北京航空航天大学: B3I062410 计算机组成

所有讨论 ▾ 搜索所有帖子

显示所有 ▾ 近期活动 ▾

写零号寄存器时的转发问题 2
报告的提交 2
我的单步调试方法 3
【P3】课上测试差一周期 8
关于寄存器堆内部转发问题 2
添加指令的问题 3
关于进行有符号乘除的判断 6
计数器的初始化问题 5
顶层视图能否发一下？ 5
DM能否像I/O设备一样放在桥后面？ 1
由：教员
能不能发p5.3课上我自己的代码？ 6

此帖对所有人可见。

我的流水线处理器设计方法

2票

由 **15231099** 发表于6 天 以前的讨论

分类

所有的分类原则都是为了能够描述指令的行为在某些方面具有的一致性。而在实际中我们可能要根据不同控制器的需要找到最细化（通过组合同时满足各种需求）的分类方法。我觉得高老板提出的分类方式是十分高明的。

	R_CALC	I_CALC	MD_CALC	LOAD	STORE	BRANCH	JR	JALR	JAL
ID:R						RS RT	RS	RS	
EX:R	RS RT	RS	RS RT	RS	RS				
ME:R					RT				
WB:W	RD:MEI	RT:MEI		RT:WBI				RD:EXI	31:EXI

这个是我用的分类，这个分类的原则是**具有相同的寄存器读写行为**，每一行表示在某个阶段用到的寄存器（需求）和在回写阶段写入的寄存器（冒号后面是结果最先产生的位置）。分类中充分利用零号寄存器读写无hazard的特性，可以将很多指令归并。但是对于特殊指令也要特殊处理（比如JR类包含BLEZ, BGEZ）。

进而大概对于转发和暂停需要的分类就是这样的了（读写HILO的地方有待改进）：

Type					
RS_IDI	BRANCH	JR	JALR		
RT_IDI	BRANCH				
RS_EXI	R_CALC	I_CALC	MD_CALC	LOAD	STORE
RT_EXI	R_CALC	MD_CALC			
RT_MEI	STORE				
RW_HI_LO	MFHI	MFLO	MTHI	MTLO	

具体实现上来说，我的分类是用宏来写的，目的是为了减少代码的重复（这个重复只是对于写代码的人来说，对于机器生成的组合逻辑电路一点都不会少）。举个例子代码大概长这样

```
`define IS_MD_CALC(Inst) \
   (Inst[`OPCODE] == `OPCODE_R \
&& (Inst[`FUNCT] == `FUNCT_MULT \
 || Inst[`FUNCT] == `FUNCT_MULTU \
 || Inst[`FUNCT] == `FUNCT_DIV \
 || Inst[`FUNCT] == `FUNCT_DIVU))

`define READ_RS_IDI(Inst) \
    ( `IS_BRANCH(Inst) \
    ||`IS_JR(Inst) \
    ||`IS_JALR(Inst) \
    )
```

暂停

暂停是为了处理数据实在无法转发的情况的，如果完全解决了暂停的问题，转发也会变得很简单。

总的来说暂停的原因有两大类：数据冲突、资源（乘除器）占用。

按照分类，具有写入行为的有R_CALC, I_CALC, LOAD, JALR, JAL。R_CALC, I_CALC在ME段数据已可转发；LOAD要到WB段数据才可转发；JAL, JALR在EX段已可转发。也就是说只有ID段和EX段的数据需求的冲突问题。

对于乘除器，因为现在的乘除运算对结果有覆盖（如果有MADD之类的就得另外考虑了），连续两个乘除指令并不需要暂停（也就是说乘除器的Start信号并不引起暂停）。

按照高老板的需求时间模型，得出不能满足而需要暂停的地方。

每一行对应的代码大概是这样的

```
wire _IDI_RS = `READ_RS_IDI(IDI) && IDI[`RS] != 0 && (
    `IS_R_CALC(EXI) && EXI[`RD] == IDI[`RS] ||
    `IS_I_CALC(EXI) && EXI[`RT] == IDI[`RS] ||
    `IS_LOAD(EXI)   && EXI[`RT] == IDI[`RS] ||
    `IS_LOAD(MEI)   && MEI[`RT] == IDI[`RS]
);
```

转发

因为已经解决的暂停的问题了，所以剩下的所有冲突都能用转发来解决。我的思路是对于每一级（ID EX ME WB）都有对应的写入目标和写入数据，这个时候不妨利用每一级的控制器产生每一级的写使能RegWr，写入目标RegDst，写入数据类型RegWrData控制信号，然后给出每一级的写入数据。

```
assign EXO_WriteDst  = `MUX_4(EXC_RegDst, EXI_Inst[`RT],
EXI_Inst[`RD], 5'd31, 5'b0);
assign MEO_WriteDst  = `MUX_4(MEC_RegDst, MEI_Inst[`RT],
MEI_Inst[`RD], 5'd31, 5'b0);
assign WBO_WriteDst  = `MUX_4(WBC_RegDst, WBI_Inst[`RT],
WBI_Inst[`RD], 5'd31, 5'b0);
assign END_WriteDst  = `MUX_4(END_RegDst, END_Inst[`RT],
END_Inst[`RD], 5'd31, 5'b0);

assign EXO_WriteData = `MUX_4(EXC_Data, FDT_EXI_RS, 0,
EXI_PC + 4, 0);
assign MEO_WriteData = `MUX_4(MEC_Data, MEI_ALUOut, 0,
MEI_PC + 4, 0);
assign WBO_WriteData = `MUX_4(WBC_Data, WBI_ALUOut,
WBI_DMOut, WBI_PC + 4, 0);
```

这里注意到了一个冲突的问题，就是唯一一个在ME段的读取的转发问题。我的方案是增加以及流水线寄存器，保存上上条指令和它的写入数据（END_WriteData）。

这样就有了每一级的写入数据了，于是可以构造这样的一个转发控制器

MUX	CTL	0	1	2	3
FDT_IDI_RS	FTL_IDI_RS	IDO_Reg_1	EXO_WriteData	MEO_WriteData	WBO_WriteData
FDT_IDI_RT	FTL_IDI_RT	IDO_Reg_2	EXO_WriteData	MEO_WriteData	WBO_WriteData
FDT_EXI_RS	FTL_EXI_RS	EXI_Reg_1		MEO_WriteData	WBO_WriteData
FDT_EXI_RT	FTL_EXI_RT	EXI_Reg_2		MEO_WriteData	WBO_WriteData
FDT_MEI_RT	FTL_MEI_RT	MEI_Reg_2	END_WriteData		WBO_WriteData

代码长这样（注意优先顺序）

```
assign FTL_IDI_RS = (IDI[`RS] == 0 || ~`READ_RS_IDI(IDI)) ?
2'd0 :
    EXI_RegWr && EXI_RegDst == IDI[`RS] ? 2'd1 :
    MEI_RegWr && MEI_RegDst == IDI[`RS] ? 2'd2 :
    WBI_RegWr && WBI_RegDst == IDI[`RS] ? 2'd3 : 2'd0;

assign FDT_IDI_RS = `MUX_4(FTL_IDI_RS, IDO_Reg_1,
EXO_WriteData, MEO_WriteData, WBO_WriteData);
assign FDT_IDI_RT = `MUX_4(FTL_IDI_RT, IDO_Reg_2,
EXO_WriteData, MEO_WriteData, WBO_WriteData);
assign FDT_EXI_RS = `MUX_4(FTL_EXI_RS, EXI_Reg_1, 0,
MEO_WriteData, WBO_WriteData);
assign FDT_EXI_RT = `MUX_4(FTL_EXI_RT, EXI_Reg_2, 0,
MEO_WriteData, WBO_WriteData);
assign FDT_MEI_RT = `MUX_4(FTL_MEI_RT, MEI_Reg_2,
END_WriteData, 0,              WBO_WriteData);
```

这样在每一级使用到寄存器数据时都用转发数据的MUX取代就能完全转发了。

小结

这个设计的好处在于对于暂停和转发的考量，只需要在分类的时候考虑清楚就好了。也就是说分类分好了，就不存在hazard的问题。

变态指令实战

最后提一下我见过的最糟糕的两种指令：MOVZ、BGEZAL。这两者我是在P5.2课下就已经想清楚也做好了的，所以课上测试还算是比较轻松。但是，我真的不认为让人用课上那短短的时间去想一个鲁棒可行的实现方法是多么正常的想法：），难道课上测试不是应该只是要证明这是我自己做的玩意就足够了么。

我的方法的关键是**指令在流水线上是可被修改的**，看一段代码就能明白了

```
// Special HACK for MOVZ MOVN BGEZAL BLTZAL
assign IDO_InstHack = IDI_Inst[`OPCODE] != `OPCODE_R ?
IDI_Inst :
IDI_Inst[`FUNCT] == `FUNCT_MOVZ && FDT_IDI_RT != 0 ?
`DISABLE_WRITE(IDI_Inst) :
IDI_Inst[`FUNCT] == `FUNCT_MOVN && FDT_IDI_RT == 0 ?
`DISABLE_WRITE(IDI_Inst) :
IDI_Inst[`OPCODE] == `OPCODE_REGIMM && IDI_Inst[`RT] ==
`RT_BLTZAL && $signed(FDT_IDI_RS) >= 0 ?
`DISABLE_WRITE(IDI_Inst) :
    IDI_Inst[`OPCODE] == `OPCODE_REGIMM && IDI_Inst[`RT] ==
`RT_BGEZAL && $signed(FDT_IDI_RS) <  0 ?
`DISABLE_WRITE(IDI_Inst) :
IDI_Inst;
```

DISABLE_WRITE具体是怎样的就任凭想象了（换成NOP也是可以的）。这样ID段往流水线寄存器写入的就是一个写屏蔽的指令（我称之为HACK）。这样做对整个系统的结构调整是极小的，要做的只有分类上的调整了。

值得注意的是，MOVZ, MOVN都提前到ID段处理的（因为这指令放在后面去HACK也是会有问题的），所以这个设计的代价是可能造成暂停。不过其结果的产生也提前到了EX段，所以此处的转发也是能减少暂停的。

2条回复

添加回复

15061134
5 天 以前
0票

最后的跟我课上测试的想法一模一样。

不过如果不是使用分布式译码的控制器的话（让RegWr信号一直往下流），想到这个方法（写屏蔽）也不是难事：）

5 天 以前之前由 **15231099** 提交

添加评论

15071030
5 天 以前
0票

我感觉咱俩的设计从头到尾都几乎一模一样。。。那两条指令的处理我也是用的变指令，虽然理论上会浪费一个周期

:)一模一样是不是夸张了点。。。

4 天 以前之前由 **15231099** 提交

一模一样，确实多少有些夸张，呵呵。但也还好，因为一旦找到方法，你就会发现方法最终大体差不多，特别是当目标为追求最精致结果后。

为什么没有表情包呢？！吐个槽。。。

4 天 以前之前由 **Gao_XiaoPeng** 教员 提交

3 天 以前之前由 **qiuqiu** 教员 提交

添加评论

显示所有的回复

回复：

预览

提交