计算机组成原理实验报告

一、CPU 设计方案综述

总体设计综述

本 **CPU** 为 **Verilog** 实现的流水线 **CPU**，支持指令集包含 `addu,subu,ori,lw,sw,beq,lui,j,jal,jr,nop` 。为了实现这些功能，**CPU** 主要包含了 `ALU,Comp,Ctrl,Datapath,DM,Ext,GRF,Hazard,IFU,IM,PC` ，这些模块分成 5 级。

关键模块定义

IFU

信号名	方向	信号描述
`clk`	I	时钟信号
`reset`	I	同步复位信号
`stall`	I	暂停信号
`Branch`	I	分支信号
`Jump`	I	跳转信号
`PCBranch[31:0]`	I	分支地址
`PCJump[31:0]`	I	跳转地址
`Instr[31:0]`	O	输出指令
`PC[31:0]`	O	输出地址

序号	功能名称	功能描述
1	取指令	根据 `PC` 地址将指令读出
2	复位	当复位信号有效时，将 `PC` 设置为 `0x00003000`
3	暂停	将 `PC` 暂停，输出指令不变
4	计算下一个 `PC`	若 `Jump` 有效，则 `PC = JumpAddr` 若 `Branch` 有效，则 `PC = BranchAddr` 否则，`PC = PC + 4`

IM

信号名	方向	信号描述
`PC[31:0]`	I	当前 `PC` 地址
`Instr[31:0]`	O	对应地址的指令

PC

信号名	方向	信号描述
`clk`	I	时钟信号
`reset`	I	同步复位信号
`stall`	I	暂停信号
`NPC_Sel`	I	控制下一个 `PC` 信号
`NPC[31:0]`	I	`Branch` 或 `Jump` 的 `PC` 信号
`PC[31:0]`	O	计算后的 `PC`

ALU

信号名	方向	信号描述
`AluCtrl[3:0]`	I	控制运算方式信号
`A[31:0]`	I	第一运算数
`B[31:0]`	I	第二运算数
`S[4:0]`	I	移位数
`D[31:0]`	O	运算结果数

序号	功能名称	功能描述
1	算术运算	将 `A` 和 `B` 进行算术运算
2	按位逻辑运算	将 `A` 和 `B` 按位进行逻辑运算
3	移位运算	将 `B` 按照 `S` 的移位数进行移位

Ctrl

信号名	方向	信号描述
`Instr[31:0]`	I	当前指令
`RegDst[1:0]`	O	选择 `rd` 、 `rt` 或 `$31` 作为写入目标寄存器
`Ext_Op[1:0]`	O	控制扩展方式
`Branch[2:0]`	O	检测条件偏移指令
`DataDone[2:0]`	O	判断当前指令得到最终数据的级数
`AluCtrl[3:0]`	O	控制运算方式
`AluSrc`	O	控制参与运算的数值
`RegWrite`	O	寄存器堆写入使能
`MemToReg`	O	控制 `DM` 数据写入寄存器堆
`MemWrite`	O	`DM` 写入使能
`Jump`	O	检测 `J` 指令信号
`Link`	O	将 `PC + 4` 写入寄存器信号
`Return`	O	将寄存器的地址写入 `PC` 信号
`Tuse_rs[2:0]`	O	使用 `rs` 的最大级数
`Tuse_rt[2:0]`	O	使用 `rt` 的最大级数
`Tnew[2:0]`	O	最新数据产生的级数

`funct`	`100000`	`100010`	`-`	`-`	`-`	`-`	`-`	`-`	`000000`	`-`	`001001`	`001000`	`-`
`opcode`	`000000`	`000000`	`001101`	`100011`	`101011`	`000100`	`001111`	`001000`	`000000`	`000011`	`000000`	`000000`	`001001`
Instruction	`addu`	`subu`	`ori`	`lw`	`sw`	`beq`	`lui`	`addi`	`nop`	`jal`	`jalr`	`jr`	`j`
`RegDst[1:0]`	`rd`	`rd`	`rt`	`rt`	x	x	`rt`	`rt`	`rd`	`$31`	`rd`	x	x
`AluSrc`	0	0	1	1	1	1	1	1	0	0	0	0	x
`RegWrite`	1	1	1	1	0	0	1	1	1	1	1	0	0
`MemToReg`	0	0	0	1	0	0	0	0	0	0	0	0	x
`MemWrite`	0	0	0	0	1	0	0	0	0	0	0	0	0
`Branch[2:0]`	x	x	x	x	x	`equal`	x	x	x	x	x	x	x
`Jump`	0	0	0	0	0	0	0	0	0	1	1	1	1
`Link`	0	0	0	0	0	0	0	0	0	1	1	0	0
`Return`	0	0	0	0	0	0	0	0	0	0	1	1	0
`ALUCtrl[3:0]`	`ADD`	`SUB`	`OR`	`ADD`	`ADD`	x	`ADD`	`ADD`	`SLL`	x	x	x	x
`Ext_Op[1:0]`	x	x	`zero16`	`sign16`	`sign16`	`sign16`	`upper`	`sign16`	x	x	x	x	x
`Tuse_rs[2:0]`	E	E	E	E	E	D	x	E	E	x	D	D	x
`Tuse_rt[2:0]`	E	E	x	x	M	D	x	x	x	x	x	x	x
`Tnew[2:0]`	M	M	M	W	x	x	E	M	M	D	D	x	x

Datapath

信号名	方向	信号描述
`clk`	I	时钟信号
`reset`	I	同步复位信号
`stall_PC`	I	`PC` 暂停信号
`stall_FD`	I	`FD` 寄存器暂停信号
`stall_DE`	I	`DE` 寄存器暂停信号
`stall_EM`	I	`EM` 寄存器暂停信号
`stall_MW`	I	`MW` 寄存器暂停信号
`clr_FD`	I	`FD` 寄存器清空信号
`clr_DE`	I	`DE` 寄存器清空信号
`clr_EM`	I	`EM` 寄存器清空信号
`clr_MW`	I	`MW` 寄存器清空信号
`ForwardSel_D1[4:0]`	I	D 级 `RD1` 转发信号
`ForwardSel_D2[4:0]`	I	D 级 `RD2` 转发信号
`ForwardSel_EA[4:0]`	I	E 级 `A` 转发信号
`ForwardSel_EB[4:0]`	I	E 级 `B` 转发信号
`ForwardSel_MD[4:0]`	I	M 级写入数据转发信号
`ForwardSel_W[4:0]`	I	W 级写入数据转发信号
`Branch_D[2:0]`	I	D 级 `Branch` 判断信号
`ExtCtrl_D[1:0]`	I	D 级扩展控制信号
`RegDst[1:0]`	I	D 级 `A3` 控制信号
`Jump_D`	I	D 级跳转信号
`Return_D`	I	D 级跳转寄存器信号
`Link_D`	I	D 级写入 `PC + 8` 信号
`AluCtrl_E[3:0]`	I	E 级 `ALU` 控制运算信号
`AluSrc_E`	I	E 级控制运算数据信号
`Link_E`	I	E 级写入 `PC + 8` 信号
`MemWrite_M`	I	M 级 `DM` 写入使能信号
`Link_M`	I	M 级写入 `PC + 8` 信号
`MemToReg_W`	I	W 级控制写入 `DM` 数据信号

信号名	方向	信号描述
`Link_W`	I	W 级写入 `PC + 8` 信号
`RegWrite_W`	I	W 级 `GRF` 写入使能信号
`Instr_D[31:0]`	O	D 级指令
`Instr_E[31:0]`	O	E 级指令
`Instr_M[31:0]`	O	M 级指令
`Instr_W[31:0]`	O	W 级指令
`PCplus4_D[31:0]`	O	D 级 `PCplus4` 信号
`PCplus4_E[31:0]`	O	E 级 `PCplus4` 信号
`PCplus4_M[31:0]`	O	M 级 `PCplus4` 信号
`PCplus4_W[31:0]`	O	W 级 `PCplus4` 信号
`PCplus8_D[31:0]`	O	D 级 `PCplus8` 信号
`PCplus8_E[31:0]`	O	E 级 `PCplus8` 信号
`PCplus8_M[31:0]`	O	M 级 `PCplus8` 信号
`PCplus8_W[31:0]`	O	W 级 `PCplus8` 信号
`A3_E[4:0]`	O	E 级 `A3` 地址
`A3_M[4:0]`	O	M 级 `A3` 地址
`A3_W[4:0]`	O	W 级 `A3` 地址

DM

信号名	方向	信号描述
`clk`	I	时钟信号
`reset`	I	同步复位信号
`MemWrite`	I	写入使能信号
`Addr[31:0]`	I	数据存储器地址
`WriteData[31:0]`	I	写入数据
`PC[31:0]`	I	当前 `PC` 地址
`DMOut[31:0]`	O	对应输出数据

序号	功能名称	功能描述
1	复位	当 `reset` 信号有效则将 `DM` 数据清零
2	写数据	当 `MemWrite` 有效且时钟上升沿时，将 `WriteData` 数据写入指定地址
3	读数据	从 `Addr` 读出数据

Ext

信号名	方向	信号描述
`ExtCtrl[1:0]`	I	控制扩展方式信号
`imm16[15:0]`	I	16 位立即数
`imm[31:0]`	O	扩展后的立即数

序号	功能名称	功能描述
1	扩展	按照 `ExtCtrl` 信号指定扩展数据和方式

GRF

信号名	方向	信号描述
`clk`	I	时钟信号
`reset`	I	同步复位信号
`RegWrite`	I	写入使能信号
`A1[4:0]`	I	指定 32 个寄存器中的一个，输出其中数据到 `RD1`
`A2[4:0]`	I	指定 32 个寄存器中的一个，输出其中数据到 `RD2`
`A3[4:0]`	I	指定 32 个寄存器中的一个，写入 `WriteData` 数据
`WriteData[31:0]`	I	输入数据
`PC[31:0]`	I	`PC` 地址
`RD1[31:0]`	O	`A1` 指定寄存器中的数据
`RD2[31:0]`	O	`A2` 指定寄存器中的数据

序号	功能名称	功能描述
1	复位	当复位信号有效时，寄存器清零
2	写数据	读出 `A1,A2` 的寄存器数据到 `RD1,RD2`
3	读数据	当 `RegWrite` 有效时且时钟上升沿时，将 `WriteData` 写入指定寄存器内

Hazard

信号名	方向	信号描述
`Instr_D[31:0]`	I	D 级指令信号
`DataDone_D[2:0]`	I	D 级数据处理终点信号
`Link_D`	I	D 级写入 `PC + 8` 信号
`Instr_E[31:0]`	I	E 级指令信号
`DataDone_E[2:0]`	I	E 级数据处理终点信号
`A3_E[4:0]`	I	E 级 `A3` 地址
`Link_E`	I	E 级写入 `PC + 8` 信号
`Instr_M[31:0]`	I	M 级指令信号
`DataDone_M[31:0]`	I	M 级数据处理终点信号
`A3_M[4:0]`	I	M 级 `A3` 地址
`Link_M`	I	M 级写入 `PC + 8` 信号
`Instr_W[31:0]`	I	W 级指令信号
`DataDone_W[2:0]`	I	W 级数据处理终点信号
`A3_W[4:0]`	I	W 级 `A3` 地址
`Link_W`	I	W 级写入 `PC + 8` 信号
`ForwardSel_D1[4:0]`	O	D 级 `RD1` 转发信号
`ForwardSel_D2[4:0]`	O	D 级 `RD2` 转发信号
`ForwardSel_EA[4:0]`	O	E 级 `A` 转发信号
`ForwardSel_EB[4:0]`	O	E 级 `B` 转发信号
`ForwardSel_MD[4:0]`	O	M 级 `DM` 写入数据转发信号
`ForwardSel_W[4:0]`	O	W 级 `GRF` 写入数据转发信号
`stall_PC`	O	暂停 `PC` 信号
`stall_FD`	O	暂停 `FD` 寄存器信号
`stall_DE`	O	暂停 `DE` 寄存器信号
`stall_EM`	O	暂停 `EM` 寄存器信号
`stall_MW`	O	暂停 `MW` 寄存器信号
`clr_FD`	O	清除 `FD` 寄存器信号
`clr_DE`	O	清除 `DE` 寄存器信号
`clr_EM`	O	清除 `EM` 寄存器信号

信号名	方向	信号描述
`clr_MW`	O	清除 `MW` 寄存器信号

Comp

信号名	方向	信号描述
`A[31:0]`	I	`RD1` 信号
`B[31:0]`	I	`RD2` 信号
`Comp_Out[2:0]`	O	比较后输出信号

序号	功能名称	功能描述
1	数据比较	若 $A > B$ 则`Comp_Out = 100` 若 $A = B$ 则 `Comp_Out = 010` 若 $A < B$ 则 `Comp_Out = 001`

重要机制实现方法

1. 跳转

   利用 `Branch` 和 `Jump` 来判断下一个时钟沿写入的 `PC` 值

2. 转发

   利用 $Tuse_{rs}$ 或 $Tuse_{rt} \geq$ 当前级数且 $Tnew \leq$ 当前级数，表示数据已经生成，可以直接转发

3. 暂停

   若 $Tuse_{rs} > Tnew$，则将处理器暂停，直到数据生成为止

二、测试方案

典型测试样例

汇编代码

```
ori $t0, $zero, 8
addu $t1, $t0, $zero
ori $t0, $zero, 8
ori $t2, $zero, 12
addu $t1, $t0, $zero
ori $t0, $zero, 8
ori $t1, $zero, 20
ori $t2, $zero, 4
ori $t3, $zero, 12
ori $t4, $zero, 16
sw $t0, 0($t1)
lw $t5, 0($t1)
ori $t6, $t5, 13
ori $t0, $zero, 8
ori $t1, $zero, 20
ori $t2, $zero, 4
ori $t3, $zero, 12
ori $t4, $zero, 16
sw $t0, 0($t1)
lw $t5, 0($t1)
ori $t7, $zero, 20
ori $t6, $t5, 13
ori $t0, $zero, 4
ori $t1, $zero, 8
ori $t2, $zero, 12
ori $t3, $zero, 16
sw $t0, 0($zero)
sw $t1, 0($t0)
sw $t2, 0($t1)
sw $t3, 4($t1)
lw $t4, 0($t0)
lw $t5, 0($t4)
lw $t5, -4($t0)
addu $zero, $zero, $t1
lw $t6, 0($t5)
lw $t6, 4($t0)
ori $s0, $zero, 1
ori $s1, $zero, 2
lw $t7, 0($t6)
```

机器码

```
34080008
01004821
34080008
340a000c
01004821
34080008
34090014
340a0004
```

```
340b000c
340c0010
ad280000
8d2d0000
35ae000d
34080008
34090014
340a0004
340b000c
340c0010
ad280000
8d2d0000
340f0014
35ae000d
34080004
34090008
340a000c
340b0010
ac080000
ad090000
ad2a0000
ad2b0004
8d0c0000
8d8d0000
8d0dfffc
00090021
8dae0000
8d0e0004
34100001
34110002
8dcf0000
```

期望结果

```
@00003000: $ 8 <= 00000008
@00003004: $ 9 <= 00000008
@00003008: $ 8 <= 00000008
@0000300c: $10 <= 0000000c
@00003010: $ 9 <= 00000008
@00003014: $ 8 <= 00000008
@00003018: $ 9 <= 00000014
@0000301c: $10 <= 00000004
@00003020: $11 <= 0000000c
@00003024: $12 <= 00000010
@00003028: *00000014 <= 00000008
@0000302c: $13 <= 00000008
@00003030: $14 <= 0000000d
@00003034: $ 8 <= 00000008
@00003038: $ 9 <= 00000014
@0000303c: $10 <= 00000004
@00003040: $11 <= 0000000c
@00003044: $12 <= 00000010
@00003048: *00000014 <= 00000008
@0000304c: $13 <= 00000008
@00003050: $15 <= 00000014
@00003054: $14 <= 0000000d
@00003058: $ 8 <= 00000004
```

```
@0000305c: $ 9 <= 00000008
@00003060: $10 <= 0000000c
@00003064: $11 <= 00000010
@00003068: *00000000 <= 00000004
@0000306c: *00000004 <= 00000008
@00003070: *00000008 <= 0000000c
@00003074: *0000000c <= 00000010
@00003078: $12 <= 00000008
@0000307c: $13 <= 0000000c
@00003080: $13 <= 00000004
@00003088: $14 <= 00000008
@0000308c: $14 <= 0000000c
@00003090: $16 <= 00000001
@00003094: $17 <= 00000002
@00003098: $15 <= 00000010
```

汇编代码

```
ori $t0, $zero, 8
ori $t1, $zero, 20
jal change1
ori $t2, $ra, 8
ori $t3, $zero, 14
change1:
    ori $t4, $zero, 18
ori $t5, $zero, 22
ori $t6, $zero, 26

ori $t0, $zero, 8
ori $t1, $zero, 20
jal change2
ori $t2, $zero, 4
ori $t3, $zero, 14
change2:
    ori $t4, $ra, 18
ori $t5, $zero, 22
ori $t6, $zero, 26

ori $t0, $zero, 4
ori $t1, $zero, 8
ori $t2, $zero, 12
ori $t3, $zero, 16
subu $t4, $t0, $t1
addu $t5, $t0, $t1
beq $t5, $t2, changeb1
ori $s0, $zero, 1
ori $s1, $zero, 2
changeb1:
    ori $s2, $zero, 3
    ori $s3, $zero, 4

addu $t6, $t0, $t1
ori $s0, $zero, 1
beq $t6, $t2, changeb2
ori $s0, $zero, 1
ori $s1, $zero, 2
changeb2:
    ori $s2, $zero, 3
    ori $s3, $zero, 4

addu $t7, $t0, $t1
ori $s0, $zero, 1
ori $s1, $zero, 2
beq $t7, $t2, changeb3
ori $s0, $zero, 1
ori $s1, $zero, 2
changeb3:
    ori $s2, $zero, 3
    ori $s3, $zero, 4

subu $t5, $t1, $t2
beq $t4, $t5, changeb4
ori $s0, $zero, 1
```

```
ori $s1, $zero, 2
changeb4:
    ori $s2, $zero, 3
    ori $s3, $zero, 4

subu $t6, $t1, $t2
ori $s0, $zero, 1
beq $t4, $t6, changeb5
ori $s0, $zero, 1
ori $s1, $zero, 2
changeb5:
    ori $s2, $zero, 3
    ori $s3, $zero, 4

subu $t7, $t1, $t2
ori $s0, $zero, 1
ori $s1, $zero, 2
beq $t4, $t7, changeb6
ori $s0, $zero, 1
ori $s1, $zero, 2
changeb6:
    ori $s2, $zero, 3
    ori $s3, $zero, 4
```

机器码

```
34080008
34090014
0c000c05
37ea0008
340b000e
340c0012
340d0016
340e001a
34080008
34090014
0c000c0d
340a0004
340b000e
37ec0012
340d0016
340e001a
34080004
34090008
340a000c
340b0010
01096023
01096821
11aa0002
34100001
34110002
34120003
34130004
01097021
34100001
11ca0002
34100001
```

```
34110002
34120003
34130004
01097821
34100001
34110002
11ea0002
34100001
34110002
34120003
34130004
012a6823
118d0002
34100001
34110002
34120003
34130004
012a7023
34100001
118e0002
34100001
34110002
34120003
34130004
012a7823
34100001
34110002
118f0002
34100001
34110002
34120003
34130004
```

期望结果

```
@00003000: $ 8 <= 00000008
@00003004: $ 9 <= 00000014
@00003008: $31 <= 00003010
@0000300c: $10 <= 00003018
@00003014: $12 <= 00000012
@00003018: $13 <= 00000016
@0000301c: $14 <= 0000001a
@00003020: $ 8 <= 00000008
@00003024: $ 9 <= 00000014
@00003028: $31 <= 00003030
@0000302c: $10 <= 00000004
@00003034: $12 <= 00003032
@00003038: $13 <= 00000016
@0000303c: $14 <= 0000001a
@00003040: $ 8 <= 00000004
@00003044: $ 9 <= 00000008
@00003048: $10 <= 0000000c
@0000304c: $11 <= 00000010
@00003050: $12 <= fffffffc
@00003054: $13 <= 0000000c
@0000305c: $16 <= 00000001
@00003064: $18 <= 00000003
```

```
@00003068: $19 <= 00000004
@0000306c: $14 <= 0000000c
@00003070: $16 <= 00000001
@00003078: $16 <= 00000001
@00003080: $18 <= 00000003
@00003084: $19 <= 00000004
@00003088: $15 <= 0000000c
@0000308c: $16 <= 00000001
@00003090: $17 <= 00000002
@00003098: $16 <= 00000001
@000030a0: $18 <= 00000003
@000030a4: $19 <= 00000004
@000030a8: $13 <= fffffffc
@000030b0: $16 <= 00000001
@000030b8: $18 <= 00000003
@000030bc: $19 <= 00000004
@000030c0: $14 <= fffffffc
@000030c4: $16 <= 00000001
@000030cc: $16 <= 00000001
@000030d4: $18 <= 00000003
@000030d8: $19 <= 00000004
@000030dc: $15 <= fffffffc
@000030e0: $16 <= 00000001
@000030e4: $17 <= 00000002
@000030ec: $16 <= 00000001
@000030f4: $18 <= 00000003
@000030f8: $19 <= 00000004
```

思考题

1. 在采用本节所述的控制冒险处理方式下，`PC` 的值应当如何被更新？请从数据通路和控制信号两方面进行说明。

   指令进入 D 级后，若是 `Jump` 指令则直接进行跳转，若是 `Branch` 指令则条件满足下进行跳转，若发生数据冒险，则利用转发或暂停进行调整。

2. 对于 `jal` 等需要将指令地址写入寄存器的指令，为什么需要回写 `PC + 8`？

   因为 `PC + 4` 是延迟槽内的指令，所以下一个指令地址应该是 `PC + 8`

3. 为什么所有的供给者都是存储了上一级传来的各种数据的流水级寄存器，而不是由**ALU**或者**DM**等部件来提供数据？

   为了不延长每一级的延迟，所有转发数据都应从流水级寄存器来进行传输，若使用 `ALU` 或者 `DM` 提供数据则会加长当前级的延长时间，并影响处理器的效能。

4. **“转发（旁路）机制的构造”**中的 **Thinking 1-4**

   Thinking 1：如果不采用已经转发过的数据，而采用上一级中的原始数据，会出现怎样的问题？试列举指令序列说明这个问题。

   会照成数据冒险，处理器不能跟着指令原意操作，最终产生错误的结果。

   Thinking 2：我们为什么要对**GPR**采用内部转发机制？如果不采用内部转发机制，我们要怎样才能解决这种情况下的转发需求呢？

   为了将数据能够正确传输所以对 `GPR` 使用内部转发机制。若不采用内部转发机制，可以利用直接在外部转发将 W 级的数据到 D 级的流水寄存器。

   Thinking 3：为什么**0**号寄存器需要特殊处理？

   因为 0 号寄存器无论写入什么都会保持 0 。

   Thinking 4：什么是**“最新产生的数据”**？

   指的是上前几个有关联写入寄存器的数据在哪个流水级寄存器产生，并且取最靠近当前级数的流水级寄存器。

5. 在**AT**方法讨论转发条件的时候，只提到了**“供给者需求者的A相同，且不为0”**，但在**CPU**写入**GRF**的时候，是有一个**we**信号来控制是否要写入的。为何在**AT**方法中不需要特判**we**呢？为了用且仅用**A**和**T**完成转发，在翻译出**A**的时候，要结合**we**做什么操作呢？

   因为 `we` 无效时，可以将 `A3` 设为 0 ，这样 `we` 无效也能保证数据不会通过转发传过来。

6. 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

   `addu->addu`：直接转发解决

   `addu->beq` ：暂停一周期再转发解决

   `lw->beq`：暂停两周期再转发解决

   `lw->addu`：暂停一周期再转发解决

   `jal->addu`：直接转发解决

   `lui->beq`：直接转发解决

   `lui->addu`：直接转发解决

   随机测试覆盖率虽然大，但是必须思考可能会发生的冲突情况，所以手动测试是必要的，而且将同一个寄存器的使用率提高可以制造更多的冲突，所以完全依靠随机生成的测试并不能把所有的情况都测出来。

   对于这次的流水线处理器转发和暂停处理方式都依靠着 `DataDone` 这个信号进行判断，在指令输入进来后并定位运算后数据的流水级寄存器出口，以便判断暂停和转发的时机。