计算机系统

MIPS 微系统

当我们完成 P6 以后，可以说我们已经完成了一个复杂度相当高的 CPU，这个 CPU 可以利用流水线技术并行指令，并按照指令的内容完成计算，修改寄存器中的值。那么是不是完成这个 CPU 就是掌握计算机组成这门课程的全貌了呢？我们说并不是这样。

CPU 的全称是 Central Processing Unit，也就是“中央处理器”。也就是说，CPU 的功能就是数据的“加工与处理”。而计算机系统不仅要完成数据的加工处理，还要完成诸如输入，输出，存储，网络等多种功能，也就是说，**“CPU 不等价于计算机系统”**。在 P7 我们要完成的任务，就是实现一个简单的计算机系统，也就是“MIPS 微系统”。

外设

外设即外部设备，他们可以看做是与 CPU 地位平等的一组设备。在计算机系统中，CPU 负责数据的加工处理，而外设则负责输入（鼠标，键盘），输出（显示屏，扬声器），存储（硬盘，U 盘），网络（网卡）等。我们的 MIPS 微系统中包括的外设主要有 3 种，即计时器、存储器、中断发生器：

- 计时器（Timer）：计算机系统中的计时部件，可以按照配置定时地产生时钟中断。
- 存储器（Memory）：计算机系统中的存储部件，用于存储指令和数据。我们在 P6 的时候已经接触过了。
- 中断发生器（InterruptGenerator）：抽象的计算机系统外设，会随机的产生外部中断信号，产生的中断信号在 CPU 响应前会持续置高。

正如 CPU 一样，这些外设也可以用 Verilog 语言对其进行建模。最终我们整个微系统都可以用 Verilog 语言建模。为了让同学们在 P7 有一个更好的实现体验，我们在这一章会只进行设计概念上的介绍，而在下一章介绍具体的实现规格和细节。希望同学们在这一章先熟悉一些系统设计的基本理念，然后在下一章具体的实现中去进一步体悟。

支持异常处理流的 CPU

之前的 CPU

在 P6 我们完成的 CPU 已经具有了很好的功能，他可以顺序的执行指令（有时会发生跳转）。那么我们考虑，这样的一个 CPU 还有哪些缺陷，我们认为主要有两点：

- 此时的 CPU 是没有办法处理错误的指令的。例如 `add` 指令，当两个源操作数相加发生溢出的时候，按照规范这是**异常**的情况。我们在 P6 的时候默认“溢出加”，会得到一个错误的答案。我们会在 P7 对这个问题进行解决。
- 此时的 CPU 是没有办法实现与外设复杂的交互的，当计时器向 CPU 传递一个信号的时候，CPU 是没有办法立刻响应这个信号并做出相应的处理的。

直观地说，P6 设计出的 CPU 运行时的可靠性和完备性都不令人满意。如果运行在其上的程序有一些 bug，CPU 既不能检测到这些 bug 并向使用者报告，也不能做出保证正确性的处理，CPU 无法满足现实场景下的多种功能。

异常处理流

异常处理流指的是，CPU 在执行程序的指令的时候，会发生一些“事件”，改变程序的原有流向，让 `PC` 跳转到特定的地址。

异常处理流可以很好的解决上面的问题。当指令执行错误时，可以产生一个“事件”。那么 CPU 就会跳转到一个处理这种执行错误的程序上执行，在处理结束后再跳转回原来的程序（不一定是“事件”来临时的地址）。对于外设的信号，我们可以将其视为一个“事件”，当“事件”来临时，CPU 会跳转到一个响应这个“事件”的程序处进行响应，在处理结束后再次跳转回原来的程序。

异常处理流可以用下面的这张图直观地表示，可以将其理解为 **“发生位置不确定的过程调用”**。

概念辨析

为了降低同学们实现的困难，我们参照《See MIPS Run Linux》制定了以下概念规范。需要强调的是，这些概念的名字仅在 P7 的实现中有效。不同的参考资料对于相同的事物可能会给出不同的概念名称，所以可能指导书会与教材、参考资料或者授课 PPT 的概念存在冲突。

在 P7 实验中请以指导书为准，在其他场景中请具体分析。

我们之前提到的“事件”一共有两种：

概念	定义	举例
内部异常	由于指令执行错误导致的“事件”	加法溢出，除法除零等
外部中断	由于外部设备信号导致的“事件”	计时器信号，键盘输入等

因为这两类“事件”的处理具有一定的共同性，所以我们统一称他们为“异常”，如下图所示：

为了响应异常，CPU 会自动跳转到某一特定的地点（将 PC 修改为特定值），然后进行异常处理。这里进行异常处理的程序叫做“异常处理程序”，是软件（在计组中表现为一段汇编代码），是不属于 MIPS 微系统内的。在课程平台的自动测试中，评测数据将包含 handler，但是我们仍要编写它来进行本地测试。

思考题

请查阅相关资料，说明鼠标和键盘的输入信号是如何被 CPU 知晓的？

思考题

请思考为什么我们的 CPU 处理中断异常必须是已经指定好的地址？如果你的 CPU 支持用户自定义入口地址，即处理中断异常的程序由用户提供，其还能提供我们所希望的功能吗？如果可以，请说明这样可能会出现什么问题？否则举例说明。（假设用户提供的中断处理程序合法）

软硬件接口

接口的理解

正如前面所描述的，在 P7 要完成的不再是 CPU 这个单独的功能模块，而是 CPU 与多个外设组成的计算机系统，各个模块之间需要进行合作。那么应该如何进行合作呢？

合作的前提就是“约定”。参与合作的人必须对其他人给出可以让自己完成功能的“约定”。比如说在“老师和学生上课”这个合作关系下，老师需要给出自己的上课时间和下课时间。如果老师并不告诉学生这个信息，这个合作就是进行不下去的。在计算机系统里，我们称这种“约定”为“接口”。

接口的设计

接口设计重要的一个方向就是要**足够简洁。**依然用“老师和学生上课”的例子，“我在 8:00 上课，你们需要在这个时刻到达” 和 “我在 7:30 到达食堂吃早餐，然后吃 20 分钟烧麦，10 分钟后到达教室给你们上课，你们需要在这个时刻到达” 两种“接口”都是可以让学生正常上课，但是学生肯定会更喜欢第一种，因为这种接口的设计简洁，学生并不需要了解老师吃早饭的细节。

那么是什么带来了这种简洁性呢？并不是因为第一个老师不吃早饭导致的，两个老师都是吃早饭的，但是第一个老师贴心的计算出了自己到达教室的时间，而第二个老师将自己早上的所有细节都暴露了出去。

同时，这种简洁性并非是功能元件本身的简洁性导致的，而是功能元件在向外界提供接口的时候，隐藏了自己内部的复杂的实现细节，只提供给外界一种简单的接口。这就是著名的**“高内聚，低耦合”**原理。

这个“隐藏内部实现细节，向外部提供接口”的行为也叫做**“封装”。**需要强调的是，为了实现接口的简洁性，在模块的内部需要进行一些实现。也就是说，为了实现接口的简洁性，需要在模块内部付出额外的努力。

外设的接口设计-系统桥

外设的种类是无穷无尽的，而 CPU 的指令集却是有限的。我们并不能总是因为新加入了一个外设，就专门为这个外设增加新的 CPU 指令。我们希望的是，尽管外设多种多样，但是 CPU 可以用统一的方法访问它们。为了实现这个目标，我们设计了系统桥。

系统桥是连接 CPU 和外设的功能设备，它会给 CPU 提供一种接口，使得 CPU 可以像读写普通存储器一样（即按地址读写）来读写复杂多变的外设。系统桥统一且简化了 CPU 的对外接口，CPU 不必为每种外设单独提供接口，符合高内聚，低耦合的设计思想。

在 P7 中，CPU 对于 DM、Timer 和 InterruptGenerator 的访问都是需要通过系统桥的。

CPU 的接口设计-封装成单周期 CPU

P6 的 CPU 是一个五级流水的设计。也就是说，同一时刻，可能会运行 1~5 条指令，这取决于是否阻塞等条件。这些具体的 CPU 实现细节，软件是并不关心的。当我们编写汇编语言的时候，是不需要考虑我们的 CPU 是否会发生阻塞，是否会有转发等实现细节的。而软件之所以可以这么轻松，是因为 CPU 实现了一个封装，即**“将复杂的多级流水线 CPU 封装成了简单的单周期 CPU”。**在计算机系统中，将 CPU 封装成单周期是理解 P7 任务的关键。

任务清单

P7 与之前的 project 相比，涉及的内容较多，所以在实现的时候很容易手忙脚乱，这里列出完成 P7 需要的事宜：

任务	解释
计时器	课程组提供实现代码，只需要结合代码和文档理解应用即可。
系统桥	为 CPU 提供统一的访问外设的接口，需要按规格自行实现。
协处理器 CP0	设置 CPU 的异常处理功能，反馈 CPU 的异常信息，需要按规格自行实现。
内部异常检测与流水	CPU 需要具有可以检测内部指令执行错误的能力。

任务	解释
外部中断响应	CPU 需要具有初步响应外部中断信号的能力。
异常处理指令	在异常处理程序中，会有一些特殊的指令需要实现。
单周期 CPU 的封装	让 CPU 从外部看上去是一个单周期 CPU。
异常处理程序	利用 MARS 编写简单的异常处理程序用于测试。

最后的架构图如图所示：

外设的实现

计时器

在 P7 这个简单的 MIPS 微系统中，计时器是一种外部设备，其主要功能就是**根据设定的时间来定时产生中断信号**，是我们系统的中断来源之一。

在今年的教程中，我们向同学们提供已实现的计时器 Verilog 源代码。 timer 内部需要定义多个程序员可见寄存器，如 **CTRL**、**PRESET** 等，也需要定义若干用于完成功能的内部寄存器（程序员不可见），详情请参考设计文档：CO 定时器设计规范-1.0.0.4.pdf。

中断发生器

这是课程组抽象简化现实中外设后得到的一种外设，会在不确定的时刻产生一个中断信号（就好像电脑并不知道谁会在什么时候敲击键盘一样），并持续置高。直到微系统做出响应，才变回低位。

对中断发生器的响应是通过系统桥来实现的，通过 `store` 类指令访问地址 `0x7F20`，就可以达到响应中断的目的。

中断发生器的实现并不需要同学们来完成，不同的中断发生器（中断信号产生的规则不一样）都是在测试的 tb 上实现的。同学们只需要确保自己的 P7 微系统，具有以下两个能力，就可以满足这个方面的测试：

- 微系统可以通过外部端口接受外部中断信号（在计时器部分已经实现了）。
- 微系统可以通过访问地址 `0x7F20` 的 `store` 类指令，改变对应的微系统输出信号（`m_int_addr`，`m_int_byteen`），即系统桥实现正确。

系统桥

怎样使外设与 CPU 进行沟通呢？采用划分地址空间的办法后，与外设沟通和与 DM 沟通的方式类似，通过一个 CPU 视图下的内存地址，读写相应数据即可达到与外设沟通的目的。而这个所谓的内存，在外设中，实际上只是若干寄存器。系统桥传入对地址的访问请求后，我们通过系统桥内部的转换代码，将请求转变为对相应寄存器的读写操作。

下表是规定的地址空间设计，测试程序也将以此为根据编写。需要注意的是，P7 与《See MIPS Run Linux》和 PPT 中给出的 MIPS 系统地址范围是不同的，而与 MARS 相同，这主要是为了能够让你能更好的验证设计。

条目	地址或地址范围	备注
数据存储器	0x0000_0000~0x0000_2FFF	
指令存储器	0x0000_3000~0x0000_6FFF	
PC 初始值	0x0000_3000	
异常处理程序入口地址	0x0000_4180	
计时器 0 寄存器地址	0x0000_7F00~0x0000_7F0B	计时器 0 的 3 个寄存器
计时器 1 寄存器地址	0x0000_7F10~0x0000_7F1B	计时器 1 的 3 个寄存器
中断发生器响应地址	0x0000_7F20~0x0000_7F23	

注意实现系统桥时，其必须作为独立 module 来实现，不能包含在 CPU 内部。关于系统桥的具体编写，请大家参考该文件 L15-支持 IO.pdf。

思考题

为何与外设通信需要 Bridge?

思考题

请阅读官方提供的定时器源代码，阐述两种中断模式的异同，并分别针对每一种模式绘制状态移图。

异常处理流的实现

CP0 的寄存器

CP0 协处理器是 P7 新引入的功能模块，我们需要用这个模块完成两个主要功能，一个是对异常进行配置，一个是记录异常的信息。CP0 有很多个寄存器用来配置或者记录，我们只需要实现其中的几个，如下所示：

寄存器	编号	功能
SR	12	配置异常的功能。
Cause	13	记录异常发生的原因和情况。
EPC	14	记录异常处理结束后需要返回的 PC。

每个寄存器都是 32 位的，我们只需要其中的几位，列表如下：

寄存器	功能域	位域	解释
SR（State Register）	IM（Interrupt Mask）	15:10	分别对应六个外部中断，相应位置 1 表示允许中断，置 0 表示禁止中断。这是一个被动的功能，只能通过 `mtc0` 这个指令修改，通过修改这个功能域，我们可以屏蔽一些中断。

寄存器	功能域	位域	解释
SR（State Register）	EXL（Exception Level）	1	任何异常发生时置位，这会强制进入核心态（也就是进入异常处理程序）并禁止中断。
SR（State Register）	IE（Interrupt Enable）	0	全局中断使能，该位置 1 表示允许中断，置 0 表示禁止中断。
Cause	BD（Branch Delay）	31	当该位置 1 的时候，EPC 指向当前指令的前一条指令（一定为跳转），否则指向当前指令。
Cause	IP（Interrupt Pending）	15:10	为 6 位待决的中断位，分别对应 6 个外部中断，相应位置 1 表示有中断，置 0 表示无中断，将会每个周期被修改一次，修改的内容来自计时器和外部中断。
Cause	ExcCode	6:2	异常编码，记录当前发生的是什么异常。
EPC	-	-	记录异常处理结束后需要返回的 PC。

当发生异常的时候，CPU 会自动将异常信息写入 CP0 的相应寄存器（如 `Cause` 和 `EPC`）。异常处理程序会访问相应寄存器，来了解异常的信息以进行异常处理。

同学们可以按规范自行设计 CP0，一个参考的 CP0 的端口声明如下：

端口	方向	位数	解释
clk	IN	1	时钟信号。
reset	IN	1	复位信号。
en	IN	1	写使能信号。
CP0Add	IN	5	寄存器地址。
CP0In	IN	32	CP0 写入数据。
CP0Out	OUT	32	CP0 读出数据。
VPC	IN	32	受害 PC。
BDIn	IN	1	是否是延迟槽指令。
ExcCodeIn	IN	5	记录异常类型。
HWInt	IN	6	输入中断信号。
EXLClr	IN	1	用来复位 EXL。
EPCOut	OUT	32	EPC 的值。
Req	OUT	1	进入处理程序请求。

异常码

在异常处理程序中，我们需要通过访问 `Cause` 寄存器的 `ExcCode` 域来获得异常的原因，在 P7 中我们需要实现的异常有这样几种（除此之外，比较常见的还有陷入，断点调试等）：

ExcCode 的编码必须遵守规范，不然在评测的时候可能会出现问题。

异常与中断码	助记符与名称	指令与指令类型	描述
0	`Int`（外部中断）	所有指令	中断请求，来源于计时器与外部中断。
4	`AdEL`（取指异常）	所有指令	PC 地址未字对齐。
PC 地址超过 `0x3000 ~ 0x6ffc`。			
`AdEL`（取数异常）	`lw`	取数地址未与 4 字节对齐。	
`lh`	取数地址未与 2 字节对齐。		
`lh`，`lb`	取 Timer 寄存器的值。		
load 型指令	计算地址时加法溢出。		
load 型指令	取数地址超出 DM、Timer0、Timer1、中断发生器的范围。		
5	`AdES`（存数异常）	`sw`	存数地址未 4 字节对齐。
`sh`	存数地址未 2 字节对齐。		
`sh`，`sb`	存 Timer 寄存器的值。		
store 型指令	计算地址加法溢出。		
store 型指令	向计时器的 Count 寄存器存值。		
store 型指令	存数地址超出 DM、Timer0、Timer1、中断发生器的范围。		
8	`Syscall`（系统调用）	`syscall`	系统调用。
10	`RI`（未知指令）	-	未知的指令码。
12	`Ov`（溢出异常）	`add`，`addi`，`sub`	算术溢出。

参考资料

CP0 设计及其相关指令的实现，以及硬软件在中断处理上的协同是 P7 中最有挑战性的部分。仅阅读教程中的简要介绍远远不够，因此课程组放出一些推荐阅读的资料，希望同学能加以研究，尝试去理解其中的思路。

推荐资料列表：

1. L13-MIPS 系统结构-V1.pdf
2. 《See MIPS Run Linux》中相关章节
3. 《计算机组成与设计：硬件/软件接口》中相关章节
4. Google / Bing 等搜索引擎
5. 讨论区 P7 答疑帖

封装成单周期 CPU

宏观 PC

我们的需求是要让我们的 CPU 从外部看上去是一个单周期的 CPU（具体的原因在前一章有阐述）。但是实际上，我们的 CPU 是一个五级流水的并行 CPU。所以本质上我们要实现一套机制，来让我们的 CPU 满足这个需求。

为了检验同学们的实现效果，我们引入宏观 PC 这一概念。宏观 PC 表示整个 CPU “宏观”运行指令所对应的PC地址。所谓“宏观”指令，表示该指令之前的所有指令序列对 CPU 的更新已完成，该指令及其之后的指令序列对 CPU 的更新未完成。具体实现上，宏观 PC 通常上是以某一个流水级 PC 作为界限，作为输出端口输出出来。这个流水级一般是你 CP0 所在的流水级。

精确异常

对于异常，我们能明确指出是哪条指令导致了异常，并称这条指令为**异常受害指令**。精确异常的特性是，在异常受害指令**前面的所有指令都执行完毕**，而**受害指令及其后续指令都像从来没有开始**（准确说是当异常处理结束后重新执行这些指令，与未发生异常时执行这些指令的效果一样）。这样的处理思路使得从使用者的角度来看， CPU 执行异常处理是顺序执行的，从而隐藏了流水线设计的细节。

清空流水线

为了达到精确异常的效果，我们需要在异常发生的时候清空流水线，以避免宏观 PC 之后的指令被执行。清空流水线一方面是要清空宏观 PC 之后的指令所在的流水线寄存器，即插入 `nop`。

在了解了这点以后，我们可以总结一下我们对于流水线寄存器的控制。流水线寄存器需要接受多种控制信号，如复位信号，阻塞信号，刷新信号，请求信号。所以有可能同时会有多个信号控制同一个寄存器，那么寄存器该展现怎样的行为呢？这是一个需要考虑的事情。

例如在 D 级处于被阻塞状态时发生 `Req`，那么 D 级流水线寄存器就应该立刻被清空，而不是保持原值；正在 `Req` 的时候发生 `reset`，那么 CPU 应该立刻复位，而不是进行异常处理。此处处理不当可能造成评测时缺少中断的情况。实现上，可按如下优先级：

信号	优先级
reset	最高，复位大于一切。
Req	次高，中断请求比内部阻塞重要。

信号	优先级
flush / stall	最低，流水线信号，外部人员看不到。

接下来我们考虑，哪些寄存器中的位段需要优先级。只有两个，一个是 PC 寄存器，原因之前论述过了；另一个是 CP0 `Cause` 寄存器的 `BD` 位。它在 flush 的时候需要保持原来的信息，因为在外部去看的话，会发现宏观 PC 是相同的，但是延迟槽标记是不同的，这显然是不正确的。如果在延迟槽指令被阻塞时产生中断，并且 `nop` 没有流水延迟槽标记，那么 `EPC` 就会被设置错误的值，无法通过评测。

清空流水线的另一个方面就是避免异常受害指令和其之后的指令产生影响（比如写寄存器，写 DM），这一点将在下一小节"确定 CP0 的位置"中讨论。

确定 CP0 的位置

CP0 需要放置在某个具体的流水级上，我们认为宏观 PC 所在的流水级就是 CP0 的流水级。为了满足宏观 PC 的性质，CP0 所处的位置不能够太靠前，比如设置在 F 级，那么异常会发生在之后的流水级，那么就检测不到这个异常了（如果宏观 PC 就是异常受害指令的话）。但是也不能太靠后，比如在 W 级，因为我们需要清除受害指令之后的指令造成的影响，但是此时 `store` 类指令已经修改了外设，清除影响较为困难。

因此，请根据所学挑选你的 CP0 在流水线上的位置，需要强调，没有标准答案或者最优答案。

流水异常码

异常信号 ExcCode 应该流水到 CP0 所在的流水级，而不能直接提交到 CP0。

这是因为 CPU 实际上是并行的，直接提交到 CP0 可能会导致后面的指令发生异常的时间比前面指令发生异常的时间要早。例如 `sw` 后接 `j` 指令。如果这两个都是异常指令，那么 `j` 在 D 级产生异常，`sw` 在 M 级产生异常（假设这个异常是超范围了）时不流水，就将先处理 `j` 异常，显然不符合我们的要求，因为 `sw` 异常被忽略了（`sw` 继续往后流，前面的流水级开始流异常处理程序，等异常返回之后，就会直接到跳转目标指令了，`sw` 的异常没有得到处理）。我们将异常信号流水以后，就可以先处理 `sw` 异常，然后运行到 `j`，再处理 `j` 异常。

写入 EPC

发生异常的一个重要行为是将中断指令的 PC 写入 EPC，就像函数跳转之前，要将返回地址写入 `$ra`。更严谨的说，对于异常情况只要考虑异常指令是不是延迟槽指令，如果是延迟槽指令，那么存的是异常指令的 PC - 4，如果不是，那么就存异常指令的 PC。

这样造成的结果就是，返回的时候将重新执行异常指令（如果异常处理程序不对 EPC 进行修改的话）。这里的 PC 指的都是宏观 PC。

思考题

倘若中断信号流入的时候，在检测宏观 PC 的一级如果是一条空泡（你的 CPU 该级所有信息均为空）指令，此时会发生什么问题？在此例基础上请思考：在 P7 中，清空流水线产生的空泡指令应该保留原指令的哪些信息？

思考题

为什么 `jalr` 指令的两个寄存器不能相同，例如 `jalr $31, $31`？

异常处理程序

eret 没有延迟槽

`eret` 承担了跳转功能，但是 `eret` 是没有延迟槽的。也就是说测试数据中可能出现 `eret` 指令后紧跟另一条非 `nop` 指令的情况。你的设计应该保证 `eret` 的后续指令不被执行。

异常处理程序的结构

异常处理程序是由软件实现的，我们只需要提供接口而无需自己实现。同时，了解异常处理程序是十分有必要的。

异常和中断处理流程可以概括成如下步骤（需要强调的是，这些步骤只是为了让同学们更好的理解处理程序的结构，我们在实际测评中并不保证下述的步骤都执行，也不保证不包含在下述步骤里的结构不出现）：

- Step 1：构造异常处理环境，保存现场。
- Step 2：读取 `Cause` 和 `EPC` 寄存器，判断错误类型。
- Step 3：根据异常类型和其他属性执行对应处理。
- Step 4：恢复现场。
- Step 5：使用 `eret` 指令从异常处理返回。

下面列出了一个简要的发生算数溢出时的程序，同学们可以结合源码进行参考和理解。

```
# 程序首先从这里运行
.text
    # 只允许外部中断
    ori $t0, $0, 0x1001
    mtc0 $t0, $12

    # 算术溢出
    lui $t0, 0x7fff
    lui $t1, 0x7fff
    add $t2, $t0, $t1

end:
    beq $0, $0, end
    nop

.ktext 0x4180
_entry:
    # 保存上下文
    j _save_context
    nop

_main_handler:
    # 取出 ExcCode
    mfc0 $k0, $13
    ori $k1, $0, 0x7c
    and $k0, $k0, $k1

    # 如果是中断，直接恢复上下文
    beq $k0, $0, _restore_context
```

```
    nop

    # 将 EPC + 4，即处理异常的方法就是跳过当前指令
    mfc0 $k0, $14
    addu $k0, $k0, 4
    mtc0 $k0, $14
    j _restore_context
    nop

_exception_return:
    eret

_save_context:
    ori $k0, $0, 0x1000     # 在栈上找一块空间保存现场
    addiu $k0, $k0, -256
    sw $sp, 116($k0)        # 最先保存栈指针
    move $sp, $k0

    # 依次保存通用寄存器（注意要跳过 $sp）、HI 和 LO
    sw $1, 4($sp)
    sw $2, 8($sp)
    # ......
    sw $31, 124($sp)
    mfhi $k0
    mflo $k1
    sw $k0, 128($sp)
    sw $k1, 132($sp)

    j _main_handler
    nop

_restore_context:
    # 依次恢复通用寄存器（注意要跳过 $sp）、 HI 和 LO
    lw $1, 4($sp)
    lw $2, 8($sp)
    # ......
    lw $31, 124($sp)
    lw $k0, 128($sp)
    lw $k1, 132($sp)
    mthi $k0
    mtlo $k1

    # 最后恢复栈指针
    lw $sp, 116($sp)

    j _exception_return
    nop
```

利用 MARS 验证异常处理框架

尽管在 MARS 中，我们只能针对内部异常进行模拟，无法模拟外部中断。但由我们对内部异常与外部中断的了解可以知道，两者的处理是类似的。因此我们可以在 MARS 中先验证中断/异常处理的框架是否正确（我们可以构造一条产生异常的指令，如溢出，再观察 MARS 能否进入 Exception Handler），至于我们如何处理这个错误，则是次要问题。

P7 提交要求

整体要求

- MIPS 处理器须为流水线设计，MIPS 微系统须支持中断和异常。
- 除本文明确的规范和补充声明外，MIPS 微系统设计以《See MIPS Run Linux》（下文简称《SMRL》）作为标准。《SMRL》的标准与 MARS 的行为存在一定差异，在测试时不以 MARS 为准。
- P7 较前几个 Project 为同学预留了更多需自主设计的内容，最终 P7 的实现因人而异。只要满足所给出的设计约束、行为规范和 MIPS 基本设计规范，任何设计都被认为是正确的。
- **此章节主要包含实现细节与评测要求，一些基本概念或定义请结合前面的教程理解。**

顶层模块接口

- MIPS 微系统接口（请顶层模块严格满足该要求）：

```
module mips(
    input clk,                    // 时钟信号
    input reset,                  // 同步复位信号
    input interrupt,              // 外部中断信号
    output [31:0] macroscopic_pc, // 宏观 PC

    output [31:0] i_inst_addr,    // IM 读取地址（取指 PC）
    input  [31:0] i_inst_rdata,   // IM 读取数据

    output [31:0] m_data_addr,    // DM 读写地址
    input  [31:0] m_data_rdata,   // DM 读取数据
    output [31:0] m_data_wdata,   // DM 待写入数据
    output [3 :0] m_data_byteen,  // DM 字节使能信号

    output [31:0] m_int_addr,     // 中断发生器待写入地址
    output [3 :0] m_int_byteen,   // 中断发生器字节使能信号

    output [31:0] m_inst_addr,    // M 级 PC

    output w_grf_we,              // GRF 写使能信号
    output [4 :0] w_grf_addr,     // GRF 待写入寄存器编号
    output [31:0] w_grf_wdata,    // GRF 待写入数据

    output [31:0] w_inst_addr     // W 级 PC
);
```

- 相较于 P6 的顶层模块新增了以下 4 个接口：
  - `interrupt`：外部中断信号。由中断发生器产生，每次中断信号会持续到处理器响应该信号。请注意，处理该中断信号的方式应和处理 Timer 产生的中断不完全相同，具体见前面的教程。
  - `macroscopic_pc[31:0]`：详细概念见宏观 PC。我们保证评测过程中宏观 PC 仅用于定位指令，作为产生外部中断信号的条件。
  - `m_int_addr[31:0]`：中断发生器待写入地址。当该信号命中中断发生器响应地址，且字节使能信号有效时，视为响应外部中断。

- `m_int_byteen[3:0]`：中断发生器字节使能信号，当该信号任意一位置位时视为有效。

硬件约束

- 顶层模块中应该至少包含 CPU、Bridge、Timer0、Timer1 四个功能部件。
- 地址空间：见系统桥。
- 主要部件：
  - CPU：在 P6 基础上进行增量开发，增加 CP0 协处理器，支持异常和中断等。
  - CP0：见CP0约束。
  - Bridge：须作为独立的 module，不包括在 CPU 中；访问外设均须通过系统桥。
  - IM：容量为 **16KiB**（**4096** × 32bit）。
  - DM：容量为 **12KiB**（**3072** × 32bit）。
  - Timer：定时器官方源代码已经给出，无需自行设计实现。
  - 中断发生器：
    - 中断信号依据宏观 PC 产生，依据相应的待写入地址和字节使能信号关闭，具体实现参考下发的 tb。
    - 由于其内部并没有真正的存储单元，我们规定读出的数据始终保持 0，且写入时除了响应中断外不会产生其他影响。

CP0 约束

- 协处理器位置：不作明确要求，自行设计。
- 输出要求：写入时无需 display。
- 为了支持异常和中断，必须实现的寄存器包括：**SR、CAUSE、EPC。**
- 寄存器规范：
  - CP0 寄存器的初始值均为 0，未实现位始终保持 0。
  - 当进入中断或异常状态时，均需要将 EXL 置为 1，用以屏蔽中断信号（注意《SMRL》中并没有指定进入中断时 EXL 的值）；当退出中断或异常状态时，也均需要将 EXL 置为 0，取消屏蔽中断信号。
  - Cause 寄存器的 IP 域每周期写入 HWint 对应位的值。
  - 当进入中断或异常状态时，需要将受害指令的 PC 写入 EPC。

指令约束

- 处理器应支持如下指令集：

```
nop, add, sub, and, or, slt, sltu, lui
addi, andi, ori
lb, lh, lw, sb, sh, sw
mult, multu, div, divu, mfhi, mflo, mthi, mtlo
beq, bne, jal, jr,
mfc0, mtc0, eret, syscall
```

- 在 P6 基础上新增了 `mfc0, mtc0, eret, syscall` 四条新指令。
- `eret` 具有跳转的功能但是没有延迟槽，你的设计应该保证 `eret` 的后续指令不被执行。

- `syscall` 指令行为与 MARS 不同，无需实现特定的输入输出功能，只需直接产生异常并进入内核态。

中断异常约束

- 异常入口：《SMRL》的表 5.1 中定义了 MIPS 的异常入口，但考虑到简化设计以及与 MARS 保持一致，我们只支持 0x4180 一个入口地址，所有异常与中断都将从这里进入。
- 嵌套中断异常：本实验不要求支持中断异常嵌套的情况。
- 优先级：中断优先级高于异常优先级，即当有异常提交至 CP0 寄存器时，若有中断发生，则硬件应先响应中断，并重新执行受害指令及其后续指令；若没有中断发生，则处理异常。
- 精确异常：
  - 除下面的情况外，对所有中断异常的处理都应遵循精确异常的处理规则。
  - 在进入中断或异常状态时，如果受害指令及其后续指令

    已经改变

    了 MDU 的状态，则无需恢复。假设 CP0 在 M 级，MDU 在 E 级，考虑以下情况：
    - `mult` 在 E 级启动了乘法运算，流水到 M 级时产生了中断，此时无需停止乘法计算，其它乘除法指令同理。
    - `mthi` 在 E 级修改了 HI 寄存器，流水到 M 级时产生了中断，此时无需恢复 HI 寄存器的值，`mtlo` 同理。
    - `mult` 在 E 级，受害指令在 M 级，此时还未改变 MDU 状态，不应开始乘法计算，其它乘除法指令同理。
    - `mthi` 在 E 级，受害指令在 M 级，此时还未改变 MDU 状态，不应修改 HI 寄存器的值，`mtlo` 同理。
- 中断规范：
  - Timer0 输出的中断信号接入 HWInt[0] (最低中断位)，Timer1 输出的中断信号接入 HWInt[1]，来自中断发生器的中断信号接入 HWInt[2]。
  - 规定中断产生时的受害指令为宏观 PC 对应的指令，此时应将宏观 PC 写入 EPC。
- MIPS 微系统需要支持的异常：

ExcCode	助记符	描述
0	Int	中断。
4	AdEL	取数或取指时地址错误。
5	AdES	存数时地址错误。
8	Syscall	系统调用。
10	RI	不认识的（或者非法的）指令码。
12	Ov	自陷形式的整数算术指令（例如 `add`）导致的溢出。

- 补充说明：
  - 分支跳转指令无论跳转与否，延迟槽指令为受害指令时 `BD` 均需要置位。
  - 发生取指异常或 `RI` 异常后视为 `nop` 直至提交到 CP0。

- 跳转到不对齐的地址时，受害指令是 PC 值不正确的指令（即需要向 EPC 写入不对齐的地址）。
- 对于未知指令的判断仅需考虑 opcode（和 R 型指令的 funct），且仅需判断是否出现在 P7 要求的指令集中，同时保证未知指令的测试用例中 opcode 和 funct 码的组合一定没有在 MARS 的基本指令集中出现。

官方测试说明

- 为便于进行测试，我们允许从 0x417C 直接前进到 0x4180，此种情况下 CPU 行为与 P6 一致，不应有中断响应等其他行为。
- 测试数据规范：
  - 测试时不会出现跳转到未加载指令的位置的情况。
  - `eret` 只会出现在中断处理程序中，后可能紧跟另一条非 `nop` 的指令。
  - 测试程序保证不会写入 Cause，但可能写入 SR 和 EPC。
  - 测试程序只会通过指令 `sb $0, 0x7f20($0)` 访问中断发生器（响应中断），且只会在中断处理程序中访问。
  - 中断处理程序会对寄存器和内存进行读写来验证 CPU 的正确性。
  - 中断处理程序执行过程中保证不出现异常，且不会产生中断。
- 官方 tb 示例：
  - 外设不给予中断时，使用的 tb 为：下载链接。
  - 评测机通过检测同学们的宏观 PC 给予中断信号并对中断进行测试，例如此下载链接中的 tb 会在处理器的宏观 PC 第一次到达 0x3010 时给予 CPU 一个中断信号。

官方 Mars

- 课程组修改了 Mars，增加了输出运行信息等功能，支持课程 P7 要求的异常和定时器中断，供同学们本地测试，下载链接。

思考题汇总

思考题

1、请查阅相关资料，说明鼠标和键盘的输入信号是如何被 CPU 知晓的？

2、请思考为什么我们的 CPU 处理中断异常必须是已经指定好的地址？如果你的 CPU 支持用户自定义入口地址，即处理中断异常的程序由用户提供，其还能提供我们所希望的功能吗？如果可以，请说明这样可能会出现什么问题？否则举例说明。（假设用户提供的中断处理程序合法）

3、为何与外设通信需要 Bridge？

4、请阅读官方提供的定时器源代码，阐述两种中断模式的异同，并分别针对每一种模式绘制状态移图。

5、倘若中断信号流入的时候，在检测宏观 PC 的一级如果是一条空泡（你的 CPU 该级所有信息均为空）指令，此时会发生什么问题？在此例基础上请思考：在 P7 中，清空流水线产生的空泡指令应该保留原指令的哪些信息？

6、为什么 `jalr` 指令为什么不能写成 `jalr $31, $31`？

思考题

1、[P7 选做] 请详细描述你的测试方案及测试数据构造策略。