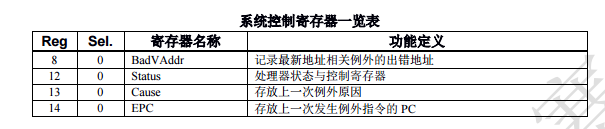
# 龙芯杯内部报告-CPO/异常/缓存

## 需求分析

### CP0协处理器

依据2018年第二届龙芯杯A03文档的规范，要求完成的CP0系统控制寄存器有4个，如下表：



如果要运行操作系统，必须要更多的寄存器。以LINUX为例，至少需要以下系统控制寄存器：

/\*Linux 必备的一些CP0寄存器\*/

CP0\_Index

CP0\_Random

CP0\_EntryLo0

CP0\_EntryLo1

CP0\_Context

CP0\_BadVAddr

CP0\_Count

CP0\_EntryHi

CP0\_Compare

CP0\_Status

CP0\_Cause

CP0\_EPC

CP0\_PRId

CP0\_EBase

CP0\_Config

CP0\_Config1

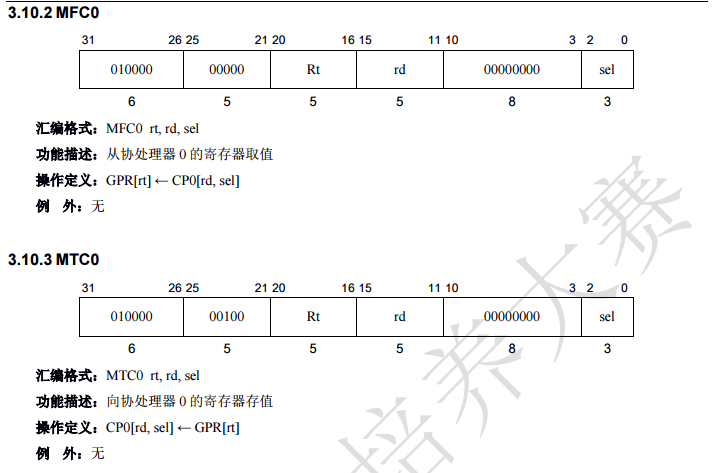
CP0是记录系统当前状态的重要寄存器组，设计时至少要考虑寄存器的复位、读出和写入。

复位通常都是置为0，但注意也有一些位段复位有初值，要根据实际的情况考虑。

读出一般有两种，一是硬件系统需要使用到部分位的值来进行控制，这时一般通过独立的模块输出读出；另一是软件系统通过CP0操作指令读出，这需要选择指令要求的寄存器值送至固定的输出。前者更为普遍，所以CP0模块有非常多的输出。

写入与读出类似，也有硬件操作和软件操作两种。不同的是硬件操作的写入不像读那么普遍，通常只有异常、中断发生时才会有硬件的写入，而且写入位置是特定的一些寄存器；而软件写入是频率很高的操作，和写通用寄存器一样，指令给出寄存器代号和写入的值，用一个指令周期完成。值得注意的是，有些位软件可能无权写，所以即使是特权级的写指令也只能操纵大部分位段。

需要实现的CP0操作指令在2018年龙芯杯A03文档中规定如下：



Sel字段是CP0寄存器在相同rd字段下的又一选择，由rd+sel拼接成的8位是CP0寄存器的唯一代号。

可以参考MIPS系统结构中更详细的规范。

### 异常

依据2018大赛的规范文档，要求处理的例外与中断如下表。硬件中断以及复用的时间中断虽然也要求实现，但在测试中没有加以检查。

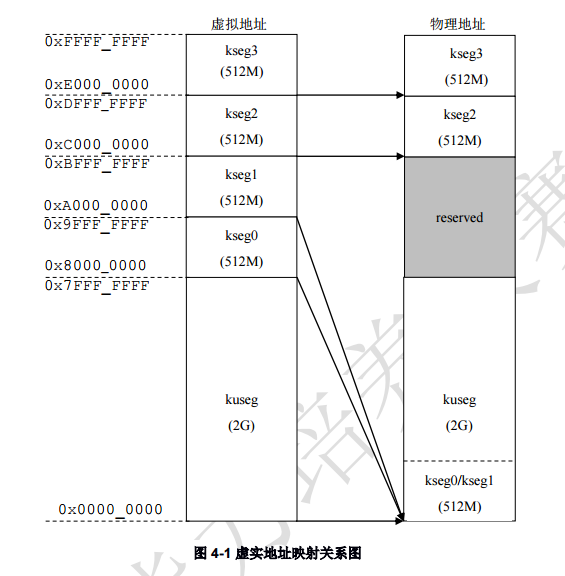
|  |  |  |  |
| --- | --- | --- | --- |
| 例外(中断)处理程序入口地址 | | **0xbfc00380** | |
| **例外类别** | **简称与Code** | **触发条件** | **处理要求** |
| 中断例外 | Int(0x00) | 软件或硬件修改CP0中的位 | 跳转到入口地址，记录ExcCode |
| 地址错例外(读) | AdEL(0x04) | 用于读操作的地址不对齐(不是需要的字、半字地址) | 跳转到入口地址，记录ExcCode，记录发生错误的地址 |
| 地址错例外(写) | AdES(0x05) | 用于写操作的地址不对齐(不是需要的字、半字地址) | 跳转到入口地址，记录ExcCode，记录发生错误的地址 |
| 整型溢出例外 | Ov(0x0c) | ADD\ADDI\SUB发生溢出 | 跳转到入口地址，记录ExcCode |
| 系统调用例外 | Sys(0x08) | SYSCALL指令被执行 | 跳转到入口地址，记录ExcCode |
| 断点例外 | Bp(0x09) | BREAK指令被执行 | 跳转到入口地址，记录ExcCode |
| 错误指令例外 | RI(0x0a) | 遇到无法识别的指令 | 跳转到入口地址，记录ExcCode |

例外与中断的优先级如下：



### 缓存

依据2018大赛的规范文档，系统存储的地址分配方案如下图所示，这一方案是兼容MIPS规范设计的。在功能测试、性能测试等初赛阶段的测试中，在虚地址kseg1段包含了两个存储从设备。一是主存储器RAM，二是承担板上各个硬件读写功能的confreg模块。所有指令和数据都存储在RAM中，而confreg可以操作板上的灯、按键、数码管等硬件，使得通过软件的指令可以操纵这些硬件外设。



根据大赛规定，这些模块都可以支持SRAM、AXI两种不同的接口协议，但如果使用SRAM简单接口，所获得的分数要减去一半。所以要求参赛队伍尽量都使用AXI接口协议。另外，在功能测试阶段，存储器响应会产生一个随机延迟，即存储器不会在当前时钟周期响应AXI接口的访存请求，而会随机等待若干个周期，在响应时传输数据前也会有随机延迟。这就要求设计AXI接口时必须支持随机延迟功能。

同时，2018大赛规定，在性能测试阶段，存储器对读访问固定25周期时延，对写访问固定3周期时延。这意味着系统运行时如果频繁地访问RAM主存，将会极大影响性能测试的发挥。这相当于对所有队伍提出了2个要求，一是几乎必须设计Cache，这样能够最大程度减少对主存的访问，减少读写的时延影响；二是在Cache的基础上，让每次访问传输比较多的数据，使得每次25周期时延“更有价值”，这需要使用AXI接口的多次突发连续传输功能，设计时要考虑这一功能的实现。另一点是Cache最好是写回工作模式的，这样还可以规避更多的写访问时延。总之，2018大赛规则虽然没有规定具体的存储方案，但对参赛队伍的存储方案设计还是提出了比较高的要求。

我们来比较一下具体的性能差距：如果没有cache，AXI总线每次传送1个字。如果需要读写访问32个字的连续地址，会产生32次读请求和32次写请求。时延方面需要32\*25+32\*3=896个时钟周期；另外AXI数据传送需要1个周期/字，AXI握手耗时假设4周期/次，则传送方面需要32\*(1+4)=160个时钟周期。合计约1056个周期，平均33个周期才能读写访问一个字。

而如果使用16个字一行的组相连Cache，写回模式。则时延方面只发生在2次读不命中的调入时，仅仅需要50个周期。写请求将在Cache范围内单周期完成，不需要访问主存，没有时延。传送时延上，每16个字一次传送，4个周期的握手耗时，20个周期可以传送一行，2次传送也只需要40个周期。加上单周期Cache读写的64个周期，合计约154个周期，平均4.8个周期可以读写访问一个字。

可见，设计Cache、支持连续传送的AXI总线协议能够带来近10倍的性能提升，实际情况可能还不止如此。但硬件外设相关的Confreg模块由于要求硬件立即显示结果，不能把数据写入Cache，而必须是Uncache模式的。因而Cache的设计需要考虑部分地址空间需要Uncache直接访问。

总结来说，存储方案的需求如下：

1. 支持大赛要求的地址分配方案；
2. 实现AXI协议访存；
3. AXI总线支持多次连续突发传输，支持数据Cache、指令Cache间的仲裁；
4. 需要设计Cache，使每行多个字，使用AXI协议(或其他协议再加上转接桥)；
5. Cache最好工作在写回模式；
6. 支持Uncache访问。

## 解决方案设计

### CP0协处理器

我们的CP0协处理器设计复制了清华大学2017年比赛开源代码，其开源的版本可以支持LINUX操作系统的运行，但一些地方设计不符合2018年大赛规范，我们进行了修改。但由于最后没有尝试LINUX系统在自己平台上运行，所以不清楚修改后是否仍满足LINUX操作系统的需要。如果要清华大学完整的参考版本，请访问<https://github.com/z4yx/NaiveMIPS-HDL>

主要的修改部分包括输入输出名称的修改，增加支持的CP0寄存器，将硬件中断同步部分移入CP0模块，修改部分复位的初值，修改写入逻辑等。

### 异常

我们的异常处理模块使用了简化的清华大学2017年比赛开源代码，其开源的版本可以支持LINUX操作系统的运行。如果要清华大学完整的参考版本，请访问：

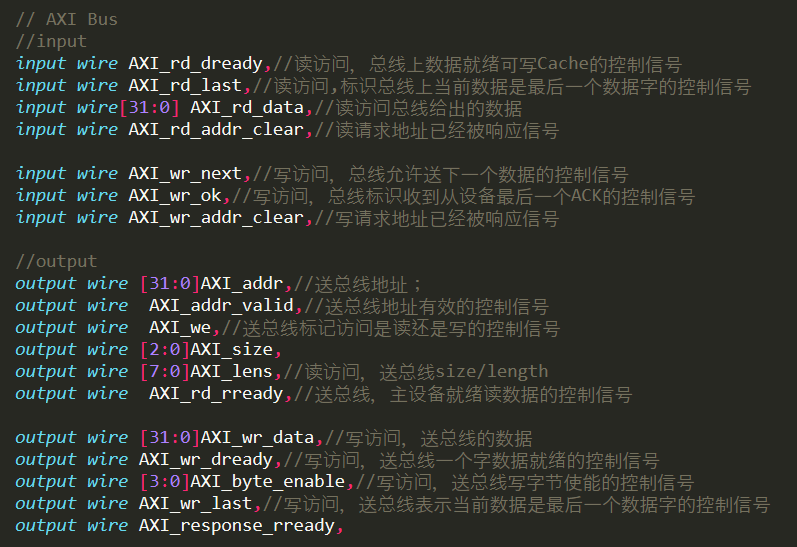
<https://github.com/z4yx/NaiveMIPS-HDL>

### 缓存

Cache缓存设计时的几个基本思路是：模块化逐级实现、与AXI接口模块沟通时序状态机、采用全参数化设计。但从最后的成品效果来看，这种设计也带来模块细分过多，布线困难，关键路径过长、资源消耗过多等问题，在后期如果要实现Cache指令也非常困难，所以可以考虑重新规划Cache的实现。（这段可以回过头再看）

我们的设计是Cache(dcache\_with\_uncache.v)和AXI接口模块(AXI\_interface.v)共同构成一个独立模块(cache\_axi.v)，这样对CPU而言，只需要将访存命令发给cache\_axi联合体，即可获得返回数据，屏蔽了具体访存的细节。如果存在延迟，则由cache\_axi模块向CPU发送停机信号，一直等待到访存成功为止。由于大赛框架中访存不会出错，总线不会出错，所以可以这样简化，实际情况如此设计CPU会完全死机。

这一方案的好处是cache\_axi部分可以独立开发，独立测试。CPU在设计时可以先采用简化SRAM接口，提前开始调试。直到cache\_axi完善后安装即可。由于Cache和AXI两个部分是不同两人开发的，所以如何沟通联系二者的“内部协议”尤为关键。在Cache侧可以看到大量AXI接口I/O，它们在AXI\_\_interface侧也是对称的，这些都是为了实现二者的“内部协议”。Cache还有一部分接口是对CPU开放的，用于接受CPU的请求。Cache的设计要求就是满足CPU的访问请求，并在必须访问主存或外设时，通过状态机实现的一定时序配合AXI接口模块，完成AXI协议。



我们需要实现的是一个组相联Cache，写回模式。所以必然会有大量的存储体，它们组成Cache行、组等等，具体的容量大小可能需要使用参数设置，方便调整Cache的大小。所以整个Cache的设计采用了完全参数化设置，实例化模块时全部采用generate关键词，定义接口位宽时全部采用标识符参数。

为了达到完全参数化的要求，需要对Cache各级模块进行封装。我们一共设计了4层封装，第一层是一个Cache行，它只有一个参数，即Cache行包含多少个字，称为行大小；第二层是由若干个Cache行构成的Cache组，它也有一个参数，是组相连的路数，路数越大每个Cache组就包含越多的Cache行；第三层是整个Cache，它也有一个参数，是整个Cache的容量大小，其定义了总共有多少个Cache组。除了存储体外这一层还包括与AXI接口模块沟通的时序状态机。第四层是Cache和Uncache两个模块构成的整个dcache\_with\_uncache，其中uncache部分仅仅有一个状态机，不包含存储体。要实现的是根据地址信息，选择当前访问是否通过Cache。

第一层封装的模块名称是CacheLine，第二层封装的模块名称是CacheGroup，第三层封装的模块名称是DCache或UnCache，第四层封装的模块名称是DCache\_with\_UnCache 和ICache(实际未使用)。

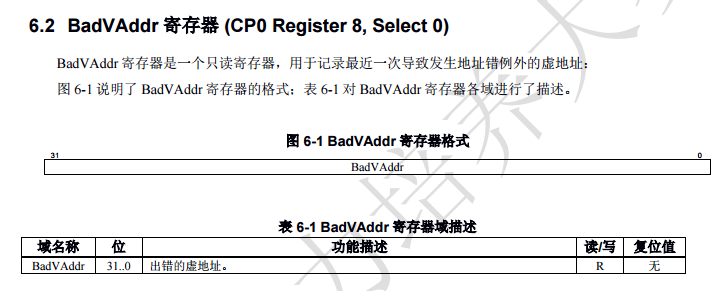
## 详细设计

### CP0协处理器

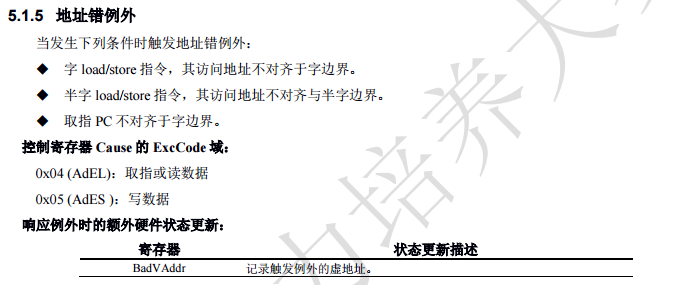
这里重点说明2018大赛所要求的4个CP0寄存器的复位、读出、写入逻辑。同时结合cp0\_reg.v文件中的源码加以说明。

#### CP0\_BadVAddr

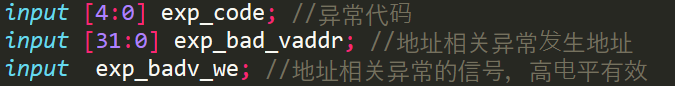
大赛要求实现如下



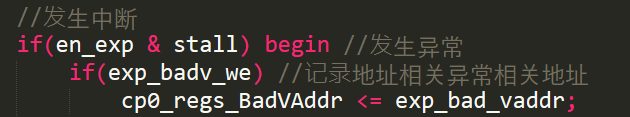
这里的只读是指软件只能读取该寄存器的值，不能写入寄存器的值。复位值无可以认为复位0即可。其写入是硬件完成的，硬件对CP0寄存器组的写入一般都发生在中断响应时，大赛具体要求如下：



即发生AdEL、AdES两种例外时，要将触发例外的虚地址同时写入BadVaddr寄存器中，因此就需要若干输入输出。

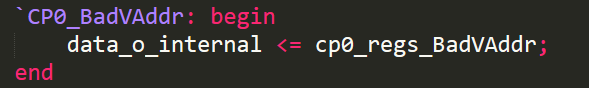


Exp\_badv\_we标识了发生地址相关例外，也就作为CP0更新BadVAddr寄存器的使能信号。具体的硬件写入逻辑如下，该逻辑表示当发生异常，且系统非停机状态(stall停机信号，低电平有效)，如果Exp\_badv\_we信号标识了地址相关例外，则写入寄存器。



CP0\_BadVAddr没有软件写入操作，所以不在其他任何地方赋值；其没有固定的复位值，所以也不需要上电复位。

读出其中的值使用CASE语句，只需要将整个寄存器的值赋值到输出即可。该寄存器没有其他硬件的读出端口。



#### CP0\_Status

大赛要求实现如下





BEV位是用来标识启动时异常向量的。该位有一个硬件的读出接口boot\_exp\_vec，用于控制异常跳转的基址：



其在异常处理模块中的控制逻辑如下：



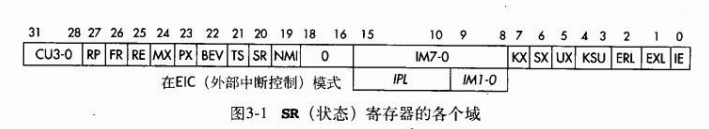
异常处理程序的入口地址是基址+偏移得到的，也就是说该位为1时，系统使用固定基址BFC00200。根据2018大赛要求，这一位可以设为常量1 ，因为其要求所有异常入口地址都是基址BFC00200+偏移0x180。如果要运行操作系统，该位要在系统引导完成后，再修改为0。

IM是中断屏蔽字段，其完全由软件控制，正常读写即可。

EXL是用于例外发生后，标识例外的状态，并陷入内核态的。MIPS系统的内核态由Status寄存器的4个位控制，当且仅当这4个位值为1000时，系统处于用户态，其余所有情况下都是内核态。CP0模块有一个独立的硬件输出user\_mode标识了当前的系统特权级，如下：



可以参考《MIPS体系结构透视》这本书，这4个位分别是2位的KSU，1位的ERL和EXL。

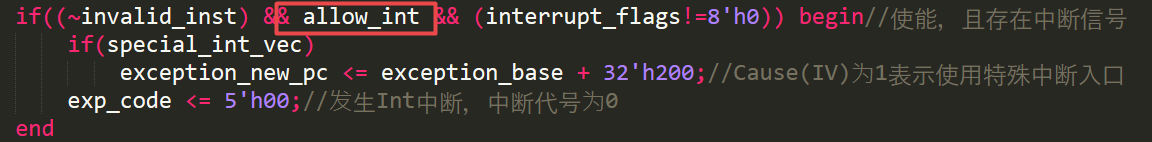


当KSU为10时，系统才能处于用户态，而2018大赛A03文档中约定KSU常为00，所以可以得知其要求系统一直工作在核心态下。虽然如此EXL位就没有陷入核心态的功能，但需要了解在操作系统中，当EXL被异常置为1时，系统会离开原来的用户态（如果是用户态的话）。\*该段文字的数字为2进制

EXL的另一功能就是标识例外状态，为1表示系统已经进入中断处理程序了。除了自动处于核心态以外，此时系统不能够响应外部硬件软件中断，所以需要屏蔽。这里的屏蔽不是给中断屏蔽字IM设值，而是在异常处理模块中通过电路设计屏蔽。同时为了保护现场，防止中断返回出错，例外状态下EPC和Cause两个寄存器的值不能够被修改。

Allow\_int是CP0允许硬件软件中断标识的输出接口，就是用于异常处理模块进行屏蔽。在下图异常处理模块中，响应中断需要检查allow\_int是否为1。





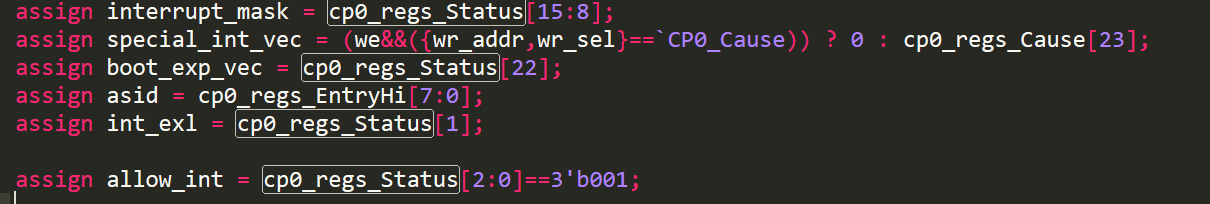
allow\_int也包含了IE的功能，如果IE为0的话，allow\_int将为0，也就是屏蔽所有中断。

最后总结，Status寄存器复位的初值为0x00400001。

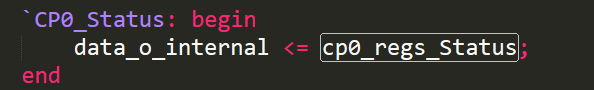


其硬件读出有以下输出接口：

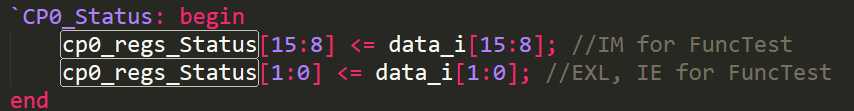




软件读出只需读出整个寄存器值即可：



软件写入只能操作IM和EXL、IE字段：



硬件也可以写入，仅仅在中断发生时(或中断返回ERET指令执行时)。如果发生中断，且系统没有停机，则EXL位置1。根据非阻塞赋值的特点，取EXL旧值，如果是0才修改EPC和Cause寄存器相应位段。如果是ERET指令发出clean\_exl信号，则EXL恢复为0。



#### CP0\_Cause

大赛要求实现如下



Cause寄存器是用于记录与中断、例外有关的信息的，包括例外的编号，中断源，是否在延迟槽发生等等。

BD位标识了是否在延迟槽发生中断。根据要求2018大赛的系统设计是支持延迟槽机制的。延迟槽是接在跳转指令之后的一条缓冲指令，使得跳转指令执行时不会浪费周期。一般指令导致的例外在处理后将返回该指令处继续执行，此时指令上下文已经被破坏。如果延迟槽指令触发的例外返回原处，则上一条的跳转指令将无法找回，相当于没有进行跳转，程序会错误地继续向下执行。所以延迟槽例外必须返回到延迟槽之前的跳转指令处继续执行，而不是返回延迟槽的位置。

TI位是用于位IP7的硬件中断源复用标记的。IP7既可以由系统外部的硬件中断源触发，也可以由系统内部的时间中断触发，所以需要一个位用以标记是时间中断还是外部中断。在2018大赛的初赛要求中，没有涉及外部硬件中断的部分；同时我们没有运行能够进行进程调度的操作系统，所以外部硬件中断和时间中断都没有起到作用，因而要求没有那么高。在设计时，我们把IP7简化为无论谁触发都把TI置为1，都视为时间中断。这当然是不符合MIPS规范的，但在本次大赛中没有影响。如果测试涉及硬件中断、时间中断时，就必须改正，或者要运行支持进程调度的操作系统，也必须改正。

如上文所述，IP7—IP0是系统的8个中断源，其中IP7-IP2是6个硬件中断源，IP7还复用为时间中断源（但时间中断所需的寄存器没有要求实现）。在2018大赛规范中它们只能软件读取，这显然是不对的。当外部硬件中断触发后，要清除中断就需要软件在中断处理程序中将其清0，这需要能够写入。因此可以认为2018大赛规范中没有要求实现硬件中断。而IP1和IP0是2个软件中断，它们要求能够用软件读写，事实上测试程序也检测了软件中断的正确性，必须按照要求实现。

ExcCode是例外的编号，在异常模块中应该对发生的例外生成符合MIPS规范的编号，并在响应异常时将其写入CP0。这个编号硬件可以写入，但软件不能通过指令写。软件可以通过指令读取编号，从而调用相应的中断处理程序。

主要编号如下：

软硬件中断INT：0x00

载入的地址出错AdEL：0x04

写入的地址出错AdES：0x05

系统调用Syscall：0x08

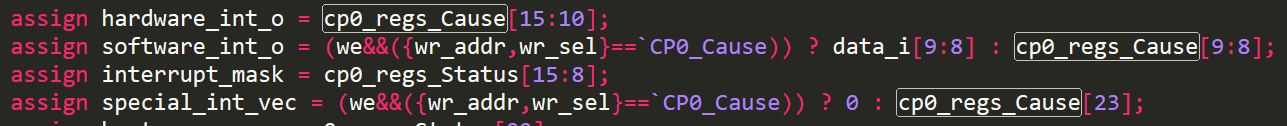
调试器调用Bp：0x09

未定义指令RI：0x0a

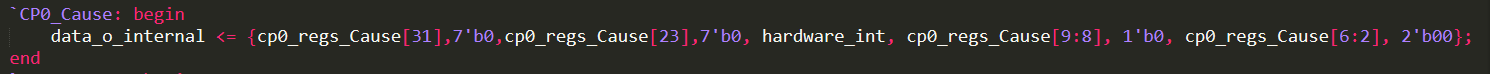
整数运算溢出Ov：0x0c

最后总结，Cause寄存器复位的初值为0x00000000。

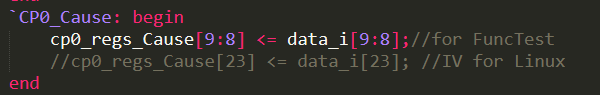
其硬件读出有以下输出接口：



软件只能读出已实现的部分位：



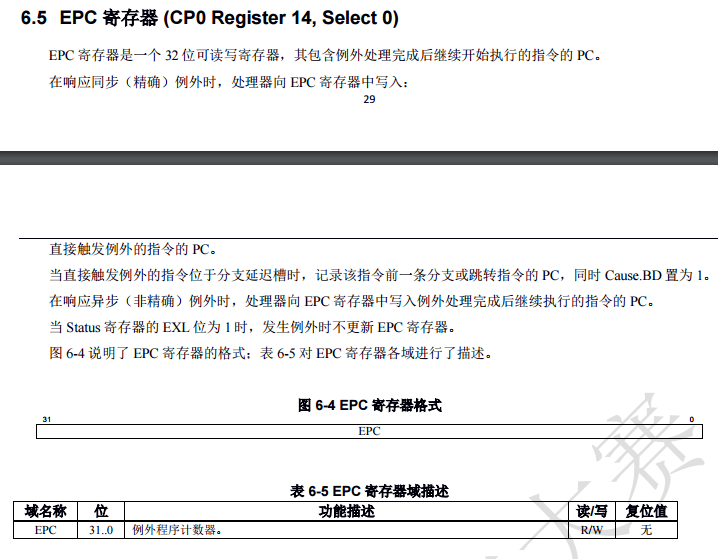
软件写入只能操作软件中断的中断源：

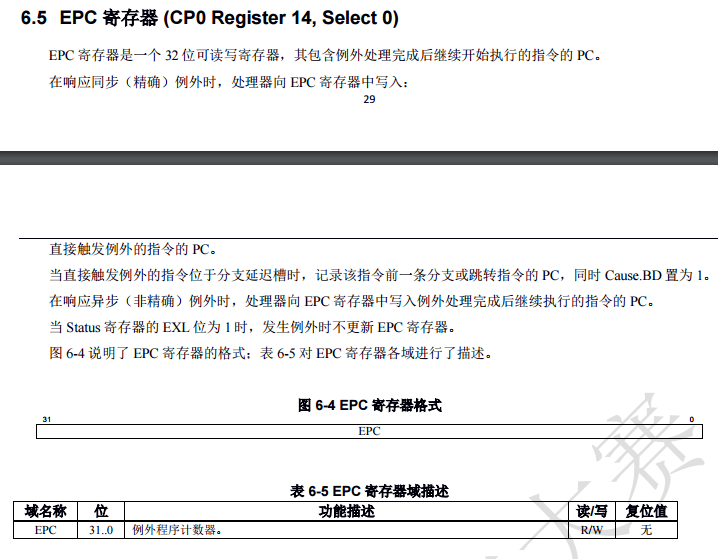


硬件需要在发生中断时记录中断编号、延迟槽发生中断、是否是计时器中断等。异常的编号只要发生异常就进行修改，但延迟槽中断的标记位只能在非异常处理状态下，即EXL位为0时才能修改，以保证程序处理完异常能返回正确的位置。这里对计时器中断进行错误简化，把IP7的所有硬件中断都视为时间中断，这是不正确的。



#### CP0\_EPC



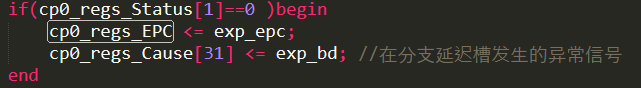


EPC是一个很简单的CP0寄存器，其保存了中断发生的地址，在此不多赘述。

它的值可以通过硬件接口读出，因为很多地方都会使用到。



当发生异常时，如果EXL位为0，需要记录异常模块发来的EPC的值。



通过软件，该寄存器可以写入和读出，这些操作都是针对整个寄存器的。

### 异常

异常模块是一个完全组合逻辑模块，它没有时序部分。它被设计在MEM段，如果发生异常等于发生了一次跳转，因此IF\ID\EX段的指令和数据必须全部清空，但WB段的指令已经执行完成。依照MIPS标准规范，WB段的指令执行时如果有写寄存器组信号发出，该信号只能维持一个时钟周期，之后必须撤除。但我们的数据通路设计没有考虑这一点，WB段的写信号会错误地维持多个周期，这不符合规范。

ERET指令不是一个真正的异常，但它也需要修改EXL位、跳转到EPC寄存器保存的返回地址处，行为非常类似于一个异常，所以出于简化起见可以将其视为一个伪异常，一并处理。与MMU模块(TLB表)相关的异常也是操作系统运行的必需，但本设计版本由于没有使用MMU进行地址映射，所以删去了这一部分的异常处理逻辑。

异常模块的注释较为详细，在此不再赘述。

### 缓存

#### CacheLine

Cacheline是C ache最小粒度上的存储体，其存储体包括2^OFFSET\_WIDTH个32位的数据寄存器，TAG\_WIDTH位的TAG标识，脏位，有效位等。我们令Cacheline的数据总是读出的，所以没有读使能信号，其输出端口总是显示存储体的值；写入则由一个写使能信号wr\_write控制，写入时数据部分依据字节使能信号更新，TAG、脏位和有效位必须更新。

wr\_byte\_enable是一个4位宽的使能信号，表示写入数据时4个字节中哪些要被修改，哪些维持原值，这个控制信号由上层传送而来；RST复位时需要把Cache整体清空，将脏位、TAG、有效位都置为0。

#### CacheGroup

Cachegroup是cache组相联中功能最重要的一层。它内部包含多个Cache行，上层通过地址的INDEX段选中某一组后，Cachegroup需要在组内依据TAG和有效位来进行匹配，找到需要操作的Cache行。这会有3种结果，如果匹配命中，即要操作的地址就在缓存中，则要把读写操作引导到对应的行上；如果匹配不命中，则需要进行调入操作。如果这一组没有满，则要把调入时的写操作引导到空行上；如果该组已满，则需要执行淘汰策略，这里使用的是伪随机淘汰算法，这是资源最为节省的淘汰算法。如果要淘汰的行数据是脏的，则首先告知上层需要写回，将需要淘汰的行写回主存，然后将该行的VALID置为0。这样就回到了上一种情况，进行调入即可。

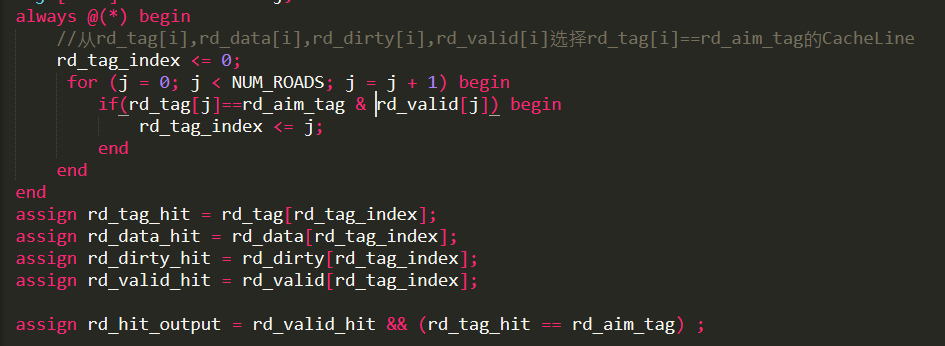
这个设计有一个问题，由于我们没有运行LINUX操作系统，所以没有更深入地考虑。LINUX操作系统要求支持MIPS指令集中的Cache指令，因为启动后LINUX的引导程序要对Cache进行初始化操作。Cache指令有一个功能是使得某一组的(由地址的INDEX段指定)所有行全部失效，这是一个批量操作。如果它们之中有脏的数据，还要依次把它们全部写回缓存。显然，我们设计的Cache组模块只支持一次引导一个Cache行的操作，而且必须由TAG字段来进行匹配，如果没有命中操作行会聚焦到空行上。这种设计不支持对整个组批量操作、批量写回。如果要运行LINUX操作系统，Cache的功能必须加以改进，以支持Cache指令的执行。这主要有两种思路，一是如果遇到Cache指令，进行一种特殊的循环，依次把每一行检查一遍，进行写回或失效；二是遇到Cache指令先把所有脏的行写回，然后整个Cache组统一失效。当然，除此之外肯定还有其他思路！

下面逐段分析模块中的代码的实现。

第一部分使用generate for语句对各个Cache行进行实例化。NUM\_ROADS是上层传递的一组多路的设置参数，如果是2路组相连，则CacheLine模块会被实例化两次。小写i是循环变量，它指定第一个CacheLine的接口下标全部为0，而另一个接口下标全部为1。一般2路组相连的性能已经足够，不太需要更大的组相连路数。

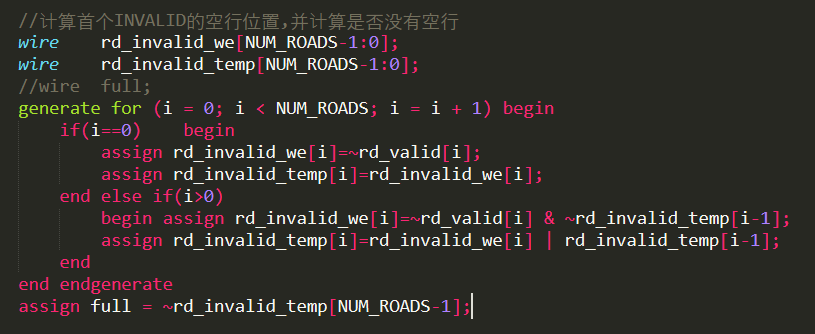
每个CacheLine都有输入输出，其中输入直连到CacheGroup的输入，wr\_write向量选中了唯一要写入的行；输出则有3种，分别为命中、空行、随机淘汰，要根据实际情况选择一个作为CacheGroup的输出。

首先获取命中情况下的数据：这里使用for循环轮询所有行的TAG字段，看是否和需要的rd\_aim\_tag相同，且有效位是否为1。如果是则将其下标j记下，命中情况的选择就是下标J这一行的输出。默认情况下，j=0；rd\_hit\_output记录了是否命中的结果。



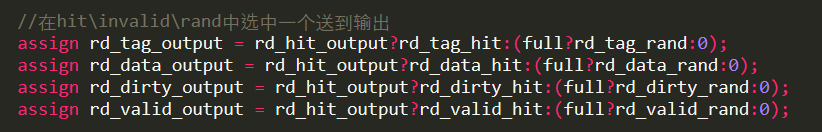
接下来获取随机行的数据：rand是外部传入的随机数，将其直接作为下标即可选中随机一行。这个随机数会在每次Cache访问时自增1，溢出后恢复为0，相当于一个伪随机数。

接下来计算空行位置的过程比较复杂：首先建立两个与行数等长的数组，从头开始遍历，如果第一次发现某行是空行，在该空行位置数组rd\_invalid\_we中标记该行为1，在另一个空行标记数组rd\_invalid\_Temp中标记已经查找到空行标志为1。如果已经查找到空行了，则之后的遍历过程空行位置数组全为0，空行标记数组均拷贝1。这样如果有空行，空行标记数组rd\_invalid\_Temp最后一位会为1，空行位置数组rd\_invalid\_we会有唯一一个1选中该空行；如果没有空行，空行标记数组rd\_invalid\_Temp最后为0，空行位置数组rd\_invalid\_we全为0。

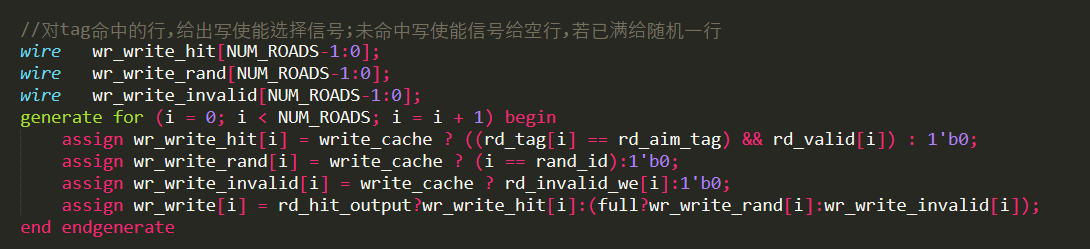


Full表示组是否已满，而rd\_invalid\_we标记出了第一个空行，这将在之后作为写使能信号。

接下来构成输入输出。输出是根据实际情况选择的数据，如果命中，即rd\_hit\_output为1，则将命中情况下的数据送到输出；否则如果已满，即full为1，则将随机一行的数据送到输出；其他情况将空行数据送到输出。命中标记和full本身也作为输出，用于上级模块的控制。



输入除了写使能信号外，其他都是直连的。写使能信号也是根据命中、满等标记来给出的，选中的是对应需要操作的cache行。Generate for语句除了对模块批量实例化外，还可以批量生成assign语句，可以理解为更强大的宏。



#### DCache

DCache是Cache运行的逻辑层，它要根据Cache的情况，响应CPU发来的访存请求。必要时命令CPU停机，向AXI总线接口请求数据或写回数据。在DCache代码里有大量的信号分发代码，这些主要用于扇出优化。是否有这些代码不影响运行逻辑，它们只是将一个单一信号拷贝多份，分发到各个存储体而已，可以理解为信号的复制。

DCache的运行逻辑是：每当CPU发来一个读写请求时，在存储体中查找这一地址是否存在。如果是，则将读写操作引导到对应的Cache组、Cache行上，在本时钟周期完成读写操作；如果在存储体中没有对应地址的数据，则进行不命中的处理，将一行新数据调入Cache。

如果需要调入的目标组还有空行，则在空行处写入即可。首先让CPU暂停运行，向AXI总线发送地址握手请求；等待到地址握手成功后开始接受总线数据；直到一行数据传送结束后，向AXI总线发送确认信号，释放总线。在传输过程中，每个时钟周期都会将上一个周期的数据写入存储体；传输结束后有一个单独的时钟周期来改变Cache的控制位信息；调入结束后，撤销CPU暂停信号，系统重新开始运行。

如果需要调入的目标组已满，势必要考虑将淘汰的数据写回。如果淘汰的行数据不脏，可以直接丢弃；如果数据已被修改过，则要在调入之前先将淘汰行的数据写回主存。具体过程和读入类似，首先发送地址握手请求；等待地址握手成功；在每个字发送后，要等待AXI总线的确认信号，才能继续发送下一个字的数据，否则暂停等待；如果不使用连续突发传输机制的话，每次传输只能传输一个字。所以尽量要实现连续突发时序，在所有字传送结束时，伴随最后一个数据帧要发送ACK确认。写回完成后，有一个单独的时钟周期来改变Cache的控制位信息，可将VALID位置为0，这样就变为有空行状态，但暂时不需要撤销停机信号。

另外还实现了Cache指令的一部分功能，包括命中时强制失效、强制数据同步等。这些操作只要稍微修改控制信号的逻辑表达式即可。但没有实现Cache指令依据INDEX字段批量处理整个Cache组的功能，如果要运行LINUX系统必须考虑实现该指令全部必要的功能。

注意，在设计时一定要充分认识到各个信号是完全并行变化的，所有信号可以看做是一个大型状态机，在每个时钟沿发生一次变化。状态机中的赋值语句都应该是非阻塞的，这样才和状态机外的赋值语句时序上一致。

这一部分代码逻辑比较零散，在源代码中写有较详细注释，在此不过多赘述。

#### DCache\_with\_UnCache

DCache\_with\_UnCache模块主要是为了增加Uncache访问功能。其根据地址解析的控制信号，将访问请求发送到DCache模块或Uncache模块。所以其内部包含DCache和Uncache模块的实例化。Uncache是一个只实现了AXI状态机的简单模块，在AXI总线上依照每次传送一个字的方式传输。

值得一提的是，Uncache模块判断CPU发来的请求是否是一个新的请求的依据也是来自CPU的。通过采样上一周期前一流水段的停机信号，可以判断当前周期的请求是否更新过，这样避免了在Uncache模块中加设标志位记录的问题，否则会浪费一个时钟周期。