Verilog设计流水线CPU

一、支持的指令集：

{LB、LBU、LH、LHU、LW、SB、SH、SW、ADD、ADDU、SUB、SUBU、 MULT、 MULTU、 DIV、 DIVU、 SLL、 SRL、 SRA、 SLLV、SRLV、SRAV、AND、OR、XOR、NOR、ADDI、ADDIU、ANDI、ORI、XORI、LUI、SLT、SLTI、SLTIU、SLTU、BEQ、BNE、BLEZ、BGTZ、  
BLTZ、BGEZ、J、JAL、JALR、JR、MFHI、MFLO、MTHI、MTLO}

二、主要模块：

|  |  |  |  |
| --- | --- | --- | --- |
| PC | I | clk | 时钟信号 |
| I | rst | 同步复位（清零） |
| I | en | 使能信号（暂停） |
| I [31:0] | nextPC | PC要跳转的值 |
| O [31:0] | PC | PC的值 |
| IM | I [31:0] | A | 要读取的指令地址 |
| O [31:0] | D | 读出的指令 |
| IM\_RF | I | clk | 时钟信号 |
| I | rst | 同步复位（清零） |
| I | en | 使能信号（暂停） |
| I [31:0] | nextInstr | RF阶段指令更新 |
| I [31:0] | nextPC | RF阶段PC更新 |
| O [31:0] | Instr | RF阶段指令 |
| O [31:0] | PC | RF阶段指令对应的PC |
| instrDecoder | I [31:0] | instr | 要识别的指令 |
| O | isInstr | 相应的识别结果 |
| RF | I | clk | 时钟信号 |
| I | rst | 同步重置 |
| I [4:0] | rsAddr | 要读的寄存器 |
| I [4:0] | rtAddr |
| I [4:0] | WA | 要写的寄存器 |
| I [31:0] | WD | 写入的数据 |
| I | regWrite | 写有效信号 |
| O [31:0] | rsData | 读出的数据 |
| O [31:0] | rtData |
| RF\_ALU | I | clk | 时钟信号 |
| I | rst | 同步复位 |
| O [31:0] | rsValue | rs寄存器的值 |
| O [31:0] | rtValue | rt寄存器的值 |
| O | regWrite | 寄存器写信号 |
| O [4:0] | regWriteAddr | 寄存器写地址 |
| O [31:0] | regWriteData | 写寄存器数据 |
| O [4:0] | rsCrash | rs冲突的指令 |
| O [4:0] | rtCrash | rt冲突的指令 |
| O [31:0] | instr | 指令内容 |
| O [4:0] | Tnew | 算出寄存器写数据还需要的拍数（若不用写寄存器则为0） |
| O [31:0] | PC | ALU阶段指令对应的PC |
| ALU | I [31:0] | Opr1 | 第一个操作数 |
| I [31:0] | Opr2 | 第二个操作数 |
| I [5:0] | aluOp | 进行的运算 |
| O [31:0] | calcRes | 运算结果 |
| MULT\_DIV | I | clk | 时钟信号 |
| I | rst | 同步复位信号 |
| I | start | 开始工作信号 |
| I [31:0] | Opr1 | 第一个操作数 |
| I [31:0] | Opr2 | 第二个操作数 |
| I [31:0] | instr | 执行的指令 |
| O | busy | 是否在工作信号 |
| O [31:0] | HI | HI寄存器中的值 |
| O [31:0] | LO | LO寄存器中的值 |
| ALU\_DM | I | clk | 时钟信号 |
| I | rst | 同步复位 |
| O [31:0] | rsValue | rs寄存器的值 |
| O [31:0] | rtValue | rt寄存器的值 |
| O [4:0] | rsCrash | 与rs冲突的指令 |
| O [4:0] | rtCrash | 与rt冲突的指令 |
| O [31:0] | regWD | 写寄存器值 |
| O | regWrite | 寄存器写信号 |
| O [4:0] | regWA | 寄存器写地址 |
| O [31:0] | calcRes | ALU阶段的计算结果 |
| O [31:0] | instr | 指令内容 |
| O [4:0] | Tnew | 得到写入寄存器值还需要的拍数 |
| O [31:0] | PC | 访问内存阶段指令对应的PC |
| DM | I | clk | 时钟信号 |
| I | reset | 同步复位 |
| I [31:0] | A | 要访存的地址 |
| I [3:0] | dmWrite | 每一位代表每个字节是否写入 |
| I [31:0] | WD | 写入的数据 |
| O [31:0] | RD | 读出的数据 |
| DM\_RF | I | clk | 时钟信号 |
| I | rst | 同步复位 |
| O [31:0] | rsValue | rs寄存器的值 |
| O [31:0] | rtValue | rt寄存器的值 |
| O [4:0] | Tnew | 产生结果所要的拍数 |
| O [31:0] | instr | 指令内容 |
| O | regWrite | 寄存器写信号 |
| O [4:0] | regWA | 寄存器写地址 |
| O [31:0] | regWD | 寄存器写数据 |
| O [31:0] | PC | 写寄存器阶段指令对应的PC |

模块沿数据通路依次摆放。

用中间流水寄存器保存每一阶段的计算结果，再通过一定逻辑（转发等）传给下一级流水线寄存器；其中转发统一从realRegWriteData中转，realRegWriteData是regWriteData经过少量逻辑处理后得到的写寄存器数据。

每一阶段功能说明：

|  |  |
| --- | --- |
| PC\_IM阶段 | PC根据isStop和isBranch信号，决定是否跳转到nextPC或PC+4 |
| IM取指令 |
| 读出的指令给IM\_RF |
| IM\_RF由isStop信号决定是否暂停 |
| RF读阶段 | 用instrDecoder识别指令的op和aluOp段，并产生指令的类别信号 |
| 确定PC的跳转信号和跳转地址 |
| 确定PC和IM\_RF的暂停信号，使得Tuse>=Tnew，从而保证之后可以转发 |
| 确定暂停信号isStop |
| 计算指令的算出寄存器结果拍数Tnew和用到寄存器拍数Tuse |
| 检测冲突，并计算冲突指令的位置 |
| 确定寄存器写控制信号 |
| 确定寄存器写地址 |
| 转发更新rsValue、rtValue |
| 更新regWriteData |
| 流水冲突信号 |
| ALU阶段 | 流水冲突信号 |
| 转发更新rsValue、rtValue |
| 根据指令译码情况进行指定的计算 |
| 乘除模块运算，乘为5个周期，除为10个周期 |
| 流水calcRes |
| 更新regWriteData |
| 更新下一阶段的Tnew |
| DM阶段 | 根据冲突信号进行转发写内存数据 |
| 根据指令译码情况确定字节写使能信号dmWrite |
| 访问内存 |
| 流水calcRes |
| 流水dmReadData和regWriteData |
| RF写阶段 | 根据指令译码情况和calcRes（计算出的地址）得出realRegWriteData |
| 根据写使能regWrite和写数据realRegWriteData写RF即可 |

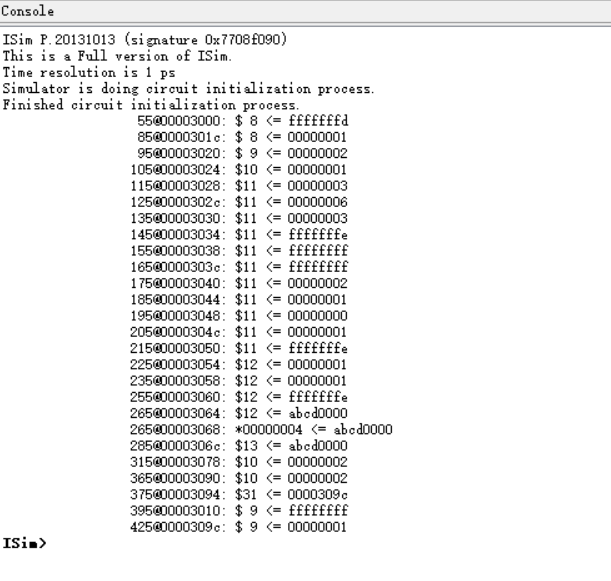
三、测试方案

1. 由于模块代码比较简单，故先人工检查各模块的代码。

2. 测试各条指令是否正常，Mars代码如下：

|  |
| --- |
| addiu $t0, $0, -3 #t0=-3  j main  nop  addiu $t0, $0, -1 #test j  func: addiu $t1, $0, -1  jr $ra #test jr  nop    main:  addiu $t0, $0, 1 #t0 = 1  #test addiu  addu $t1, $t0, $t0 #t1 =2  #test addu  subu $t2, $t1, $t0 #t2=1  #test subu  ori $t3, $t0, 2 #t3=3  #test ori  sll $t3, $t3, 1 #t3=6  #test sll  srl $t3, $t3, 1 #t3=3  #test sll  addiu $t3, $0, -2 #t3=-2  sra $t3, $t3, 1 #t3=-1  #test sra  srav $t3, $t3, $t0 #t3=-1  #test srav  sllv $t3, $t0, $t0 #t3=2  #test sllv  srlv $t3, $t3, $t0 #t3=1  #test srlv  sltu $t3, $t3, $t0 #t3=0  #test sltu  sltiu $t3, $t3, 1 #t3=1  #test sltiu  addiu $t3, $0, -2 #t3=-2  slt $t4, $t3, $0 #t4=1  #test slt  slti $t4, $t3, -1 #t4=1  #test slti  mult $t4, $t3  mflo $t4 #t4=-2  #test mult  lui $t4, 0xabcd #t4 = 0xabcd0000  #test lui  sw $t4, 4($0) #dm[0] = 0xabcd0000  #test sw  lw $t5, 4($0)#t5 = 0xabcd0000  #test lw  beq $t0, $0, B1 #t0=1  nop  addiu $t2, $0, 2  B1: #test beq  bne $t0, $0, B2  nop  addiu $t2, $0, 2  B2: #test bne  bltz $t0, B3  nop  addiu $t2, $0, 2  B3: #test bltz  jal func #test jal  nop  addiu $t1, $0, 1 |

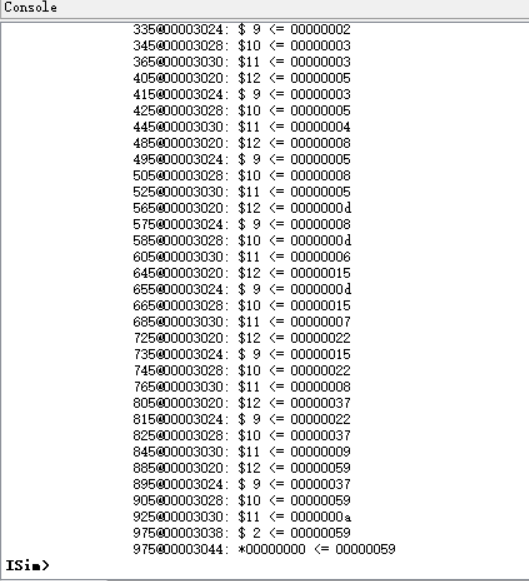
将机器码导入code.txt文件再用ISim测试，查看输出信息逐条比对即可



3. 测试实际代码

测试计算裴波那契数

|  |
| --- |
| j main  nop  fib:  addu $t0, $a0, $0 #get argument N  addiu $t1, $0, 0 #fib[-1]=0  addiu $t2, $0, 1 #fib[0]=1  addiu $t3, $0, 0 #i=0  loop\_in: beq $t3, $t0, loop\_out  nop  addu $t4, $t1, $t2  addu $t1, $t2, $0  addu $t2, $t4, $0  j loop\_in  addiu $t3, $t3, 1  loop\_out:  jr $ra #return fib[N]  addu $v0, $t2, $0  main:  jal fib  addiu $a0, $0, 10  sw $v0, 0($0) #dm[0] = fib[10] |



1. 思考题

* 为什么需要有单独的乘除法部件而不是整合进ALU？为何需要有独立的HI、LO寄存器？

因为乘除法需要的时间过长，若直接整合成组合电路会导致时频大幅度下降，故一般分解为多个时钟周期进行乘除法的计算，为时序逻辑，需要与ALU分开；独立的HI、LO寄存器用来储存乘除法的计算结果，只能被乘除相关的指令读写，与普通寄存器的行为不同，故要独立开来。

* 参照你对延迟槽的理解，试解释“乘除槽”。

某某槽的意思是若某条指令需要执行n个周期，则在这n个周期中可以同时执行其他与之独立的指令。乘除槽即在乘除法计算的n个周期中还能执行的n条指令。

* 举例说明并分析何时按字节访问内存相对于按字访问内存性能上更有优势。（Hint： 考虑C语言中字符串的情况）

例如C语言中字符串的存储和操作，按字节访问可以节省内存。

* 在本实验中你遇到了哪些不同指令类型组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？如果你是手动构造的样例，请说明构造策略，说明你的测试程序如何保证覆盖了所有需要测试的情况；如果你是完全随机生成的测试样例，请思考完全随机的测试程序有何不足之处；如果你在生成测试样例时采用了特殊的策略，比如构造连续数据冒险序列，请你描述一下你使用的策略如何结合了随机性达到强测的效果。此思考题请同学们结合自己测试CPU使用的具体手段，按照自己的实际情况进行回答

数据冲突只由“写后读”，即写普通寄存器的指令和读普通寄存器的指令之间的冲突；利用AT法转发解决；测试样例通过写普通寄存器指令与读普通寄存器的指令组合来构造。

* 为了对抗复杂性你采取了哪些抽象和规范手段？这些手段在译码和处理数据冲突的时候有什么样的特点与帮助？

为使得流水信号更加规范，采用二维数组的方式，给每个阶段分别编号为1~4，相应阶段的信号就为signal[i]；为使得转发更加规范，规定转发一律从realRegWriteData中转发，即只从每个阶段的一个信号中转发。

这些手段首先可以简化译码，即可用generate语句处理译码。其次是可以简化转发的复杂度，即检测到哪个阶段冲突直接将对应阶段的realRegWriteData连过来就行了。