MIPS 32 流水线设计文档

# 概述

## 设计目标

1. 实现经典的五段流水线CPU；
2. 基本兼容MIPS32r1指令集体系结构，能够执行简单的通用程序。

## 设计基础

在开始流水线设计之前，需要保证以下功能部件已经理解设计原理并且被正确实现：

1. 支持基本MIPS算数逻辑运算的ALU；
2. 桶形移位器；
3. 通用寄存器组。

此外，还需要了解MIPS指令的基本格式，能阅读指令文档并理解指令行为。

## 设计步骤

### 确定功能部件

设计流水线，首先需要明确的是流水线中的最基本元素，功能部件的种类以及数量，以及各个功能如何连接形成数据通路。

首先，为了能够执行简单的程序，需要实现常用的MIPS指令。所以，需要根据实现的指令种类，确定需要哪些功能部件。本设计实现的MIPS指令及其需要功能部件如下表所示（通用寄存器组、指令地址寄存器PC以及指令存储器因为所有指令都会用到所以省略）：

|  |  |  |
| --- | --- | --- |
| 指令类型 | 指令名称 | 需要的功能部件 |
| 算数逻辑指令 | ADD | ALU(加法)， |
| ADDI | ALU(加法)，符号扩展电路 |
| ADDIU | ALU(加法)，无符号扩展电路 |
| SUB | ALU(减法) |
| SUBU | ALU(减法)，无符号扩展电路 |
| AND | ALU(逻辑按位与) |
| ANDI | ALU(逻辑按位与)，无符号扩展电路 |
| OR | ALU(逻辑按位或) |
| ORI | ALU(逻辑按位或)，无符号扩展电路 |
| XOR | ALU(逻辑按位亦或) |
| XORI | ALU(逻辑按位亦或)，无符号扩展电路 |
| NOR | ALU(逻辑按位或非) |
| SLT | ALU(减法)，条件判断电路 |
| SLTU |
| SLTI | ALU(减法)，条件判断电路，符号扩展电路 |
| SLTIU | ALU(减法)，条件判断电路，无符号扩展电路 |
| 移位指令 | SLL[V] | 桶形移位器 |
| SRL[V] |
| SRA[V] |
| 其他运算指令 | CLO | ALU(前导1) |
| CLZ | ALU(前导0) |
| 分支跳转指令 | BEQ | 条件判断电路，跳转地址计算逻辑，PC选择电路 |
| BEQL |
| BGEZ |
| BGTZ |
| BLEZ |
| BLTZ |
| BNE |
| 绝对跳转指令 | J | 绝对地址计算电路，PC选择电路 |
| 过程调用指令 | JAL |
| JR | PC选择电路 |
| 数据移动指令 | MOVN | 判零电路 |
| MOVZ |
| LUI | 位拼接电路 |
| 乘除法指令 | MUL | 乘法器 |
| MULT | 带符号乘法器，LO寄存器 |
| MULTU | 无符号乘法器，LO寄存器 |
| DIV | 带符号除法器 |
| DIVU | 无符号除法器 |
| MFHI | HI寄存器 |
| MTHI |
| MFLO | LO寄存器 |
| MTLO |
| 存储相关指令 | LB | 数据存储器（符号扩展电路） |
| LBU |
| LH |
| LHU |
| LW |
| SB |
| SH |
| SW |
| LWL | 数据存储器，地址对齐电路 |
| LWR |
| SWL |
| SWR |
| 异常相关指令 | MFC0 | CP0寄存器组 |
| MTC0 |
| SYSCALL | CP0寄存器组，PC选择电路 |
| ERET |
|  | NOP |  |

所以需要的基本功能部件有：PC寄存器、指令存储器、指令译码器、ALU、符号扩展部件、条件判断部件、分支地址计算部件、乘除法器、HI/LO寄存器、数据寄存器、CP0寄存器组、Load/Sotre指令地址对齐部件。剩下的一些功能通过选择器和简单的组合逻辑即可实现，不再赘述。

为了提高效率，并使得CPU设计更接近常规实现，指令和数据共享一个存储器，并在其上划分出指令cache和数据cache以提高访存速度以及消除结果冒险。

以上即为了完成基本的指令执行所需要的功能部件。由于设计采用了流水线，所以需要流水段寄存器IFID、IDEX、EXMEM、MEMWB，以及检测控制冒险和Load-Use冒险的控制单元以及为了消除其他写后读数据冒险的转发单元。

### 规划数据通路

确定功能部件的种类和数量后，需要确定每条指令的执行路径，即每条指令的数据通路。对于可以重叠的部分，则通过指令译码器生成控制信号，配合选择器进行多路复用，具体的例子有：

1. R-Type指令使用Rd字段作为目标寄存器地址，而I-Type指令使用Rt字段作为目标寄存器地址。对指令的Rd字段和Rt字段连线进行进行多路复用，根据指令类型选择其一送入通用寄存器组的写地址口；
2. R-Type指令使用GPR[ Rt ]作为第二操作数，而I-Type指令使用（无）符号扩展后Imm作为第二操作数，对二者进行多路复用，选择其一送入ALU的第二操作数端口。

用译码器产生的控制信号和选择器合并重叠的数据通路之后，需要将这个完整的数据通路划分到各个流水段。

五段流水线将一条指令的执行划分为“取指(IF)、译码(ID)、执行(EX)、访存(MEM)、写回(WB)”五个阶段。本流水线对数据通路的划分如下表所示：

|  |  |
| --- | --- |
| 流水段 | 段内行为 |
| 取指 | 1. 更新PC寄存器 2. 从指令cache取出指令 3. 将指令写入IFID流水段寄存器 4. 将PC + 4送入分支预测器计算下个周期待发射的指令地址 |
| 译码 | 1. 将IFID流水段寄存器的指令送入译码器进行译码 2. 根据译码结果取得寄存器操作以及立即数的符号扩展 3. 将需要的信号和数据写入IDEX流水段寄存器 |
| 执行 | 1. 根据来自IDEX的控制信号选择操作数送入ALU和移位器进行计算 2. 根据溢出结果和控制信号计算新的寄存器写使能 3. 计算非对齐地址的写使能并对齐待写入数据 4. 根据控制信号选择结果作为EX段的结果和其它需要的数据写入EXMEM流水段寄存器 |
| 访存 | 1. 条件判断并计算出正确的下条指令地址 2. 根据控制信号访问数据cache 写入或读出数据 3. 对齐非对齐地址读出的内存数据 4. 将需要的信号和数据写入MEMWB流水段寄存器 |
| 写回 | 1. 执行对GPR或CP0寄存器的写入操作 |

### 控制流水线

上一小节对简述了本流水线设计对指令执行的数据通路的流水段划分策略，但是数据通路的各段行为只涉及与指令执行直接相关的部件以及流水段寄存器还有分支预测器。这一小节则会介绍流水线特有的功能部件转发单元和控制单元的跨流水段行为。

转发单元虽然跨越流水段，但是只作用于EX段。它接受来自MEM段和WB段的数据、寄存器写使能以及寄存器地址，判断EX段的来自GPR[ Rs ]和/或GPR[ Rt ]的数据是否应该被MEM段或者WB段的数据覆盖。这主要通过写使能以及匹配地址来判断。需要注意的是MEM段的指令在WB段的指令后执行，其数据是更新的，所以MEM段的转发优先级要高于WB段。需要理解的是这些数据实际上是直接从EXMEM和MEMWB流水段寄存器送入转发单元的。也就是是上个周期EX和MEM段获得的数据。所以MEM段访存获得的数据是无法及时转发给EX段的，此即Load-Use冒险。易见转发单元无法解决Load-Use冒险。

控制单元负责对整个流水线的控制。它识别控制冒险、Load-Use冒险以及cache miss，从而阻塞和/或冲刷特定流水段寄存器，保证指令按需提交（即进入WB段的顺序正确）以及执行时能获得正确的数据。

控制冒险主要通过判断MEM段分支地址计算单元计算出来的正确的下条指令地址（注意不是单纯的分支指令地址）并与实际紧跟着进入流水线的EX段指令地址是否一致来确定。一旦发生控制冒险，冲刷IFID、IDEX、EXMEM流水段寄存器，修改PC选择子，让正确的指令在下个周期进入流水线，而中间的错误指令都无效化。

Load-Use冒险主要通过判断ID段Rs和/或Rt是否与EX段的LW类指令的Rd一致来判断。一旦发现Load-Use冒险，阻塞IFID流水段寄存器，冲刷IDEX流水段寄存器，保证下个周期ID段往前的流水段保持不变而EX插入一个气泡（无效信号）。虽然此时依然没有办法让ID段的译码获得正确的操作数，但是下个周期LW进入WB段，可以通过转发将load到的数据正确地送入EX段。同时可以在这里断言：“Load指令只会在WB段被转发数据”。

另外需要注意的是，由于控制单元每个周期都在检查MEM段分支地址计算模块给出的正确下地址与EX段的PC进行匹配，如果其中一个流水段是或者两个都是气泡，则会给控制冒险的判断带来干扰。如果忽视气泡，则存在一种极端情况，即分支预测器预测了一条错误的指令，且这个指令与前一条指令产生了Load-Use冒险，这两个指令之间会插入一个气泡，从而导致这个控制冒险无法被监测到。为了检测这种情况，需要额外传递一些信息到MEM段并最终送入控制单元。

控制单元除了为了保证正确性而阻塞和/或冲刷流水线外，还需要在检测到跳转类指令时修改PC来源的选择子，从而实现跳转实现。这里需要注意的是绝对跳转指令J和SYSCALL是在ID段就可以跳转，而其他跳转类指令，则要到MEM段才能判断是否跳转。由于进入MEM段的指令一定是正确的，所以具有更高的优先级来控制PC来源。

在下一小节中，还将介绍控制单元在MIPS异常机制中的功能。

### 实现异常机制

MIPS的异常机制可以这样简单理解：CP0中status寄存器的字段控制CPU是否响应异常。一旦响应异常，需要向CP0的cause寄存器写入exec\_code，用来表明异常类型；修改status寄存器以关闭异常响应；同时保存返回地址到CP0的EPC寄存器，PC修改为中断向量，跳转执行异常处理程序。

CP0作为一个功能模块，其作用与GPR相似，除了写寄存器优先级的判断----因为有控制单元和MTC0两个都会写CP0的寄存器----以外并没有复杂的控制逻辑。所以这一套中断响应操作由控制单元完成。

需要关注的一个点是EPC的保存，这是关乎正确性的地方。由于控制单元对所有指令都执行控制冒险检查，所以通过分支地址计算单元得出的正确的下条指令地址十分适合作为异常返回的地址。但是MEM段不可避免地会出现气泡，此时没办法知道正确的下条指令地址，而且EX段也可能是气泡，所以并不能非常轻松地获得这个时候的异常返回地址。为了解决这个问题，在控制单元内额外存储了上一个正确提交的指令PC，正确提交像前面提到的那样，即一条指令进入MEM段，实际的提交时刻应该在WB段的结束，但是MEM段足以保证正确性，所以可以从这个正确的指令重新开始执行（存疑）。

在介绍完流水线的整体设计思路后，下面解释一个关于设计考量的问题。

|  |
| --- |
| Q：既然延迟槽是汇编器一定会插入的，为什么使用分支预测器而不是延迟槽？ |
| A：虽然延迟槽是编译MIPS32r1程序时的默认选项。但是为了实现1周期的分支延迟损失，需要在ID段就完成条件判断，使得ID段功能膨胀，时延增加。由于比较参数来自寄存器，更是需要往ID段转发数据，同时Load-Use冒险将阻塞两个周期。在异常处理中，对延迟槽内的指令也要做检测和特别处理。综上所述，延迟槽并不是一个非常简洁，同时也不是现在通行的设计思路。故本设计采用更加常用的分支预测器。 |

## 关于本文档

本文档由概述，设计过程，代码实现，调试策略，以及若干关于实现细节的附录组成。概述部分已经介绍了关于设计的一些思路性内容。之后的章节将更多地关注具体的过程性内容以及细节解释。

文档的前半部分相当于对设计的过程记录和记忆巩固，后半部分相当于对代码的详细说明以及对MIPS体系结构的理解性内容，可用于阅读Verilog代码时加深理解。

# 设计过程

## 绘制框图

在实际编码之前，需要绘制基本的流水线框图。以框图来知道代码的编写，可以保证准确率和编码效率。

常规情况下，使用Visio画图是一个不错的选择，但是Visio在自动折线以及端点吸附上，操作不是非常方便。实际设计时，使用Logisim进行框图的绘制，Logisim的获取地址在[http://www.cburch.com/logisim](http://www.cburch.com/logisim/)。Logisim不仅可以自动折现以及端点吸附，还可以检查总线宽度与端口位宽是否吻合。此外，选中一条连线，可以轻松地观察到所有连接的端口。Logisim还可以进行电路仿真，不过这里并不需要用到。

首先绘制的是各个流水段寄存器以及指令执行所需要的基本功能部件。设定好各个部件的端口名以及位宽，为之后的总体连线做准备。

接下来连结使用各个功能部件，使得流水线框图基本成型。此时开始考虑跨流水段的功能部件，比如控制单元和转发单元，所需要的接口，并将它们在合适的位置接入流水线。

基本的流水线框图完成后，在其上还进行了几番迭代。主要原因是对流水线以及冒险控制的思考更加深入，考虑到了各种边界情况。为了处理这些边界情况，往往需要在流水线框图上增加连线以让某些部件获取更多信息。只是越到后面增加连线和部件的难度越大。

在绘制流水线框图的过程中，还需要时刻记录对流水线设计的思考以及各个功能部件具体的行为描述，因为流水线框图只描述了流水线宏观的组织结构，即顶层的连线。具体的功能部件行为则记录在分散的文档中。最终有一个总的文档用来描述流水线中出现的所用信号的含义，作为编写代码时的备忘。

## 编写代码

有了功能部件的行为文档做支持，代码编写可以分配给多个人独立完成，设计与实现分开进行也有利于在实现时审校模块设计的正确性。

为了方便多人协作编写代码，可以使用Git作为版本管理工具，同时在GitHub上托管项目代码，本设计的项目地址在<https://github.com/NJU-CS-SYS/ArchLab2015>。协作开发的基本流程如下：

1. 配置开发环境：Git、Vivado、交叉编译等；
2. 从项目托管网站（远程仓库）上clone项目到本地；
3. 在本地进行开发和调试；
4. 确认完成编码目标后在本地提交修改；
5. 向远程仓库推送更新的代码；
6. 如果发生冲突，获取最新的远程仓库代码，尝试合并代码以消除冲突。

实现功能模块以外，还需要对功能模块进行仿真测试。但是单独的功能仿真只能验证功能模块（比如控制单元）的代码实现是否符合设计文档的要求。至于设计是否正确，还需要将所有功能部件加入流水线后做整体测试才能验证。

由于最终连结流水线需要将所有功能部件模块实例化并用wire或者reg类型变量进行连接，连结时要清楚使用的变量作用于那个流水段、来自哪个功能部件、应该输送到哪里，所以规定一套命名规范显得尤为重要。

需要注意的是，只有流水线顶层模块才有比较多的变量名，所以功能模块内部并不一定需要也不一定适用针对流水线的命名规范，但是，功能模块的端口名称应该做到尽可能不加修改地表明其在顶层流水线中需要连结的信号所处的流水段和部件来源。

一开始的命名规范是这样的（中括号内的内容表示可选）：

|  |  |  |
| --- | --- | --- |
| 命名范式 | 例子 | 含义 |
| [来源\_]对象[\_性质][\_行为] | ifid\_rs\_addr | 来自ifid流水段寄存器的Rs地址 |
| cu\_pc\_src | 来自控制单元的PC来源选择子 |
| operand\_A\_after\_forwarding | 经过转发的ALU的第一次操作数 |
| operand\_B\_after\_selection | 经立即数选择的ALU第二操作数 |

这样的命名规范能够增加代码的自解释性，即便脱离文档直接阅读代码，只要具有相近的背景知识就能够明白变量名背后的含义。

但是在实践过程中出现了一些差错。因为绘制框图时流水段寄存器的端口命名没有很好地区分流水段以及流水段寄存器的名称，即idex流水段寄存器的输出端口都带有idex前缀而exmem流水段寄存器的输入端口都带有idex前缀。虽然直连的输出和输入端口保持变量名一致不是什么坏事，但是使用的前缀带来了很大的误导。一开始的想法是，用流水段寄存器输出端口的前缀来表明信号的来源。但是指令译码器为了区分哪些信号是只在ID段使用的，哪些信号是送入idex流水段寄存器进行传递的，所以将送入idex流水段寄存器的信号名加上了前缀idex。这造成了名冲突，为了解决这个问题，同时保证变量名的表达性，对于流水段寄存器的输出信号，使用流水段名作为前缀而不是流水段寄存器的名字。这样，idex向exmem流水段直接传递的信号名都是ex前缀，而exmem向memwb流水段寄存器直接传递的信号名都是mem前缀。但是由于这是在编写代码时临时采取的编码规范，而整合流水线的顶层模块主要参考之前绘制好的框图，所以出现了不少纰漏，主要错误集中在错误地在ID段之后使用了idex前缀的信号名导致连线错误。

此外由于变量名过多，且Verilog缺少有效的静态分析工具，所以经常出现所谓的typo，即拼写错误，而Verilog对这种没定义的变量，则是默认为位宽为1的信号。所以应该尽量使用由代码提示的编辑器，避免使用Vivado自带的编辑器！一般比较好的选择是Notepad++以及VIM/EMACS。Notepad++自带对已有符号的提示，VIM使用Ctrl-N有类似效果，都是避免typo的好工具。

## 调试代码

最基本的功能仿真在各个功能部件实现时已经做过，不过其只能验证实现的正确性，而不能验证设计的正确性。这里所讲的调试关注的是设计的正确性，即一条指令从发射到提交的执行过程正确性、多条指令之间产生冒险与转发关系的处理机制正确性以及异常处理机制的正确性。

调试在平台方面可以分为两大类。一种是通过ISE（如Vivado）的对流水线顶层模块进行功能仿真；另一种是直接烧录到开发板上进行测试。后者对流水线设计与实现正确性的要求很高。如果不能保证bug足够少，用这种测试方法会比较困难，因为每一次对程序的修改都要经过综合（Synthesis）、实现（Implementation）、产生二进制码流（Bistream Generation）等过程，时间消耗大且能获取的信息少。所以优先采用第一种平台，即使用功能仿真进行调试。

调试在策略方面又分为两大方向，一个方向直接使用通用程序，创造足够复杂的执行场景进行验证；另一个方向则手动编写汇编代码，精心设计测试样例，在简化的场景下有的放矢地对边界情况进行测试。

两个测试策略都需要使用助教提供的工具链，主要用于将C源文件和汇编代码文件编译成二进制可执行代码后将其反汇编，同时生成可用于Vivado的内存初始化文件。工具链的下载地址及使用方法参见<https://github.com/jackyangNJ/testbench4mips>。

由于代码托管在GitHub上使用Git进行版本管理，所以应该尽量避免每个人工作环境的差异带来的源代码修改。而用于读入内存初始化文件的Verilog系统任务$readmemh必须使用内存初始化文件的绝对路径来进行读取。为了避免每个人在内存模块设计代码里硬编码自己的工作路径，代码中的内存初始化文件名写成宏定义的形式。宏在Vivado仿真环境里进行定义。

|  |
| --- |
| Q：如何在仿真时定义宏？ |
| A：在Vivado默认窗口布局下，选择Simulation Settings。在弹出的对话框中找到Verilog options，点击Defines添加宏名以及定义。 |

以上是调试的准备工作，下面介绍调试的具体工作。

### C源代码调试

对于C源代码的测试，需要注意的是流水线CPU相当于是裸机状态，缺少运行时环境，尤其是栈空间的分配。所以，需要准备一份特殊的start.S汇编文件，在里面实现基本的运行环境设计，主要就是将栈指针$sp分配到内存的高地址边界，以提供最大可能的栈空间。之后再调用main函数执行程序。为了保证先设置好栈空间在进入main函数，需要做两件事：

1. 修改链接脚本，将start.S中的入口函数的名字作为entry point；
2. 保证start.S中的入口函数位于可执行文件开头的部分。

第一点只是为了符合常规的编译思路，关键是第二点。并没有什么方法能很好的控制代码生成的位置。即便是entry point，也不一定就位于可执行文件的起始部分。但是在实践中，只要保证入口函数在start.S的开头，并且没有其它汇编文件，则基本能满足第二点条件。

如何检验一个程序是否被正确执行？首先需要做的就是宏观语义上的断言，即一个程序执行到这里应该是什么状态，这需要预先知道测试样例对输入会产生什么样的结果（这里不使用输出这个词，因为本CPU设计暂时无法进行IO）以及什么样的中间状态。可以准备两个死循环函数，或者无限递归函数（下面统称trap函数），一个叫good，一个叫bad，在断言失败是调用bad，若正常退出则在start.S里进入good。通过观察仿真界面的波形图确定PC最终稳定在good还是bad来判断一个程序是否被正确执行。需要提醒的是，之后MEM段往后的PC才是可以参考的。而且无论以哪种方式实现trap，都会带来气泡或无关指令。也就是，进入trap的模式应该是MEM段的PC在很长一段时间内交替地变成trap里的PC和一个气泡或无关指令对应的值。

断言是一把双刃剑。一方面，它能及时地检查出错误，方便调试；另一方面，插入断言意味着至少插入了一条分支跳转指令和一条过程调用指令。这种指令的插入可能导致一些极端的冒险场景被消除，从而给bug绪了一命。此外，断言也不能检查出所有的错误，比如，跳转类指令错误地实现，以至于可以很快结束程序，进入good trap。所以，不能盲目相信断言和测试用例。为了弥补断言的局限性，需要利用更高级的测试手段，这点将在之后的小节中介绍。

目前通过测试的程序主要有：冒泡排序，采用两种划分算法的快速排序，交换函数，求和函数。测试样例的具体实现参见代码仓库:

<https://github.com/NJU-CS-SYS/ArchLab2015/tree/master/testcase>

### 手写汇编指令调试

汇编指令测试又可以分为两部分。

首先是单周期所使用的汇编测试代码，其主要测试一条指令的执行结果是否正确，每个测试单元涉及操作数的载入、待测试指令的执行、检查指令结果是否符合预期、陷入错误死循环或进入下一个测试单元。

这个测试在C源代码测试之前进行，确实检查出了一些问题，在DEBUG后能够通过。但是还是不够充分的。主要问题在于它没有非常有针对性的检测转发和冒险问题，没有考虑每条指令在流水线中会具有什么单周期里不具有的新特性，同时测试指令也不是很全面。所以，重新设计的针对单条指令的测试是有必要的。此外，像系统调用和中断机制这种暂时缺少基础设施如中断服务程序和外设的待测试内容，暂时也只能通过针对性的汇编和仿真测试进行，直接从C源代码构建这样的测试场景的是比较困难的。

对新增加的以及容易出现错误的指令额外手写汇编代码进行测试，涉及的指令包括分支指令、jr、syscall、eret等。以分支指令为例介绍其具体的测试情况，下面的文字内容展现了手写汇编测试的整套流程，其他待测指令也是类似的：

#### 列表

|  |  |
| --- | --- |
| 指令 | 功能描述 |
| bgez | 若寄存器rs的值大于等于0，则指令的转移地址由有符号偏移量左移2位来决定，即将offset左移两位之后，进行符号拓展并与当前PC相加。 |
| bgtz | 若寄存器rs的值大于0，则指令的转移地址由有符号偏移量左移2位来决定，即将offset左移两位之后，进行符号拓展并与当前PC相加。 |
| blez | 若寄存器rs的值小于等于0，则指令的转移地址由有符号偏移量左移2位来决定，即将offset左移两位之后，进行符号拓展并与当前PC相加。 |
| bltz | 若寄存器rs的值小于0，则指令的转移地址由有符号偏移量左移2位来决定，即将offset左移两位之后，进行符号拓展并与当前PC相加。 |

#### 测试：

以Bltz为例，首先把4、8、16、27、28、29号寄存器的值分别置为0、1、-1、0、0、0。然后按下表书写汇编代码：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PC(相对) | Opcode | Rs | Rt | Offset(十进制) |
| 0 | 000001 | 00100 | 00000 | (32) |
| 4 | Load指令给$27赋值3 | | | |
| 8 | 000001 | 01000 | 00000 | (24) |
| 12 | Load指令给$28赋值3 | | | |
| 16 | 000001 | 10000 | 00000 | (16) |
| 20 | Load指令给$29赋值3 | | | |
| 24 | 24到28死循环 | | | |
| 28 |
| 32 | 下一个测试 | | | |

查看寄存器的值$27为0，$28为0，$29为3正确。

其他分支指令的测试类似，只是将操作码000001替换为自己的即可，不再追溯。

与单周期指令测试相比，这里针对同一指令进行了连续多次测试。

## 高级调试工具设计

上文提到过，断言是有局限的，其插入指令会导致冒险场景的消失，同时错误的指令跳转和条件判断可能直接跳过断言。

此外，断言在调试效率上也是有限的。虽然一检测到错误就会陷入bad trap，控制指令执行流的复杂度。但是既有的复杂度往往已经很高，很难追踪到到底是哪里发生了错误。

由于本设计是CPU的教学实验，所以可以利用正确的指令集处理机实现（然而如果是在开发原创的指令集实现，则没有这么幸运了），比对二者的执行流，找到第一个不一样的地方，则可以快速定位错误。执行流只需要看提交的PC的顺序，无论是获取还是比较，都可以通过软件自动进行，效率远高于发现bad trap后再到波形图里人肉寻找错误点。

获取执行流的一种方法是使用qemu + gdb的组合。gdb使用display /i $pc则可以执行一条指令就打印一次指令对应的汇编代码。qemu需要特定的版本，可以在工程目录的ram\_init\_gen下面执行make qemu，如果没有安装需要版本的qemu，则会报错，然后根据提示信息去安装对应的qemu即可。

|  |
| --- |
| Q：qemu除了要安装指令集对应的版本，还有其他特别的要求吗？ |
| A：常规的qemu模拟的是对应指令集的裸机，普通的可执行文件不能直接运行，至少要准备一个boot loader进行装载。有一类user mode的qemu，可以直接执行二进制文件而不需要磁盘镜像，调试使用的即这种qemu。这种qemu处于保护模式，必须使用虚拟地址进行编译链接。同时要规避默认的初始化栈空间行为。这里可以在入口函数后再写一个针对qemu的入口函数，针对make qemu写特定的链接脚本。 |

如果start.S设计得当的话，针对CPU调试用的程序和用来参照的程序对应指令地址的地位应该是一样的。启动qemu时需要以调试模式启动，等待gdb启动并登录约定好的端口。之后，在gdb中使用单步执行功能观察正确的指令执行流，并与波形图的进行比较。

但是目前没有找到方法以从qemu中直接获取完整执行流并输出到文件。比对的工作依然是人肉进行的，虽然脱离了不稳定的断言，但是效率提升有限。更好的选择则是spim。spim中step与gdb的单步执行类似，可以自动执行一定量的指令，而且能输出指令PC和对应汇编代码。可以将这些输出复制到一个文件中，作为正确的执行流。spim的缺点是只能执行特定格式的汇编代码。这导致反汇编的代码无法直接在spim上执行，可以先用gcc生成完整的汇编代码然后使用脚本剔除spim不能识别的编译指令解决这个问题。由于MEM段的PC一定是正确的，所以只需要将MEM段的PC用Verilog的系统任务$display逐条打印下来即可。

spim还有个小问题：由于它直接执行汇编代码，PC无法通过编译进行控制从而像qemu那样容易对应，而且spim对跳转类指令的延迟槽处理与正常的CPU不一样，更不用说本流水线设计了。所以逐指令比对虽然可以用代码实现，但是工作量不小。一个简化的方法是比较执行流的基本块。基本块的定义参见<https://en.wikipedia.org/wiki/Basic_block>一般意义上的基本块针对静态代码，而这里则是使用基本块的划分原则，划分出“动态”基本块。基本块的大小只与指令执行流有关，而与地址空间无关，设定一定的阈值，则可以忽视延迟槽的影响。如果使用更加有效的划分规则“PC如果不是顺序递增的，一定是遇到了分支、跳转，认为这是一个基本块的结束；如果遇到一个nop且nop指令的前后两条指令的pc不是相差8，那么认为这是一个基本块的结束”则可以更加简单地划分基本块并消除延迟槽的影响。

如果一个指令执行流是相同的，那么它们的动态基本块长度一定对应相同；而如果一个指令执行流是不同的，那么它们的动态基本块长度可能不同。只要检测出长度第一次不同的基本块，就能较准确且快速地定位到出错的地方。

如果想要简单地执行测试工具，则执行项目根目录下的test.py脚本，第一个参数是要测试的C源代码文件名，在testcase中挑一个，第二个参数可选，是对应的高级调试工具，有qemu和spim两个选项。spim选项最终会生成可供spim执行的汇编代码，需要自行获得两份执行流文本并用tools/cmp.py进行比较。qemu选项会直接启动完qemu进入gdb界面。

# 附录A. CACHE设计说明

## 需求分析

### 为什么需要cache？

在设计流水线cpu的时候，存储器必不可少的一部分，而在现代cpu中，cache是cpu内部关于访存的必不可少的部件，原因有2：

* 冯·诺依曼体系结构的瓶颈之一就是访存：cpu速度与存储器速度不相称，导致cpu经常等待存储器，这浪费了大量的计算性能。
* 一些统计结果显示，常用的软件中有25%的load类指令，有12%的store类指令，如果不使用instruction cache + data cache的话，流水线执行过程中会遭遇大量的访存冲突，速度也会受到较大的影响。

### 需要什么样的cache？

分析我们设计的流水线可以得出cpu向存储器暴露以下接口：

输出信号（从cpu输出到存储器）

* 指令地址 ：cpu在每个周期都会进行取指操作；
* 数据地址 ：cpu在进行内存读写时会指定读写地址；
* 数据读信号 ：如果是load类指令，cpu会传出数据读取的请求；
* 数据写信号 ：如果是store类指令，cpu会传出写入数据的请求；
* 字节写使能信号 ：如果是store类指令，cpu会指定要写入的字节；
* 写入数据 ：如果是store类指令，cpu会提供要写入的数据；

输入信号（从存储器到cpu）

* 指令 ：即取值得到的结果；
* 数据 ：如果是load类指令，存储器需要返回指定地址处的数据；
* 访存阻塞 ：如果数据在请求发出的周期无法准备好，则需要阻塞流水线。

cache需要实现这些目标：

* 当指定地址的数据存在于cache中时，能向cpu传送正确的数据（取值和load）或正确的修改数据（store）。
* 当指定地址的数据不存在于cache中时，在一定的时钟周期内让cpu阻塞，并从主存载入相应的数据块。

### 如何满足需求？

从上述目标中可知（也是常识），cache是在cpu和主存之间的代理人：

因此cache需要提供面向cpu的接口和面向主存的接口。其中，cache面向CPU的接口实际上等同于上文提到的CPU面向存储器的接口，因为cache已经充当了“代理”存储器的作用，此处不再赘述。而cache面向主存的接口如下：

输出（从cache到主存）

* 地址 ：读写的数据的地址；
* 写入数据 ：当cache进行写回时，需要提供写入主存的数据；
* 使能 ：当访问主存时，使能信号应该有效；
* 写使能 ：当cache进行写回时，写使能信号应该有效；

输入（从主存到cache）

* 就绪信号 ：当主存收到读写请求后，如果不能在该周期内准备好数据，则就绪信号应该为无效；当主存准备好数据或写入数据完成后，就绪信号应该有效；
* 读出数据 ：当cache进行数据块载入时，主存应该返回相应的数据。

cache面向主存的六个接口是根据cache的需求制定的，而不是面向特定的ram标准制定的。因此在将cache与某种ram连接时，还应该将ram提供的接口进行封装，然后再与cache相连接。

## 设计概述

### cache顶层设计

根据上述的需求分析，我设计cache的顶层结构如下：

cache\_manage\_unit是cache的顶层模块，提供了面向cpu和面向主存的接口，记录了cache的状态（在我的设计中，为了处理cache miss将cache划分成了不同的状态，会在后文中详细说明），初步处理cpu的请求和两部分cache的反馈信号，生成了一些与cache状态无关的控住信号、地址和数据。cache\_manage\_unit包含了3个子模块：cache\_control、instruction cache 和data cache。

cache\_control是纯组合逻辑，cache\_manage\_unit将初步处理得到的信号和cache的状态传给cache\_control，cache\_control根据cache的状态返回最终的控制信号，以及cache在下周期的新状态。cache\_control的实现细节和cache的miss处理高度相关，因此后文会先介绍cache miss的处理流程，再在代码说明部分中详细解释cache\_control模块完成的工作。

instruction cache和data cache是将二路组相联的cache实例化得到的，关于二路组相联cache的实现会在后面详细说明。

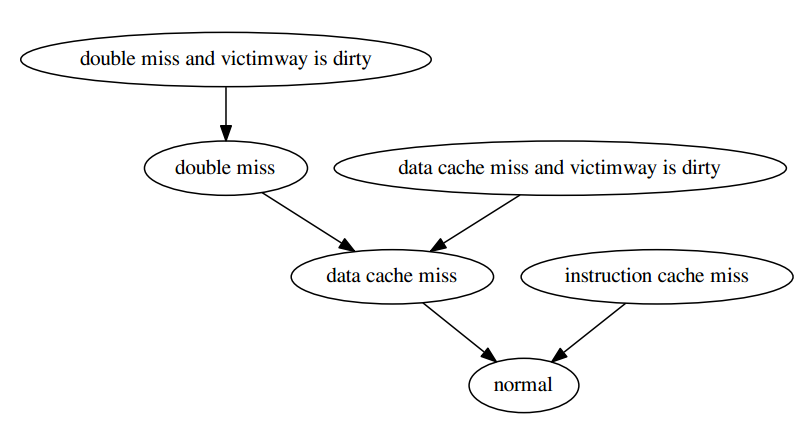
### miss的处理

cache\_manage\_unit将2块cache的输出信号和cache的当前状态送入cache\_control模块，cache\_control模块是纯组合逻辑，会根据输入得到相应的控制信号，控制cache的状态转移和miss处理。控制cache在不同的状态下完成不同的工作的控制信号是在cache\_control模块中完成的。

根据instruction cache和data cache的miss、dirty的组合情况，我将cache的状态划分为以下6种：

1. normal ：表示没有异常发生，cache正常运转；
2. data cache miss ：表示正在处理data cache miss的情况，即正在从主存加载数据块到data cache。如果normal状态下检测到单纯的（单纯的意思是没有instruction cache miss，且不需要写回脏块）data cache miss便会进入该状态。
3. instruction cache miss：表示正在处理instruction cache miss，即正在从主存加载数据块到instruction cache。如果normal状态下检测到单纯的instruction cache miss，便会计入该状态。
4. double miss：在normal状态下，在同一周期发现了data cache miss且instruction cache miss便会进入该状态。在该状态下，先处理instruction cache miss，处理完之后便会进入data cache miss状态。
5. data cache miss and victimway is dirty：在normal状态下发现了data cache miss，而且需要被替换的块是脏块，便会进入此状态。在该状态下，先写回脏块，处理完之后便会进入data cache miss状态。
6. double miss and victimway is dirty：这是最坏的情况，即在normal状态下发现instruction cache和data cache都发生了miss，而且data cache需要替换的是脏块。在该状态下，先写回脏块，然后便会进入double miss状态。

根据上述描述，这是cache miss处理过程中的状态图（这是非标准的状态图，仅仅用于示意异常处理的过程）：



另外，在后文中为方便期间，将nromal称为“正常状态”，其它5个状态称为“异常状态”，请不要与cpu的“异常”相混淆。

### 二路组相联的cache的设计

二路组相联cache是我们的cache的主体，它的设计需要兼顾两方面：一方面是cache manage unit的需求，另一方面是下层实现的兼容。cache实际上是一个耦合度很高的器件，在我的设计过程中，既不是单纯的自顶向下的，又不是单纯的自底向上的，而是兼而有之的，因此本层cache既需要向上提供足够的信息，又需要考虑下层实现的难易程度。在综合各方面考虑之后，二路组相联cache的层次结构如下所示：

一个二路组相联cache由2个直接映射的cache组成，使用的替换算法是随机替换算法。

假设已经有直接映射的cache，如何用2个直接映射的cache拼出一个**二路组相联**的cache？想象一下，如果自己是手拿2个直接映射cache的管理器，这个时候送过来一个地址并且是一个读请求，那么该做什么？首先，查看2块cache分别有没有命中，如果有一个命中，则输出相应的值（不会2块同时命中）；如果2块都miss，则需要从内存加载相应的数据块。加载数据块需要将数据**写入**cache中。这会引出2个问题：

1. 如果cpu送来的是一个写请求，那么也会写入数据块，两种写入有什么区别呢？
2. 现在有2块cache，写入到哪一块cache中呢？

接下来围绕这2个问题进行讨论：

#### 写入操作的区分----cmp信号

如果cpu发起了一个写请求，该如何处理？仍然假设我们是一个管理器，拿着2个cache。首先，我们应该查看两块cache是否有命中的，如果有则将送入的数据写入到命中的cache。（暂时不用考虑miss时该怎么处理，在此处这不是重点）注意，这里的写入操作必须建立在cache命中的基础上。

如果我们需要从内存加载一个快，该如何处理？将地址送到主存接口，等待主存准备好相应的数据，然后将数据块写入某一个cache（如何选择会在下面介绍）。在这里暂时不考虑脏块怎么处理。注意，这里的写操作无视cache的命中情况，强制写入。

这是直接映射的cache本身的问题：cpu的写入操作依赖于cache hit，而加载数据块时的写入操作则是强制的。所以直接映射的cache必须向外暴露一个接口：用于告知它“是否需要比较”，如果需要比较，则在tag hit的情况下才写入，否则不写入。为此，我们引入了cmp信号。

#### 替换哪一行？----victimway

在cache miss时，不得不从主存加载数据块，加载到2块cache中的哪一块呢？为此，在这里我采用了这样的策略：

1. 如果2块cache的valid都是0，也就是说里面都是无效数据，那么选择第0块吗，done；
2. 如果2块cache中有一块的valid是0，那么选择valid为0的这一块，done；
3. 如果2块cache的dirty都是0，那么随机选一块，done；
4. 如果2块cache中一个dirty为0，那么选择dirty为0的这一块，done；
5. 如果2块cache的dirty都是1，那么随机选一块，done；

采用这样的策略，可以让每次cache miss的开销尽可能小。另外，这里的“随机”并不是真随机，而是在cpu中使用一个1位寄存器，不断翻转来模拟随机。

### 直接映射的cache的设计

直接映射cache是二路组相联cache的主要部件。我的直接映射的cache参考了wisconsin的[cs552](http://pages.cs.wisc.edu/~david/courses/cs552/S12/includes/cache-mod.html)这门课的实验。在这个实验课程中，有一种“堆砌原料的思想”：

什么是原料：在我们的头脑中，cache总是以“行x列”的二维方式组织起来的，ICS课程可能会给我们留下这样印象：cache是以“行”进行组织的，每行有valid bit、dirty bit和tag bits，然后将这些“行”堆叠起来就得到了我们要的cache，在这种背景下实现的“行”便是原料。而这个实验中的原料是以“列”的方式进行组织的：先实现一个cache memory module，它是按照列来组织的，将这个模块作为原料，分别实例化为valid bit列、dirty bit列、tag列和数据块便是对原料的堆砌。同时对这些部分进行堆砌，添加适当的控制逻辑之后，便得到了一个直接映射的cache。

这样做会不会有什么问题？看上去valid、dirty、tag和数据都分离了，怎么将它们对应起来呢？只需要明白一点：cache的深度一旦确定下来，那么所有的模块的深度都确定下来了。如果我需要实现一个深度为512的cache，那么我一定需要512个valid位，512个dirty位，512个tag，512块数据，需要9位的地址宽度，地址将会送给所有的模块：将9’b000000000送给valid列，输出口的信号便是第一行的valid值，同样的，dirty列送出的也是第一行的dirty值。所以原本在“同一行”的数据现在虽然分离到了不同的模块，在不同的列中，但是我用一个确定的地址去访问时，总是能得到同一行的数据。

## 详细实现

不同于设计概述部分，实现部分的解释与众多信号是相关的，因此必须了解底层的实现才能理解上层的控制逻辑，所以此部分采用自底向上的顺序。

### 模块介绍

* cache\_mem ：上文中所述的“原料”模块，是在实例化过程中可以通过重写参数来修改规格的寄存器堆。对应层次图中的 cache memory。
* cache\_mem\_word ：每个数据块中包含了8个4 bytes的字，而cache\_mem\_word module就是一列4 bytes的字，每个byte又是通过实例化一个cache\_mem模块得到的。对应层次图中的data-word。
* cache\_oneline ：直接映射的cache，在cache\_oneline module中实例化了3次cache\_mem，分别用于valid-column，dirty-column和tag-module，另外实例化了8个cache\_mem\_word作为cache的数据块。此外，cache\_oneline module中有很多控制逻辑，用于完成命中判定、数据读写等。对应层次图中的direct-mapped-cache。
* cache\_2ways ：按照1.1.2中提到的方案，将2个direct-mapped-cache拼装成一个2-way-set-associative-cache，主要的任务是将2块cache的信号进行封装，以及选择被替换的块。对应层次图中的2-way-set-associative-cache。
* victimway\_sel :实现cache替换算法的模块。
* cache\_manage\_unit ：将2个2-way-set-associative-cache实例化为instruction cache和data cache；在内部通过状态机的方式实现各种情况下cache miss的处理。给出面向cpu和面向主存的接口。
* cache\_control ：根据cache\_manage\_unit当前的状态，得到相应的控制信号的模块。

下面是各个模块的具体实现的代码和解释：

### cache\_mem

#### 参数

ADDR\_WIDTH ：地址宽度；（后略）

MEM\_DEPTH ：cache的深度，由地址宽度决定；（后略）

DATA\_WIDTH ：数据宽度，不同的单元所需要的宽度不同。（后略）

#### 变量

##### 输入：

* clk ：时钟。（cache模块中的clk都是同一时钟，后略）
* rst ：复位信号。（cache模块中的rst都是同一时钟，后略）
* write ：写使能信号。
* data\_in：输入数据，宽度由DATA\_WIDTH参数决定。
* addr ：输入地址，宽度由ADDR\_WIDTH参数决定。

##### 输出：

* data\_out ：输出数据，宽度由DATA\_WIDTH参数决定。

##### 内部寄存器

* mem ：cache的存储空间，深度为MEM\_DEPTH，由ADDR\_WIDTH决定，宽度为DATA\_WIDTH。
* i ：数字，宽度为ADDR\_WIDTH，用于仿真时的初始化，在综合时无意义。

##### 内部wire

无

#### 代码

always @(posedge clk) begin

if(rst) begin

for(i=0;i<MEM\_DEPTH;i=i+1) begin

mem[i] <= 0;//在rst为0时，初始化为0；只在仿真时有效，综合时 //应该将它注释掉

end

end

if(!rst && write) begin

mem[addr] <= data\_in;//将送入模块的数据写入到寄存器中

end

end

### cache\_mem\_word

#### 参数

ADDR\_WIDTH ;

MEM\_DEPTH ;

DATA\_WIDTH

#### 变量

##### 输入：

* write ：写使能信号。
* data\_in ：输入数据，宽度由DATA \_WIDTH决定 。
* addr ：地址，宽度由ADDR\_WIDTH决定。
* byte\_w\_en ：字节写使能信号，用于每一个byte的写使能。

##### 输出：

* data\_out ：输出的数据，宽度由DATA \_WIDTH决定 。

##### 内部寄存器

无

##### 内部wire

byte\_out[3:0] ：每一个byte输出的值。

#### 代码

cache\_mem #(ADDR\_WIDTH,MEM\_DEPTH,8) byte0(clk,rst,write&&byte\_w\_en[0],data\_in[7:0],addr,byte\_out[0]);

cache\_mem #(ADDR\_WIDTH,MEM\_DEPTH,8) byte1(clk,rst,write&&byte\_w\_en[1],data\_in[15:8],addr,byte\_out[1]);

cache\_mem #(ADDR\_WIDTH,MEM\_DEPTH,8) byte2(clk,rst,write&&byte\_w\_en[2],data\_in[23:16],addr,byte\_out[2]);

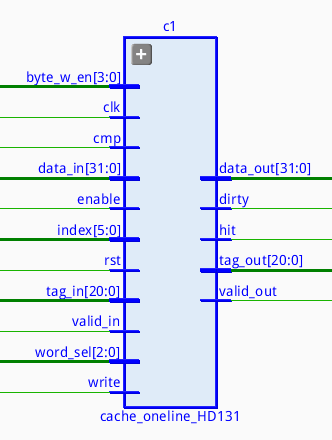
cache\_mem #(ADDR\_WIDTH,MEM\_DEPTH,8) byte3(clk,rst,write&&byte\_w\_en[3],data\_in[31:24],addr,byte\_out[3]);

/\*用参数#(ADDR\_WIDTH,MEM\_DEPTH,8)实例化4个cache\_mem模块，需要注意的是每一个byte的写使能信号都是*相应的字节写使能信号*与上*总的写使能信号*\*/

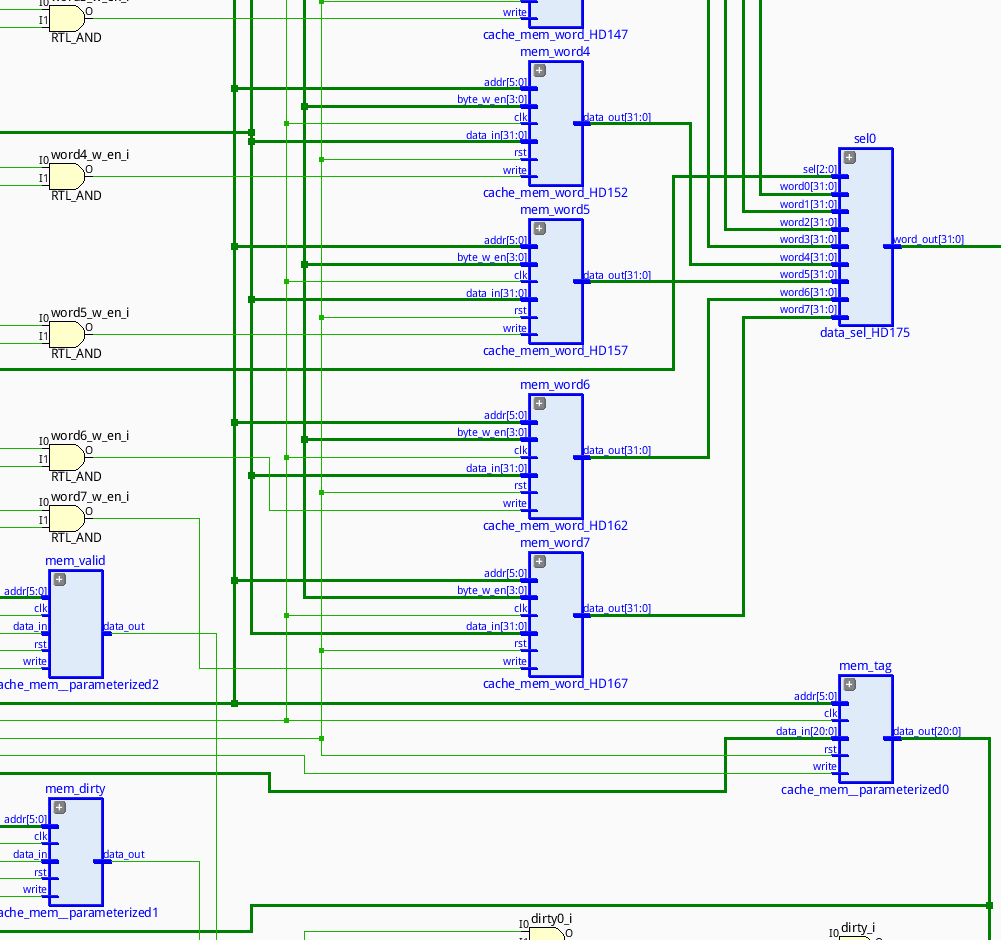
assign data\_out = {byte\_out[3],byte\_out[2],byte\_out[1],byte\_out[0]};

### cache\_oneline

本模块的外部接口：



本模块内部包含的模块：



#### 参数

OFFSET\_WIDTH ：内存地址中用于块内寻址的字段的宽度；

INDEX\_WIDTH ：内存地址中用于映射到cache的地址的字段的宽度；

CACHE\_DEPTH ：1<<INDEX\_WIDTH，表示cache的深度；

TAG\_WIDTH ：内存地址中用于比较，以确认是否命中的字段的宽度。

#### 变量

##### 输入：

* input enable：使能信号。一方面，当cpu不访问data cache的时候，通过关闭相应的使能信号，避免意外的cache miss发生；另一方面，为以后拼装成2-way-set-associative cache做铺垫，因为在发生替换时，2路cache中只会有1路被写入。
* input [INDEX\_WIDTH-1:0] index：即内存地址中的index字段，用于cache内寻址。
* input [OFFSET\_WIDTH-1:0] word\_sel：即内存地址中用于cache块内寻址的字段，用于确认使用cache块中的哪一个字。
* input cmp：在1.1.2.1中解释过，用于区分正常读写和块替换。
* input write：写使能信号
* input [TAG\_WIDTH-1:0] tag\_in ：内存地址中tag字段。
* input [31:0] data\_in ：输入数据，在写使能有效的时候有意义。
* input valid\_in ：当进行数据块载入时，应该将valid位置为1；另外，cache初始化时，应该将valid置为0。
* input [3:0] byte\_w\_en ：字节写使能信号。

##### 输出：

* output hit ：表示输入的tag和内部的tag相同，注意直接映射的cache中的hit信号不是真正意义上的“hit”，因为输出的hit信号没有包含valid位的信息，这与二路组相联的cache输出的hit信号不同。
* output dirty ：表示该数据块被修改过。
* output [TAG\_WIDTH-1:0] tag\_out ：输出的tag信息。
* output [31:0] data\_out ：输出数据，当write无效时有意义。
* output valid\_out ：输出的“有效”信号，与hit一起表示cache是否命中。

##### 内部寄存器

无

##### 内部wire

* go ：表示enable信号有效且rst信号无效，即cache处于应该工作的状态下。
* match ：match为1表示输入的tag和内部的tag相同。
* wordi\_w\_en ：一共8个信号，用于表示单个数据块内的八个字的写使能，由word\_sel决定。
* dirty\_override：dirty column的写使能信号。
* tag\_override ：tag的写使能信号。
* valid\_override：valid column的写使能信号。
* dirty\_in ：重写到dirty colune的值。
* dirty\_bit ：dirty column输出的值。
* valid\_bit ：valid column输出的值。

#### 代码

assign dirty\_override = go & write & (match|~cmp);

//当cmp无效时，即载入数据块时，需要写入dirty列

assign tag\_override = go & write & ~cmp;

//只有载入数据块时，tag才会被重写。

assign valid\_overide = go & write & ~cmp;

//只有载入数据块时，valid才会被重写。

assign dirty\_in = cmp;

//当写使能有效，且cmp有效时，表示cpu发出了写请求，且命中了，写入后该数据块变为脏块。

cache\_mem\_word #(INDEX\_WIDTH) mem\_word0(clk, rst, word0\_w\_en, data\_in, index, word0, byte\_w\_en);

……省略部分代码……

cache\_mem\_word #(INDEX\_WIDTH) mem\_word7(clk, rst, word7\_w\_en, data\_in, index, word7, byte\_w\_en);

/\*用INDEX\_WIDTH作为参数实例化了8个mem\_word作为一个内存数据块，INDEX\_WIDTH参数代表的是cache的地址宽度，决定了cahe的深度\*/

cache\_mem #(INDEX\_WIDTH,CACHE\_DEPTH,TAG\_WIDTH) mem\_tag(clk, rst, tag\_override, tag\_in, index, tag\_out);

/\*用#(INDEX\_WIDTH,CACHE\_DEPTH,TAG\_WIDTH)实例化了一个cache\_mem作为tag，写使能信号是tag\_override，输入的数据是tag\_in，输入的地址是index（cache内部总是用index作为地址，后略），输出数据是tag\_out，用于和tag\_in比较得到match \*/

cache\_mem #(INDEX\_WIDTH,CACHE\_DEPTH,1) mem\_dirty(clk, rst, dirty\_override, dirty\_in, index, dirty\_bit);

/\*实例化一个cache\_mem作为dirty列，写使能信号是dirty\_override，输入的数据是dirty\_in，输出数据是dirty\_bit\*/

cache\_mem #(INDEX\_WIDTH,CACHE\_DEPTH,1) mem\_valid(clk, rst, valid\_overide, valid\_in, index, valid\_bit);

/\*实例化一个cache\_mem作为valid列，写使能信号是valid\_override，输入的数据是valid\_in，输出数据是valid\_bit\*/

assign hit = go & match;

//cache处于正常状态，且cache内部的tag与输入的tag一致

assign dirty = go & dirty\_bit & (~write | ( cmp & ~match ));

/\*此处的条件是有冗余的，原课程网站上的实例代码是这样的，但是我的cache设计中只有cache替换时才会用到dirty信息，因此只需要go信号和dirty\_bit信号就可以了\*/

assign valid\_out = go & valid\_bit & (~write | cmp);

/\*valid\_out的条件也是有冗余的，因为valid信号只在cmp有效时才有意义\*/

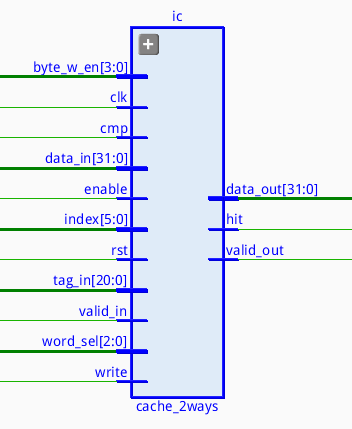
data\_sel sel0(word\_sel,word0,word1,word2,word3,word4,word5,

word6,word7,data\_out);

//根据word\_sel字段选择输出的字。

### cache\_2ways

本模块的外部接口：



#### 参数

与cache\_oneline的参数相同

#### 变量

##### 输入：

与cache\_oneline的输入相同

##### 输出：

与cache\_oneline的输出相同，但是下列信号的意义不同：

* output hit ：本模块中的hit是包含了valid信息的，与cache\_oneline的hit输出的意义不同。

##### 内部寄存器

reg victimway\_ff ：在1.1.2.2中提到了，在某些条件下需要用随机替换的方案替换cache，通过一位寄存器的方式实现伪随机。

##### 内部wire

在本模块内部定义的wire型变量大多是2个单路cache不同的输入或输出。

* victimway ：由victimway\_sel模块计算出来的被替换的cache。
* valid0,valid1 ：分别表示2路cache中地址为index的行是否有效。
* hit0,hit1 ：分别表示2路cache中地址为index的行是否命中。
* dirty0,dirty1 ：分别表示2路cache中地址为index的行是否有脏块。
* data0,data1 ：2路cache输出的32位的数据，当write无效时有意义。
* tag0,tag1 ：2路cache中地址为index的行的tag。
* write0,write1 ：输入到2路cache的写使能信号，总是仅有一个有效。
* enable0,enable1 ：输入到2路cache的使能信号，总是仅有一个有效。

#### 代码

victimway\_sel vs0(rst, enable, cmp, valid0,valid1,dirty0,dirty1,victimway\_ff,victimway);

/\*实例化victimway\_sel模块，将各种需要的信息送入，得到victimway作为被替换的cache\*/

always @ (posedge clk) begin

victimway\_ff <= victimway;

//伪随机

end

/\*以下信号直接送入cache的2路：

index, word\_sel, cmp, tag\_in, data\_in, valid\_in, byte\_w\_en, clk, rst

\*/

cache\_oneline #(OFFSET\_WIDTH,BLOCK\_SIZE,INDEX\_WIDTH,CACHE\_DEPTH,TAG\_WIDTH) c0(enable0, index,word\_sel,

cmp, write0, tag\_in, data\_in, valid\_in, byte\_w\_en, clk, rst, hit0, dirty0, tag0, data0, valid0);

//实例化cache\_oneline得到第1路

cache\_oneline #(OFFSET\_WIDTH,BLOCK\_SIZE,INDEX\_WIDTH,CACHE\_DEPTH,TAG\_WIDTH) c1(enable1, index, word\_sel,

cmp, write1, tag\_in, data\_in, valid\_in, byte\_w\_en, clk, rst, hit1, dirty1, tag1, data1, valid1);

//实例化cache\_oneline得到第2路

assign enable0 = cmp ? enable : ~victimway;

assign enable1 = cmp ? enable : victimway;

/\*当cmp有效时，表明要进行cache替换：当victimway为1时，1号cache的行被替换；反之，0号cache的行被替换。当cmp为1时，2路cache的使能信号均有效。\*/

assign write0 = write &(cmp ? (valid0 & hit0) : ~victimway\_ff);

assign write1 = write &(cmp ? (valid1 & hit1) : victimway\_ff);

/\*首先，必须总的写使能信号有效，2路cache的写使能信号才可能有效。其次，如果cmp无效，则表示正在从主存载入数据块，由victimway决定写入哪一路cache；如果cmp有效，则仅当cache命中时才会写入，由于cache\_oneline的hit输出是不包含valid信息的，所以hit&valid才是我们需要的“hit”信号\*。/

assign hit = (valid1 & hit1) | (valid0 & hit0);

//二路中如果有一路命中便是命中

assign dirty = dirty0 & dirty1;

//二路均为脏，当发生miss的时候有意义，说明不得不进行写回

assign data\_out = cmp ? ((hit0 & valid0) ? data0 : data1) : (victimway\_ff ? data1 : data0);

/\*data\_out仅当write无效时才有意义。如果cmp无效，说明在进行写回操作，这时候根据victimway选择应该写回的数据块；如果cmp有效，说明在响应cpu的读请求，如果一路cache命中，则输出该路cache传出的数据，如果2路cache均miss，则data\_out是无意义的\*/

assign valid\_out = valid0 | valid1;

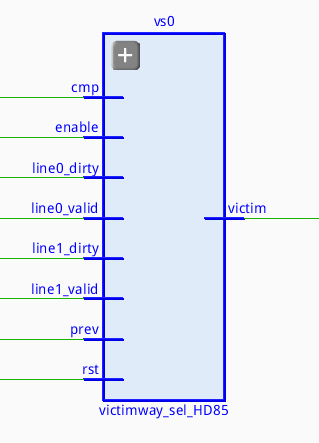
/\*在我的实现中,为了让cache\_manage\_unit既可以实例化cache\_2ways，也可以实例化cache\_oneline作为instruction cache和data cache，在cache\_manage\_unit中判定是否hit的时候，总是使用(hit&valid)以兼容cache\_oneline，而这里将valid\_out赋值为(valid0|valid1)则是为了对此进行再兼容\*/

assign tag\_out = cmp ? ((hit0&valid0) ? tag0 : tag1) : (victimway\_ff ? tag1 : tag0);

//此处输出的tag\_out意义不大，目前没有用处。

### victimway\_sel

本模块的外部接口：



#### 参数

无

#### 变量

##### 输入：

* input enable ：同二路cache的使能信号。
* input cmp ：同二路cache的cmp信号。
* input line0\_valid, line1\_valid：2路cache的valid信号。
* input line0\_dirty, line1\_dirty：2路cache的dirty信号。
* input prev ：对应于cache\_2ways module中的victimway\_ff，用于维护伪随机机制。

##### 输出：

* output victim ：本次应该被替换的路，仅当cache miss发生时才有意义。

##### 内部寄存器

无

##### 内部wire

reg v ：对应输出victim

#### 代码

assign go = ~rst & enable & cmp;

/\*当cmp信号无效时，说明正在进行写回或者加载操作，此时victimway不能发生变化，因为同一个数据块一定位于一路cache中\*/

always @(\*) begin

if(!go) begin

if(rst) begin

v = 0;

end

else begin

v = prev;

end

end

else begin

if(!line0\_valid && !line1\_valid)begin

//两路cache均无效，v=0表示替换第一路

v = 0;

end

else begin

if(line0\_valid && line1\_valid)begin

//两路cache均有效

if(!line0\_dirty && !line1\_dirty) begin

//没有哪一路有脏块

v = ~prev；

end

else begin

if(line0\_dirty && line1\_dirty)begin

//两路cache均含有脏块

v = ~prev;

end

else begin

if(line0\_dirty)begin // 第一路cache为脏

v = 1;

end

else begin // 第二路cache为脏

v = 0;

end

end

end

end

else begin

if(line0\_valid)begin //第二路cache无效

v = 1;

end

else begin //第一路cache无效

v = 0;

end

end

end

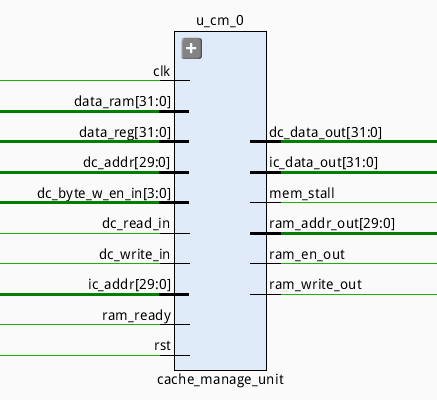
end

end

assign victim = v;

### cache\_manage\_unit

本模块的外部接口：



#### 参数

同cache\_oneline和cache\_2ways，且在本模块中会用本模块的参数重写cache\_2ways的相关参数，所以要调整cache的参数只需要在本模块中修改即可。

#### 变量

##### 输入：

* input [29:0] ic\_addr,dc\_addr ：分别送入instruction cache和data cache的主存地址（不是cache内部的地址，cahce内部的地址是index字段）。另外为了简短起见，**后文的instruction cache和data cache分别用ic和dc代替**。
* input [31:0] data\_ram,data\_reg ：从主存送来的数据和从cpu送来的数据，前者只在从主存加载数据块时有意义，而后者只在cpu执行store指令的时候有意义。
* input dc\_write\_in,dc\_read\_in ：分别表示dc是否被写入，或者被读取，前者仅在cpu执行store指令时有效，而后者仅在cpu执行load指令的时候有效。可以发现，在很多情况下这两个信号都是无效的，所以dc的enable信号在很多情况下都是无效的。
* input ram\_ready ：由主存送过来的信号，表示主存已经准备好了数据或者已经写入了数据。
* input [3:0] dc\_byte\_w\_en\_in ：dc的字节写使能信号，从cpu送来的，主要用于sw、sb、swl、swr等指令的实现。

##### 输出：

* output [31:0] dc\_data\_out,ic\_data\_out ：从dc和ic读出的数据，既可能送到cpu（在正常状态下），又可能送到主存（在异常状态下）。
* output mem\_stall ：表示发生了cache miss，必须让cpu流水线停止。
* output ram\_en\_out ：在异常状态下，进行写回或者载入操作时，需要与主存进行交互，此时主存的使能信号应该有效。
* output ram\_write\_out ：在进行写回操作时，主存的写使能信号应该有效。
* output [29:0] ram\_addr\_out ：送入到主存的地址，低二位应该补0。因为cache与主存进行数据交换时，总是4字节对齐的。

##### 内部寄存器

* reg [2:0] status ：当前的状态，在[1.3 cache miss处理](#_cache_miss的处理)中介绍了cache可能处在的6中状态，status就是用来存储这些状态的。
* reg [2:0] counter ：3位的计数器，因为cache的一个数据块中包含了8个字，在异常处理的写回或者载入阶段，每次都需要写回或者载入8个字。conuter寄存器一方面可以充当cache内部的计数器，另一方面还可以作为送给主存的30位地址的低3位。counter在异常处理周期总是从0变为7。
* reg write\_after\_load ：当cpu执行一条store指令时，如果发生了cache miss，将会迎来8个周期的载入。在原本的设计中存在bug：载入完成后直接将mem\_stall拉低，紧接着cpu便进入了下一周期，

##### 内部wire

* wire ic\_enable,dc\_enable ：ic和dc的使能信号，ic使能信号长期有效，而dc的使能信号仅在cpu进行访存操作时才有效。
* wire ic\_cmp,dc\_cmp ：ic和dc 的cmp信号，在正常周期cmp有效，在处理miss时cmp无效。
* wire ic\_write,dc\_write ：ic和dc的写使能信号，在正常周期里，ic的写使能信号应该长期无效，仅在ic miss的处理周期有效。
* wire ic\_data\_sel,dc\_date\_sel：送入ic和dc的数据的选择端，逻辑较为复杂，在代码解释部分说明。
* wire ic\_valid\_2ic,dc\_valid\_2dc：送入ic和dc的valid信号，用于cache的初始化（写0）和数据块的载入（写1）。
* wire [OFFSET\_WIDTH-1:0] ic\_word\_sel, dc\_word\_sel :送入ic和dc的cache地址的低3位。在正常周期是主存地址的低3位，在异常周期则是counter的值。
* wire [3:0] status\_next ：下一周期status寄存器的期望值，由cache\_control模块计算。
* wire [3:0] counter\_next ：下一周期counter寄存器的期望值，由cache\_control模块计算。
* wire [3:0] ic\_byte\_w\_en, dc\_byte\_w\_en ：ic和dc的字节写使能。一般的，ic的字节写使能是全部有效，因为数据块总是整块载入的，且ic不会被cpu写入。
* wire [31:0] ic\_data2ic, dc\_data2dc ：送入ic和dc的数据。
* wire [29:0] ram\_addr\_ic ：进行ic miss处理时，从主存载入数据块时，送入主存的30位地址。
* wire [29:0] ram\_addr\_dc ：进行dc miss处理时，从主存载入数据块时，送入主存的30位地址。
* wire [29:0] ram\_addr\_dc\_wb ：进行dc miss处理时，写回数据块时，送入主存的30位地址。
* wire [OFFSET\_WIDTH-1:0] ic\_offset, dc\_offset ：送入ic和dc的cache数据块内偏移量。
* wire [INDEX\_WIDTH-1:0] ic\_index, dc\_index ：送入ic和dc的cache地址。
* wire [TAG\_WIDTH-1:0] ic\_tag, dc\_tag ：送入cache的tag字段，分别送入ic和dc，用于比较以确认cache是否miss。
* wire [TAG\_WIDTH-1:0] ic\_tag\_out, dc\_tag\_out ：从cache中返回的tag字段，仅dc\_tag\_out有用，用于数据块的写回。
* wire loading\_ic ：表示cache正在载入数据块到ic中，这个信号是为了保证dc和ic的数据一致性而添加的。
* wire ic\_hit, ic\_valid, ic\_dirty ：ic送出的hit、valid和dirty信号
* wire dc\_hit, dc\_valid, dc\_dirty ：dc送出的hit、valid和dirty信号

#### 代码

wire loading\_ic = status ==`STAT\_IC\_MISS || status == `STAT\_DOUBLE\_MISS;

/\*当状态为 ic miss或double miss时加载数据块到ic中，而此时dc中可能存在相应的数据块，可以直接从dc加载；如果dc中的相应数据块是脏块，那么不从dc中加载还会产生数据不一致的错误\*/

assign ic\_tag = ic\_addr[29:29-TAG\_WIDTH+1];

assign dc\_tag = (~loading\_ic) ? dc\_addr[29:29-TAG\_WIDTH+1] : ic\_tag;

/\*当ic miss发生时，需要查看dc中是否存在相应的数据块，如果存在则将相应的数据块载入到ic中。所以当loading\_ic有效时，应该将ic\_tag送入dc中进行比较，根据dc送出的hit信号判断dc中是否有相应的块。\*/

assign ic\_index = ic\_addr[29-TAG\_WIDTH:OFFSET\_WIDTH];

assign dc\_index = (~loading\_ic) ? dc\_addr[29-TAG\_WIDTH:OFFSET\_WIDTH] : ic\_index;

//类似的，为了检查dc中有无ic所需的块，需要将ic的index字段送入dc

assign ic\_offset = ic\_addr[OFFSET\_WIDTH-1:0];

assign dc\_offset = dc\_addr[OFFSET\_WIDTH-1:0];

assign ram\_addr\_ic = {ic\_tag,ic\_index,counter};

assign ram\_addr\_dc = {dc\_tag,dc\_index,counter};

/\*以上两个地址用于ic和dc miss时的处理，将相应的地址送入主存，其中tag和index来自主存地址，而counter则是cache内部的计数器。开始载入数据块时，counter为0，后面的周期里conuter递增到7，这样同一个数据块中的7个字将被按顺序加载到cache中\*/

assign ram\_addr\_dc\_wb = {dc\_tag\_out,dc\_index,counter};

/\* 当dc需要写回脏块时，该地址有意义，需要注意的是该地址的tag字段来自于dc内的tag字段，而非主存地址的tag字段\*/

assign ram\_addr\_out = ram\_addr\_sel[1] ? ram\_addr\_dc\_wb : (ram\_addr\_sel[0] ? ram\_addr\_dc : ram\_addr\_ic);

/\*选择合适的送入主存的地址，ram\_add\_sel是两位信号，11表示进行dc的写回，00表示进行dc的载入，01表示进行ic的载入。\*/

cache\_control cctrl (dc\_read\_in, dc\_write\_in, ic\_offset, dc\_offset, dc\_byte\_w\_en\_in,

ic\_hit, ic\_valid,/\*ic's output\*/

dc\_hit, dc\_dirty, dc\_valid,/\*dc's output\*/

status, counter,/\*status\*/

ic\_enable, ic\_word\_sel, ic\_cmp, ic\_write, ic\_data\_sel, ic\_byte\_w\_en, ic\_valid\_2ic,/\*to ic\*/

dc\_enable, dc\_word\_sel, dc\_cmp, dc\_write, dc\_data\_sel, dc\_byte\_w\_en, dc\_valid\_2dc,/\*to dc\*/

ram\_addr\_sel, ram\_en\_out, ram\_write\_out,

status\_next, counter\_next

);

//实例化[cache\_control](#_cache_control)模块，用于产生各种控制信号

assign ic\_data2ic = ic\_data\_sel ? data\_ram : dc\_data\_out; //0:load from dc

/\*选择送入ic的数据，ic仅在载入数据块时才会被写入，data\_ram是从主存送来的数据，而dc\_data\_out是从dc中送来数据，当ic所需要的数据存在dc中时，直接从dc载入\*/

assign dc\_data2dc = dc\_data\_sel ? data\_ram : data\_reg;

/\*选择送入dc的数据，送入dc的数据有2种来源：从主存加载数据块，或是cpu执行store指令\*/

cache\_2ways ic(ic\_enable, ic\_index, ic\_word\_sel, ic\_cmp, ic\_write, ic\_tag, ic\_data2ic,

ic\_valid\_2ic, ic\_byte\_w\_en, clk, rst, ic\_hit, ic\_dirty, ic\_tag\_out, ic\_data\_out, ic\_valid);

//实例化ic

cache\_2ways dc(dc\_enable, dc\_index, dc\_word\_sel, dc\_cmp, dc\_write, dc\_tag, dc\_data2dc,

dc\_valid\_2dc, dc\_byte\_w\_en, clk, rst, dc\_hit, dc\_dirty, dc\_tag\_out, dc\_data\_out, dc\_valid);

//实例化dc

always @(posedge clk) begin

if(rst) begin

//初始化3个状态寄存器的值

status <= `STAT\_NORMAL;

counter <= 3'd7;

write\_after\_load <= 0;

end

else begin

if(status == `STAT\_NORMAL) begin

/\*处于正常状态下，status和counter写入cache\_control计算出的下周 期的期望值，而write\_after\_load寄存器则被复位\*/

status <= status\_next;

counter <= counter\_next;

write\_after\_load <= 0;

end

else begin

//处于异常状态下

if(dc\_write\_in) begin

/\*如果cpu送入的写请求有效，那么需要将write\_after\_load置位，以配合后面的mem\_stall的产生逻辑，使cache 写发生miss的处理结束后多阻塞一个周期，让期望写入的值能够写入到dc中\*/

write\_after\_load <= 1;

end

if(ram\_ready || (loading\_ic && dc\_hit)) begin

/\* 当载入数据块时，如果是ic所需数据块在dc中能找到，则不需要访问ram，并且每个时钟周期都能载入一个字，不需要额外的等待；如果是从主存载入，则每次都需要等待ram\_ready信号有效时才能请求下一个字，进入下一个状态\*/

status <= status\_next;

counter <= counter\_next;

end

end

end

end

assign mem\_stall = (status != `STAT\_NORMAL) || (status\_next != `STAT\_NORMAL) || write\_after\_load ;

/\*当cache本周期处于异常处理状态或是下周期会进入异常处理状态时，必须让cpu阻塞。另外，如果是cpu执行store指令导致的cache miss，在cache载入了相应的数据块后，还应该让cpu多阻塞一个周期，使数据能够写入cache中，这就是write\_after\_load的意义\*/

### cache\_control

本模块的外部接口：



#### 参数

无

#### 变量

##### 输入：

* input dc\_read\_in ：需要读取dc，即cpu在执行load指令；
* input dc\_write\_in ：需要写入dc，即cpu在执行store指令；
* input ic\_hit\_in ：ic命中；
* input ic\_valid\_in ：ic中被映射到的行可用；
* input dc\_hit\_in ：dc命中；
* input dc\_dirty\_in ：dc中被映射到的行中包含脏块；
* input dc\_valid\_in ：dc中被映射到的行可用；
* input status\_in ：cache当前的状态；
* input counter\_in ：cache计数器当前的值；
* input ic\_word\_sel\_in ：表示要访问的字在ic的数据块内的偏移量，仅在正常周期有意义；
* input dc\_word\_sel\_in ：表示要访问的字在dc的数据块内的偏移量，仅在正常周期或处理ic miss的周期有意义；
* input dc\_byte\_w\_en\_in ：dc的字节写使能。

##### 输出：

* output ic\_enable\_out ：送入ic的使能信号；
* output ic\_cmp\_out ：送入ic的cmp信号；
* output ic\_write\_out ：送入ic的write信号；
* output ic\_data\_sel ：送入ic的数据的选择信号；
* output ic\_valid\_out ：送入ic的valid信号；
* output dc\_enable\_out ：送入dc 的使能信号；
* output dc\_cmp\_out ：送入dc的cmp信号；
* output dc\_write\_out ：送入dc的写使能信号；
* output dc\_data\_sel ：送入dc的数据的选择信号；
* output dc\_valid\_out ：送入dc的valid信号；
* output ram\_en\_out ：送入主存的使能信号；
* output ram\_write\_out ：送入主存的写使能信号；
* output ram\_addr\_sel ：送入主存的地址的选择信号；
* output status\_next ：cache下一个周期的状态；
* output counter\_next ：计数器下一个周期的值；
* output ic\_word\_sel\_out ：送入ic的块内偏移量，在正常模式下来自指令的主存地址，在异常模式下来自计数器。
* output dc\_word\_sel\_out ：送入dc的块内偏移量，在正常模式下来自数据的主存地址，在异常模式下来自计数器或者指令的主存地址。
* output ic\_byte\_w\_en ：送入ic的字节写使能信号；
* output dc\_byte\_w\_en ：送入dc的字节写使能信号。

##### 内部寄存器

在本模块中定义了很多reg类型的变量，但是本模块实际上是纯组合逻辑，这些reg类型的变量只是用作中间变量，他们与输出信号一一对应，最后都会连接到输出信号上。也可以直接将输出信号定义为output reg，就可以不用定义这些变量了。

##### 内部wire

无

#### 编写思路

本模块的代码主体是一个case语句块，case的条件是输入的cache状态，即status\_in。本模块编写复杂，但是方法很机械：在正常模式下需要考虑ic和dc 的hit、dirty和valid信号的情况，选下接下来进入什么状态；在异常模式下需要考虑怎么处理异常，什么条件下进入其他状态。因为本模块的编写方法十分机械，所以不使用“代码+注释”的方式解释，而是给出“问题描述+解决方案”。

需要考虑两方面的问题：第一、在正常状态下，在什么条件下进入其他状态，在什么条件下保持正常状态。第二、在异常状态下该做什么处理，什么条件下进入其他状态。

##### 正常状态

首先，考虑正常状态在什么条件下进入其他状态，列出表格：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 信号 | dc\_hit | dc\_dirty | ic\_hit | 下个状态 |
|  | 0 | 0 | 0 | double miss |
|  | 0 | 0 | 1 | dc miss |
|  | 0 | 1 | 0 | double miss + dc dirty |
|  | 0 | 1 | 1 | dc miss + dc dirty |
|  | 1 | 0 | 0 | ic miss |
|  | 1 | 0 | 1 | normal |
|  | 1 | 1 | 0 | ic miss |
|  | 1 | 1 | 1 | normal |

如果在正常状态下且下一周期仍然为正常状态，那么本周期cache的输出值是有效的（如果是读请求），写入值能正常写入（如果是写请求）。这时候需要考虑本周期输出的控制信号（如果下一周期不是正常周期，这些信号也会被输出，但是是无意义的）：

* ic\_enable ：每个周期都会取指令，所以ic\_enable有效；
* ic\_cmp\_out ：正常周期的cmp信号恒为有效；
* ic\_write\_out ：此信号只在载入数据块到ic时有效，此时无效；
* ic\_data\_sel\_out ：此信号只在载入数据块到ic时有意义，此时无关；
* ic\_valid\_out ：此信号只在载入数据块到ic时有意义，此时无关；
* ic\_word\_sel\_out ：处于正常状态，取值为ic\_word\_sel\_in ；
* ic\_byte\_w\_en\_out ：此信号只在载入数据块到ic时有效，此时无效；
* dc\_enable\_out ：取值为(dc\_read\_in | dc\_write\_in)，即只有在访存周期，dc的使能信号才会有效。
* dc\_cmp\_out ：正常周期的cmp信号恒为有效；
* dc\_write\_out ：如果dc\_write\_in有效，则该信号有效；
* dc\_data\_sel\_out ：取值为0，意义是选择来自寄存器堆的数据；
* dc\_valid\_out ：无关；
* dc\_word\_sel\_out ：处于正常状态，取值为dc\_word\_sel\_in；
* dc\_byte\_w\_en\_out ：处于正常状态，取值为dc\_byte\_w\_en\_in；
* ram\_addr\_sel\_out ：无意义；
* ram\_en\_out ：无效；
* ram\_write\_out ：无效；

这样，就能保证cache在未发生miss的情况下能正常运转，在发生miss的情况下进入相应的异常状态进行处理。

接下来讨论异常处理方法。虽然异常状态有5种，但是实际上异常处理过程中只会进行3种操作：从dc写回数据、载入数据块到dc、载入数据块到ic。在2种dc dirty的异常状态下，都进行脏块写回操作；而在double miss和ic miss的情况下，都进行ic数据块的载入，仅在dc miss这个状态下进行dc数据块的载入。因此下面的内容介绍这3种异常处理操作的控制信号。

##### 脏块写回

脏块写回时，总是读取dc，然后将数据写回主存，需要注意这几点：不影响ic的内容；总是写回一个数据块中所有的字，所以应该使用counter作为地址的低位。

* ic\_enable, ic\_cmp\_out, ic\_write\_out, ic\_data\_sel\_out, ic\_valid\_out, ic\_word\_sel\_out, ic\_word\_sel\_in, ic\_byte\_w\_en\_out ：脏块写回与ic无关，所以上述信号种ic\_enable无效，其余信号都是无关信号。
* dc\_enable\_out ：取值为1，因为本周期必定访问dc；
* dc\_cmp\_out ：取值为0，因为总是需要读取选定的某一行，不需要比较
* dc\_write\_out ：写回过程中不写dc，无效；
* dc\_data\_sel\_out ：无关；
* dc\_valid\_out ：无关；
* dc\_word\_sel\_out ：取值为counter，counter在异常处理过程中总是从0变为7，保证经历8个周期把整个数据块写回；
* dc\_byte\_w\_en\_out ：写回过程中不写dc，无效；
* ram\_addr\_sel\_out ：取值为3，意义是选择ram\_addr\_dc\_wb作为送入主存的地址；
* ram\_en\_out ：有效；
* ram\_write\_out ：有效；

##### 载入dc数据块

总是读取主存，写入dc，需要注意这几点：不影响ic的内容；总是从主存读取一个数据块中所有的字，所以应该使用counter作为地址的低位。

* ic\_enable, ic\_cmp\_out, ic\_write\_out, ic\_data\_sel\_out, ic\_valid\_out, ic\_word\_sel\_out, ic\_word\_sel\_in, ic\_byte\_w\_en\_out ：载入dc数据块与ic无关，所以上述信号种ic\_enable无效，其余信号都是无关信号。
* dc\_enable\_out ：有效，因为本周期必定访问dc；
* dc\_cmp\_out ：无效，因为总是写入选定的某一行，不需要比较
* dc\_write\_out ：有效，因为载入dc数据块总是写dc；
* dc\_data\_sel\_out ：取值为1，表示选择来自主存的数据；
* dc\_valid\_out ：有效，因为载入完成后该行cache变为有效；
* dc\_word\_sel\_out ：取值为counter，counter在异常处理过程中总是从0变为7，保证经历8个周期把整个数据块写入cache中；
* dc\_byte\_w\_en\_out ：取值为4’b1111，因为总是写cache；
* ram\_addr\_sel\_out ：取值为1，意义是选择ram\_addr\_dc作为送入主存的地址；
* ram\_en\_out ：有效，读主存；
* ram\_write\_out ：无效。

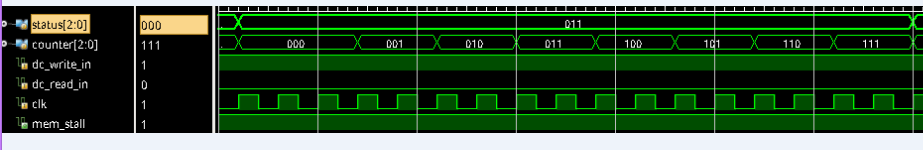
##### 载入ic数据块

载入ic数据块时，总是写入ic，但是数据的来源可能是主存也可能是dc，需要对此做出判断，以确保数据的一致性。

* ic\_enable\_out ：有效，因为要载入ic数据块时总是要写ic
* ic\_cmp\_out ：无效，因为总是写入选定的某一行，不需要比较
* ic\_write\_out　 ：有效；
* ic\_data\_sel\_out　 ：ic送入dc的地址命中时为１（选择来自dc的数据），否则为０；这个选择信号是用于保证ic和dc的一致性的；
* ic\_valid\_out ： 有效，因为载入数据块后，ic相应的行变为有效；
* ic\_word\_sel\_out ：与conuter相同；
* ic\_byte\_w\_en\_out ：4’b1111，因为总是会写入；
* dc\_enable\_out　 ：有效，因为可能需要从dc读取数据；
* dc\_cmp\_out ：有效，因为需要确定ic送来的地址在dc中是否命中；
* dc\_write\_out ：无效，因为一定不会写dc；
* dc\_data\_sel\_out ：无关；
* dc\_valid\_out ：无关；
* dc\_word\_sel\_out ：与conuter相同，当ic送来的地址在dc中命中时，总是将dc中的数据块的第i个字写入到ic中的数据块的第i个字；
* dc\_byte\_w\_en\_out ：无效，因为一定不会写ic；
* ram\_en\_out ：当dc未命中时为1，否则为0；
* ram\_addr\_sel\_out ：取值为0，ram\_addr\_ic作为输入到ram的地址；
* ram\_write\_out ：无效。

##### 3种处理的时序

1. 脏块写回



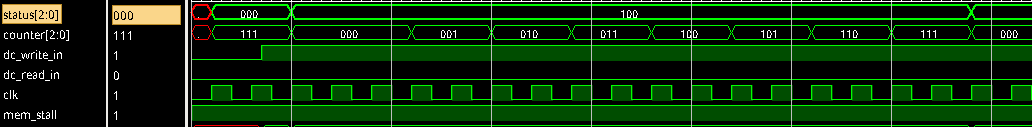
上图是dc miss且需要写回脏块时的波形。status 3’b011代表的是dc\_miss且替换脏块，在此状态下完成脏块的写回，经历8个周期，counter从0变为7，同一个数据块中8个字被依次写回主存。

1. 载入dc数据块



上图是A的波形图的后续部分，在写回脏块之后，cache进入dc\_miss的状态(3’b010)，从主存中载入8个数据块到dc中，counter再次经历从0到7的变化。需要注意的是mem\_stall信号（最下面的信号）在载入数据块完成之后还多保留了一个周期的高态，是因为dc\_write\_in有效，即该访存请求是一次写入请求，这里让流水线多阻塞一个周期，保证数据的写入。

1. 载入ic数据块



上图是double miss时的波形。状态3’b100表示的是double miss，在此状态下完成ic的载入。从主存中载入8个数据块到ic中，counter经历从0到7的变化。然后进入dc miss的状态，处理dc的数据块加载。

## 数据一致性

### 基本措施

ic与dc的最常见的一致性问题是在使用loader的时候发生的：当loader将程序加载到内存中时，由于dc采用的是写回策略，所以程序很可能还存在于cache中，并没有修改相应的主存，如果这时候执行载入的程序，那么很可能执行的是主存中的旧的指令。为了解决这种不一致性，我在cache\_manage\_unit模块中增加了一个信号表示正在加载数据块到ic中：

wire loading\_ic = status ==`STAT\_IC\_MISS || status == `STAT\_DOUBLE\_MISS;

当loading\_ic信号有效时，送入dc的tag和index是指令地址的相应字段，而不是数据地址的相应字段：

assign ic\_tag = ic\_addr[29:29-TAG\_WIDTH+1];

assign dc\_tag = (~loading\_ic) ? dc\_addr[29:29-TAG\_WIDTH+1] : ic\_tag;

assign ic\_index = ic\_addr[29-TAG\_WIDTH:OFFSET\_WIDTH];

assign dc\_index = (~loading\_ic) ? dc\_addr[29-TAG\_WIDTH:OFFSET\_WIDTH] : ic\_index;

如果发现dc hit，那么就从dc中载入块到ic中，避免了上述的不一致性。

### 特殊情况

还有一种比较特殊的情况是执行了某一段代码之后，它还存在于ic中，紧接着对应地址被写入了新的数据，这样ic中的数据仍然是旧值。目前我对这种情况采用的是鸵鸟策略，因为我认为这种情况发生的概率太低，目前不会发生。

如果这种情况发生了，应该这样处理：当dc miss时，如果cpu送入的写使能信号有效，当从主存中载入块到dc时，将数据地址的高位送给ic，如果hit则将index映射到的行的valid位写为0。这样处理之后，如果某个块先被写入，然后被执行，那么1中的策略会使他执行正确的值；如果它先被执行再被写入，那么ic中的块会被置为无效，下次执行时会重新载入新的指令。

### 极端情况

但是在2中仍然没有解决这种情况：某个块先被执行，再被写入，再被执行，此时ic和dc中相应的行都被变为valid，如果再对他进行写入，再执行，那么此时会执行到旧的指令。我认为目前的cache设计结构难以解决这个问题，不过我认为在我们的实验中这种情况不会发生。

## 测试方案

### 基本测试

基本测试是对二路组相联的cache的测试，主要的测试对象是脏块能否正常写回。测试文件是cache\_test.v。测试方案如下：

1. 实例化一个二路组相联的cache作为测试对象；
2. 寻找一个物理地址addrX，写入值valX；
3. 读取地址addrX的值，看是否为valX；
4. 寻找一个物理地址addrY，使它与addrX映射到同一行，写入值valY；
5. 读取地址addrY的值，看是否为valY；
6. 寻找一个物理地址addrZ，使它与addrX、addrY映射到同一行，写入值valZ（此时addrX或addrY的值应该被写回了）；
7. 读取地址addrZ的值，看是否为valZ；
8. 再反复读取addrX、addrY和addrZ处的值。观察波形图应该可以看见，cache被反复的替换；另外，此过程中valX、valY、valZ必须为之前写入的值，否则有错。

### 通用测试

由于cache的控制信号很多，要想手写大量的测试样例是不现实的。最终的检测途径是将cache加入流水线中，运行实际的程序，目前我们通过了规模较大的冒泡排序和快速排序，也就是说我们的cache是经受住了考验的。

# 附录 B. 控制单元设计说明

控制单元检测流水线的冒险，控制流水段寄存器的冲刷和阻塞以及异常处理

## 检测的冒险

### 检测load-use冒险

根据输入的id段得到的rs寄存器地址和最终的rt寄存器地址，以及ex段继承自上一流水段的rd寄存器地址和mem\_read访存信号，如果ex段是访存指令（mem\_read有效），且rd与rs和rt中的一个相等，则通过组合逻辑使得pc\_stall, ifid\_stall, idex\_flush有效，在下一个周期在ex段插入气泡，而if段和id段阻塞，mem段和wb段正常执行。

### 检测控制冒险

分支指令在mem段算出跳转条件是否成立，比较mem段分支地址单元提供的正确下地址和ex段给出的预测器给出的预测下地址，如果不一样，使得ifid\_flush, idex\_flush, exmem\_flush有效，同时通过组合逻辑使得pc\_src选择mem段的正确的跳转地址作为下个周期写入pc的值，并且使得bpu\_w\_en有效，送入mem段的当前pc以及正确的跳转地址，通知分支预测单元进行更新。注意由于分支预测器可能允许非跳转指令误撞，所以cu每个周期都在进行有意义的分支冒险检测。

一般情况下控制单元看到EX或MEM中存在气泡则不检测控制冒险。但是分支预测器可能预测出错误的指令并与前面的指令产生Load-Use冒险，此时必须检测控制冒险。此时MEM段不是跳转类指令而EX段是气泡，要看ID段的PC与MEM段的是否一致。

**绝对跳转指令和过程调用指令**在id段能判断出来，根据输入的指示信号，将ifid\_flush有效，pc\_src改为选择绝对跳转地址或寄存器跳转指令。

**分支和绝对跳转**，分支和load-use的冒险可能同时检测到，由于此时分支指令之后的指令都是错的，所以最优先生成分支指令发生冒险时的信号，而绝对跳转和load-use都需要id段的信息，所以不可能同时发生。

**检测syscall**：

id段生成的syscall信号，但是此时id段可能位于错误分支中。所以要将该信号传递到mem段再触发，因为如果能成功传递到exmem流水段寄存器，说明这个syscall是正确的要执行的指令。冲刷和阻塞信号同分支指令冒险的情况。输出对应的中断向量地址，pc\_src改成选择中断向量。cu\_cp0\_w\_en有效， cu\_exec\_code输出syscall对应的8，cu\_epc\_src选择exmem段的pc。不会与分支跳转冒险同时发生。

**检测eret/ret：**

id段判明，exmem段触发，效果同控制冒险，不过pc\_src改成epc/$31， 并冲刷前三个流水段寄存器，使得ifid\_flush, idex\_flush, exmem\_flush有效，。不可能分支跳转冒险同时发生。

**检测外部中断：**

外部中断信号从cp0传入，判明发生中断后，将cu\_pc\_src修改为中断向量的选择端，cu\_cp0\_w\_en置为有效，cu\_exec\_code赋值为0（表示是外部中断）,根据当前的分支冒险检测结果让cu\_epc选择ex段的pc还是正确跳转地址。

中断向量由cu计算，其他的如跳转地址可以直接连线过去。

## 输入

### Id\_ir

从id段送入的rs的地址，用于检查和ex段的数据依赖

### ex\_nop

用来忽视气泡造成的不必要的冒险，以及检测分支预测错误同时产生Load-Use冒险的控制冒险。

### mem\_nop

用来忽视气泡造成的不必要的冒险

### mem\_jmp

用于判断是否可能会有分支预测错误同时产生Load-Use冒险的控制冒险

### mem\_stall

由存储器送入的阻塞信号，cu负责进一步对流水段寄存器生成阻塞信号

### ifid\_rs\_addr [4:0]

从id段送入的rs的地址，用于检查和ex段的数据依赖

### real\_rt\_addr [4:0]

从id段送入的经过选择的rt的地址，用于检查和ex段的数据依赖

### idex\_rd\_addr [4:0]

从ex段送入的经过选择的真正的目标寄存器地址，用于检查和id段的数据依赖

### idex\_mem\_read

从ex段送入的读内存指示，在其有效的情况下数据冒险结果才有效

只需要检查load-use冒险

### predicted\_idex\_pc [31:0]

从ex段送入的pc地址，在mem段是分支跳转指令的情况下，此pc是分支预测器给出的预测地址，用于检查控制冒险

### predicted\_ifid\_pc [31:0]

用于判断由分支预测器送入的指令与前一条指令产生Load-Use冒险后是否发生控制冒险

### target\_exmem\_pc [31:0]

从mem段送入的计算偏移量并且经过条件选择的正确跳转地址，用于检查控制冒险和确定返回地址.

### cp0\_intr

由cp0的cause寄存器送入的中断组合信号，是8个中断比特的或

用于指示cu改变pc\_src，提供中断向量地址，情况保存epc并冲刷流水段寄存器

### id\_jmp

由id送入的绝对跳转指示信号（包括j和jal），指示cu改变pc\_src并准备冲刷ifid流水段寄存器

### exmem\_eret

由exmem送入的异常返回信号，指示cu改变pc\_src，冲刷前三个流水段寄存器

### exmem\_syscall

由id生成，传送到exmem的系统调用指示信号，指示cu改变pc\_src，提供中断向量地址，视情况保存epc并冲刷流水段寄存器

## 输出

### cu\_pc\_src [3:0]

选择pc下址：

0 – 绝对跳转地址

1 – 寄存器跳转地址

2 – 中断向量

3 – epc

4 – 正确跳转地址

5 – bpu预测结果（含正常+4）

### cu\_pc\_stall

### cu\_ifid\_stall

### cu\_idex\_stall

### cu\_exmem\_stall

### cu\_wb\_stall

各寄存器阻塞信号

### cu\_ifid\_flush

### cu\_idex\_flush

### cu\_exmem\_flush

各流水段寄存器冲刷信号

### cu\_cp0\_w\_en

cp0写使能，写入cause，关闭中断(移动status)，发生异常时有效

### cu\_exec\_code [4:0]

写入cause的原因：

0 – 外部中断

8 – 系统调用

### cu\_epc

送入cp0的中断返回地址

检测到mem\_syscall时，将epc赋值为predicted\_idex\_pc,检测到外部中断的时候，根据分支冒险的情况来确定epc。

### cu\_vector [31:0]

中断向量地址，不同的中断类型可能不同。（目前默认为0x80000180）

### cu\_bpu\_w\_en

bpu写使能，发生控制冒险时有效，用于更新bpu

## 程序说明

程序中每次将所有输出信号进行初始化， 具体如下：

cu\_pc\_src = 4'b0101;

cu\_pc\_stall = 1'b0;

cu\_ifid\_stall = 1'b0;

cu\_idex\_stall = 1'b0;

cu\_exmem\_stall = 1'b0;

cu\_ifid\_flush = 1'b0;

cu\_idex\_flush = 1'b0;

cu\_exmem\_flush = 1'b0;

cu\_cp0\_w\_en = 1'b0;

cu\_exec\_code = 5'b00000;

cu\_epc = 32'h00000000;

cu\_vector = 32'h80000180;

bpu\_write\_en = 1'b0;

后面按照信号说明来进行输出信号的修改。

# 附录 C. 转发单元设计说明

## 功能描述

因为I-type指令要在Wb段才将结果写回寄存器，但是如果紧接着的一条或者第二条指令要使用相关寄存器中的数据，在ID段从寄存器组中取数时，那些数据还没有被写回寄存器组，即在I-type指令写回之前就要拿到数据，这时就需要根据前后流水段指令对寄存器的读写情况，生成转发控制信号，控制选择ALU输入端的输入数据。

如果是出现load-use数据冒险，CU单元会控制流水线阻塞相应的流水段，在load-use之间插入气泡，然后通过转发Mem/wb流水段寄存器中的值即可解决。

主要的转发规则是：如果ID/Ex流水段指令的寄存器读地址与Ex/Mem、Mem/Wb流水段指令的寄存器写地址相同，则进行转发。

又因为lwl, lwr等类型的指令写使能可能不是全有效的，所以wb段的写回数据，只是部分改写目标寄存器的数据。为了保证转发的正确性，还需要将寄存器数据送入转发单元，由转发单元进行一次改写，将改写后的数据转发到ALU前的选择器。

有3种转发情况：不转发、从Ex/Mem段转发、从Mem/Wr段转发。

## 输入输出信号

### 输入

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **信号名** | **位宽** | **来源** | **使能** | **含义** |
| rs\_data | 32 | Id/ex流水段寄存器 | — | ID段从寄存器中取出的数据 |
| rt\_data | 32 | Id/Ex流水段寄存器 | — | ID段从寄存器中取出的数据 |
| rs\_addr | 5 | Rs地址 | — | rs源操作数地址 |
| rt\_addr | 5 | Rt地址 | — | rt源操作数地址 |
| exmem\_byte\_en | 4 | ExMem流水段寄存器 | 高有效 | Ex/mem流水段中数据写使能 |
| exmem\_rd\_addr | 5 | Exmem流水段寄存器 | — | Ex/mem流水段中数据写地址 |
| memwb\_byte\_en | 4 | MemWr流水段寄存器 | 高有效 | Mem/Wr流水段中数据写使能 |
| memwb\_rd\_addr | 5 | MemWr流水段寄存器 | — | Mem/Wr流水段中数据写地址 |
| memwb\_data | 32 | MemWr流水段寄存器 | — | Mem段从存储器中取出的数据 |

### 输出

|  |  |  |
| --- | --- | --- |
| **信号名** | **位宽** | **含义** |
| input\_A | 32 | wb段的转发数据与rs原始数据经过处理后的值 |
| input\_B | 32 | wb段的转发数据与rt原始数据经过处理后的值 |
| A\_sel | 2 | ID段操作数A的选择控制信号。00：不转发；01：  从Ex/Mem流水段转发；10：从Mem/Wr段转发。 |
| B\_sel | 2 | ID段操作数B的选择控制信号。00：不转发；01：  从Ex/Mem流水段转发；10：从Mem/Wr段转发。 |

## 转发情况判别

### 从Ex/Mem段转发

exmem\_byte\_en有效，

若 exmem\_rd\_addr与rs\_addr相同，

则 A\_sel输出为 2’b01

若 exmem\_rd\_addr与rt\_addr相同。

则 B\_sel输出为 2’b01

### 从Mem/Wr段转发

memwb\_byte\_en不全无效，

若 memwb\_rd\_addr与rs\_addr相同，

则 A\_sel输出为 2’b10

若 memwb\_rd\_addr与rt\_addr相同。

则 B\_sel输出为 2’b10

### 不转发

上述情况都不满足即为不转发，A\_sel和B\_sel的输出都为2’b00

## 信号输出逻辑

### 优先级

由于转发情况可能同时出现，在输出信号生成时需要确定优先级，先考虑优先级高的情况。

考虑到，让两条涉及转发的指令尽可能的靠近，所以正确的优先级顺序为：

从Ex/Mem段转发 > 从Mem/Wb段转发 > 其他。

### A\_sel输出逻辑

若exmem\_byte\_en有效 且exmem\_rd\_addr与rs\_addr相同：

A\_sel输出为2’b01

否则：

若memwb\_byte\_en有效 且 memwb\_rd\_addr与rs\_addr相同：

A\_sel输出为2’b10

否则：

A\_sel输出为2’b00

### B\_sel输出逻辑

若exmem\_byte\_en有效 且exmem\_rd\_addr与rt\_addr相同：

B\_sel输出为2’b01

否则：

若memwb\_byte\_en有效 且 memwb\_rd\_addr与rt\_addr相同：

B\_sel输出为2’b10

否则：

B\_sel输出为2’b00

## 数据改写逻辑

exmem\_byte\_en指示ALU运算结果是否可以写入，不会出现部分使能的情况。

所以数据改写仅发生在执行lwl系列指令过程中。

根据memwb\_byte\_en[3:0]的值，如果某一位的值为1，即该字节写使能有效，则rs\_data和rt\_data相应字节上的数据都修改为memwb\_data相应字节上的数据；如果某一位的值为0，字节写使能 无效，则rs\_data和rt\_data相应字节上的数据不修改。最后将修改后的数据送到input\_A和input\_B的输出。

# 附录 D. 分支预测器设计说明

## 功能说明

分支预测器位于流水线的最前端，它根据IF段提供的下一条指令PC给出预测的下一条指令PC，并且能够根据控制单元给出的信号，动态地修改预测结果。

## 模块说明

分支预测器的端口如下表所示

|  |  |  |  |
| --- | --- | --- | --- |
| 性质 | 端口名 | 位宽 | 说明 |
| 输入 | clk | 1 | 全局同步时钟 |
| reset | 1 | 全局复位信号 |
| bpu\_w\_en | 1 | 控制冒险发生标志，同时也是预测表状态更新使能 |
| current\_pc | 32 | 用于索引预测表项，获取预测PC |
| tag\_pc | 32 | 用于索引需要更新的预测表项 |
| next\_pc | 32 | 用于更新对应预测表项的预测PC |
| 输出 | predicted\_pc | 32 | 预测的下条指令PC |

### 预测机制

IF段总是将该段的当前PC加4作为下一条指令PC送入分支预测器的输入端current\_pc。分支预测器会截取current\_pc的一段作为下标，以直接映射的方式索引分支预测表。一个分支预测表的表项具有以下信息：

1. valid：标志该表项的有效性。有效的场合，直接使用存储的预测PC作为下条指令的PC输出；
2. predict：标志该表项对应的预测PC是否可以修改；
3. pc：存储的预测PC，注意其语义并不是分支指令的跳转地址，而是预测的地址，所以可以是顺序地址，也可以是跳转地址。

预测逻辑是纯组合逻辑。分支预测器通过current\_pc取出表项后，直接观察valid位，如果有效就直接输出预测PC；如果valid位无效则直接输出current\_pc作为下条指令PC，相当于不预测。

注意预测逻辑并不像cache那样会比对标签（因为并没有存储标签），所以有可能出现正常的顺序执行指令错误地预测出其他指令的地址，这时分支预测器会直接输出错误的PC。但是控制单元对每一条指令都会进行控制冒险的监测，所以可以保证指令提交时顺序的正确性。

另外要提醒的是一旦valid位有效，它就不会再无效。对应的表项将一直承担预测的任务，包括顺序执行指令的和分支指令的。

分支预测器的预测行为可用如下的框图直观地表示：

### 更新机制

只有在控制冒险发生时分支预测器才会更新预测PC。控制单元检测到控制冒险后，将bpu\_w\_en有效，将MEM段的PC和分支地址计算模块得到的正确的下条指令PC分别送入分支预测器的tag\_pc和next\_pc端口。分支预测器在bpu\_w\_en有效的周期，根据tag\_pc直接索引到对应的预测表项，首先将其valid位置为有效，表明这个表项从此与预测任务相关联了。然后检查predict位，如果为0，则将next\_pc写入表项的预测PC，由于所有predict位在复位时设置为0，所以一个新的valid有效的预测表项刚好predict位为0，允许next\_pc写入。如果predit位为1，则不写入，表示分支预测器允许一次错误。无论predict是否为0，在更新逻辑的末尾，都要将predict翻转，以实现延迟更新的效果。也就是说：

1. 如果当前predict为0，预测PC被更新，翻转predict为1，保证第二次预测错误时预测PC不被更新，即容忍一次预测错误；
2. 如果当前predict为1，预测PC不被更新，此次不更新的语义即容忍一次错误预测，但是要翻转predict为0，表明下次连续预测错误就要被更新。

分支预测器只有在同一个预测表项连续两次预测错误时才会更新预测PC。所以平时bpu\_w\_en无效的场合，表明tag\_pc对应的预测表项提供的预测PC是正确的，这时就要将表项的predict位置为1。需要注意的是，如果无视valid位直接将tag\_pc对应的表项的predict位置1，会影响valid位刚有效的表项的预测PC的写入，所以只有valid位有效的表项的predict位才能在bpu\_w\_en无效的情况下置1。

分支预测器的更新逻辑可以总结为如下的表格：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bpu\_w\_en | valid位 | predict位 | 新predict位 | 预测PC |
| 1 | 1 | 1 | 0 | 保持不变 |
| 0 | 1 | 修改为next\_pc |
| 0 | 一定为0 | 1 | 修改为next\_pc |
| 0 | 1 | 1 | 1 | 保持不变 |
| 0 | 1 | 保持不变 |
| 0 | 一定为0 | 0 | 为无效值 |

## 行为分析

分支预测器在默认情况下，也就是预测表项valid位无效时，直接输出current\_pc，即默认不预测，让IF段对应指令物理上的下一条指令（PC+4）进入流水线，对于分支指令来说，相当于默认不跳转。所以一个预测表项第一次有效时，预测PC填入的一定是分支指令的跳转目标地址。该存储了跳转地址的分支预测器下一次修改预测PC时，写入的PC性质将无法确定:它可能是原来的分支指令的顺序下地址，也可能是其它映射到同一表项的指令的下地址。

## 常见误区

常见的分支预测器，存储的是跳转与否的决策函数，以及跳转指令地址。一般是用输入PC预测出对应的下一条PC地址，并使用标签准确匹配待预测的PC。

在我们的分支预测器实现中，为了减少硬件资源消耗，并没有存储标签，所以无论是跳转指令还是顺序执行指令都有可能被预测；并且我们不是存储跳转决策函数以及跳转地址，而是直接存储预测地址；此外我们用于索引预测表项的输入PC并不是预测PC逻辑上的上一条指令PC，而是上一条指令PC+4。

所以，用于索引更新表项的tag\_pc，虽然来自MEM段，但是并不是MEM段的指令对应的PC，而是其PC+4。如果没有加4，则总是往真正需要更新的预测表项的上一项写入正确的预测地址，结果是分支预测器一直预测错误。

## 仿真测试

首先通过针对分支预测器模块的仿真文件进行单元测试。测试的主要项目是分支预测期的更新机制。即保持current\_pc和tag\_pc恒定的情况下，在特定的时机将bpu\_w\_en有效，对next\_pc进行多次更改，观察预测PC是否确实被更新以及容忍一次错误的策略有没有通过波形图反应出来。

之后将分支预测器接入流水线，通过执行有一定复杂度的测试程序，在对整个流水线的测试中，验证分支预测器功能的正确性。

# 附录 E. 分支地址计算部件设计说明

## 概述

在我们的CPU设计中，为了使流水线更加均匀，我们选择用4个周期完成branch指令，并且通过分支预测来降低控制冒险带来的cost，在EX段计算出分支目标地址（加法器）和分支条件标志位（ALU），在MEM段得到最终的目标地址。在MEM段的一个分支处理单元负责选择最终的目标地址。

## 模块信号说明

|  |  |
| --- | --- |
| Exmem\_branch | 是否为分支指令 |
| Exmem\_condition | 分支条件 |
| Exmem\_target | 分支目标地址 |
| Exmem\_pc\_4 | 当前PC+4 |
| Exmem\_lf | Less Flag |
| Exmem\_zf | Zero Flag |
| Final\_target | 真正的目标地址 |

## 具体功能

本模块要完成的工作与单周期CPU类似，但是有1点不同：

由于MIPS有延迟槽，而我们要做分支预测，所以我们会让编译器保证分支指令的下条指令一定为nop，以免发生错误。为了避免这些nop指令带来吞吐量下降，我们在分支预测时，如果预测不跳转，我们会将下条指令的pc设置为当前pc+8。所以在此单元中，如果这是一条分支指令，但是分支条件不满足，那么输出的Final\_target是pc+8而不是pc+4；

用伪代码表示：

If not branch:

Final\_target = pc +4

Else if condition\_satisfy(condition,lf,zf):

Final\_target = Exmem\_target

Else:

Final\_target = pc+8

# 附录 F. LOAD和STORE相关部件设计说明

## 概述

本次设计中，我们将load和store系列指令的字节写使能发生模块放到了EX段，还把store系列指令的数据移位器放到了EX段，load系列指令的移位器仍然在MEM段。（注意：为了冲刷流水段方便，我们的写使能信号都是1有效，0无效。）

## load系列指令的写使能信号

Load系列指令：

|  |  |
| --- | --- |
| LB | GPR[rt] ← sign\_extend(memory[addr]) , addr[1:0]可以为0~3 |
| LBU | GPR[rt] ← zero\_extend(memory[addr]) , addr[1:0]可以为0~3 |
| LH | GPR[rt] ← sign\_extend(memory[addr]) , addr[1]可以为0或1；addr[0]必须为0 |
| LHU | GPR[rt] ← zero\_extend(memory[addr]) , addr[1]可以为0或1；addr[0]必须为0 |
| LW | GPR[rt] ← memory[addr] , addr[1:0]必须为0 |
| LWL | 见手册 |
| LWR |  |

LB、LBU、LH、LHU这4条指令会对取出的数进行扩展，LW的取数位宽本身为4byte，所以他们对应的字节写使能都是4’b1111。而LWL和LWR指令的偏移量和addr[1:0]有关：

LWL：

|  |  |
| --- | --- |
| addr[1:0] | Reg\_byte\_write\_en |
| 0 | 1111 |
| 1 | 1110 |
| 2 | 1100 |
| 3 | 1000 |

LWR:

|  |  |
| --- | --- |
| addr[1:0] | Reg\_byte\_write\_en |
| 0 | 0001 |
| 1 | 0011 |
| 2 | 0111 |
| 3 | 1111 |

## store系列指令的写使能信号

Store系列指令：

|  |  |  |
| --- | --- | --- |
| 指令 | 行为 | 数据位宽 |
| SB | memory[addr] ← GPR[rt] , addr[1:0]可以为0~3 | 1byte |
| SH | memory[addr] ← GPR[rt] , addr[1]可以为1或0，addr[0]必须为0 | 2bytes |
| SW | memory[addr] ← GPR[rt] , addr[1:0]必须为0 | 4bytes |
| SWL | 见手册 | 不确定 |
| SWR |  |  |

store系列指令中的SB和SH指令不会对数据进行拓展。出于简化存储器设计的考虑，我希望送给存储器的**地址总是4字节对齐**的，即只有30位，然后再通过**字节写使能信号**控制每个字节是否被写入，通过移位得到要送入存储器的数据。

用一个例子来进一步说明我的想法：SB指令的addr计算结果为0x40000001，需要在0x40000001处写入一个字节，要写入的数据是0xff。按照我的思路，应该把地址(0x40000001 & 0xfffffffc)送给存储器，将字节写使能信号4’b0100送给存储器（大端方式），将数据0x0000ff00送给存储器。

SW：1111

SB：

|  |  |
| --- | --- |
| addr[1:0] | mem\_byte\_write\_en |
| 0 | 1000 |
| 1 | 0100 |
| 2 | 0010 |
| 3 | 0001 |

SH：

|  |  |
| --- | --- |
| addr[1] | mem\_byte\_write\_en |
| 0 | 1100 |
| 1 | 0011 |

SWL：

|  |  |
| --- | --- |
| addr[1:0] | mem\_byte\_write\_en |
| 0 | 1111 |
| 1 | 0111 |
| 2 | 0011 |
| 3 | 0001 |

SWR：

|  |  |
| --- | --- |
| addr[1:0] | mem\_byte\_write\_en |
| 0 | 1000 |
| 1 | 1100 |
| 2 | 1110 |
| 3 | 1111 |

## STORE指令的数据移位器

SW：不需要移位

SB: out = in << addr [1:0] \* 8

SH: out = in << addr[1] \* 16

SWL: out = in >> addr [1:0]

SWR: out = in << ~addr [1:0]

注：in表示来自寄存器的数据，out表示输出的数据

## LOAD指令的数据移位器

LW：不需要移位

LB: in = in >> addr [1:0] \* 8; out = in [7]24 + in [7:0]

LBU: in = in >> addr [1:0] \* 8; out = 024 + in [7:0]

LH: in = in >> addr [1]\* 16; out = in [15]16 + in [15:0]

LHU: in = in >> addr [1]\* 16; out = 016 + in [15:0]

LWL: out = in << addr [1:0]

LWR: out = in >> ~addr [1:0]

注：in表示来自存储器的数据，out表示输出的数据；取自存储器的数据宽度都是4bytes

# 附录 G. 乘除法功能部件说明

## 功能概述

乘法除法单元实现以下9个指令

DIV , DIVU , MUL , MULT , MULTU , MFHI , MFLO , MTHI , MTLO共9个指令。

在乘除法单元中，集成了HI寄存器和LO寄存器以及乘除法功能单元。

从乘除法操作码Md\_op中获得具体操作要求，根据操作要求执行乘除法操作或者MFLO , MFHI, MTLO , MTHI操作。将结果从Res\_out输出。

由于除法的运算会涉及到多个周期，所以在执行除法的时候，要停掉cpu中其他部件的执行，所以通过Md\_stall有效来告知cu，执行相关的停顿操作。

## 输入

Md\_op[3:0] 乘除法操作码

4’b0001 DIV

4’b0010 DIVU

4’b0011 MFHI

4’b0100 MFLO

4’b0101 MTHI

4’b0110 MTLO

4’b0111 MUL

4’b1000 MULT

4’b1001 MULTU

Rs\_in[31:0] rs操作数输入

Rt\_in[31:0] rt操作数输入

## 输出

Res\_out[31:0] 结果输出

Md\_stall 由乘除法单元送出的阻塞信号，和存储器输出的mem\_stall信号做或操作，输入到cu中

## 数据冒险相关

如果在指令1改变了HI寄存器的值， 紧接着的指令2要读取HI寄存器的值。因为在指令1中，HI寄存器的值已经写回，所以指令2读到的值就是最新的值，不存在RAW冒险。写读LO寄存器的情况是一样的。

如果指令1和紧接着的指令2都用写HI寄存器，因为Hi寄存器发射是按此发射，按序到达乘除法单元，所以不存在WAW冒险。写LO寄存器的情况是一样的。

如果指令1读HI寄存器，指令2写HI寄存器，因为指令时按序发射，按序到达乘除法单元，所以不存在WAR冒险。读写LO的情况是一样的。

在乘法除法器中涉及到通用寄存器的操作，其冒险处理由FU和CU处理。

# 附录 G. CP0设计说明

异常处理单元，实现STATUS, CAUSE和EPC寄存器。

以下描述中断均指外部中断。

## 寄存器格式：

### STATUS寄存器[31:0] 编号12

记录当前cp0状态信息，主要包括中断信息，是否允许中断和系统调用。

第0位记录外部中断允许位，1表示允许中断，0表示屏蔽中断。

第8-15位记录中断请求。暴露给外设。

其他位不做定义

初始化为32’h1

### CAUSE寄存器[31:0] 编号13

记录系统调用号和中断掩码

第2-6位记录系统调用号。

第8-9位对应status寄存器中相应位置，分配给软件中断。

第10-15位对应status寄存器中相应位置，分配给硬件中端。

其他位不定义

初始化为32’h1

### EPC寄存器[31:0] 编号14

记录中断返回地址, 在异常处理时记录了中断返回地址。

初始化为32’h0

### COUNT寄存器[31:0] 编号 9

时间计数器，流水线时钟信号的函数，以固定速度增加。达到32’hffffffff就清零

初始化为32’h0

## 输入：

### Wb\_cp0\_w\_en

mtc0写使能，高电平有效，由id段产生，memwb提供。

### Cu\_cp0\_w\_en

cu的异常写使能，高电平有效，cu提供。

### Epc [31:0]

记录中断返回地址，cu提供。

### Id\_cp0\_src\_addr [4:0]

cp0源寄存器地址，id译码出来直接送入，读出来的数据在id段使用， id译码单元提供。mcf0指令使用。

### Wb\_cp0\_dst\_addr [4:0]

cp0寄存器地址，memwb提供，mtc0指令使用。

### Ex\_data [31:0]

来自ex段的寄存器数据，memwb提供，是即将要写入cp0[wb\_cp0\_dst\_addr]的数据。

### Cu\_exec\_code [4:0]

来自cu的异常号。写入cause[6:2]

### Clk

时钟信号

## 输出

### Cp0\_data [31:0]

mfc0指令请求的数据，提供给ID

### Cp0\_epc [31:0]

中断返回地址，epc寄存器的值。提供给cu

### Cp0\_intr

组合逻辑，中断信号。提供给cu

## cp0行为：

### 时钟计数器增加

每次有效脉冲，count寄存器加一

### 检测到中断

中断存在的定义如下：

1. status[i]值为1(8<= i <=15)
2. cause[i]值为1
3. status[0]值为1

以上三个条件同时成立。

检测到中断之后，从cp0\_intr端口将中断抛出

### MTC0

wb\_cp0\_w\_en使能时，将ex\_data数据写入cp0[wb\_cp0\_dst\_addr]，如果wb\_cp0\_dst\_addr为14，即epc寄存器，并且cu\_cp0\_w\_en使能，则mtc0无效

### MFC0

将cp0[wb\_cp0\_src\_addr]数据从cp0\_data输出

### 捕获到系统调用

当cu\_cp0\_w\_en使能时，捕获系统调用，将cu\_exec\_code[4:0]的值写入cause[6:2]，提供给上层软件处理。保存epc

上述行为除时钟计数器外优先级由低到高。

# 附录 H. 存储器说明

## 概述

在设计图中，设计了IM和DM的接口，实现的时候可以考虑使用不同的存储元件，也可以采用相同的存储元件。前者更简单，后者容易引起阻塞，但后者可以更方便地使用loader。如果将IM和DM合并，则最好加上cache。另外，总线的IO也是通过DM接口来实现的：将一些地址映射到总线上，用于与外设进行交互。

## 指令存储器接口

### 输入

clk：存储器时钟

addr：指令地址

### 输出

instr：32位指令

mem\_stall：存储器无法在一个周期内完成取指，用此信号告知控制单元：流水线应该阻塞。

## 数据存储器接口

### 输入

clk：存储器时钟

read：表示本周期要取数据，设置这个信号的原因是并不是每个周期都要从存储器取数，为了防止错误的地址导致不必要的cache miss，甚至fatal

write：表示本周期要向存储器写入数据。

addr：存取数据的地址，某一确定范围内的地址代表的是总线，另一部分是显存，剩下的则是内存。宽度为30位，总是按照4字节对齐

data\_in：本周期要写入存储器的数据

mem\_byte\_w\_en：决定4个字节的数据是否写入到存储器。

### 输出

data\_out：读出存储器的数据

mem\_stall：存储器无法在一个周期内完成数据的存取。一般的，总线读写操作会引发阻塞

# 附录 I. 流水线顶层模块信号说明

注：所有流水段寄存器有clk, reset, cu\_flush, cu\_stall输入，下略。

## ifid流水段寄存器

输入为32位指令，指令PC和指令PC+4。输出为各种指令格式对应字段的信号。

输出id\_nop表明该流水段寄存器是否被冲刷。

（注：可以直接在内部完成J指令跳转地址的拼接）

## idex流水段寄存器

一般地，我们认为译码段不会成为最长的一个流水段，所以我们考虑尽可能在译码段多完成一些工作。所以我们将寻址方式比较特殊的指令在ID段进行处理，使他们在以后的流水段中表现出的行为与其他指令一致。由于流水段寄存器中所存的信号大部分来自译码部件，但是其中大部分都不会在译码段使用，所以下文分别说明译码段所使用的信号和为后面的周期准备的信号。

### 本流水段使用的信号

#### id\_rt\_zero\_sel

此信号用于选择通用寄存器组的Rt\_addr的源：因为bgez等指令需要用0与Rs中的值进行比较，但是这类指令中的一部分的Rt字段不全为0，所以我们需要手动修改输入到ALU中的op\_B的数据。

在流水线中我们还需要进行冒险检测，并且冒险检测最好在译码阶段完成，我们需要将Rt的地址送到控制单元进行分析。然而这类指令的Rt\_addr字段实际上都不是标注源操作数用的，它们的Rt\_addr字段不应该造成冒险，如果将非0的Rt\_addr送到控制单元，控制单元还得为这类指令开出特例才能保证不会“误判”。为了减轻控制单元的负担，我们在译码阶段确定该分支指令的Rt字段是否有（作为寄存器地址的）意义，如果没有，则把0送给通用寄存器组和控制单元。

#### id\_cp0\_src\_addr [3:0]

mfc0指令的源操作数地址。

#### id\_rt\_data\_sel

mfc0指令的源操作数来自cp0，而其他指令的源操作数来自寄存器组，此信号用于在取自cp0的数据和取自GPR中的数据间进行选择。如果该指令是mfc0则选择来自cp0的数据。

#### id\_jump

jump指令跳转所需要的信息在ID段完全可以获得，所以jump指令在ID段生效，在ID段将跳转信息送到控制单元。

id\_eret和id\_syscall与jump类似

#### id\_jr

jump register指令使用GPR[Rs]的值作为下一条指令的PC。通知控制单元改变pc\_src。

#### id\_imm\_ex [1:0]

选择立即数的扩展方式以及为lui指令开的特例。

#### id\_ctrl

此信号线标注的宽度为1，实际上它的意义是在ID段未使用到的控制信号，直接送入到idex寄存器中。我考虑到画图时idex寄存器的输出肯定会标注出所有的控制信号，如果在输入处也标注一次，显得比较冗余，所以我选择只画了一根线，但是实现的时候还是应该按照指令要求完成。关于id\_ctrl中包含的信号，见下文。

（注：在编写代码时，拟将这些信号显式地声明为指令译码器（ID）的输出端口和IDEX流水段寄存器的输入端口）

#### id\_rd\_addr\_sel

rd的地址有三种可能:

1. 指令中的Rt字段（mfc0包含在这里）
2. Rd字段
3. 5’b11111（jal等指令会将地址写入31号寄存器）

### 其它周期使用的信号

#### 控制信号

这些控制信号用图中用id\_ctrl表示，从译码器传送到idex寄存器，具体包含的控制信息在idex寄存器的输出中可以看到。

##### idex\_mem\_w

表示该指令写存储器。

##### idex\_mem\_r

表示该指令读存储器（避免触发不必要的cache miss）。

##### idex\_reg\_w

表示该指令写寄存器，主要是为冒险检测提供方便。

##### idex\_branch

传送到exmem寄存器，用于分支判断。

##### idex\_condition [2:0]

表示该分支指令所使用的条件。

##### idex\_of\_w\_disen

表示该指令在overflow时，不会写入寄存器，主要针对add等不允许溢出的算术指令。

##### idex\_exres\_sel [1:0]

选择将ALU或移位器的结果传到exmem寄存器（此前图上还花了一个pc+4作为选择器输入端，似乎没必要？）（看起来是有必要的，不然哪里让。

##### idex\_B\_sel

选择ALU的B操作数的来源：Rt或是经过扩展的立即数。

##### idex\_rs\_addr、idex\_rt\_addr、idex\_rd\_addr

寄存器地址，用作转发和冒险检测（仅Rd）判断依据。

##### idex\_load\_sel [2:0]

用于表示这是load系列指令中的哪一条，用于字节写使能信号的生成和数据移位。

##### idex\_store\_sel [2:0]

用于表示这是store系列指令中的哪一条，用于字节写使能信号的生成和数据移位。

##### idex\_cp0\_dst\_addr [4:0]

写入cp0的目的寄存器地址

##### idex\_cp0\_w\_en

cp0写使能，由mtc0产生

##### idex\_syscall

系统调用指示信号，由syscall产生

##### ex\_nop

表明该流水段寄存器是否被冲刷。

##### 与同单周期下含义相同的指令

idex\_alu\_op、idex\_shamt\_sel、idex\_shift\_op、idex\_shamt

#### 数据

##### idex\_op\_A，idex\_op\_B

可能送往ALU作为A B操作数的数据（这里其实应该用Rd和Rt命名）

##### idex\_pc

当前指令的pc

##### idex\_pc\_4

当前pc+4

##### idex\_imm\_ext

经过扩展的立即数

## exmem流水段寄存器

EX段完成的主要功能与单周期cpu的功能类似，详细说明略，EX段所需要的控制信号在idex段寄存器部分说明，EX段产生的新数据和控制信号在此说明。

### 输入

#### 控制信号

##### dex\_mem\_r、idex\_mem\_w、idex\_load\_sel、idex\_branch、idex\_condition

继承自idex寄存器的信号EX段产生的控制信号。

##### alu\_lf, alu\_zf, alu\_of

来自ALU的标志信息。

##### real\_rd\_addr

经过选择后的目的寄存器地址。

##### reg\_byte\_w\_en

字节写使能生成模块得到的寄存器写使能信号，如果of和idex\_of\_disen都有效，则全部置为0

##### mem\_byte\_w\_en

字节写使能生成模块得到的存储器写使能信号

##### ex\_nop

来自EX段的是否是气泡

##### ex\_jmp

来自EX段的是否是跳转指令

#### 数据

##### idex\_pc, idex\_pc\_4

继承自idex寄存器的数据。

##### adder\_target

来自加法器，分支指令目标地址。

##### ex\_res

ALU或者桶形移位器的结果。

##### aligned\_rt\_data

store移位器生成的经过对齐的数据。

### 输出

#### 送入data\_mem\_interface的信号

exmem\_mem\_w、exmem\_mem\_r、alu\_res（作为地址）、rt\_data（作为写入数据）、mem\_byte\_w\_en（字节写使能）

#### 送入控制单元的信号

exmem\_pc（和final\_target送入控制单元，控制单元根据当前EX段的指令的pc判断是否预测正确，对错误进行矫正，并且反馈到分支预测器。）

mem\_nop 送入MEM段的是否是气泡

mem\_jmp 送入MEM段的是否是跳转指令

#### 送入分支地址计算模块的信号

exmem\_branch、exmem\_condition、exmem\_target、exmem\_pc\_4、exmem\_lf、exmem\_zf（详见该模块说明）

#### 送入load移位器的信号

alu\_res低2位，exmem\_load\_sel（选择移位、扩展方式，详见该模块文档）

## memwb流水段寄存器

### 输入

#### exmem\_mem\_r

继承自exmem，用于选择写回到寄存器的数据：从存储器中取得的数据（load系列指令） 或 EX段得到的数据（其它指令）。

#### exmem\_reg\_w

继承自exmem，寄存器总写使能，一些不允许溢出的运算指令需要，可以用它来避免不必要的冒险（？）

#### reg\_byte\_w\_en [3:0]

继承自exmem，寄存器字节写使能。

#### exmem\_rd\_addr [4:0]

继承自exmem，目的寄存器地址（也可用于mtc0的目的寄存器，图上还没画相关电路和mtc0的sel信号；也还没有画要传递的异常信号）。

#### mem\_data [31:0]

mem段产生，来自存储器的数据。

#### ex\_data [31:0]

继承自exmem，来自EX段的数据。

#### exmem\_cp0\_dst\_addr [4:0]

继承自exmem，要写入的cp0寄存器的地址。

#### exmem\_cp0\_w\_en

继承自exmem，cp0寄存器写使能

#### aligned\_rt\_data [31:0]

继承自exmem，本质上就是GPR[Rt]的数据，对特殊的load指令进行了移位处理。

### 输出

与输入相同，exmem前缀换成memwb前缀。

#### 送入cp0的信号

memwb\_cp0\_dst\_addr[4:0], exmem\_cp0\_w\_en, aligned\_rt\_data[31:0]

#### 送入寄存器堆和转发单元的信号

memwb\_mem\_r选择memwb\_memdata[31:0]和memwb\_exdata[31:0]的输出, mem\_reg\_w(不用送转发), reg\_byte\_w\_en[3:0], memwb\_rd\_addr[4:0]

## 指令译码器ID

ID前缀的输出信号只在ID段使用，IDEX前缀的输出信号送入IDEX流水段寄存器。

这里只列出了指令译码器本身需要输出的控制信号，其他如寄存器地址的选择放在id段指令译码器的外部完成。

### 输入

#### ifid\_instr [31:0]

直接将整个指令送入进行译码

### 输出

#### id\_cp0\_src\_addr [3:0]

mfc0索引cp0寄存器组的编号，来自rd

#### id\_rt\_data\_sel

选择rt源操作数来源：gpr, cp0

#### id\_jump

绝对跳转指示信号

#### id\_imm\_ext [1:0]

符号扩展指示信号:

2‘b00: 无符号扩展

2’b01: 符号扩展

2’b10: lui扩展

2’b11: 符号扩展单元输出常数4，这个数字将来在EX段会计算出跳转地址PC+8，这是JAL需要的返回地址。当指令为JAL时，id\_imm\_ext==2’b11，同时必有idex\_exres\_sel==2’b10.

#### id\_rt\_addr\_sel

强制rt为$zero

#### id\_rd\_addr\_sel [1:0]

实际的目标地址可能是rt, rd和$31

#### id\_cp0\_dst\_addr [3:0]

mtc0索引cp0寄存器组的编号，来自rd

#### idex\_cp0\_w\_en

mtc0写使能

#### idex\_eret

异常返回指示信号

#### idex\_syscall

系统调用指示信号

#### idex\_mem\_w

主存写使能

#### idex\_mem\_r

主存读指示信号

#### idex\_reg\_w [3:0]

寄存器写使能

#### idex\_branch

分支跳转指示信号

#### idex\_condition [3:0]

条件判断方式选择信号

#### idex\_of\_w\_disen

溢出不可写信号

#### idex\_exres\_sel [1:0]

运算结果选择(alu和移位器)

0 – alu

1 – shifter

2 – PC+8

#### idex\_alu\_op [3:0]

同单周期

#### idex\_b\_sel

选择寄存器操作数和立即数

#### idex\_shamt\_sel

同单周期

#### idex\_shift\_op [1:0]

同单周期

#### idex\_shamt [4:0]

同单周期

#### idex\_rs\_addr [4:0]

转发判断依据

#### idex\_rt\_addr [4:0]

转发判断依据，可能的目标地址

#### idex\_rd\_addr [4:0]

可能的目标地址

#### idex\_load\_sel [2:0]

load类型选择，根据地址，不同的使能

#### idex\_store\_sel [2:0]

store类型选择，根据地址，不同的使能

## 转发单元FU

转发的来源有exmem流水段寄存器的alu运算数据和memwb流水段寄存器经过mem\_read选择后的写回数据，转发目的地是ex段alu要使用的rs和rt的寄存器操作数（其中rt的经过转发后的数据还需要进一步和立即数进行选择）。

由于lwl, lwr等类型的指令写使能可能不是全有效的，所以wb段的写回数据，只是部分改写目标寄存器的数据。为了保证转发的正确性，需要将寄存器数据送入转发单元，由转发单元进行一次改写。

mem段只用转发alu的计算结果，因为load-use冒险无法通过转发解决，我们在id段取操作数时检测出load-use冒险，此时访存指令执行到了ex段，正在计算地址，然后我们阻塞前半段流水线，在下个周期的ex段插入气泡，此时访存指令执行到mem段，之后流水线正常执行，访存指令已经执行到wb段，而需要写回数据的指令刚好执行到ex段（原来阻塞在id段），所以可以通过转发wb段来解决冒险。鉴于上诉情况，mem段只需要转发运算器的结果，而且没有写使能不全有效的情况。当然写使能还是要送入，因为要通过写使能全无效来决策是否可以转发。

转发单元需要知道mem段和wb段的写使能和目标寄存器地址。由于mem段数据比wb段的更新，所以优先转发mem段的数据。决定转发来源的规则如下：

对rs寄存器，如果mem段的目标寄存器即rs且写使能全有效，则选择mem段的转发数据（输出对应的选择信号）；如果mem段不满足要求，而wb段的目标寄存器即rs且写使能不全无效，则选择经过转发单元用原始数据根据写使能改写过的wb段的转发数据；如果都不满足要求，则不转发。对rt寄存器采取相同的操作。

### 输入

#### rs\_data [31:0]

原始的来自rs的数据，保证写使能无效字节的正确性

#### rt\_data [31:0]

原始的来自rt的数据，保证写使能无效字节的正确性

#### rs\_addr [4:0]

rs地址，由于检查写回寄存器是否被使用

#### rt\_addr [4:0]

rt地址，用于检查写回寄存器是否被使用

#### exmem\_w\_b\_en [3:0]

mem段回写寄存器的字节写使能，主要用于检查不允许写的情况

#### exmem\_rd\_addr [4:0]

mem段回写寄存器的地址

#### memwb\_w\_b\_en [3:0]

wb段回写寄存器的字节写使能，用于检查不允许写和部分写的情况

#### memwb\_rd\_addr [4:0]

wb段回写寄存器的地址

#### memwb\_data [31:0]

wb段回写寄存器的数据，用于和rs\_data和/或rt\_data依据memwb\_w\_b\_en做组合

### 输出

#### input\_a [31:0]

wb段的转发数据与rs原始数据经过处理后的值

#### input\_b [31:0]

wb段的转发数据与rt原始数据经过处理后的值

#### a\_sel [2:0]

#### b\_sel [2:0]

选择转发操作数：

0 – 不转发，使用原始数据

1 – 转发mem段数据

2 – 转发wb段数据

## 控制单元CU

控制单元检测流水按的冒险，控制流水段寄存器的冲刷和阻塞以及异常处理

检测load-use冒险：

根据输入的id段得到的rs寄存器地址和最终的rt寄存器地址，以及ex段继承自上一流水段的rd寄存器地址和mem\_read访存信号，如果ex段是访存指令（mem\_read有效），且rd与rs和rt中的一个相等，则通过组合逻辑使得pc\_stall, ifid\_stall, idex\_flush有效，在下一个周期在ex段插入气泡，而if段和id段阻塞，mem段和wb段正常执行。

检测控制冒险：

分支指令在mem段算出跳转条件是否成立，比较mem段分支地址单元提供的正确下地址和ex段给出的预测器给出的预测下地址，如果不一样，使得ifid\_flush, idex\_flush, exmem\_flush有效，同时通过组合逻辑使得pc\_src选择mem段的正确的跳转地址作为下个周期写入pc的值，并且使得bpu\_w\_en有效，送入mem段的当前pc以及正确的跳转地址，通知分支预测单元进行更新。注意由于分支预测器可能允许非跳转指令误撞，所以cu每个周期都在进行有意义的分支冒险检测。

绝对跳转指令和过程调用指令在id段能判断出来，根据输入的指示信号，将ifid\_flush有效，pc\_src改为选择绝对跳转地址。

分支和绝对跳转，分支和load-use的冒险可能同时检测到，由于此时分支指令之后的指令都是错的，所以最优先生成分支指令发生冒险时的信号，而绝对跳转和load-use都需要id段的信息，所以不可能同时发生。

检测syscall：

id段生成的syscall信号，但是此时id段可能位于错误分支中。所以要将该信号传递到mem段再触发，因为如果能成功传递到exmem流水段寄存器，说明这个syscall是正确的要执行的指令。冲刷和阻塞信号同分支指令冒险的情况。输出对应的中断向量地址，pc\_src改成选择中断向量。cu\_cp0\_w\_en有效，cu\_exec\_code输出syscall对应的8，cu\_epc\_src选择ex段的pc。不会与分支跳转冒险同时发生。

检测eret/ret：

id段判明，id段触发，效果同绝对跳转，不过pc\_src改成epc/$31。可能和分支跳转冒险同时发生，分支跳转冒险的信号优先输出。

检测外部中断：

外部中断信号从cp0传入，判明发生中断后，根据当前的分支冒险检测结果让pc\_src选择ex段的pc还是正确跳转地址。

中断向量由cu计算，其他的如跳转地址可以直接连线过去。

### 输入

#### mem\_stall

由存储器送入的阻塞信号，cu负责进一步对流水段寄存器生成阻塞信号

#### ifid\_rs\_addr [4:0]

从id段送入的rs的地址，用于检查和ex段的数据依赖

#### real\_rt\_addr [4:0]

从id段送入的经过选择的rt的地址，用于检查和ex段的数据依赖

#### idex\_rd\_addr [4:0]

从ex段送入的经过选择的真正的目标寄存器地址，用于检查和id段的数据依赖

#### idex\_mem\_read

从ex段送入的读内存指示，在其有效的情况下数据冒险结果才有效

只需要检查load-use冒险

#### predicted\_idex\_pc [31:0]

从ex段送入的pc地址，在mem段是分支跳转指令的情况下，此pc是分支预测器给出的预测地址，用于检查控制冒险

#### target\_exmem\_pc [31:0]

从mem段送入的计算偏移量并且经过条件选择的正确跳转地址，用于检查控制冒险

#### cp0\_intr

由cp0的cause寄存器送入的中断组合信号，是8个中断比特的或

用于指示cu改变pc\_src，提供中断向量地址，情况保存epc并冲刷流水段寄存器

#### id\_jmp

由id送入的绝对跳转指示信号（包括j和jal），指示cu改变pc\_src并准备冲刷ifid流水段寄存器

#### id\_eret

由id送入的异常返回信号，指示cu改变pc\_src，冲刷ifid流水段寄存器

#### exmem\_syscall

由id生成，传送到exmem的系统调用指示信号，指示cu改变pc\_src，提供中断向量地址，视情况保存epc并冲刷流水段寄存器

### 输出

#### cu\_pc\_src [3:0]

选择pc下址：

0 – 绝对跳转地址

1 – 寄存器跳转地址

2 – 中断向量

3 – epc

4 – 正确跳转地址

5 – bpu预测结果（含正常+4）

#### cu\_pc\_stall

#### cu\_ifid\_stall

#### cu\_idex\_stall

#### cu\_exmem\_stall

#### cu\_wb\_stall

各寄存器阻塞信号

#### cu\_ifid\_flush

#### cu\_idex\_flush

#### cu\_exmem\_flush

各流水段寄存器冲刷信号

#### cu\_cp0\_w\_en

cp0写使能，写入cause，关闭中断(移动status)，发生异常时有效

#### cu\_exec\_code [4:0]

写入cause的原因：

0 – 外部中断

8 – 系统调用

#### cu\_epc\_sel [1:0]

选择epc的来源:

|  |
| --- |
| if exmem.branch = true and cu.predict = true  epc <- exmem.target  else if exmmem.jmp = true  epc <- ifid.pc  else  epc <- idex.pc |

#### cu\_vector [31:0]

中断向量地址，不同的中断类型可能不同。

#### cu\_bpu\_w\_en

bpu写使能，发生控制冒险时有效，用于更新bpu

## 分支预测单元BPU

暂时的设计是if段的pc传到current\_pc计算正常执行的下一条地址，在缺失的情况下自动加4

由于分支指令的顺序执行地址是pc+8（越过延迟槽的nop），所以可以考虑根据冒险检测结果存储循序地址还是跳转地址。

分支预测存在由于由于误撞而取出非pc+4地址的情况，所以需要分支地址计算单元在正常情况下返回pc+4而不是pc+8或target\_pc，并且分支冒险的检测会常态化。

### 输入

#### current\_pc [31:0]

用来查询预测下地址的pc，来自if段

#### tag\_pc [31:0]

用来做标签的pc，是发生控制冒险的周期的执行到mem段的分支指令的pc

#### target\_pc [31:0]

被记录的目标跳转地址，是发生控制冒险的周期的执行到mem段的分支指令计算出的跳转地址

#### bpu\_write\_en

bpu更新使能。

### 输出

#### predicted\_pc [31:0]

预测器输出的pc下地址，pc的来源之一。

## cp0

异常处理单元，主要是含32个寄存器的寄存器组。需要一定的组合逻辑来控制数据写入。

id\_mfc0是用来选择写回数据的。

### 输入

#### interruption [7:0]

外部中断，此接口暴露给外设。

#### cu\_cp0\_w\_en

cu的异常写使能，并且用来分辨写来源。

#### cu\_exec\_code [4:0]

来自cu的异常号。异常号的写入与mtc0可能会产生冒险，可以根据使能判断决策如何写入cause寄存器。

#### epc [31:0]

记录中断返回地址，外部是一个选择器，选择不同的pc来源

#### id\_cp0\_src\_addr [4:0]

cp0源寄存器地址，id译码出来直接送入，读出来的数据在id段使用。

#### wb\_cp0\_w\_en

mtc0写使能，由id段产生，memwb流水段寄存器送入。

#### wb\_cp0\_dst\_addr [4:0]

cp0寄存器地址，memwb流水段寄存器送入，mtc0使用。

#### wb\_cp0\_data\_in [31:0]

来自ex段的寄存器数据，由memwb流水段寄存器给出，是即将要写入cp0[wb\_cp0\_dst\_addr]的数据。

### 输出

#### cp0\_data [31:0]

mfc0需要的数据

#### cp0\_epc [31:0]

中断返回地址，epc寄存器的值。

#### cp0\_intr

组合逻辑，中断信号。

## 存储器

在设计图中，设计了IM和DM的接口，实现的时候可以考虑使用不同的存储元件，也可以采用相同的存储元件。前者更简单，后者容易引起阻塞，但后者可以更方便地使用loader。如果将IM和DM合并，则最好加上cache。另外，总线的IO也是通过DM接口来实现的：将一些地址映射到总线上，用于与外设进行交互。

### 指令存储器

#### 输入

##### clk

存储器时钟

##### addr [29:0]

指令地址

#### 输出

##### instr [31:0]

32位指令

##### mem\_stall

存储器无法在一个周期内完成取指，用此信号告知控制单元：流水线应该阻塞。

### 数据存储器

#### 输入

##### clk

存储器时钟

##### read

表示本周期要取数据，设置这个信号的原因是并不是每个周期都要从存储器取数，为了防止错误的地址导致不必要的cache miss，甚至fatal

##### write

表示本周期要向存储器写入数据。

##### addr [29:0]

存取数据的地址，某一确定范围内的地址代表的是总线，另一部分是显存，剩下的则是内存。宽度为30位，总是按照4字节对齐

##### data\_in [31:0]

本周期要写入存储器的数据

##### mem\_byte\_w\_en [3:0]

决定4个字节的数据是否写入到存储器。

#### 输出

##### data\_out [31:0]

读出存储器的数据

##### mem\_stall

存储器无法在一个周期内完成数据的存取。一般的，总线读写操作会引发阻塞

## 分支地址计算模块

在我们的CPU设计中，为了使流水线更加均匀，我们选择用4个周期完成branch指令，并且通过分支预测来降低控制冒险带来的cost，在EX段计算出分支目标地址（加法器）和分支条件标志位（ALU），在MEM段得到最终的目标地址。在MEM段的一个分支处理单元负责选择最终的目标地址。

### 输入

#### exmem\_branch

是否为分支指令

#### exmem\_condition [2:0]

分支条件

#### exmem\_target [31:0]

计算出来的分支跳转地址

#### exmem\_pc\_4 [31:0]

当前pc+4

#### exmem\_lf

less flag

#### exmem\_zf

zero flag

### 输出

#### final\_target

真正的目标地址

具体功能

本模块要完成的工作与单周期cpu类似，但是有1点不同：

由于mips有延迟槽，而我们要做分支预测，所以我们会让编译器保证分支指令的下条指令一定为nop，以免发生错误。为了避免这些nop指令带来吞吐量下降，我们在分支预测时，如果预测不跳转，我们会将下条指令的pc设置为当前pc+8。所以在此单元中，如果这是一条分支指令，但是分支条件不满足，那么输出的final\_target是pc+8而不是pc+4；

用伪代码表示：

if not branch:

final\_target = pc +4

else if condition\_satisfy(condition,lf,zf):

final\_target = exmem\_target

else:

final\_target = pc+8

## load, store相关器件

本次设计中，我们将load和store系列指令的字节写使能发生模块放到了EX段，还把store系列指令的数据移位器放到了EX段，load系列指令的移位器仍然在MEM段。（注意：为了冲刷流水段方便，我们的写使能信号都是1有效，0无效。）

### load\_b\_w\_e\_gen

load系列指令的写使能信号

Load系列指令

|  |  |
| --- | --- |
| LB | GPR[rt] ← sign\_extend(memory[addr]) , addr[1:0]可以为0~3 |
| LBU | GPR[rt] ← zero\_extend(memory[addr]) , addr[1:0]可以为0~3 |
| LH | GPR[rt] ← sign\_extend(memory[addr]) , addr[1]可以为0或1；addr[0]必须为0 |
| LHU | GPR[rt] ← zero\_extend(memory[addr]) , addr[1]可以为0或1；addr[0]必须为0 |
| LW | GPR[rt] ← memory[addr] , addr[1:0]必须为0 |
| LWL | 见手册 |
| LWR | 见手册 |

LB, LBU, LH, LHU:

这4条指令会对取出的数进行扩展，LW的取数位宽本身为4byte，所以他们对应的字节写使能都是4’b1111。而LWL和LWR指令的偏移量和addr[1:0]有关：

LWL:

|  |  |
| --- | --- |
| addr[1:0] | Reg\_byte\_write\_en |
| 0 | 1111 |
| 1 | 1110 |
| 2 | 1100 |
| 3 | 1000 |

LWR:

|  |  |
| --- | --- |
| addr[1:0] | reg\_byte\_write\_en |
| 0 | 0001 |
| 1 | 0011 |
| 2 | 0111 |
| 3 | 1111 |

### store\_b\_w\_e\_gen

STORE系列指令的写使能信号

store系列指令

|  |  |  |
| --- | --- | --- |
| 指令 | 行为 | 数据位宽 |
| SB | memory[addr] ← GPR[rt] , addr[1:0]可以为0~3 | 1byte |
| SH | memory[addr] ← GPR[rt] , addr[1]可以为1或0，addr[0]必须为0 | 2bytes |
| SW | memory[addr] ← GPR[rt] , addr[1:0]必须为0 | 4bytes |
| SWL | 见手册 | 不确定 |
| SWR | 见手册 | 不确定 |

store系列指令中的SB和SH指令不会对数据进行拓展。出于简化存储器设计的考虑，我希望送给存储器的**地址总是4字节对齐**的，即只有30位，然后再通过**字节写使能信号**控制每个字节是否被写入，通过移位得到要送入存储器的数据。

用一个例子来进一步说明我的想法：SB指令的addr计算结果为0x40000001，需要在0x40000001处写入一个字节，要写入的数据是0xff。按照我的思路，应该把地址(0x40000001 & 0xfffffffc)送给存储器，将字节写使能信号4’b0100送给存储器（大端方式），将数据0x0000ff00送给存储器。

SW: 1111

SB:

|  |  |
| --- | --- |
| addr[1:0] | mem\_byte\_write\_en |
| 0 | 1000 |
| 1 | 0100 |
| 2 | 0010 |
| 3 | 0001 |

SH:

|  |  |
| --- | --- |
| addr[1] | mem\_byte\_write\_en |
| 0 | 1100 |
| 1 | 0011 |

SWL:

|  |  |
| --- | --- |
| addr[1:0] | mem\_byte\_write\_en |
| 0 | 1111 |
| 1 | 0111 |
| 2 | 0011 |
| 3 | 0001 |

SWR:

|  |  |
| --- | --- |
| addr[1:0] | mem\_byte\_write\_en |
| 0 | 1000 |
| 1 | 1100 |
| 2 | 1110 |
| 3 | 1111 |

### store\_shifter

STORE指令的数据移位器

* SW: 不需要移位
* SB: out = in << (addr [1:0] \* 8)
* SH: out = in << (addr [1] \* 16)
* SWL: out = in >> addr [1:0]
* SWR: out = in << ~addr [1:0]

注：in表示来自寄存器的数据，out表示输出的数据

### load\_shifter

LOAD指令的数据移位器

* LW：不需要移位
* LB: in = in >> (addr [1:0] \* 8); out = in [7]24 + in [7:0]
* LBU: in = in >> (addr [1:0] \* 8); out = 024 + in [7:0]
* LH: in = in >> (addr [1]\* 16); out = in [15]16 + in [15:0]
* LHU: in = in >> (addr [1]\* 16); out = 016 + in [15:0]
* LWL: out = in << addr [1:0]
* LWR: out = in >> ~addr [1:0]

注：in表示来自存储器的数据，out表示输出的数据；取自存储器的数据宽度都是4bytes

# 附录 J. 指令行为描述

## 分支跳转指令行为描述

本文档适用于以下指令：BEQ, BEQL, BGEZ, BGTZ, BLEZ, BLTZ, BNE, J, JAL, JR 共10条指令

其中BEQ , BEQL, BGEZ, BGTZ, BLEZ, BLTZ, BNE 7条branch指令为I-type指令。

J, JAL 为I-type指令。

JR指令为特殊的R-type指令。

### IF段

从PC中获得指令地址送入指令存储器，获得指令之后送入IF/ID流水线寄存器。

对于7条branch指令：

PC\_out送到BPU。由BPU来预测这条branch指令是跳转还是不跳转，BPU的输出送到一个选择器，由CU来控制选择下一条PC地址。

对于JR指令：

跟普通指令相同，没有特殊地方

### IF/ID段

IF/ID段流水线寄存器对指令进行分割，输出完整的指令ifid\_instr[31:0]提供给译码器。

对于7条branch指令：

输出ifid\_rs\_addr[4:0] , ifid\_rt\_addr[4:0]，由branch指令使用。

ifid\_rd\_addr[4:0]输出为无关输出。

输出ifid\_imm[15:0]作为偏移量提供branch指令使用。

对于j,jr,jal指令：

没有特别的地方，只是简单地传送ifid\_rs\_addr[4:0] , ifid\_rt\_addr[4:0], ifid\_rd\_addr[4:0], ifid\_imm[15:0]到后面去使用

### ID段

译码器从IF/ID流水段寄存器获得完整的指令，进行译码。

对于7条branch指令：

Id\_rt\_data\_sel输出为1’b1，选择从通用寄存器组中送出的值。

id\_rd\_addr\_sel输出为2’b01，选择从IF/ID流水段寄存器中输出的ifid\_rt\_addr[4:0]。（实际上分支指令并不会把计算结果送回该值所对应的寄存器中）。

id\_rt\_addr\_sel输出为1’b0，选择从IF/ID流水段寄存器中输出的ifid\_rt\_addr[4:0]。Id\_imm\_ext[1:0]信号有效，输出为2’b01。

Id\_ctrl输出其他相关的控制信号，其中的id\_branch信号有效，各个branch指令对应的condition[2:0]如下表：

Branch指令 Condition[2:0](3’bxxx)

BEQ 001

BEQL 001

BGEZ 011

BGTZ 100

BLEZ 101

BLTZ 110

BNE 010

位扩展单元将所有的ifid\_imm[15:0]进行符号扩展。

对于J, JAL两条指令：

Decoder中将id\_jump置为1

输出id\_jump\_addr[31:0]到PC前的选择器.

对于jal指令，将id\_rd\_addr\_sel赋值为3代表选择31号寄存器，即$ra，将来要写入返回地址,其他decoder信号无影响.

对于JR指令：

在decoder中将id\_ir置为1，id\_jump应为0，其它输出为无关输出,同时GPR中输出的rs\_data传送到pc选择器的选择端。

### ID/EX段

对于7条branch指令：

Idex\_condition[2:0]输出从译码器输出的id\_ctrl信号中的condtion[2:0]。

Idex\_exres\_sel[1:0]输出为2’b00，选择ALU的输出为计算结果。

Idex\_movz和idex\_movnz输出均为0，表明当前指令不是movz或者movnz指令。

Idex\_B\_sel选择1’b1。

Idex\_ALU\_op[3:0]输出从译码器中输出的id\_ctrl信号中的op[3:0]，均为4’b0001，表示作减法运算。

Idex\_imm\_ext[31:0]输出ID/EX流水寄存器的输入id\_imm\_ext[31:0]。

Idex\_op\_A[31:0]输出ID/EX流水寄存器的输入gpr\_rs[31:0]。

Idex\_op\_B[31:0]输出ID/EX流水寄存器的输入gpr\_rt[31:0]。

Idex\_rs\_addr[4:0]输出ID/EX流水寄存器的输入id\_ctrl信号中的相关内容，输出到转发单元。

Idex\_rt\_addr[4:0]输出ID/EX流水寄存器的输入id\_ctrl信号中的相关内容，输出到转发单元。

Idex\_branch输出1’b1。

Idex\_pc[31:0]输出到CU，用来判断branch是否预测正确。

对于jal指令：

Id\_reg\_w赋值为1, idex\_of\_w\_disen应该置为1，表示这条指令写寄存器并且溢出对其无影响，id\_reg\_w传送到exmem段寄存器

Idex\_mem\_w,idex\_mem\_r赋值为0，其他无关信号应该为默认的无效值

对于j和jr指令：

这个阶段发生跳转，跳转到相应的目标地址中

### EX段

由转发单元选择ALU的输入。

Id/ex流水段寄存器的输出idex\_ALU\_op[3:0]作为ALU的输入，控制ALU的行为。

ALU将计算结果输出（branch指令并不关心此结果），同时输出lf , zf , of标志，If, zf的结果送入ex/mem流水段寄存器。

Id/ex流水段寄存器的输出idex\_pc\_4[31:0]以及idex\_imm\_ext[31:0]相加，计算跳转的目标地址，结果送入ex/mem流水段寄存器的adder\_target[31:0]输入端口。

### EX/MEM

对于7条branch指令：

Exmem\_branch输出Idex流水段寄存器输出的branch信号。

Exmem\_condition[2:0]输出idex流水段寄存器输出的condition信号。

Exmem\_target[31:0]输出adder\_target[31:0]输入的值。

Exmem\_pc\_4[31:0]输出idex\_pc\_4[31:0]的值。

Exmem\_If和Exmem\_zf输出前面ALU计算后所得的标志位。

Exmem\_pc[31:0]为真正的跳转地址，输出到BPU。

对于jal,j,jr指令：

这个阶段不做任何事情,jal指令只是简单地将idex\_rd\_addr传送到下一个流水段寄存器

### MEM

对于7条branch指令：

ex/mem流水段寄存器输出的Exmem\_branch, Exmem\_condition[2:0], Exmem\_target[31:0], Exmem\_pc\_4[31:0], Exmem\_If, Exmem\_zf六个信号输出到分支地址计算模块，分支地址计算模块根据这些信号计算输出branch指令下一条指令真正的地址，结果送入CU。

CU判断branch指令是否预测正确，如果预测正确，流水段正常执行；如果预测错误，bpu\_write\_en写使能有效，修改BPU中的内容，将正确的跳转地址写入，同时ifid\_flush, idex\_flush, exmem\_flush信号有效，冲刷流水线。

对于jal指令：

Memwb\_reg\_w信号应该赋值为1，memwb\_rd\_addr应该是从上段传来的rd地址exmem\_rd\_addr

## 数据移动行为描述

本文档适用于以下指令：MOVZ，MOVNZ指令。

### IF段

从PC中获得指令地址送入指令存储器，获得指令之后送入IF/ID流水线寄存器。

### IF/ID段

IF/ID段流水线寄存器对指令进行分割，输出完整的指令ifid\_instr[31:0]提供给译码器。输出ifid\_rs\_addr[4:0] , ifid\_rt\_addr[4:0]，ifid\_rd\_addr[4:0]。

### ID段

译码器从IF/ID流水段寄存器获得完整的指令，进行译码。

Id\_rt\_data\_sel输出为1’b1，选择从通用寄存器组中送出的值。

Id\_rd\_addr\_sel输出为2’b01，选择从IF/ID流水段寄存器中输出的ifid\_rd\_addr[4:0] 。

Id\_ctrl输出其他相关的控制信号。

### ID/EX段

Idex\_condition[2:0]输出从译码器输出的id\_ctrl信号中的condtion[2:0]。

Idex\_exres\_sel[1:0]输出为2’b11，选择Rs寄存器的值为计算结果。

idex\_movz输出为1（movz指令），idex\_movnz输出为1（movnz指令）。

idex\_B\_sel选择1’b0。

Idex\_ALU\_op[3:0]输出从译码器中输出的id\_ctrl信号中的op[3:0]。

Idex\_op\_A[31:0]输出ID/EX流水寄存器的输入gpr\_rs[31:0]。

Idex\_op\_B[31:0]输出ID/EX流水寄存器的输入gpr\_rt[31:0]。

Idex\_rs\_addr[4:0]输出ID/EX流水寄存器的输入id\_ctrl信号中的相关内容，输出到转发单元。

Idex\_rt\_addr[4:0]输出ID/EX流水寄存器的输入id\_ctrl信号中的相关内容，输出到转发单元。

Idex\_rd\_addr[4:0]输出为指令的目标寄存器rd。

### EX段

由转发单元选择ALU的输入。

Id/ex流水段寄存器的输出idex\_ALU\_op[3:0]作为ALU的输入，控制ALU的行为。

ALU将计算结果输出，同时输出lf , zf , of标志。

### EX/MEM

Exmem\_rd\_addr[4:0]的输出目标寄存器地址。同时送入转发单元。

### MEM/WB

Memwb\_mem\_r输出为1’b1，输出mem/wb流水线寄存器的输入ex\_data[31:0]。

Memwb\_rd\_addr[4:0]目标寄存器地址。

Memwb\_reg\_w输出exmem\_reg\_w。

## 算术逻辑运算指令行为描述

本文档适用于以下指令：ADD , ADDU , ADDI , ADDIU , AND , ANDI , XOR , XORI , NOR , OR , ORI , SLT , SLTI , SLTIU , SLTU , SUB , SUBU 共17条指令

其中ADD , ADDU , AND , XOR , NOR , OR , SLT , SLTU , SUB , SUBU为R-type指令。

ADDI , ADDIU , ANDI , XORI , ORI , SLTI , SLTIU 为I-type指令。

### IF段

从PC中获得指令地址送入指令存储器，获得指令之后送入IF/ID流水线寄存器。

### IF/ID段

IF/ID段流水线寄存器对指令进行分割，输出完整的指令ifid\_instr[31:0]提供给译码器。输出ifid\_rs\_addr[4:0] , ifid\_rt\_addr[4:0]，由 R-type指令与I-type指令共用。输出ifid\_rd\_addr[4:0]仅提供给R-type指令使用， 输出ifid\_imm[15:0]仅提供I-type指令使用。

### ID段

译码器从IF/ID流水段寄存器获得完整的指令，进行译码。

Id\_rt\_data\_sel输出为1’b1，选择从通用寄存器组中送出的值。

如果是R-type指令，Id\_rd\_addr\_sel输出为2’b01，选择从IF/ID流水段寄存器中输出的ifid\_rd\_addr[4:0]，如果是I-type指令， id\_rd\_addr\_sel输出为2’b00，选择从IF/ID流水段寄存器中输出的ifid\_rt\_addr[4:0]。

id\_rt\_addr\_sel输出为1’b0，选择从IF/ID流水段寄存器中输出的ifid\_rt\_addr[4:0]。Id\_imm\_ext[1:0]仅在I-type指令输出有效，输出为2’b01。

Id\_ctrl输出其他相关的控制信号。

位扩展单元将所有的ifid\_imm[15:0]进行符号扩展，仅在I-type指令中有效。

### ID/EX段

Idex\_condition[2:0]输出从译码器输出的id\_ctrl信号中的condtion[2:0]。

Idex\_exres\_sel[1:0]输出为2’b00，选择ALU的输出为计算结果。

idex\_movz和idex\_movnz输出均为0，表明当前指令不是movz活着movnz指令。

如果是R-type指令，idex\_B\_sel选择1’b0，如果是I-type指令，idex\_B\_sel选择1’b1。Idex\_ALU\_op[3:0]输出从译码器中输出的id\_ctrl信号中的op[3:0]。

Idex\_imm\_ext[31:0]输出ID/EX流水寄存器的输入id\_imm\_ext[31:0]。

Idex\_op\_A[31:0]输出ID/EX流水寄存器的输入gpr\_rs[31:0]。

Idex\_op\_B[31:0]输出ID/EX流水寄存器的输入gpr\_rt[31:0]。

Idex\_rs\_addr[4:0]输出ID/EX流水寄存器的输入id\_ctrl信号中的相关内容，输出到转发单元。

Idex\_rt\_addr[4:0]输出ID/EX流水寄存器的输入id\_ctrl信号中的相关内容，输出到转发单元。

Idex\_rd\_addr[4:0]输出为指令的目标寄存器，R-type指令为rd , I-type指令为rt。

### EX段

由转发单元选择ALU的输入。

Id/ex流水段寄存器的输出idex\_ALU\_op[3:0]作为ALU的输入，控制ALU的行为。

ALU将计算结果输出，同时输出lf , zf , of标志。

### EX/MEM

当指令为ADD , ADDI, SUB且计算结果溢出，则exemem\_reg\_w无效。

Exmem\_rd\_addr[4:0]的输出目标寄存器地址。同时送入转发单元。

Exmem\_alu\_res[31:0]输出alu计算结果，输出到mem/wb流水段的ex\_data[31:0]。

### MEM/WB

Memwb\_mem\_r输出为1’b1，输出mem/wb流水线寄存器的输入ex\_data[31:0]，即ALU计算结果。

Memwb\_rd\_addr[4:0]目标寄存器地址。

Memwb\_reg\_w输出exmem\_reg\_w。

## 移位指令行为描述

本文档适用于以下指令：SLL, SLLV, SRA, SRAV, SRL, SRLV 共6条指令

它们均为R-type指令。

### IF段

从PC中获得指令地址送入指令存储器，获得指令之后送入IF/ID流水线寄存器。

### IF/ID段

IF/ID段流水线寄存器对指令进行分割，输出完整的指令ifid\_instr[31:0]提供给译码器。输出ifid\_rs\_addr[4:0] 在SLLV, SRAV, SRLV中被使用，ifid\_rt\_addr[4:0]，ifid\_rd\_addr[4:0]这六条指令都会使用。

### ID段

译码器从IF/ID流水段寄存器获得完整的指令，进行译码。

Id\_rt\_data\_sel输出为1’b1，选择从通用寄存器组中送出的值。

Id\_rd\_addr\_sel输出为2’b01，选择从IF/ID流水段寄存器中输出的ifid\_rd\_addr[4:0] 。

id\_rt\_addr\_sel输出为1’b0，选择从IF/ID流水段寄存器中输出的ifid\_rt\_addr[4:0]。

Id\_ctrl输出其他相关的控制信号。

### ID/EX段

Idex\_condition[2:0]输出从译码器输出的id\_ctrl信号中的condtion[2:0]。

Idex\_exres\_sel[1:0]输出为2’b01，选择移位器的输出为计算结果。

idex\_movz和idex\_movnz输出均为0，表明当前指令不是movz活着movnz指令。

Idex\_shamt\_sel，当指令为SLL, SRA, SRL时，输出为0，当指令为SLLV, SRAV, SRLV时，输出为1。

Idex\_shamt[4:0]输出id\_ctrl中的相关内容。

Idex\_shift\_op[1:0]输出id\_ctrl中的相关内容。

Idex\_op\_A[31:0]输出ID/EX流水寄存器的输入gpr\_rs[31:0]。

Idex\_op\_B[31:0]输出ID/EX流水寄存器的输入gpr\_rt[31:0]。

Idex\_rs\_addr[4:0]输出ID/EX流水寄存器的输入id\_ctrl信号中的相关内容，输出到转发单元。

Idex\_rt\_addr[4:0]输出ID/EX流水寄存器的输入id\_ctrl信号中的相关内容，输出到转发单元。

Idex\_rd\_addr[4:0]输出为 rd 。

### EX段

由转发单元选择移位器的输入。

Id/ex流水段寄存器的输出idex\_shift\_op[3:0]作为移位器的输入，控制移位器的行为。

移位器将计算结果输出。

### EX/MEM

Exmem\_rd\_addr[4:0]的输出目标寄存器地址，即rd。同时送入转发单元。

Exmem\_alu\_res[31:0]输出移位结果，输出到mem/wb流水段的ex\_data[31:0]。

### MEM/WB

Memwb\_mem\_r输出为1’b1，输出mem/wb流水线寄存器的输入ex\_data[31:0]，即移位结果。

Memwb\_rd\_addr[4:0]目标寄存器地址，即rd。

Memwb\_reg\_w输出exmem\_reg\_w。

## 乘除法指令行为描述

### MFHI

将HI寄存器中的内容从Res\_out输出

### MFLO

将LO寄存器中的内容从Res\_out输出

### MTHI

将Rs\_in的输入存储到HI寄存器中

### MTLO

将Rs\_in的输入存储到LO寄存器中

### MUL

Rs\_in 与 Rt\_in相乘，结果低32位从Res\_out输出（符号扩展）

### MULT

Rs\_in 与Rt\_in相乘， 结果低32为存放在LO寄存器中，高32为存放在HI寄存器中(符号扩展)

### MULTU

Rs\_in 与Rt\_in相乘， 结果低32为存放在LO寄存器中，高32为存放在HI寄存器中(无符号扩展)

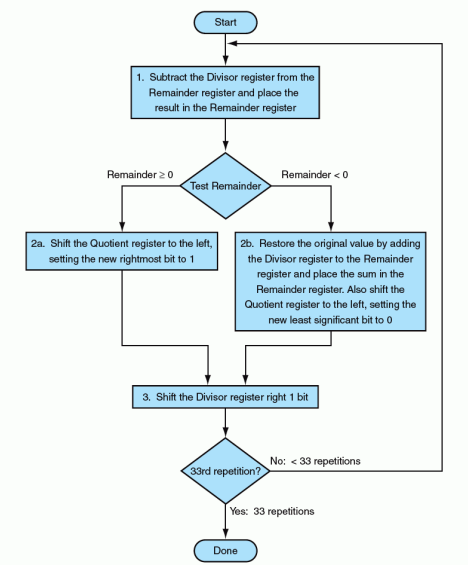
### DIV & DIVU

Rs\_in作为被除数， Rt\_in作为除数，执行除法计算， 计算结果存放在LO中，余数存放在Hi中。除数为0的情况暂不做处理。

DIV对源操作数做符号扩展， DIVU对源操作数做无符号扩展。

除法的执行要32个周期。在此期间，通过Md\_stall 有效阻塞其他部件的执行。

使用如下算法：



参考资料

Computer organization and design – the hardware / software interface