P6实验设计报告

18373085 张海渝

1. 数据通路

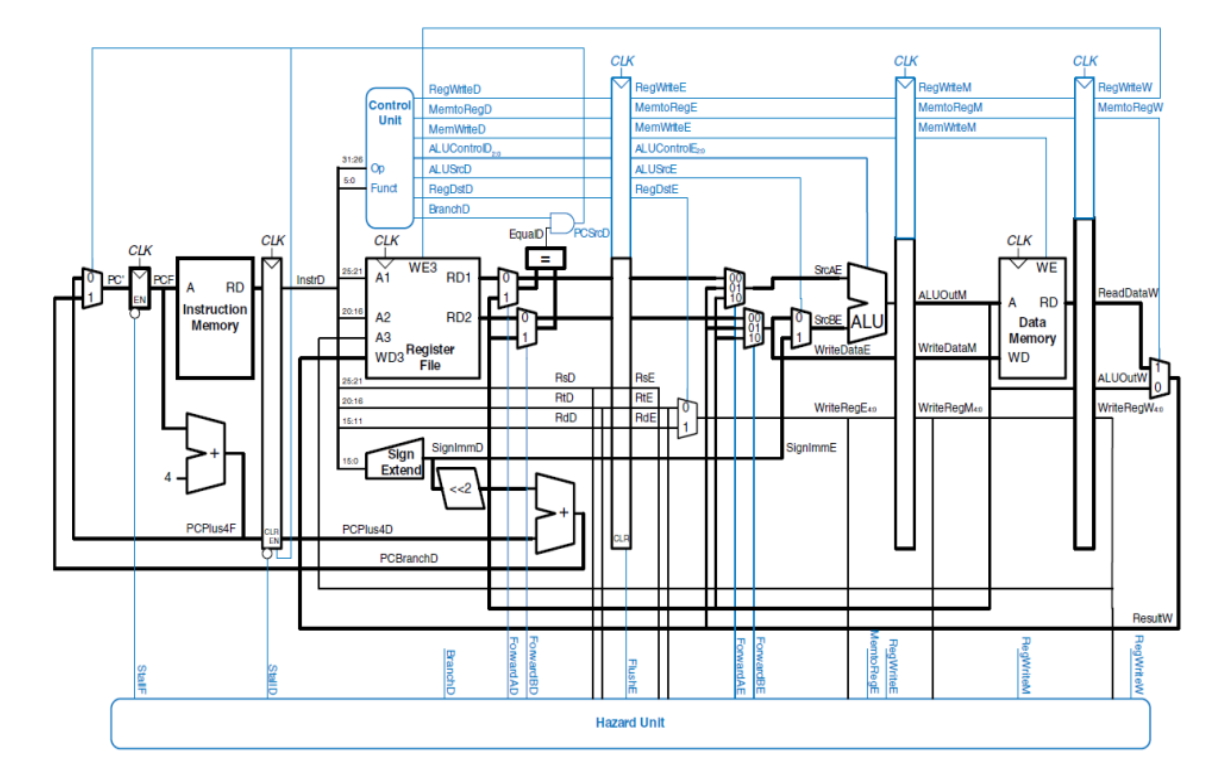


图1：数据通路（以此图为基础，之后会说明修改的地方）

1. 模块规格
2. GetNpc：（程序计数器）

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 功能名称 | 方向 | 功能描述 |
| 1 | npc[31:0] | I | 为下一PC值 |
| 2 | reset | I | 复位信号 |
| 3 | clk | I | 时钟信号 |
| 4 | En | I | 使能信号 |
| 5 | PC[31:0] | O | 当前D寄存器之前的PC值 |

模块解释：

1. 如果Reset信号有效时，用选择器将PC值复位。
2. 解释En信号：当需要暂停时，即D寄存器需要暂停的时候，需要对该模块不能向下传值，即保持D寄存器之前的PC值不被修改。
3. IM：（指令获取）

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 功能名称 | 方向 | 功能描述 |
| 1 | add[11:0] | I | 指令地址 |
| 2 | Instr | O | 输出的机器码 |

模块解释：

1. add[11:0]是当前PC值的第13-2位(因为ROM地址为12位，并且PC每次加4，相当于ROM加1)。
2. 用$readmemh载入机器码。
3. 用寄存器储存机器码。

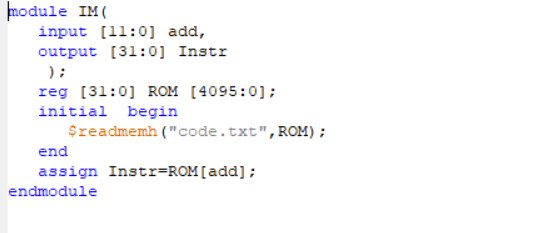


图2：IM模块代码图

1. GRF：（通用寄存器组，也称寄存器文件、寄存器堆）

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 功能名称 | 方向 | 功能描述 |
| 1 | pc[31] | I | 当前PC值 |
| 2 | A1[4:0] | I | 机器码的第25-21位，寄存器编号 |
| 3 | A2[4:0] | I | 机器码的第20-16位，寄存器编号 |
| 4 | A3[4:0] | I | 需要改变的寄存器 |
| 5 | WD3[31:0] | I | 寄存器需要改变为的值 |
| 6 | reset | I | 同步复位信号 |
| 7 | clk | I | 时钟信号 |
| 8 | WE | I | 是否写入寄存器信号 |
| 9 | RD1[31:0] | O | A1寄存器中的值 |
| 10 | RD2[31:0] | O | A2寄存器中的值 |

模块解释：

1.0号寄存器始终为0，不改变。

2.一共有32个带有使能端的寄存器。

3.需要注意：首先这一模块是在D/E级，其次PC值并不是D级的PC值而是通过W级传输过来的值，WD3以及WE都是通过W级传输过来的，还需要注意RD1,RD2并不是准确值，即并不是准确的寄存器里面的值，后面会通过转发暂停机制实现准确。

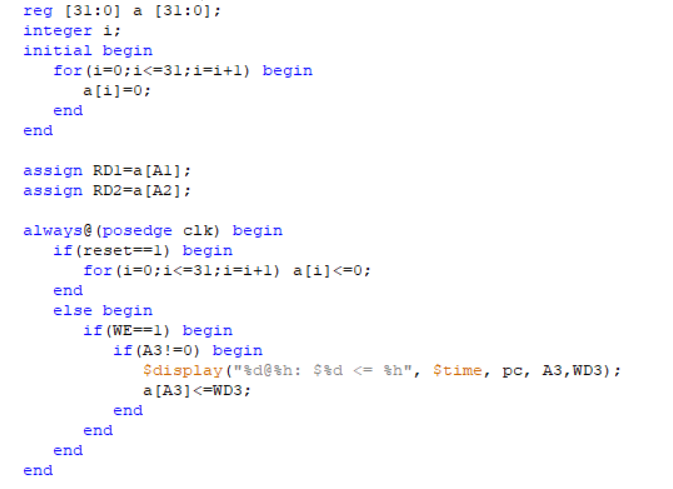


图3：GRF模块部分代码

4.ALU：（算数逻辑单元）

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 功能名称 | 方向 | 功能描述 |
| 1 | A[31:0] | I | 进行操作的第一个数 |
| 2 | B[31:0] | I | 进行操作的第二个数 |
| 3 | ALUctr[3:0] | I | 进行操作的方式 |
| 4 | ALUresult[31:0] | O | 计算后的结果 |

模块解释：

1. ALUctr=0：输出0值

ALUctr=1：加法

ALUctr=2：减法

ALUctr=3：或（or）操作

ALUctr=4:输出B的值

ALUctr=5：输出A&B

ALUctr=6：输出A^B

ALUctr=7：输出~（A|B）

ALUctr=8：输出（$signed(A)<$signed(B)）

ALUctr=9: 输出(A)<(B)

ALUctr=其他：输出0

2．此模块存在于E/M级，需要注意其两个操作数的正确与否，需要通过转发机制实现，会在后面详细说明。

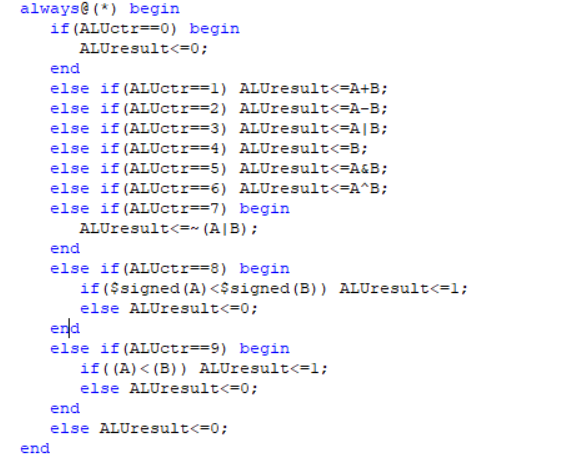


图4：ALU模块代码图

1. DM：（数据存储器）

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 功能名称 | 方向 | 功能描述 |
| 1 | addr[11:0] | I | 进行操作的数据的地址 |
| 2 | opcode[5:0] | I | 当前执行的指令 |
| 3 | byteM[1:0] | I | 存储的时候选择怎么存储 |
| 4 | din[31:0] | I | 数据 |
| 6 | clk | I | 时钟信号 |
| 7 | dout[31:0] | O | 读出的数据 |
| 8 | pc | I | 当前PC值 |

模块解释：

1.此模块通过一组寄存器，此模块在M级。

2.因为模块容量为4kb，所以进行操作的地址为M/W级ALUoutM计算结果的11-2位。且PC值为M级的PC值，会在前面进行传输下来。

3.用ans代表当前addr的32位表达值。



图5：DM模块代码图

6.EXT：（数据扩展器）

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 功能名称 | 方向 | 功能描述 |
| 1 | IMM[15:0] | I | 进行扩展的数 |
| 2 | ExtOp[1:0] | I | 扩展的方式 |
| 3 | ExtIMM[31:0] | O | 扩展后的数据 |

模块解释：

1. ExtOp=0：前16位0扩展

ExtOp=1：符号扩展至32位

ExtOp=2：后16位补0扩展

1. 用ans（一个寄存器）以及for循环的利用实现。
2. 此模块存在于D/E级，会在D/E级

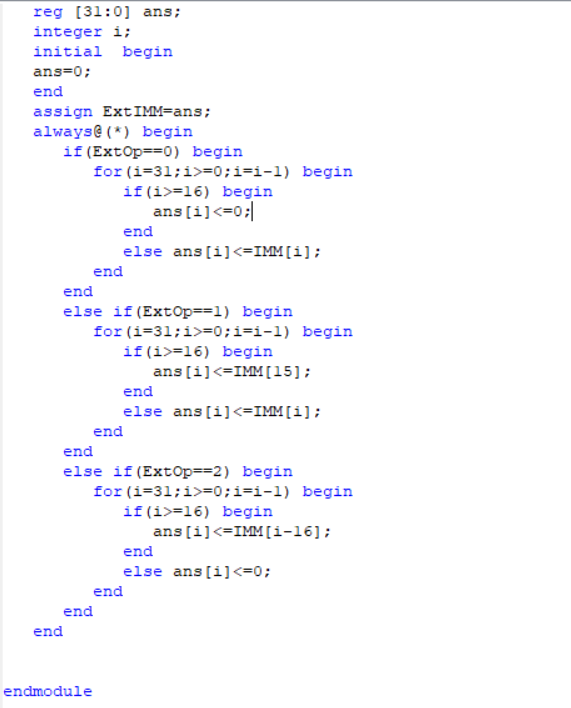


图6：EXT模块代码图

7．Control：（控制器）

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 功能名称 | 方向 | 功能描述 |
| 1 | opcode[5:0] | I | Special |
| 2 | funct[5:0] | I | Function |
| 3 | RegDst[1:0] | O | 选择需要在GRF中写入数据的寄存器 |
| 4 | ALUSrc[1:0] | O | 选择进行ALU操作的第二个操作数 |
| 5 | MemtoReg[2:0] | O | 选择存入GRF中寄存器的数据 |
| 6 | RegWrite | O | 是否在GRF中写入控制信号 |
| 7 | MemWrite | O | DM中的RAM使能端控制信号 |
| 8 | nPC\_sel[1:0] | O | 选择下一PC值信号 |
| 9 | ExtOp[1:0] | O | 数据扩展方式 |
| 10 | ALUctr[3:0] | O | ALU进行操作的方式 |

模块解释：

1. 首先通过opcode与funct选择出此时进行操作的指令方式。

2.通过if\_else语句实现控制信号的选择。

3. RegDst==2时选择31号寄存器，MemtoReg==2是选择PC+8(因为延迟槽)，为了实现jal指令。MemtoReg==3是为了Load型指令

4. npc\_sel==0: 其他指令

npc\_sel==1: beq bne blez bgtz bltz bgez

npc\_sel==2: jal or j

npc\_sel==3: jr jalr

5.此模块在D/E级操作，后面会将这些产生的控制信号，在后面需要的地方进行传输，保证正确性。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
| funct | 100000 | 100010 |  |  |  |  |  |
| opcode | 000000 | 000000 | 001101 | 100011 | 101011 | 000100 | 001111 |
|  | add | sub | ori | lw | sw | beq | lui |
| RegDst[1:0] | 1 | 1 | 0 | 0 | x | x | 0 |
| ALUScr | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| Memto [1:0] | 0 | 0 | 0 | 1 | x | x | 0 |
| RegWrite | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| MemWrite | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| nPC\_sel[1:0] | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| ExtOp[1:0] | x | x | 0 | 1 | 1 | x | 2 |
| ALUctr[2:0] | 1 | 2 | 3 | 1 | 1 | 2 | 4 |

图七：信号输出真值表（Memto代表MemtoReg）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| funct | 001000 | 000000 |  |  |
| opcode | 000000 | 000000 | 000011 | 000010 |
|  | jr | nop | jal | j |
| RegDst[1:0] | 0 | 0 | 2 | 0 |
| ALUScr | 0 | 0 | 0 | 0 |
| Memto [1:0] | 0 | 0 | 2 | 0 |
| RegWrite | 0 | 0 | 1 | 0 |
| MemWrite | 0 | 0 | 0 | 0 |
| nPC\_sel[1:0] | 3 | 0 | 2 | 2 |
| ExtOp[1:0] | 0 | 0 | 0 | 0 |
| ALUctr[2:0] | 0 | 0 | 0 | 0 |
| npc\_sel3 | 1 | 0 | 1 | 0 |
| npc\_sel4 | 1 | 0 | 0 | 0 |

图七（续）：信号输出真值表（Memto代表MemtoReg，部分，其余的同理）

8．ALU\_2(HI,LO乘除模块)

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 功能名称 | 方向 | 功能描述 |
| 1 | clk | I | 时钟信号 |
| 2 | reset | I | 复位信号 |
| 3 | start | I | 是否在开始工作 |
| 4 | A[31:0] | I | 乘除的第一个数 |
| 5 | B[31:0] | I | 乘除的第二个数 |
| 6 | sel[3:0] | I | 选择此模块干什么 |
| 7 | out[31:0] | O | 输出信号 |
| 8 | busy | O | 是否在工作的信号 |

模块解释：

1.当其中的信号start为1的时候开始工作，工作方式如下：

sel==1:进行有符号乘操作（mult），置其中的Busy=6

sel==2:进行无符号乘操作（multu）, 置其中的Busy=6

sel==3:进行有符号除操作（div）,置其中的Busy=11

sel==4:进行无符号除操作（divu）, 置其中的Busy=11

sel==5:取出HI寄存器中的值（mfhi）

sel==6:取出LO寄存器中的值（mflo）

sel==7:给HI赋值（mthi）

sel==8:给LO赋值（mtlo）

2．其输出的busy信号是为了暂停D级中的mfhi，mflo，mtlo，mthi着一些指令，因为乘除需要分别进行5，10个周期，所以需要等待。



图八：ALU\_2模块部分代码图

1. 流水线CPU各级传输及操作

1.D级

1．传输信号：InstrD（来自于Instr@F）,pc4D（pc4@F）,pcD(pc@F)。

2．使能端En,当需要暂停的时候需要将D级寄存器锁住，不再传输，当reset的时候也需要将其锁住，以免使第一条指令工作多次。

3．此级进行GRF读取操作以及EXT扩展操作和Control的操作。

4．还需要进行B/J类型指令跳转的选择，通过npc\_sel进行选择，在上面已经解释了npc\_sel的含义。

5.在进行b型,j型指令的时候在这一级会进行一次比较，从而选择npc\_sel的终值。

6.以后只用传输Control翻译出来的信号即可。（集中式译码）

2.E级

1.传输信号：A1E(A1D@D),A2E(A2D@D),A3E(A3D@D), Data1E(Data1D@D),Data2E(Data2D@D),EXTIMME(EXTIMMD@D),RegDstE(RegDstD@D),ALUSrcE(ALUSrcD@D),MemtoRegE(MemtoRegD@D),RegWriteE(RegWriteD@D),MemWriteE(MemWriteD@D),ALUctrE(ALUctrD@D),opcodeE(opcodeD@D),functE(functD@D),pcE(pcD@D)。

2．信号解释，其中A1E,A2E,A3E代表进行操作的寄存器编号，Data1E,Data2E分别为A1E,A2E为转发以后的值（不一定为当前的准确值，因为可能在D级时这两个数据未被使用，于是可能在E级的转发器中更新）。其余的为ControlD控制信号传输。

3．此级进行操作的有ALU，ALU\_2模块。

4．需要注意在此级就选择出来了会进行回写的寄存器编号，命名为WriteRegE。还需要注意在此级仍需转发，以免进行操作的数据为错误数据。

3.M级

1．传输信号：WriteDataM(WriteDataE@E),WriteRegM(WriteRegE@E),MemtoRegM(MemtoRegE@E),RegWriteM(RegWriteE@E),MemWriteM(MemWriteE@E),opcodeM(opcodeE@E),functM(functE@E),pcM(pcE@E),ALUoutM(ALUoutE@E)。

2．信号解释：WriteDataM为向DM中写入的值，ALUoutM为在E级中ALU中计算出来的值。其余的为控制信号传输。

3．此级需要进行操作的是DM模块。

4.W级

1.传输信号：ALUoutW(ALUoutM@M),DMoutW(DMoutM@M),pcW(pcM@M),MemtoRegW(MemtoRegM@M),RegWriteW(RegWriteM@M),opcodeW(opcodeM@M),functW(functM@M),WriteRegW(WriteRegM@M)。

2．其中需要选择出回写的值，记即在ALUoutW ,DMoutW, pcW+8中进行选择出最后的值。（因为延迟槽所以需要pcW+8）。选择出来值为ResultW。

1. 转发与暂停机制的实现

1.转发机制：

1.实现的方法：暴力转发。即在每一级中找到需要的更新转发的值，找到在后面需要转发过来的值，用一个选择器进行选择，其中转发需要的条件是满足寄存器编号一样且不为0，并且此时得到了该周期的值。

2.需要转发的地方：D级的GRF出来的RD1,RD2值，这是因为这是读出了其中两个寄存器的值，当进行beq的值的时候需要在D级就进行比较，所以需要最新的值。RD1转发以后的值记为Data1D,RD2转发以后的值记为Data2D，按照数据通路可知转发来自于本身的值（RD1），E级的EXIMME，offoutE，M级的ALUoutM以及W级的ResultW。用ForwardAD, ForwardBD代表选择信号。E级的Data1E,Data2E需要转发，转发至本身的值（Data1E），M级的ALUoutM以及W级的ResultW，用ForwardAE,ForwarBE表示选择信号。

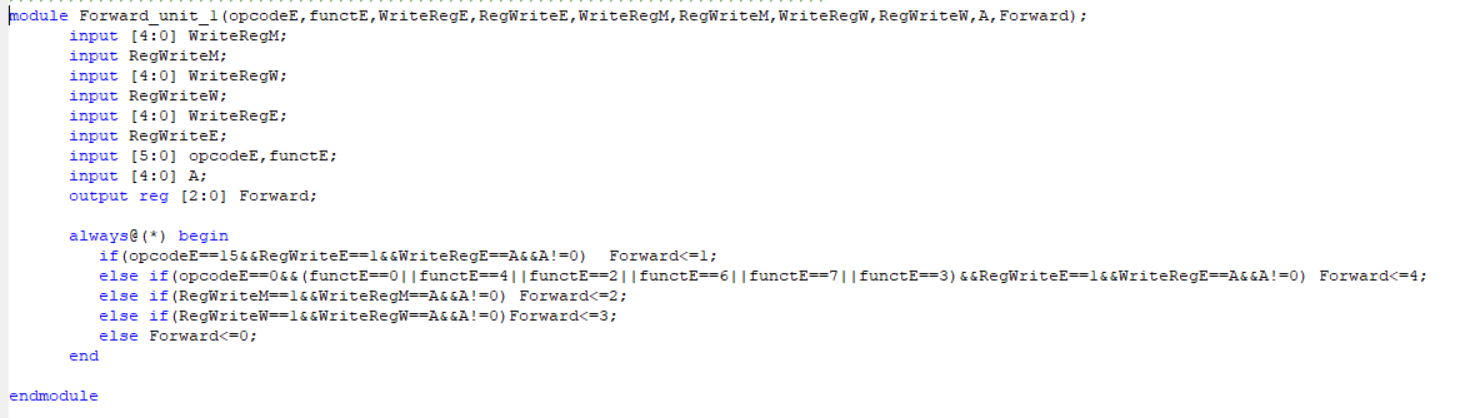


图7：Forward选择信号示意图

2.暂停机制：

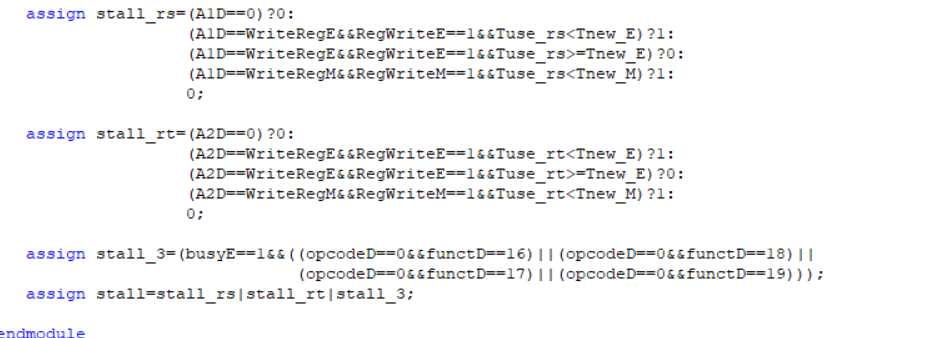
1.暂停机制是为了解决转发机制解决不了的问题。

2.首先用Tuse代表该条指令在进入D级以后的第几个周期需要用到寄存器中的值，Tnew代表在流水线中的E,M级中需要几个周期才能得到最新的值（比如b型的Tuse等于0，E级的load型等于2）。

3．当Tuse<后面的任何一级的Tnew时需要暂停，暂停时进行的操作为E级注入一个空操作，D级停止传输，pc停止传输。

4.需要注意当此时在E级的为乘除操作时，D级为mfhi，mflo，mtlo，mthi也需要暂停

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Tuse-A1 | Tuse-A2 | E-Tnew | M-Tnew |
| addu | 1 | 1 | 1 | 0 |
| subu | 1 | 1 | 1 | 0 |
| ori | 1 | 1 | 1 | 0 |
| lw | 1 |  | 2 | 1 |
| sw | 1 | 1 | 0 | 0 |
| beq | 0 | 0 |  |  |
| lui |  |  | 0 | 0 |
| jal |  |  | 2 | 1 |
| jr | 0 |  |  |  |



图九：Tuse，Tnew代码（部分代码）

1. 测试代码：

1.R型指令（addu为例，乘除包含mflo，mfhi，slt等类型指令）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **测试类型** | **前序指令** | **冲突位置** | **冲突寄存器** | **测试序列** | **解决方法** |
| R-M-RS | subu | E | rs | subu $1,$2,$3  addu $4,$1,$2 | 转发 |
| R-M-RT | subu | E | rt | subu $1,$2,$3  addu $4,$2,$1 | 转发 |
| R-W-RS | subu | E | rs | subu $1,$2,$3  nop  addu $4,$1,$2 | 转发 |
| R-W-RT | subu | E | rt | subu $1,$2,$3  nop  addu $4,$2,$1 | 转发 |
| I-M-RS | ori | E | rs | ori $1,$0,0x1111  addu $3,$1,$2 | 转发 |
| I-M-RT | ori | E | rt | ori $1,$0,0x1111  addu $3,$2,$1 | 转发 |
| I-W-RS | ori | E | rs | ori $1,$0,0x1111  nop  addu $3,$1,$2 | 转发 |
| I-W-RT | ori | E | rt | ori $1,$0,0x1111  nop  addu $3,$2,$1 | 转发 |
| LD-W-RS | lw | E | rs | lw $1,0($2)  nop  addu $3,$1,$2 | 转发 |
| LD-W-RT | lw | E | rt | lw $1,0($2)  nop  addu $3,$2,$1 | 转发 |
| LD-M-RS | lw | E | rs | lw $1,0($2)  addu $3,$1,$2 | 暂停 |
| LD-M-RT | lw | E | rt | lw $1,0($2)  addu $3,$1,$2 | 暂停 |
| LD-E-RS | lw | D | rs | lw $1,0($2)  addu $3,$1,$2 | 暂停 |
| LD-E-RT | lw | D | rt | lw $1,0($2)  addu $3,$1,$2 | 暂停 |
| JAL-M-RS | jal | D | rs | jal dfs  nop  dfs:  addu $1,$31,$0 | 转发 |
| JAL-M-RT | jal | D | rt | jal dfs  nop  dfs:  addu $1,$0,$31 | 转发 |
| JAL-W-RS | jal | D | rs | jal dfs  nop  dfs:  nop  addu $1,$31,$0 | 转发 |
| JAL-W-RT | jal | D | rt | jal dfs  nop  dfs:  nop  addu $1,$0,$31 | 转发 |

2.I型指令（ori为例，其中包含了sll等指令的测试）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **测试类型** | **前序指令** | **冲突位置** | **冲突寄存器** | **测试序列** | **解决方法** |
| R-M-RS | subu | E | rs | subu $1,$2,$3  ori $4,$1,0x1111 | 转发 |
| R-W-RS | subu | E | rs | subu $1,$2,$3  nop  ori $4,$1,0x1111 | 转发 |
| I-M-RS | ori | E | rs | ori $1,$0,0x1110  ori $2,$1,0x1111 | 转发 |
| I-W-RS | ori | E | rs | ori $1,$0,0x1110  nop  ori $3,$1,0x1111 | 转发 |
| LD-W-RS | lw | E | rs | lw $1,0($2)  nop  ori $3,$1,0x0 | 转发 |
| LD-M-RS | lw | D | rs | lw $1,0($2)  nop  ori $3,$1,$2 | 暂停 |
| LD-E-RS | lw | D | rs | lw $1,0($2)  ori $3,$1,$2 | 暂停 |
| JAL-M-RS | jal | D | rs | jal dfs  nop  dfs:  ori $1,$31,0x0 | 转发 |
| JAL-W-RS | jal | D | rs | jal dfs  nop  dfs:  nop  ori $1,$31,0x0 | 转发 |

3.LD型指令（lw为例）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **测试类型** | **前序指令** | **冲突位置** | **冲突寄存器** | **测试序列** | **解决方法** |
| R-M-RS | subu | E | rs | subu $1,$2,$3  lw $4,0($1) | 转发 |
| R-W-RS | subu | E | rs | subu $1,$2,$3  nop  lw $4,0($1) | 转发 |
| I-M-RS | ori | E | rs | ori $1,$0,0x1110  lw $4,0($1) | 转发 |
| I-W-RS | ori | E | rs | ori $1,$0,0x1110  nop  lw $4,0($1) | 转发 |
| LD-W-RS | lw | E | rs | lw $1,0($2)  nop  lw $4,0($1) | 转发 |
| LD-M-RS | lw | D | rs | lw $1,0($2)  nop  lw $4,0($1) | 转发 |
| LD-E-RS | lw | D | rs | lw $1,0($2)  lw $4,0($1) | 暂停 |
| JAL-M-RS | jal | D | rs | jal dfs  nop  dfs:  lw $4,0($31) | 转发 |
| JAL-W-RS | jal | D | rs | jal dfs  nop  dfs:  nop  lw $4,0($31) | 转发 |

4.Store型指令（sw为例）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **测试类型** | **前序指令** | **冲突位置** | **冲突寄存器** | **测试序列** | **解决方法** |
| R-M-RS | subu | E | rs | subu $1,$2,$3  sw $4,0($1) | 转发 |
| R-M-RT | subu | E | rt | subu $1,$2,$3  sw $1,0($0) | 转发 |
| R-W-RS | subu | E | rs | subu $1,$2,$3  nop  sw $4,0($1) | 转发 |
| R-W-RS | subu | E | rt | subu $1,$2,$3  nop  sw $1,0($0) | 转发 |
| I-M-RS | ori | E | rs | ori $1,$0,0x1110  sw $4,0($1) | 转发 |
| I-W-RS | ori | E | rs | ori $1,$0,0x1110  nop  sw $4,0($1) | 转发 |
| I-M-RT | ori | E | rt | ori $1,$0,0x1110  sw $1,0($0) | 转发 |
| I-W-RT | ori | E | rt | ori $1,$0,0x1110  nop  sw $1,0($0) | 转发 |
| LD-W-RS | lw | E | rs | lw $1,0($2)  nop  sw $4,0($1) | 转发 |
| LD-W-RT | lw | E | rt | lw $1,0($2)  nop  sw $1,0($0) | 转发 |
| LD-M-RS | lw | D | rs | lw $1,0($2)  nop  sw $4,0($1) | 转发 |
| LD-M-RT | lw | D | rt | lw $1,0($2)  nop  sw $1,0($0) | 转发 |
| LD-E-RS | lw | D | rs | lw $1,0($2)  sw $4,0($1) | 暂停 |
| LD-E-RT | lw | D | rt | lw $1,0($2)  sw $1,0($0) | 转发 |
| JAL-M-RS | jal | D | rs | jal dfs  nop  dfs:  sw $4,0($31) | 转发 |
| JAL-W-RS | jal | D | rs | jal dfs  nop  dfs:  nop  sw $4,0($31) | 转发 |

5.B型指令（beq为例）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **测试类型** | **前序指令** | **冲突位置** | **冲突寄存器** | **测试序列** | **解决方法** |
| R-E-RS | subu | D | rs | subu $1,$2,$3  beq $1,$2,loop | 暂停 |
| R-E-RT | subu | D | rt | subu $1,$2,$3  beq $2,$1,loop | 暂停 |
| R-M-RS | subu | D | rs | subu $1,$2,$3  nop  beq $1,$2,loop | 转发 |
| R-M-RT | subu | D | rt | subu $1,$2,$3  nop  beq $2,$1,loop | 转发 |
| I-E-RS | ori | D | rs | ori $1,$2,0x1111  beq $1,$2,loop | 暂停 |
| I-E-RT | ori | D | rt | ori $1,$2,0x1111  beq $2,$1,loop | 暂停 |
| I-M-RS | ori | D | rs | ori $1,$2,0x1111  nop  beq $1,$2,loop | 转发 |
| I-M-RT | ori | D | rt | ori $1,$2,0x1111  nop  beq $2,$1,loop | 转发 |
| LD-E-RS | lw | D | rs | lw $1,0($2)  beq $1,$2,loop | 暂停 |
| LD-E-RT | lw | D | rt | lw $1,0($2)  beq $2,$1,loop | 暂停 |
| LD-M-RS | lw | D | rs | lw $1,0($2)  nop  beq $1,$2,loop | 暂停 |
| LD-M-RT | lw | D | rt | lw $1,0($2)  nop  beq $2,$1,loop | 暂停 |
| LD-W-RS | lw | D | rs | lw $1,0($2)  nop  nop  beq $1,$2,loop | 转发 |
| LD-W-RT | lw | D | rt | lw $1,0($2)  nop  nop  beq $2,$1,loop | 转发 |
| JAL-M-RS | jal | D | rs | jal dfs  nop  dfs:  beq $4,$31,loop | 暂停 |
| JAL-M-RT | jal | D | rt | jal dfs  nop  dfs:  beq $31,$4,loop | 暂停 |
| JAL-W-RS | jal | D | rs | jal dfs  nop  dfs:  nop  beq $4,$31,loop | 转发 |
| JAL-W-RT | jal | D | rt | jal dfs  nop  dfs:  nop  beq $31,$4,loop | 转发 |

6.jr指令：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **测试类型** | **前序指令** | **冲突位置** | **冲突寄存器** | **测试序列** | **解决方法** |
| R-E-RS | subu | D | rs | subu $1,$2,$3  jr $1 | 暂停 |
| R-M-RS | subu | D | rs | subu $1,$2,$3  nop  jr $1 | 转发 |
| I-E-RS | ori | D | rs | ori $1,$2,0x1111  jr $1 | 暂停 |
| I-M-RS | ori | D | rs | ori $1,$2,0x1111  nop  jr $1 | 转发 |
| LD-E-RS | lw | D | rs | lw $1,0($2)  jr $1 | 暂停 |
| LD-M-RS | lw | D | rs | lw $1,0($2)  nop  jr $1 | 暂停 |
| LD-W-RS | lw | D | rs | lw $1,0($2)  nop  nop  jr $1 | 转发 |
| JAL-M-RS | jal | D | rs | jal dfs  nop  dfs:  jr $31 | 暂停 |
| JAL-W-RS | jal | D | rs | jal dfs  nop  dfs:  nop  jr $31 | 转发 |

6.乘除指令（mult为例）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **测试类型** | **前序指令** | **冲突位置** | **冲突寄存器** | **测试序列** | **解决方法** |
| R-E-RS | mfhi | D | HI | mfhi $1  mult $2,$3 | 暂停 |

六．思考题：

乘除模块:

1.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？

答：因为乘除模块需要进行的时钟周期为多个周期，而ALU模块进行的时钟周期为一个周期，而且单独用一个乘除模块不会影响不用乘除模块的指令，加快CPU进程，因为单独的HI，LO模块可以更高效的实现cpu性能，因为其他的指令就可以继续进行，而不会受到影响，而且如果放在GRF中，没有办法在我的设计中直接改变两个寄存器的值，所以单独是最好的选择。

2. 参照你对延迟槽的理解，试解释“乘除槽”。

答：这是为了不运用乘除模块的指令可以正常进行操作而不受乘除模块的影响，这样可以就像延迟槽一样，可以使得CPU的效率增加，并且使CPU设计不受到更多的影响

扩展模块：

1. 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）

答：C语言中的字符串可以知道，当按字节访问的时候，就像相当于访问了一个char字符，此时一个字符可以知道可以直接寻址，而你一个字，可以知道，必须要进行扩展等操作，所以按字节访问更有优势。

复杂性控制与设计风格：

1. 如何概括你所设计的CPU的设计风格？为了对抗复杂性你采取了哪些抽象和规范手段？

答：首先在译码阶段，我才用了集中式译码的手段，运用了if\_else,我自认为这种方法对我来说比宏定义更加的让我喜欢，因为这种方式虽然代码量大，但是这种方式我觉得更加易于debug，因为opcode，funct都是具体的值，方便观察。而且控制信号更加容易修改。

在选择器方面，我也选择两种方式，当选择器需要输出多个信号的时候我选择了用模块的方式实现，当只用输出一个信号的时候，我选择的就是用assign语句进行选择，我觉得这样可以是我的思路更加清晰，因为用模块进行选择的时候容易打错值。而直接用assign可以更加容易修改，但是局限性比较强。

在转发模块方面，我选择了侦测者方式，即用模块的方式。

在暂停方面我又选择了规划者方式，因为就像我刚才所说，因为输出的信号只有一位，需要的信号也不多，所以我选择了这种方式。

总的来说我选择了两种方式的综合

2. 你对流水线CPU设计风格有何见解？

答：我认为应该选择一种适合自己的方式，并且能够很好的加入指令，而且再转发等方面应该考虑各种各样的情况，以免在暂停转发方面出现问题，应该多写一些模块，比如各种的mux，这样在需要的时候能够更加方便简便，最重要的一点就是需要很好的命名格式，要不然会发生惨剧，两个小时一个bug，所以最重要的我觉得使命名格式，设计风格更命名模式有很大的关系。

在线测试：

1. 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？

答：b,j指令最复杂，因为这些需要更多的转发，其实就是用了更多的模块，其实冲突最复杂就是用的，我是通过手动一条一条分类，通过器件进行分类，比如通过ALU得到最终结果的，其实最后的转发暂停条件就是一样的，所以就进行分类以后进行测试，这样能覆盖所以的冲突。