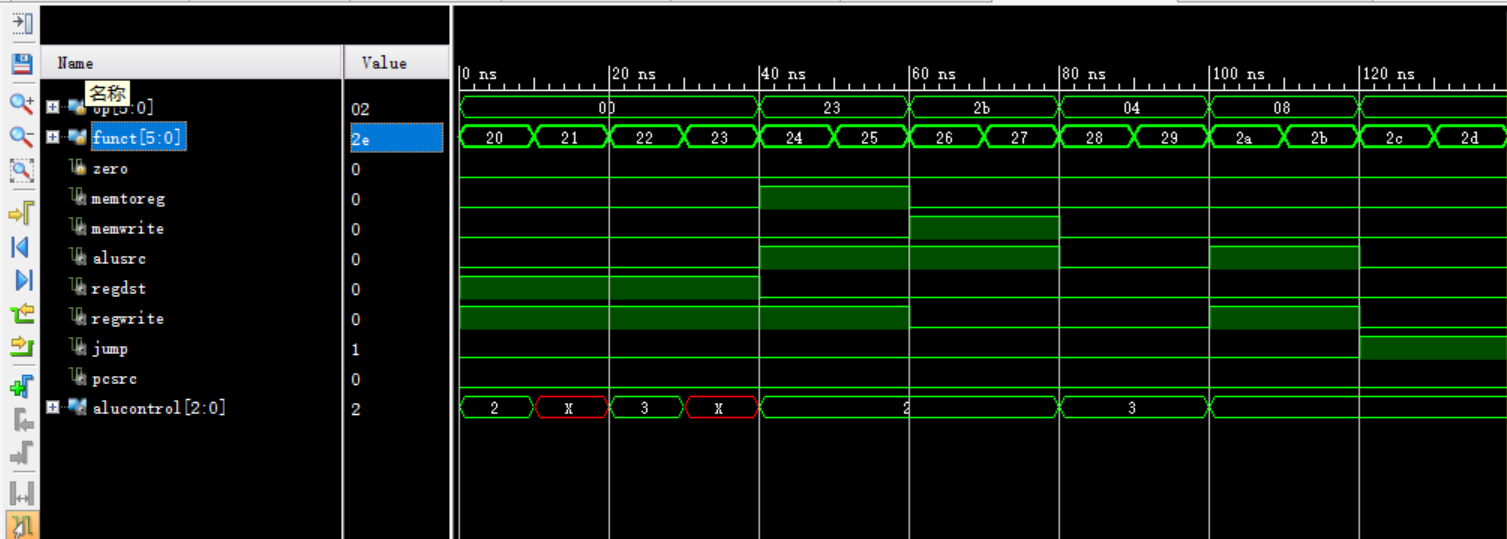




**11.module controller**

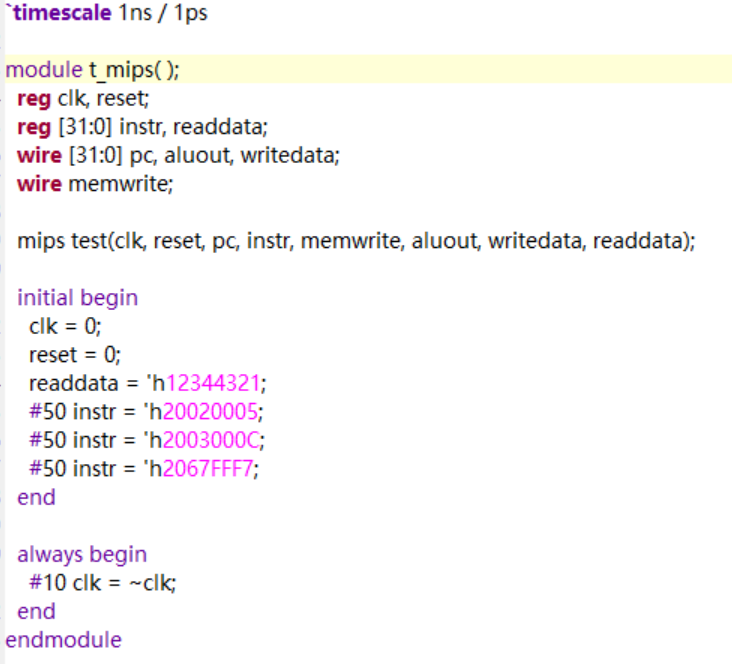
测试op不同，funct不同情况下，不同的控制位输出

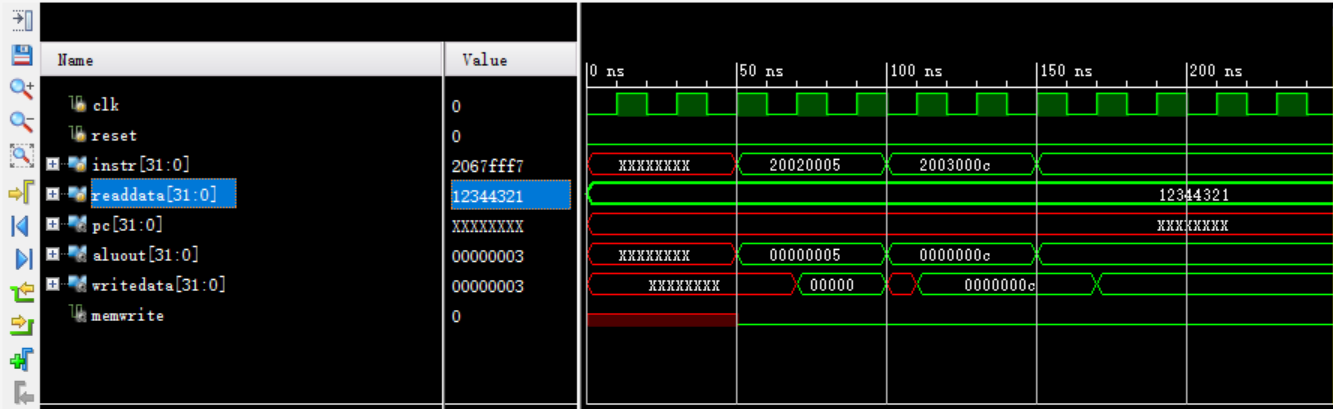




**12.module mips**

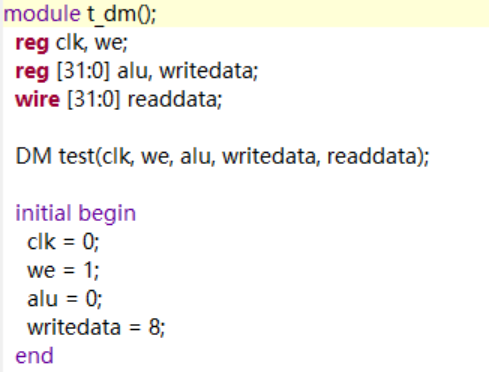
取不同的指令，并且每过10个时钟周期对clk取反

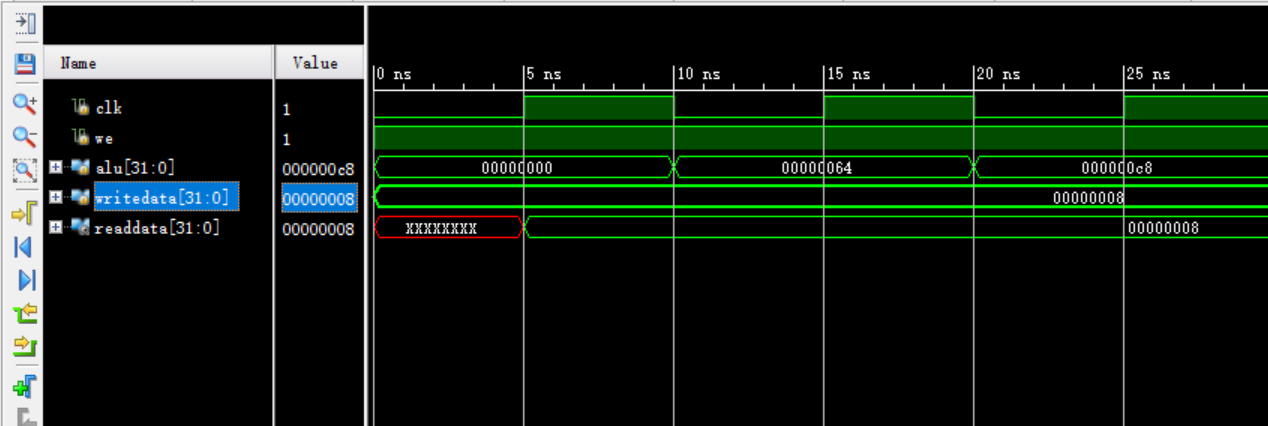


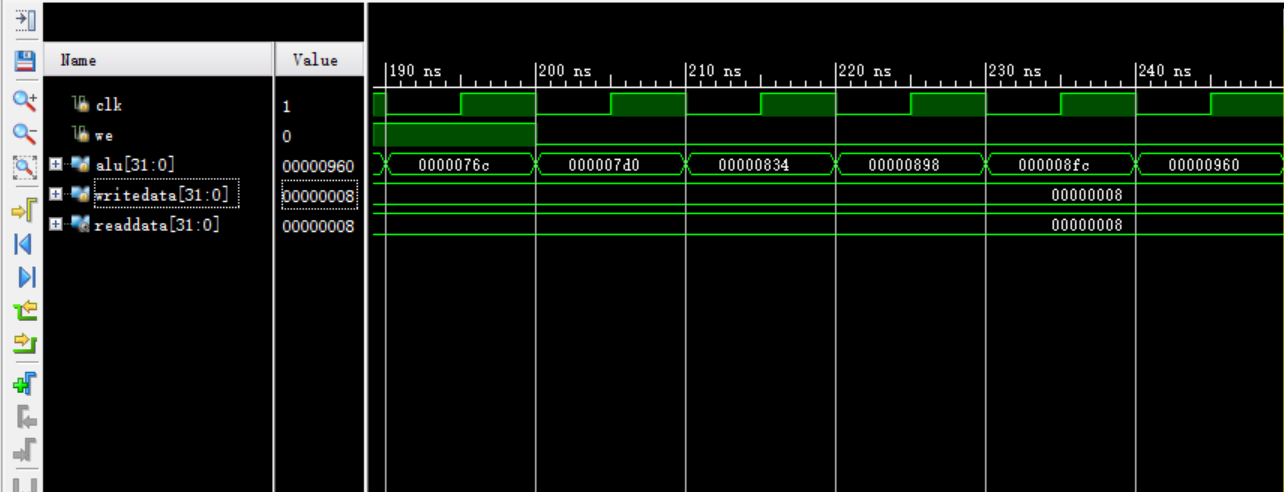


**13.module dm**

尝试将8写入内存并读出来

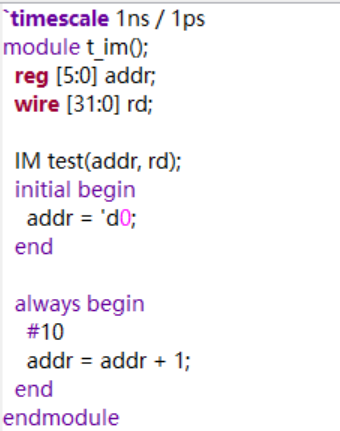






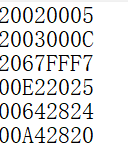
**14.module im**

测试将memfile.txt中的指令加载到内存中



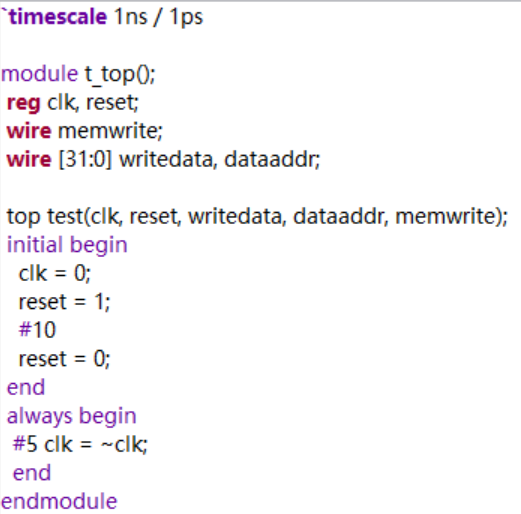


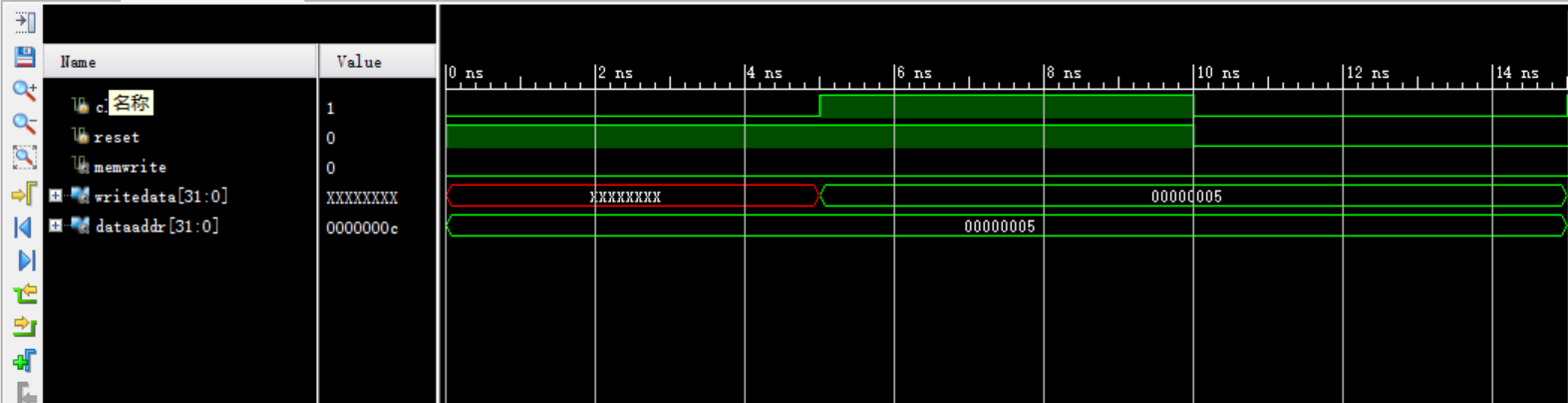
**memfile.txt：**



**15.module top**

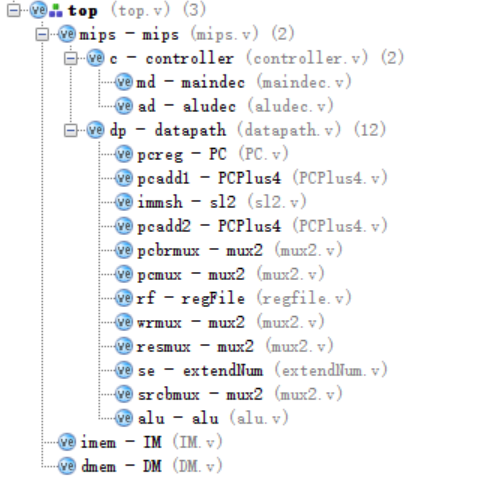
主要测试reset和clk：





**四、实验总结：**

**1.本次实验结合了上课所讲的设计方法和文档中给出的设计参考，采用了层次化的设计，这点从项目的目录结构也能看出来：**



**每一部分都是分为了数据层和控制层，数据层包括datapath、DM和IM，功能分别是数据进行处理、从内存加载或读取数据和将磁盘文件中的指令加载到内存中；控制层包括controller，主要负责读取指令之后的处理，协调各个功能单元完成功能。**

**2.本次实验中涉及的许多功能单元的设计都是之前LAB中做过的，此次LAB主要实现的就是如何利用控制单元协调这些单元工作，利用层次化设计，不仅使设计更为简单，也让实现的过程更加的清晰明确**

**五、实验中的收获与感受：**

**1.通过此次实验，更加深入的了解了CPU是如何工作的**

**2.对verilog语言运用的更加自如，比如在此次实验中用到了新的数据类型parameter**

**3.对数字部件设计的思想、流程有了更加深入的体会**

**六、对实验的反思：**

**1.我所实现的CPU基本上是按照需求文档的标准来实现的，没有做过多的优化，但是根据之前ICS课上以及本学期CSE课上所讲到的内容，我联想到是否可以利用流水线来优化CPU的性能，但是因为时间原因，没有具体的实现**

**2.此次实验，代码部分很早就完成了，但是文档迟迟没有动工，这就导致写文档有些不流畅，也耽误了老师和TA的时间，我对此深刻检讨**