**计原大实验报告**

组号302

组员：肖迪，杨煜，张艺庆

**系统整体测试**

经过测试，系统可以正常的使用，可以在我们做出来的CPU上运行监控程序，监控程序A，R，D，U，G各项命令运行正常。 在单步运行的时，通过VGA接口，显示屏幕上可以显示当前刚刚从指令寄存器取出的指令汇编代码（经过解码的），ps2小键盘运行也是正常的，可以通过小键盘的不同的按键选择不同的CPU运行的频率。系统整体通过了助教的检查验收，我们也拍摄了视频录制了系统正常运行的状态。

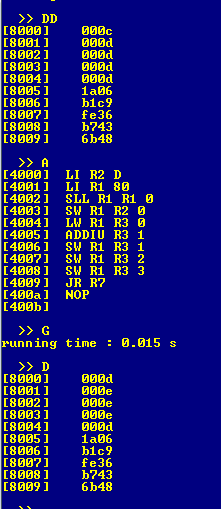
**系统运行速度测试**

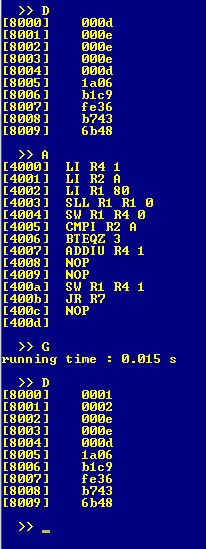
我们小组的CPU运行主频是12.5MHz，在这个频率下，运行助教的提供的五段测试代码，运行的时间如下表：

|  |  |
| --- | --- |
| 测试代码 | 运行时间（s） |
| 1 | 16．0 |
| 2 | 22．0 |
| 3 | 12．0 |
| 4 | 18．0 |
| 5 | 12．0 |

**系统编程测试**

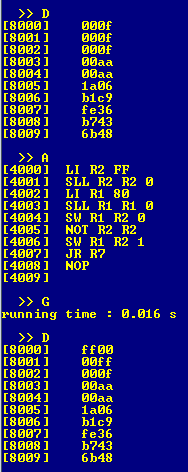
能够成功的运行监控程序，这就说明系统基本正确，但还有一些可能出现的冲突情况我们没有测试，另外还有监控程序中没有的扩展指令我们没有测试。

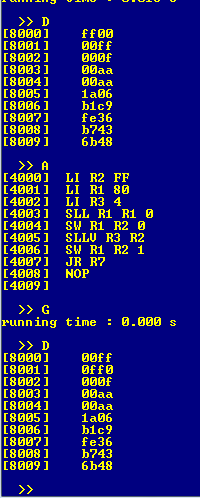
左图是**测试“load后使用”**这个数据冲突的，这个数据冲突在流水线中必须要加nop作为气泡，[4004]的load指令的目标地址是R3，然后下一条指令就要读R3，这就会产生一个冲突，从[4006]开始三条指令就是将R3的值写入[8001]开始的三个地址，可见，程序正确处理了这个情况，写入的值是加1之后的新值。

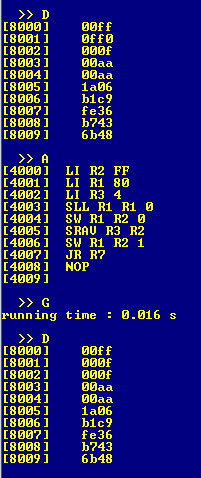
左图是**测试转发单元**，和**延迟槽**的代码：

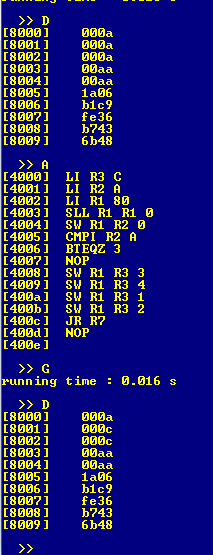
若前一条指令的目标寄存器是后一条指令的源寄存器，就会产生数据冲突，必须使用转发，在[4002]和之后一条指令之间就存在这样的关系。

延迟槽是分支和跳转指令后面的一条指令，无论分支是否执行，这条指令都要被执行，[4006]这条分支指令之后的一条指令就是延迟槽，虽然这里分支指令是被执行了，但是延迟槽中的指令照样要被执行，R4的值要被加1，所以我们看到[8001]和[8000]内存地址的结果就是不一样的，[8001]的结果被加了1，说明延迟槽是正常工作的。

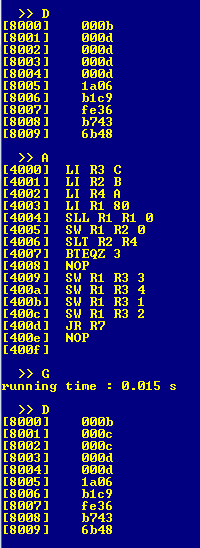
左图是测试我们302小组的第一条**扩展指令NOT**的，我们小组第一条扩展指令是NOT，测试代码中，将寄存器R2的值设置为0x00FF，然后取非，可以看到，结果变成了0xFF00，我分别将结果储存在了[8000]开头的内存中，这将是之后测试所将采用的模式。

左图是第二条**扩展指令SLLV**, 测试代码中，将寄存器R2的值设置为0x00FF，然后左移4位，结果变为0x0FF0，我分别将结果储存在了[8000]开头的内存中。

左图是测试**第三条扩展指令SRAV**的，和上一条类似，将寄存器R2的值设置为0x00FF，然后右移4位，结果变为0x000F，我分别将结果储存在了[8000]开头的内存中。



左图是测试**扩展指令CMPI的**，本条指令是比较一个寄存器和另外一个立即数的值，然后根据比较结果改变标志寄存器的T。本条指令应该结合分支指令使用，由于R2的值是0xA，而立即数的值也是0xA，所以CMPI运行之后的结果是将标志寄存器置为0，下一条[4006]的跳转指令就会产生跳转，跳转到的位置是[400a]，跳过了[4008] 和[4009]的两条将内存修改成c的指令，结果只是将[8001]和[8002]两个内存地址修改成了c，而不是不发生跳转的4个。



类似的，左图是测试**扩展指令SLT**的，这条指令也是一条根据不同的条件而改变标志寄存器的值。在[4006]的SLT指令比较了两个寄存器R2和R4的值，然后置标志寄存器为0，下一条[4007]的跳转指令就会产生跳转，跳转到的位置是[400b]，跳过了[4009] 和[400a]的两条将内存修改成c的指令，结果只是将[8001]和[8002]两个内存地址修改成了c，而不是不发生跳转的4个。

**控制信号设计**

在本次实验中，我们一共设计了14个控制信号，来控制数据通路上的十四个多路选择器选择合适的数据，我们设计控制信号和多路选择器的方法是渐进迭代的，首先，我们首先仿照书上设计出一个初版本的数据通路，然后将我们要实现的扩展指令一条一条的模拟在数据通路上的运行过程，并且给出相应的控制信号，如果现有的多路选择器能够满足实现这条指令的要求，那就继续下一条指令，只有当出现一条指令，现有的数据通路实在无法实现的时候，我们才在数据通路上添加一个新的多路选择器来满足新的指令的要求，并且我们力求做到使用最少的硬件资源来完成这件事情。14个控制信号分别是imSelector，ALUSrc2，memWrite，memRead，regDst，branch，regWrite，memToReg，op，readSpecReg，writeSpecReg，jump，ALUSrc1，rxToMem。每条指令所对应的控制信号我们在controllerDesignFiles文件夹中的controlSignal.给出了，而每个控制信号每一位都代表什么含义，是起到什么控制作用的，我们在controllerDesignFiles文件夹中的controlBitsMeaning.xls.给出了。在这里不加赘述。

这里以一条典型指令来说明设计情况：

ADDIU rx immediate

该指令将立即数的值与寄存器rx的值相加，值再存回rx寄存器。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| instruction | imSelector | ALUSrc2 | memWrite | memRead | regDst | branch | regWrite |
| ADDIU | '1000' | '01' | '00' | '00' | '00' | 0 | 1 |
|  | memToReg | Op | readSpecReg | writeSpecReg | jump | ALUSrc1 | rxToMem |
|  | 0 | '0000' | '00' | '00' | 0 | '00' | x |

imSelector：由于该指令是从[7:0]位取出立即数，然后进行符号扩展，所以控制信号是‘1000’

ALUSrc2：由于该指令设计立即数，所以ALU的第二个操作数需要读取经过符号扩展的立即数

memWrite，memRead：这条指令既不读也不写寄存器，所以这两个都是‘00’

regDst：由于该指令写回的寄存器是依靠rx控制的，所以regDst为‘00’

branch，jump：这条指令不是分支指令或者跳转指令，所以jump和branch都为0

regWrite:这条指令需要写寄存器堆，所以为1

memToReg：这条指令不是装载指令，不需要从内存中读出数据写入寄存器，所以为0

Op：这条指令是加法操作，所以ALU需要执行加法操作。

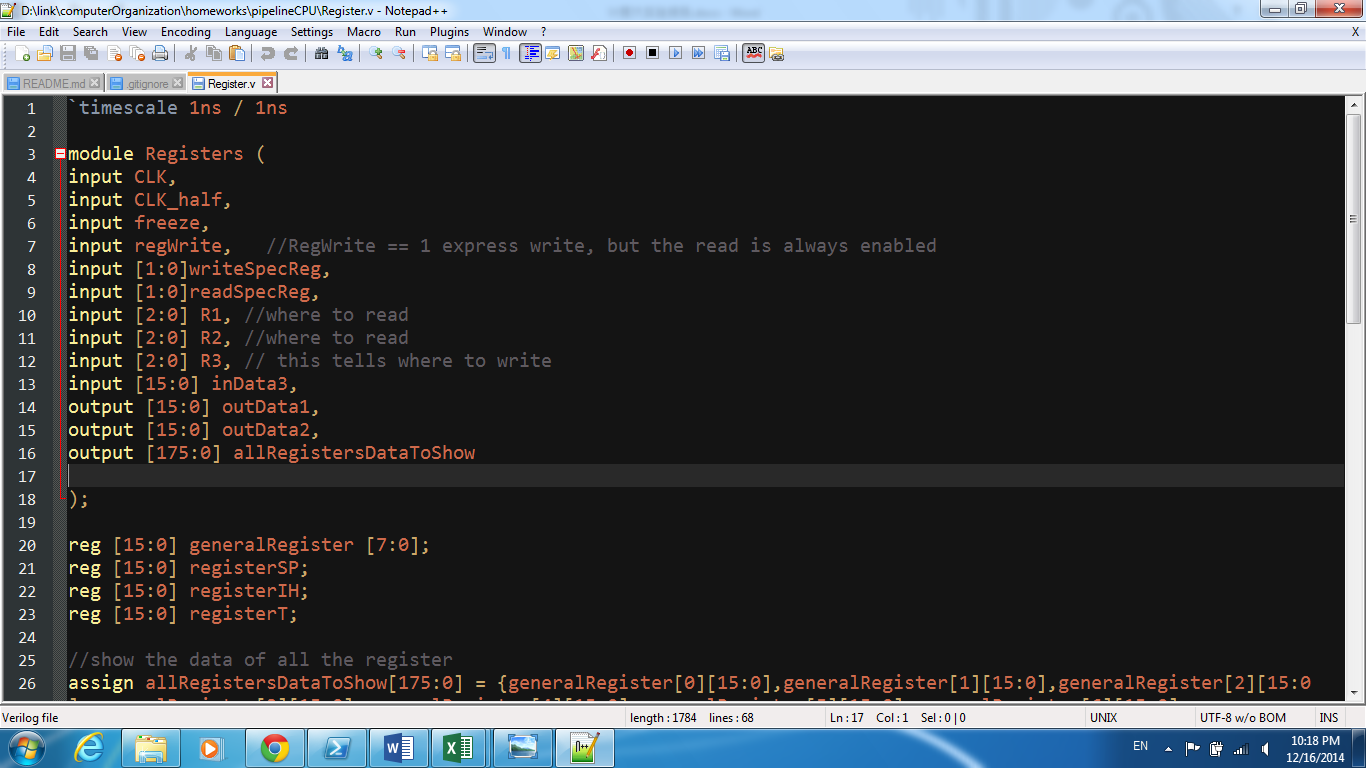
readSpecReg,WriteSpecReg: 这条指令是从通用寄存器而不是特殊寄存器中读写的，所以都为‘00’

ALUSrc1:ALU的第一个操作数是从正常的寄存器堆的第一个输出中读入的，所以是‘00’

rxToMem：不需要写内存所以这个控制信号无所谓。

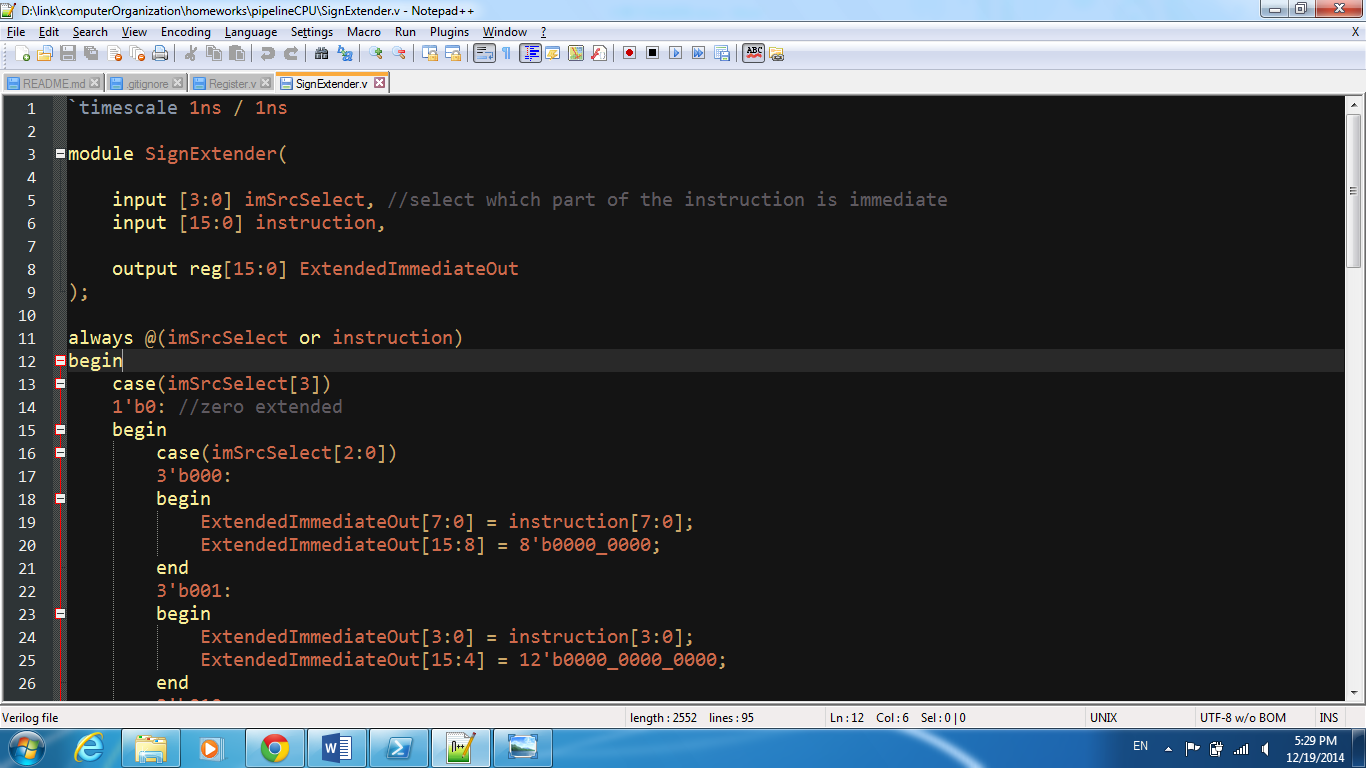
**主要模块设计**

**寄存器堆模块**



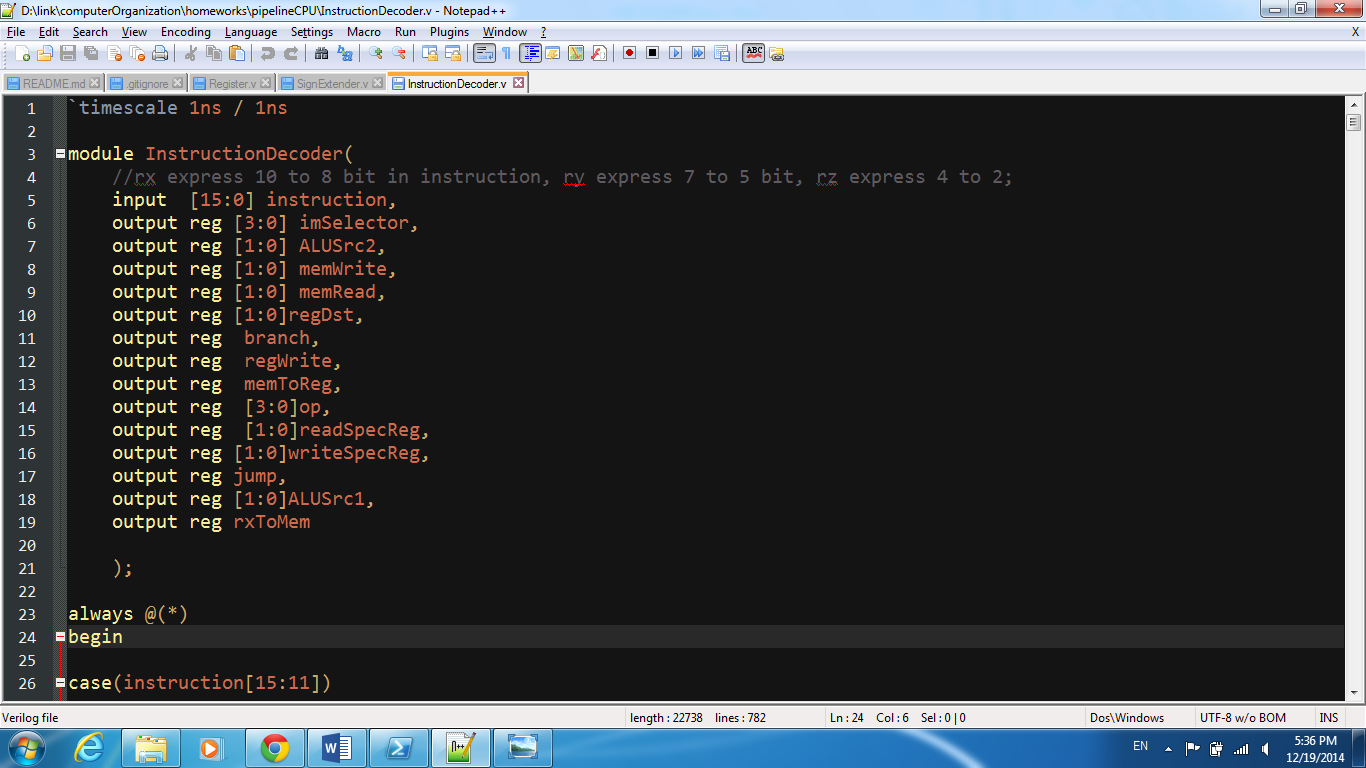
该模块实现了包括8个通用寄存器和3个特殊寄存器（T，SP，IH），每一个寄存器使用了verilog语言中的16位reg类型实现，电路本质就是触发器，读寄存器的值是通过组合逻辑读出的，所以outData1和outData2实时反应的就是最新的寄存器的值，只不过需要利用R1，R2和readSpecReg的值来确定是否要读取，同时确定读取的是哪个寄存器的值。而写入操作只是发生在主频时钟的下降沿的时候，由regWrite，writeSpecReg来确定是否要写寄存器，由R3确定写入的是哪个通用的寄存器。而读出的数据就通过outData1，和outData2来读出。

**符号扩展器模块**



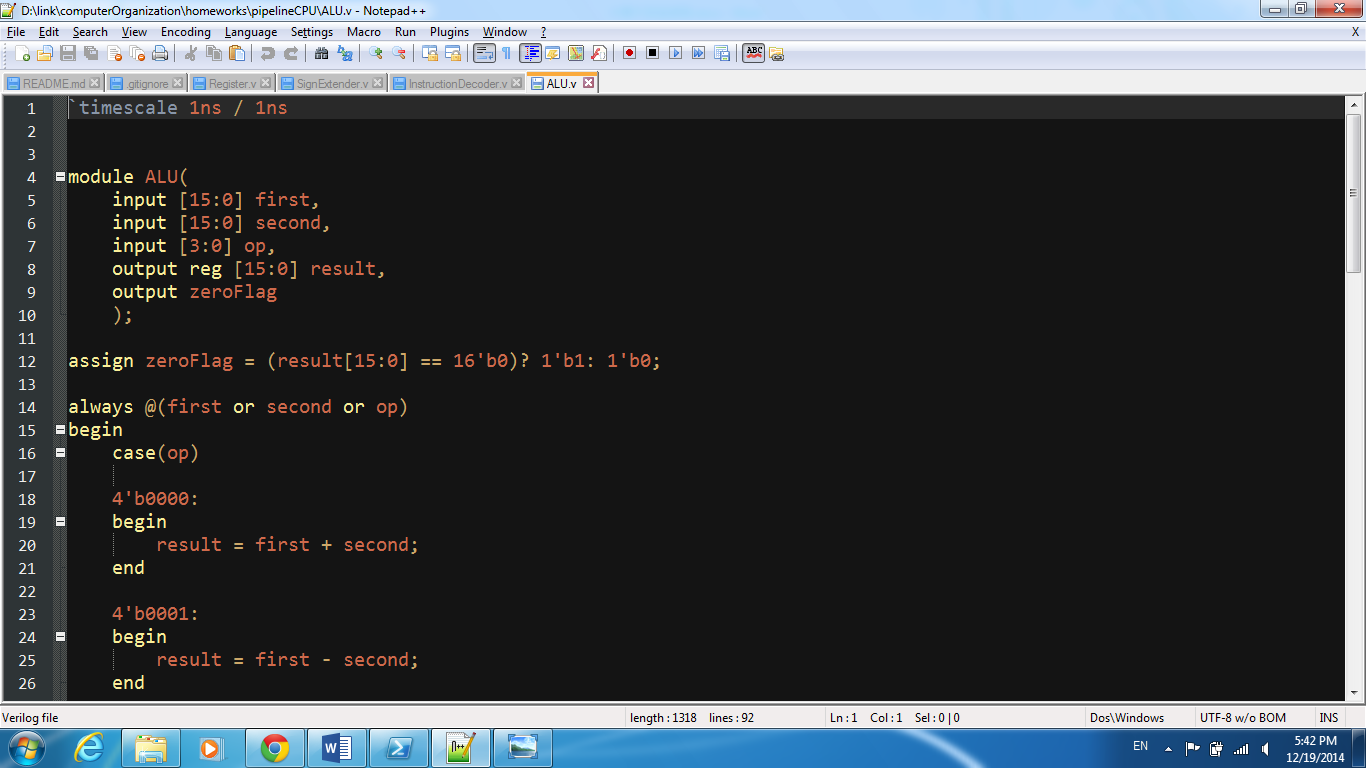
由于THUMIPS的指令集中的指令立即数可能出现的位置和位数有很多种可能，符号扩展器相对于标准MIPS来说相对复杂，但也是归根结底还是简单的组合逻辑。该组合逻辑通过解码器产生的控制信号imSrcSelect来选择和扩展指令中可能存在的立即数，扩展的方式由符号扩展和0扩展，模块的输出是16位的经过适当扩展的立即数。控制位相应的意义和对应的功能请详见controllerDesignFiles/controlBitsMeaning

**解码器模块**



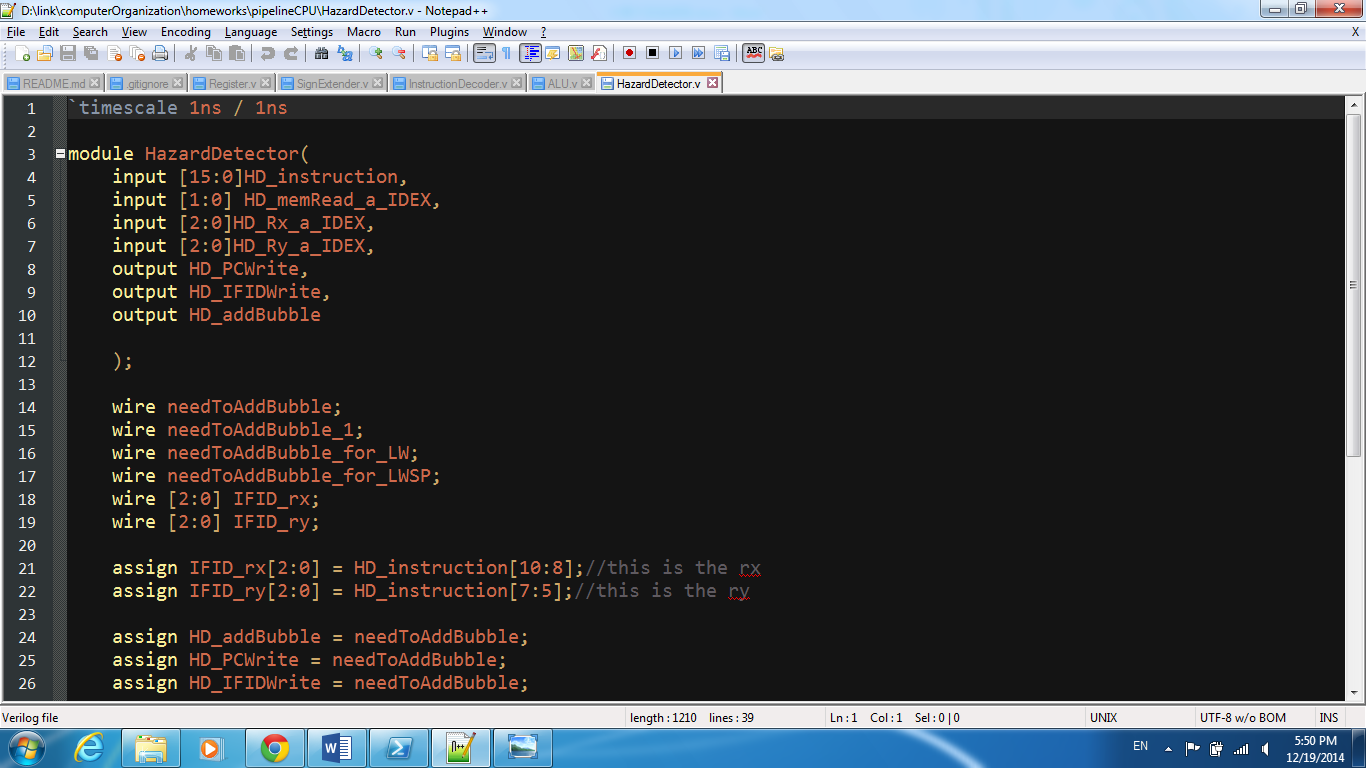
解码器模块虽然代码量比较大，但是逻辑十分清楚，是一个组合逻辑，输入是一个16位的指令，输出是该条指令相对应的14个控制信号，我们使用了verilog常用的switch-case语句来实现这个逻辑。相应的每一条指令对应的控制信号可以在controllerDesignFiles/controlSingal查到。实现这个控制器是一项比较繁杂的工作，我们在实现的额时候采用了一个人编码，另外一个人复核的实现策略，果然复核的时候检查出了上一个人的实现中的一些问题。

**ALU模块**



ALU是整个CPU的核心，但是实现起来却比较简单，他的功能就是根据操作码以不同的方式对两个操作数first和second进行操作，将结果输出到result里面，同时，若result等于全0，将zeroFlag设为1。在整个数据通路的设计中，我们为了使得总体的设计简单，稍微将ALU的可能进行的操作复杂化了。比如说，ALU中有一项看起来很奇特的操作，将输出的结果设置为第一个操作数first，也就是说操作数直接通过ALU，没有进行任何操作。这样做的目的是简化外部顶层逻辑的设计，因为有些指令，不需要ALU进行操作，本来我们可以在顶层通路的设计中加一条直接链接的线，然后在结果处加一个多路选择器，但是这样会使得顶层逻辑变得复杂，于是我们就没有这么做。这里的设计思想其实是某种平衡，若模块设计适当复杂，顶层逻辑就可以适当简单，顶层通路已经十分负责了，所以我们就希望通过子模块来实现顶层模块的功能，来降低顶层模块设计的难度。

**冒险检测单元**



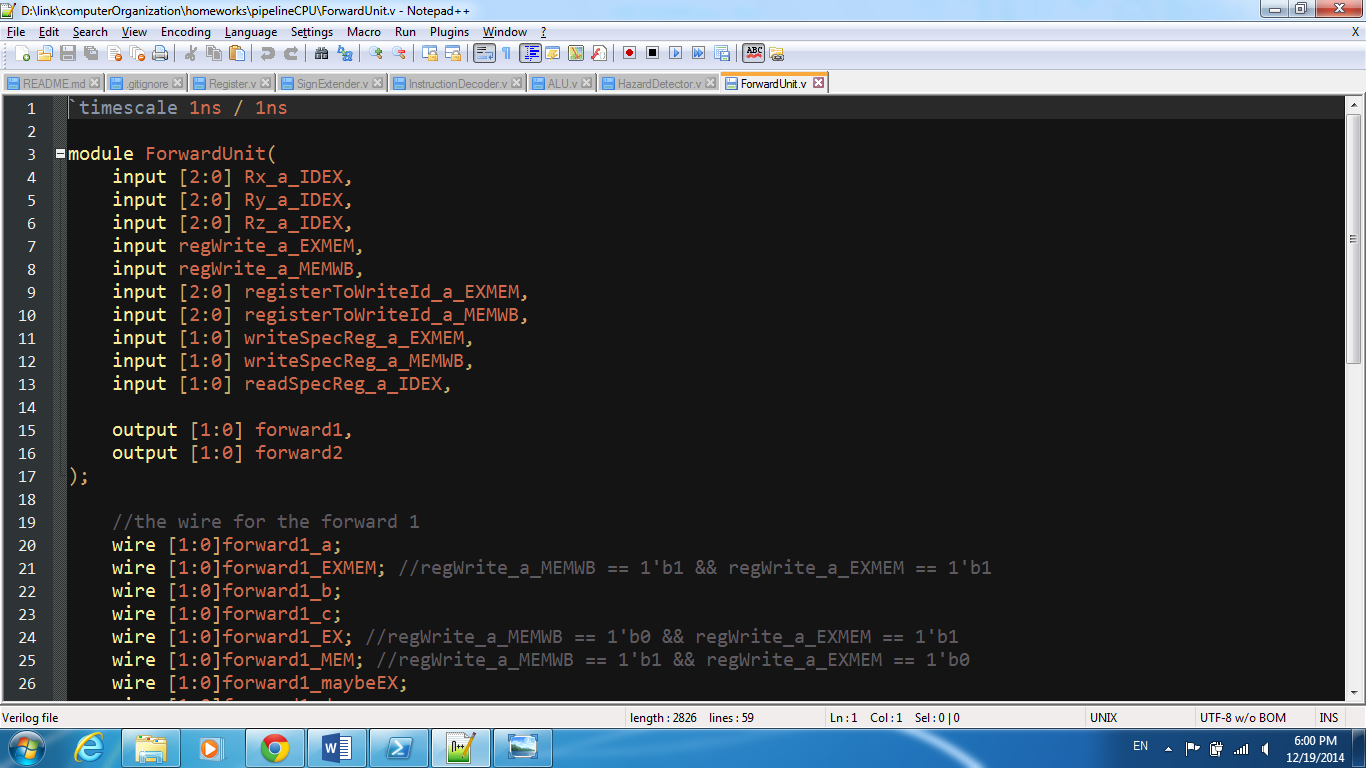
由于流水线的存在，若前一条是装载指令，后一条是指令需要用到装载指令的目标地址的值，那么这时，装载指令还没有执行，在时间上通过转发解决这个问题是不可能的，所以我们必须通过插入气泡（NOP指令）的方式来在事实上暂停流水线一个周期。形式化的表述这种情况的发生是：

HD\_memRead\_a\_IDEX &&（HD\_Ry\_a\_IDEX[2:0] == IFID\_rx[2:0] ||HD\_Ry\_a\_IDEX[2:0] == IFID\_ry[2:0]）

表示的意义就是（前一条指令是装载指令&& 装载的目的地址==本条指令的任意一个源地址）

插入气泡的方式是通过将新解码出来的指令变为NOP（所有控制信号变为0即可），然后保持PC值和已经读出到IF/ID流水寄存器的的值就可以。其中，HD\_PCWrite是用来保持的PC的，HD\_IFIDWrite是用来保持IF/ID流水寄存器的，HD\_addBubble是用来将指令变为NOP的。

**转发单元**



转发单元的出现也是由于使用了流水线本身，假如前一（二）条指令是一条需要写寄存器堆的指令，后一（二）条指令刚好要读寄存器堆，如果他们写和读的地址是同一个位置，就会发生问题，因为写指令是在最后一个阶段WB才写的，下一（二）条指令没有办法及时读取到正确的值，所以就需要通过旁路将正确的值及时的转发过去由于转发的情况较多，形式化的描述请见代码，非形式化的描述如下：

如果（前一或者二条指令是需要写寄存器的）&&（（前一条指令的写入的地址==后一条指令读取的地址）||（（前一条指令的写入的地址！=后一条指令读取的地址）&&）（前两条的指令的写入地址==后一条指令的写入的地址）），那么就使用相应的转发。

之所以有这么多复杂的条件，是因为不但前一条指令满足目的地址等于源地址的条件需要转发，前两条指令也需要转发。而且还会出现前一条和两条指令都满足转发条件的情况，这个时候需要转发最近的也是最新的那条。

转发单元虽然内在逻辑较为复杂，但是本质上还是一个组合逻辑，控制的输出也只有两个，分别命名为forward1，和forward2，分别控制ALU的两个操作数的第一次选择，他们选择ALU的操作数是通过从寄存器堆的结果读取出来还是通过转发得到。

**实验感想**

**汇编代码测试**

为了确保我们代码的正确性，在调通监控程序可以正常通过A，G指令执行代码之后，我们小组编写了一些汇编程序对程序进行了测试，测试程序在testingFiles中，我们不但测试了我们扩展指令的正确性，我们还测试了我们有没有正确的处理延迟槽，转发等。我们感觉，这样的测试是十分必要的，在测试中我们发现了一些问题，比如说原来我们没有正确实现算数右移的指令，因为在verilog语言中，只是使用>>>还不能实现算数右移，还需要我们将寄存器改为signed类型的才可以。通过测试，我们还增强了对我们自己设计的系统的信心，发现可以在我们设计的系统上跑写出来的几乎所有的程序的时候，当时心情是十分的兴奋。