



---

# MIPS CPU FPGA 路线指南

---

V0.0.0.1



2018-3-28

辣鸡 INTEL 小组  
ACM1501

## 声明

---

- 本文档仅是 FPGA 路线的一个不完备的参考，请结合课程设计指南以及本文档中提供的参考文献阅读。
- 本文档基于 Windows 10 Version 1709 系统，Vivado 2017.4 编写，不保证在其他系统版本和 Vivado 版本上的适用性。
- 不建议对自己代码能力以及独自检索信息能力没有自信的同学选择 FPGA 路线。
- 不建议无法顺利阅读理解英文报错信息以及文档的同学选择 FPGA 路线。
- 请本着严肃认真的态度选择 FPGA 路线，这虽然是实验，可不是闹着玩的。
- 由于时间所限，难免有所错漏，望海涵。
- 欢迎有能力的人参与修订该参考手册。
  - 添加仿真源代码的编写指导
  - 添加多级中断流水上板任务中，使用中断向量表实现中断入口程序地址获取的方法指导
  - 翻译链接所给的文档
  - 修正错误以及不清晰的地方
  - 添加其它有用的指导内容

## Vivado 介绍

### Product Overview & Resources

<https://www.xilinx.com/products/design-tools/vivado.html#overview>

### Getting Started

[Vivado Design Suite User Guide – Getting Started](#)

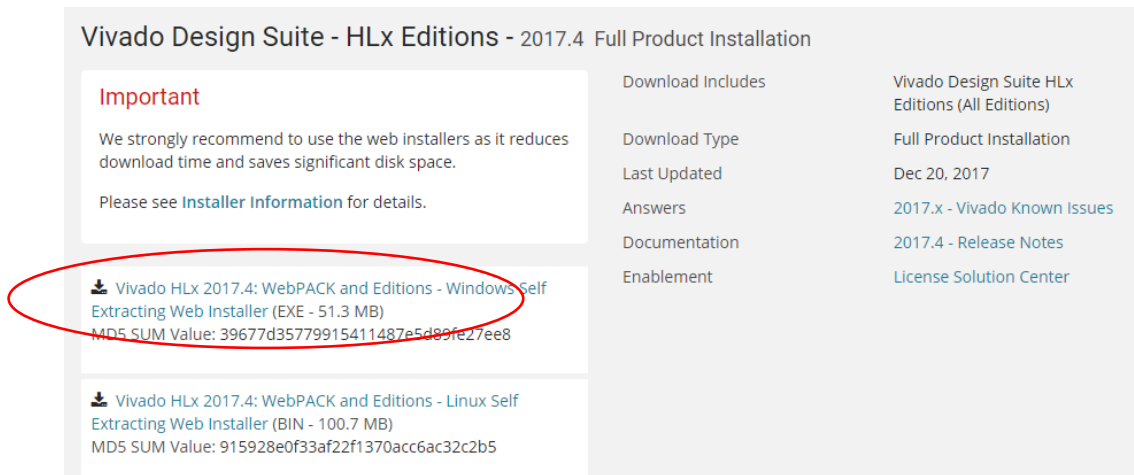
[Vivado Design Suite User Guide – Design Flows Overview](#)

或

从 Verilog 选修课的相关材料中了解 vivado 的基本使用方法。

### 最新版本 Vivado 下载与安装

下载地址：<https://www.xilinx.com/support/download.html>



Vivado Design Suite - HLx Editions - 2017.4 Full Product Installation

Download Includes	Vivado Design Suite HLx Editions (All Editions)
Download Type	Full Product Installation
Last Updated	Dec 20, 2017
Answers	2017.x - Vivado Known Issues
Documentation	2017.4 - Release Notes
Enablement	License Solution Center

**Important**

We strongly recommend to use the web installers as it reduces download time and saves significant disk space.

Please see [Installer Information](#) for details.

↓ Vivado HLx 2017.4: WebPACK and Editions - Windows Self Extracting Web Installer (EXE - 51.3 MB)  
MD5 SUM Value: 39677d35779915411487e5d89fe27ee8

↓ Vivado HLx 2017.4: WebPACK and Editions - Linux Self Extracting Web Installer (BIN - 100.7 MB)  
MD5 SUM Value: 915928e0f33af22f1370acc6ac32c2b5

（上图 of 2018 年 3 月 28 日的截图）

找到 Vivado Design Suite – HLx Editions，选择 [Vivado HLx 2017.4: WebPACK and Editions - Windows Self Extracting Web Installer](#)（上图红圈所示）并下载。

注意：由于 Xilinx 的设置，需要注册账户并登陆后方可下载。

下载完毕后打开安装文件，可以选择立即下载并安装，也可下载完整镜像准备之后安装。在选择安装版本时，建议选择 Vivado HL WebPACK（免费，且支持功能已足够使用）。

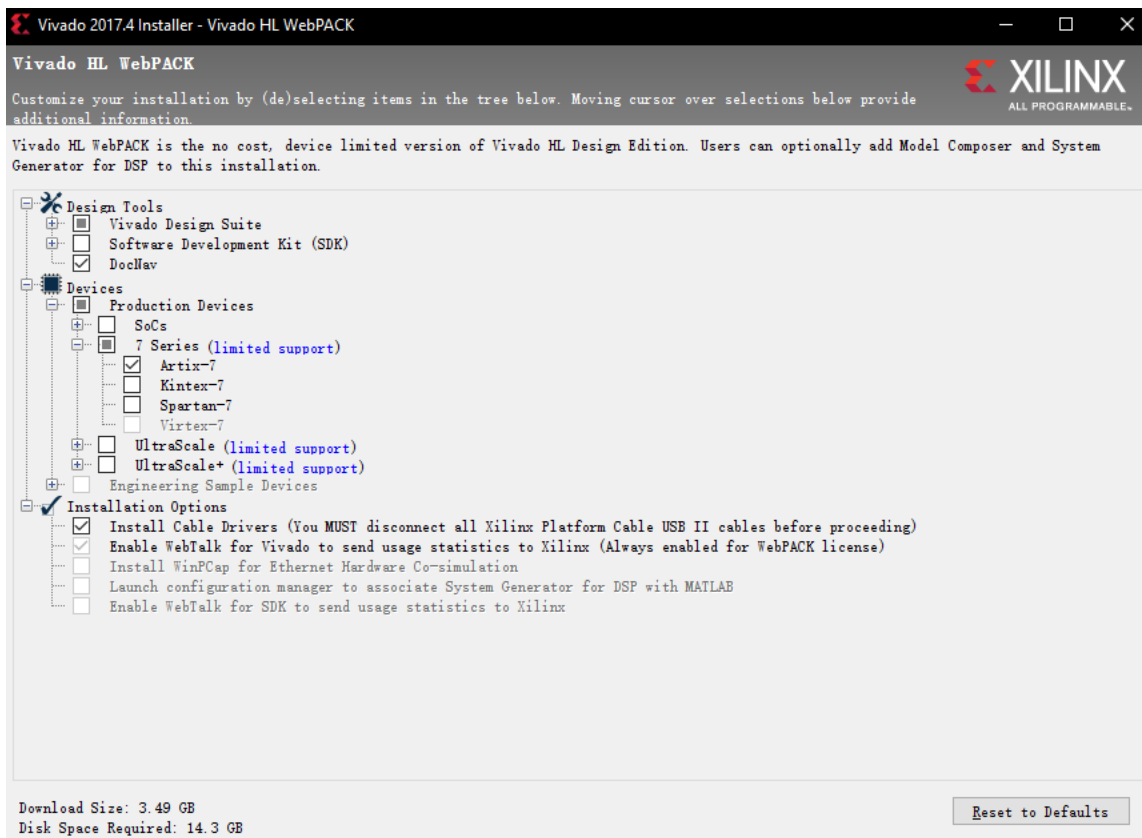
在选择安装的组件时，按照如下图的方式配置即可。Devices 由于学校提供的开发板所使用的 FPGA 属于 Artix-7 系列所以只需要选这一项就足够，需要磁盘空间如图约 14.3G。

另外，请不要安装在中文路径下，同时尽可能不要在使用中文用户名的用户环境下使用。否则可能出现闪退。

如果碰到闪退问题，首先检查用户名是否为中文，如果是请创建一个英文用户并移动到新用户下使用 vivado。

（按照之前的经验，有时可以在中文用户情况下正常使用 Vivado 一段时间，但在某个时间点后突然开始不断闪退，迁移到英文用户下可能可以解决）

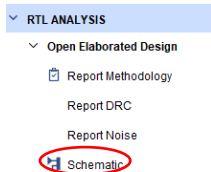
Windows 10 下创建新的本地用户的方法参见：<https://support.microsoft.com/zh-cn/help/4026923/windows-create-a-local-user-or-administrator-account-in-windows-10>



（2017.4 的安装界面）

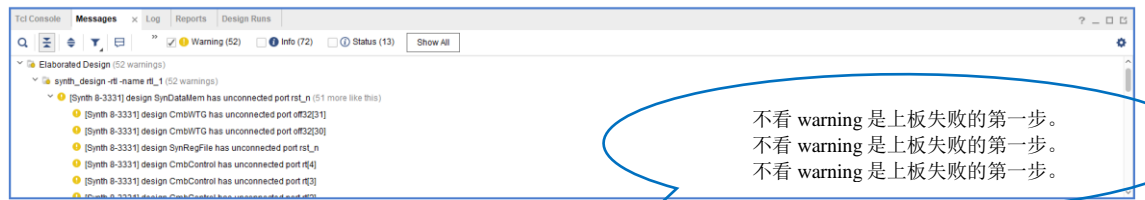
## 使用 RTL 分析检查可能存在的错误

参考链接：[https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2017\\_4/ug895-vivado-system-level-design-entry.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug895-vivado-system-level-design-entry.pdf)



如左图所示，点击 Flow Navigator 内的 RTL ANALYSIS 下的 Schematic，就可以打开电路原理图。

但在查看让人头皮发麻的电路图连线之前，请先打开在 Vivado 应用下端的 Message 窗口。



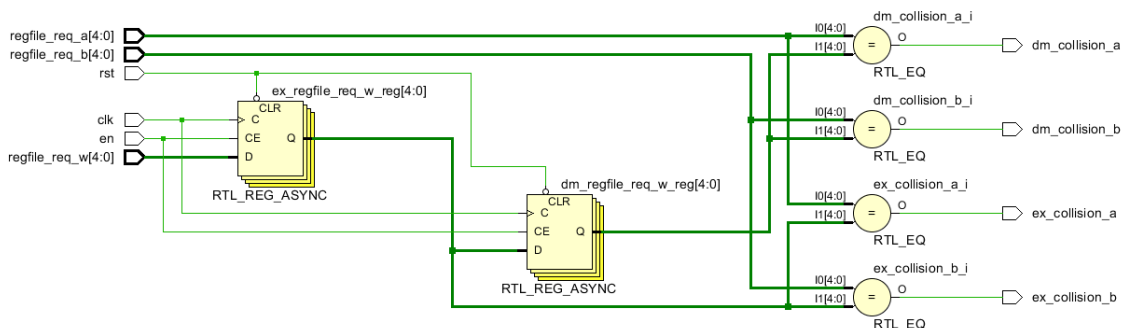
在这里，你可以看到编译你的代码时产生的 warning。

由于硬件设计语言的特殊性，并不是所有的 Verilog 代码都可以被正确地转换为电路元件（具体见下一章“代码注意事项”），部分不能被正确转换的代码会引发编译时的 warning，故仔细检查 warning 能够解决很多代码潜在的问题。

需要注意的是，Vivado 对 warning 的提示是并不是很明显，需要使用者自己留心去看。

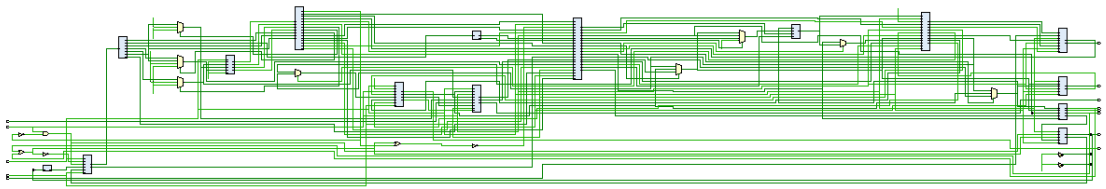
关于 warning 的细节与解决方案，请上网搜索 warning 的编号。本文档末附录列出了部分较为常见的 warning。

之后，点开原理图，可以在此检查各个部分的连线是否正常。

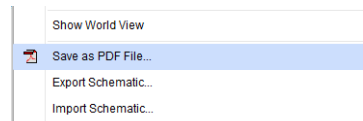


一般情况下，需要留意的是左上角是否有未连接的输入，此外，各个寄存器的时钟，异步清零以及数据输入是否正确。

可以考虑截图发给在 logisim 平台上艰苦奋斗的同学作为参考（顺便嘲讽），比如这种：

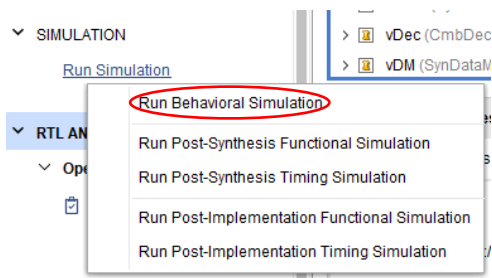


当然，右键直接作为 PDF 保存也是可以的。



## 仿真波形图的使用技巧

参考链接：[https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2017\\_4/ug900-vivado-logic-simulation.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug900-vivado-logic-simulation.pdf)



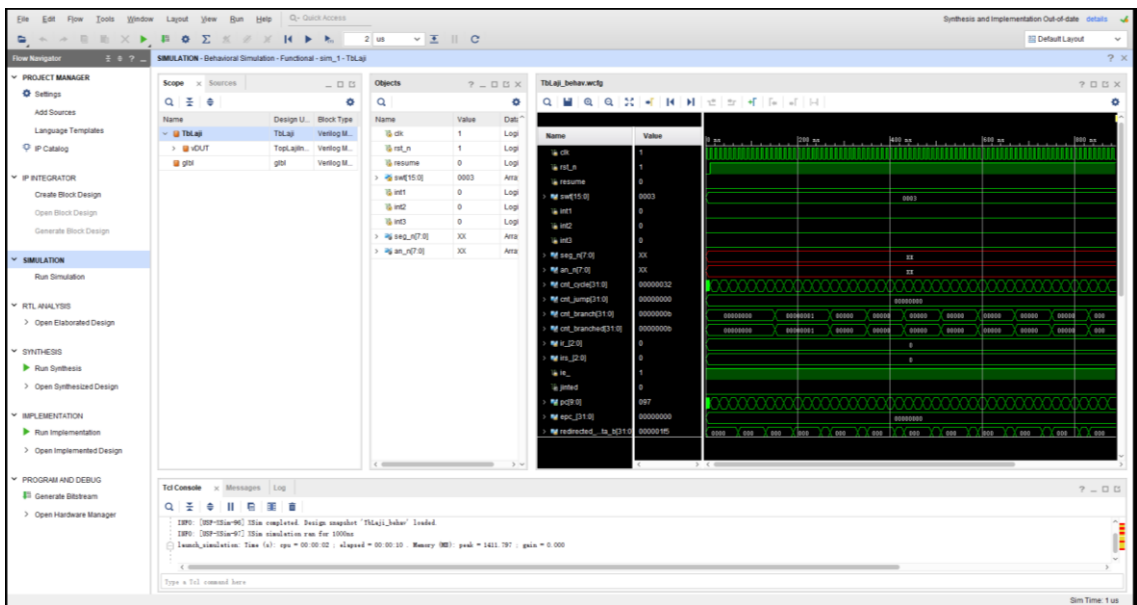
当你确认并处理了 warning 之后，就该进行行为仿真了。图中红圈即为行为仿真的选项。之后还有综合后仿真和实现后仿真，总的来说，越靠后的仿真越接近上板后的实际情况。但是考虑到普通电脑综合与实现往往需要数分钟之久，故最好先解决行为仿真中的全部问题之后再进之后的仿真。

但在进行仿真之前，我们需要编写仿真源代码。

> Simulation Sources (11)

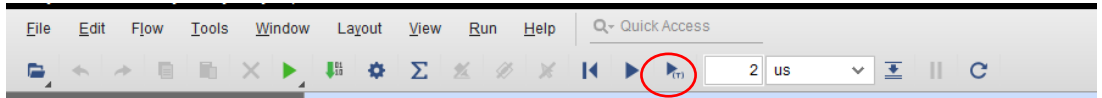
仿真源代码的编写本指南（v0.0.0.1）暂不涉及，具体见参考链接或 Verilog 语言课的课件。

运行仿真之后，界面如下：



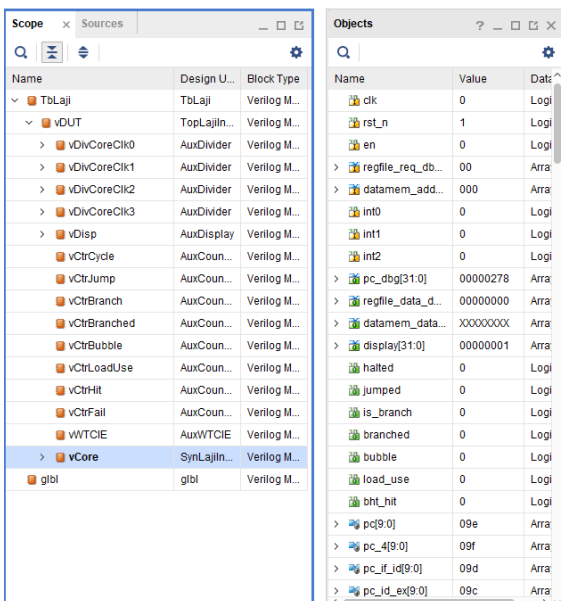
下面给出几个注意点/使用技巧：

1. 下图红圈圈出的按钮用于控制 vivado 继续仿真 2us，可通过紧跟的文本框设定继续仿真的



的时间长度。

2. 左侧的 Scope 对应你编写的代码里的各个模块，选中一个模块之后，Objects 窗口会将其中的变量全部列出。

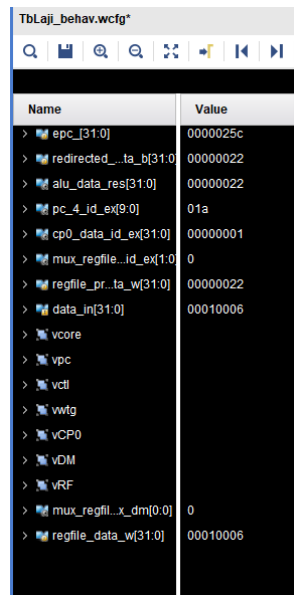



所以想要看模块内部变量的值不需要特意给模块加端口连出来。

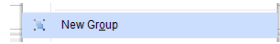
Object 和 Scope 都可以拖动到右侧仿真波形窗口中，这会让对应变量的波形在波形窗口中出现。

拖动 Scope 到波形窗口会将其所有的 Objects 都移动到添加到波形窗口中。

3. 新添加到仿真波形窗口中的变量可能暂时没有波形，此时继续仿真可以看到当前仿真到的时刻之后的波形。



也可以使用  按钮使仿真回到 0s，再使用继续仿真的按钮来查看该变量之前的波形。



4. 在波形窗口的 Name 中鼠标右击的菜单栏中添加新的 Group，可用于将变量分组。  
(一开始直接创建好 group 然后把左边的 scope 全拖进去，之后就方便多了)
5. 波形配置文件可以保存。
6. 当波形图中的某个变量被选中时，键盘的左、右方向键分别可将游标移动到被选中变量当前游标位置前一次、后一次发生变化的位置。



## 代码注意事项——如何编写可正确综合的代码

### 为什么我不能使用 Verilog 语言的所有特性？

Verilog 语言是作为硬件仿真用语言诞生的，有大量语法特性是针对仿真设计的，与 FPGA 硬件上能够支持的操作并不能完全对应。例如，时延语句不可能在硬件上实现。只有遵守一定规范编写的代码才能够被综合成为正确的硬件设计。关于 Vivado 在综合时所能够支持的语言特性及其限制，请查阅 [Vivado User Guide 901 Synthesis](#) (链接指向 Vivado 2017.4 版本的相应用户指南)。下面简单说明一些课程设计过程中很可能需要注意的要点。

### 组合与时序

使用行为建模的方式编写 Verilog 代码时，组合逻辑和时序逻辑都可以使用 `always` 块实现。使用 `always` 块实现时序逻辑时，触发条件应当使用信号的上升沿或下降沿，而组合逻辑的触发列表应包含所有用于条件判断的变量和赋值语句右侧的变量，或直接使用 `always @ (*)` 将所有可能的触发条件包含在内。在编写组合逻辑时，应注意以下问题：

- 触发列表必须包含所有可能触发电路输出或内部状态发生改变的变量，否则部分变量可能被锁存。建议在组合逻辑块中不要显式地列出触发列表。
- 在存在条件判断语句时，分支语句中被赋值的变量在所有的分支都应当进行赋值，否则将产生一个锁存。请仔细考虑此处的逻辑关系，如果需要保持原值不变考虑是否应当改写为时序逻辑。

在编写时序逻辑时，除上升沿和下降沿可以自行确定，并且异步逻辑并非必须存在，异步逻辑控制信号的个数也不固定以外，`always` 块必须采用以下结构：

```
always @ (posedge clk or posedge ACTL1 or negedge ACTL2)
begin
if(ACTL1)
    <高优先级异步逻辑>
elsif(ACTL2)
    <低优先级异步逻辑>
else
    <同步逻辑>
end
```

存在多个异步逻辑时，不同异步逻辑之间一定要存在优先级，否则可能会导致难以预料的结果。建议尽量减少异步逻辑。

## 阻塞赋值与非阻塞赋值

Verilog 语言中存在两种赋值语句，阻塞赋值与非阻塞赋值。阻塞赋值语句表示实际运行效果逻辑上相当于其后的阻塞赋值语句在该语句执行后执行；非阻塞赋值语句不能保证逻辑上语句间的执行顺序。在进行赋值时，需要注意以下问题：

- 不要混用阻塞赋值和非阻塞赋值，即使对某个变量的一部分位仅使用阻塞赋值，另一部分位只使用非阻塞赋值也不可以。混用两种赋值语句会导致 Vivado 仿真出错。
- 考虑到非阻塞赋值的语义，使用非阻塞赋值时任何一个变量不能被多次赋值，有效的赋值语句只能有一句。

错误代码范例：

```
always @ (posedge clk)
begin
a <= 1;
a <= a + 1;//语义上此时 a 的值是不定的，可能是原值+1,1 或 2。
end
```

正确代码范例：

```
always @ (posedge clk)
begin
    if (m == 4)
        a <= 1;
    else
        a <= a + 1;//尽管有 2 条给 a 赋值的语句，程序逻辑保证只有一条语句有效
end
```

- 阻塞赋值的语义与时序电路的行为并不相符，不要在时序逻辑中使用阻塞赋值。
- 建议在组合逻辑中使用阻塞赋值，与时序逻辑较好的区分，并且方便编写代码。

## case 语句

Vivado 支持对 case 语句本身的综合，只需注意以下几点：

- 不要在 case 语句的分支中使用未指定位宽的数字，否则会导致难以预料的结果。

- case 语句中靠前的分支比靠后的分支优先级更高。如果不同分支的优先级对电路的逻辑功能没有影响，而这种优先级的设定可能会影响电路的并行性，可在 case 语句前增加“(\* parallel\_case \*)”，表明希望电路能够并行，不关心优先级。

## for 循环

考虑到 for 循环可能的用途，不可能将任意 for 循环直接用硬件实现，但满足特定条件的 for 循环是可综合的。在满足以下所有条件时，Vivado 能够综合 for 循环：

- 循环界为常数
- 判断条件为<=,<,>=,>中的一个
- 步长为固定的常数
- 循环体自身可综合

正确代码示例：

```
integer i;
reg j[7:0];
for(i = 0; i < 8; i = i + 1)
j[i] = 1'b0;
```

错误代码范例 1：

```
integer i;
reg j[7:0];
for(i = 0; i < 8; i = i + j)//步长不满足条件
j[i] = 1'b0;
```

错误代码范例 2：

```
integer i , j;
for(i = 0; i < 8; i = i + j)
j = i;//循环体无法综合
```

## initial 块

initial 块用于初始化，综合后会在 FPGA 发生 global reset 时执行。但是，由于硬件本身的限制和 Vivado 的功能支持，initial 块必须满足以下条件才能够综合：

- 所有语句均用于赋初始值
- 所有初始值均为常数
- 后面的初始值不能依赖于前面在 initial 块中的赋值
- 任何一个被赋值了的变量的所有位都被赋值

请注意，`initial` 块不会在用户定义的复位信号触发时执行，不能代替异步复位逻辑。建议仅在仿真文件中和加载指令存储时使用 `initial` 块。

## 为什么行为仿真的波形与综合后仿真不一样？

行为仿真是按照 Verilog 代码的语义直接执行的，不考虑电路中的实现，而综合后仿真是按照综合后电路的实际连接进行模拟的。除了极少数情况下 Vivado 出现 bug 以外，仿真波形不一致往往是以下两种原因导致的：

- 电路延迟无法满足仿真文件中时钟的频率。

如果你的电路中部分路径延迟较大，而仿真文件中的时钟频率太快，电路无法满足要求，不能在仿真文件中的时钟频率下正常运行，可能会导致综合后仿真的结果与行为仿真不一致。这时请首先确定电路是否存在设计失误导致延迟过高(例如将非流水接口处的数据作为数据重定向的来源)，然后根据 Vivado 的延迟分析结果合理设定仿真频率。

- 编写了无法综合的代码。

另一种可能会导致行为仿真与综合后仿真波形不同的原因是编写了无法综合或无法正确仿真的代码。请修改代码，尽量减少 warning 的数量。如果此时行为仿真与综合后仿真的波形仍然不相符，建议重构代码。

## 单周期上板

---

### 重新设计 CPU

#### 重新设计的理由

为了方便达成后续的目标（流水线、中断、动态分支预测），单周期上板的代码作为第一个版本，一定要具有较好的可扩展性。Logisim 上 CPU 的实现太过于具体化。在 Logisim 上实现的时候，对于复杂的组合逻辑电路（尤其是 Control），会显式用到大量的门级电路。如果将它们直接使用门级建模的方式使用 Verilog HDL 照搬，这样的代码是十分难以进行扩展的，而且不便于调试。

这是一个总结自己 Logisim 版本 CPU 缺陷、与组员交流经验、共同得出更佳设计、减少以后走弯路可能性的过程，直接使用某一人的版本直接用 Verilog HDL 实现可能会为后续设计带来潜在的问题。

当然，如果仅以单周期成功上板为最终目标，直接使用 Verilog HDL 实现 Logisim 版本 CPU 的电路也无可厚非。对于这一类小组，请在阅读完本节后关闭该文档。

#### 确定需要编写的模块

Logisim 上关于 IO 都有现成的元件可以直接使用，但是对于 FPGA 平台，这部分都要自己实现。而且 FPGA 上不像 Logisim 可以自由调节时钟频率、可以随意查看信号。因此将所有需要编写的模块分为两大类：CPU 本身所需要的模块和其余辅助类模块。

- CPU 本身部分：原 CPU 设计本身所涉及到的模块（PC、Control、RF 等）。
- 其余部分：Project 顶层模块、辅助模块（计数器、分频器、数码管显示驱动等）。

#### 确定接口和分工

为了方便最后的整合、系统测试，应先确定各模块的原型（名称、输入端口、输出端口等），可以单独写成一个仅包含模块定义的文件，用于代码编写阶段的参考。

根据各组员的代码能力、调试能力，将各模块的编写任务分配给各组员。同时各模块编写者也应负责该模块的调试测试工作。

#### 开始编写代码

1. 按照事先决定的分工，对各模块进行编写、测试。

通过仿真，对整个系统进行调试（这个过程可能会大量修改现有的源代码）。

2. 上板测试。

3. 整理代码，统一代码风格，删除无用代码。
4. 再次上板测试，无误后即可请求验收。

### 一些提示

- 对于信号位宽，建议使用 `parameter` 或者宏来指定，不建议直接写具体的位宽。因为在后续任务中可能会修改一些模块控制信号的位宽。
- 对于 MUX，建议使用 `case` 语句实现。
- 对于 MUX 的选择信号，建议使用 `parameter` 或宏来指定，可以有效避免出错，而且方便后续任务中增加 MUX 选项。
- 尽量在正式开始编写代码前确定好各模块的端口，避免在开始编写后出现变动，这样会对整个小组的代码编写进度带来一定影响。
- 在确定接口时，一定要确定各组员都明白每个模块的作用，避免因理解出现偏差导致的 BUG。必要的时候可以使用共享文档或者其他方式写出各种约定。
- 善用 SCM/VCS（源代码管理、版本控制系统），方便代码管理，方便与组员共享代码。
- 善用较轻量的 Verilog 综合工具（例如 Icarus Verilog）进行语法检查（能节省一些时间，因为 Vivado 综合一次太花時間了）。

## 流水线（理想→气泡→重定向）

---

### 理想流水线

#### 确定各个模块所处的流水阶段

- 通常情况下，除访存指令之外，一切需要使用到来自寄存器堆的值的指令的实际执行/计算都放在 EX 段进行。

#### 确定每个阶段需要传递给下一个阶段的数据/控制信号

- 通常情况下，ID/EX 将会是最复杂的。
- 考虑到之后可能的需求，推荐将 PC 与 PC+4 从 IF 段一直传递到 WB 段。

#### Syscall 指令的处理

- 处理的方式不唯一，只要最终（即在气泡/重定向流水线中）保证下述特征即可：
  - Syscall 的显示指令能显示出正确的，最新的寄存器值。
  - Syscall 的停机指令之前执行的指令的写回操作在进入停机状态之前完成。
  - Syscall 的停机指令之后跟随的指令不会改变顶层模块的计数，此处特指 Jump/Branch 指令。

#### 开始编写流水线

- 按照流水线的划分，将原本单周期内所有会被流水模块隔开的线全部断开，并单独声明每个流水阶段内的变量。
  - 例如，alu\_op 是由控制器发出，到达 ALU 模块的 op 输入口的控制信号，由于控制器在 ID 段而 ALU 在 EX 段，它们被流水模块隔开。因此声明 alu\_op\_ex 变量，并将之替代 alu\_op 连接到 ALU 模块的 op 输入口。
  - 当然，某些线是可能是反向跨越流水模块的，比如跳转模块控制 PC 加载跳转地址的信号线。需要辨别并单独处理。
- 使用 RTL 图检查是否已经正确将各个部分分开。
- 编写各个流水模块。
  - 流水模块实际上只需要包含每一对输入输出对应的寄存器即可。
    - 当异步清零信号到来时，清空所有寄存器。
    - 当时钟信号上升沿到来时，若使能信号有效，用输入端的值更新每个寄存器。

- 在之后的气泡流水中，需要进一步添加阻塞信号和同步清零信号。
  - 可以考虑使用苟桂霖提供的自动生成工具（脚本），见附录。
- 将它们连入 CPU。
- 运行行为仿真并检查。

## 气泡流水线

### 对各模块进行一些修改

- 修改会直接修改 CPU 状态的模块，保证在气泡到来时该模块不会错误地修改 CPU 状态。
  - 为 PC 模块添加阻塞逻辑用于支持气泡插入。
  - 为流水模块添加阻塞和同步清空信号用于支持气泡插入以及跳转逻辑。
    - 阻塞电平信号有效时，寄存器内的值维持上个时钟周期的值不变。
    - 同步电平清空信号有效时，寄存器内的值在时钟到来后清空。
    - 异步清零，使能，同步清零，阻塞信号之间的优先级是？
- 结合插入气泡与跳转时需要做的工作回答。

### 如何检测数据冲突？

- 需要在 ID 段特意使用两个锁存器储存前两条指令要写入的寄存器编号吗？  
回想随流水一直传递到 WB 段的信号有哪些后再回答。
- 考虑到重定向需要，建议将以下四种冲突的信号都单独生成出来，再将它们的或作为插入气泡的信号。
  - 当前要使用的 a 寄存器为正在 EX 段的指令要写入的寄存器
  - 当前要使用的 a 寄存器为正在 DM 段的指令要写入的寄存器
  - 当前要使用的 b 寄存器为正在 EX 段的指令要写入的寄存器
  - 当前要使用的 b 寄存器为正在 DM 段的指令要写入的寄存器
- 如何检测 load-use 冲突？（见课设指导手册重定向流水部分）

### 插入气泡时，应当怎么做？

- 当需要插入气泡时，CPU 应保证 ID 段及以前的指令被暂停执行，EX 段执行的下一条指令为空操作。
- （思考题）如果不使用同步清零而使用异步清零，会有问题吗？
- （思考题）如果不使用阻塞信号而使用使能信号，会有问题吗？



控制冲突应当怎么处理？结合课程设计指南找到正确的处理方法。

使用 **benchmark** 测试自己的代码

- 如果气泡数量多于基准值，但 **load-use** 数量正确，可以直接开始编写重定向流水。

## 重定向流水线

增加重定向的数据通路

- 注意重定向一定是从流水模块的出口拉到 **ID/EX** 段流水的 **regA**，**regB** 出口（或 **ID/EX** 段流水的 **regA**，**reg** 入口）。**EX** 段的任何模块，以及 **EX-DM** 段之间的流水，都只使用重定向后的 **regA** 和 **regB** 数据。这样可以极大地简化重定向逻辑，避免为 **EX** 段内各个模块单独做重定向。
- 若使用上述重定向策略，除了 **load-store** 的特殊情况外，只需要从 **EX-DM** 段之间的流水的出口，和 **DM-WB** 段之间的流水的出口，将要写入寄存器的数据重定向回来即可。
- 如果出现 **load** 后紧跟一个有数据冲突的 **store** 指令的情况，本应从 **WB** 段重定向刚从内存中取出的数据到 **DM** 段来写入。但是为了简化重定向逻辑，可以将之判定为 **load-use**。

向控制器中加入重定向的 **MUX** 的控制信号

向流水中加入新控制信号的接口

跑通 **benchmark**

## 分支预测

### 为什么要进行分支预测

在流水线上，如果遇到了新的指令，在 IF 段是无法判断其类型的。只有当 ID 段将指令解析出来后才能判断出它是否为跳转或分支指令。而对于分支指令，为了提高流水线效率（特别是加入了重定向后），一般选择在 EX 甚至是 MEM 段计算出其是否应该跳转，并选择其相应的跳转地址。

默认情况下，我们在流水线上会不断取出下一条指令解析，也就是  $newPC = PC + 4$ ，除非跳转逻辑判断出我们应该在此处跳转。

由于我们前面已经完成了重定向逻辑，现在流水线中气泡的主要来源是分支指令“清空流水”操作。这可以在 Vivado 中通过观察流水线的每个阶段的 PC 值发现。

### 外围电路简要分析

为了解决这个问题，我们通过“动态分支预测”的方法，减少流水线清空的频率。注意到 j 系指令也应该加入 BHT 中。为了简化逻辑，我们可以把 j 系指令看做是永远跳转的 branch 指令，与 branch 系列指令统一处理。

我们需要修改的重点部分，在于 newPC 的策略，也就是将  $newPC = PC + 4$  修改为  $newPC = F(PC)$ 。这里的函数 F 就是我们所需要实现的策略。而所谓的策略修改，实际上也就是设计出一个 BHT 模块后，把原先的 +4 加法器替换掉即可。当然，这个 BHT 还需要连接配套的更新逻辑电路，见后文详解。

### BHT 模块设计思路

#### 设计思路与术语

分支预测表可以看做两个部分，“猜测逻辑”与“更新逻辑”。

“猜测逻辑”是一个组合逻辑电路，输入是当前指令的 PC，根据 BHT 表的内容进行一系列组合逻辑处理后，输出下一条指令的预测地址（这个地址可能是  $PC + 4$ ，也可能是 j 系指令和 branch 指令所对应的跳转地址  $remote\_PC$ ）。而第二个部分“更新逻辑”是一个时序逻辑电路，输入是 PC 和  $remote\_PC$  以及跳转是否成功，当然还有 en 信号指明当前是否应该更新；它在正式处理跳转时更新 BHT 历史记录。这两个部分共享 BHT 数据，但是除了数据共享外最好分开处理，以降低耦合性。

接下来，我们需要讨论 BHT 中所需要记录的信息。本质上，BHT 是一块缓存，它至少需要 Valid 位、索引、对应数据，还可能有替换策略所需要的额外信息。其中，Valid 位是为了判断该处缓存是否有效，避免未初始化的缓存对程序产生影响；索引-数据，也就是 key-value 键值对。在 BHT 中，索引显然是 PC 值，而数据是跳转目标地址(remote\_PC)和历史信息。

## 猜测逻辑

猜测逻辑会根据当前的 PC 值，给出下一条被真正执行的指令的最有可能的 PC 值。在“猜测逻辑”不涉及缓存的更新，因此我们无需关心替换策略额外信息。

我们只需要多路并行匹配，查看 8 个有效索引中是否有匹配项。如果没有，那么直接输出 PC+4 即可。如果命中缓存，那么有两种策略：其一，是根据匹配结果，使用类似于编码器的结构，得到命中缓存的编号，再用这个编号选择出对应的数据；其二，是将匹配结果扩展成“掩码”的形式，和数据进行与运算后，不匹配数据变为 0，有效数据（如果有）不变，再把它们全部或起来。查找到数据后，根据其中的历史信息输出相应的预测地址即可。

## 更新逻辑

在“更新逻辑”中，我们需要记录并更新各种数据和“策略额外信息”。实现 LRU 有多种方式，每种方式所需要的额外信息也各不相同。可以选择在每条缓存上都维护一个计数器，访问到自己时清空，访问其他缓存时加一，需要淘汰时淘汰掉最大的；也可以维护缓存的序关系，比如用 0-7 记录缓存最后访问时间的先后顺序排名，更新时修改排名，需要淘汰时淘汰掉 7 所代表的缓存；甚至可以使用矩阵维护偏序关系表，利用行列更新简化逻辑。这些方法各有优劣，可以自行设计。

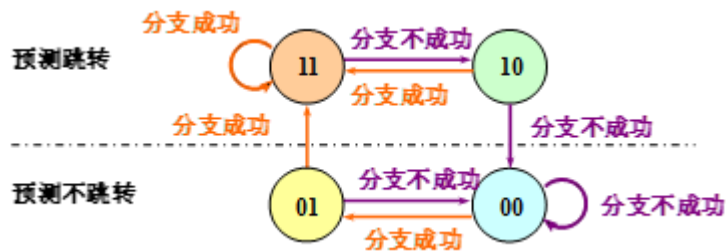
但是，不论哪种方法，大多都需要注意以下事项：

1. 重视初始化。除了 Valid 位的初值外，最好一开始就设定一个可用的缓存替换顺序，避免一次更新需要淘汰掉多个缓存。同时，切记初始化的时机应在 RST 信号的下降沿，而不是 initial 中。
2. 正确处理两部分的关系。“更新逻辑”和“猜测逻辑”类似，也需要知道更新时，当前 PC 是否在缓存中已经出现。这里的代码和“猜测逻辑”中的代码极为相似，但是切记同一时刻，“猜测逻辑”的 PC 索引和“更新逻辑”的 PC 索引来源于不同的流水线阶段，不可混淆。
3. 解耦代码。复杂的逻辑最好写在辅助的组合逻辑模块或函数中（比如 always@(\*)中），这样即使写错，也只是得到的答案有问题，不会有副作用。而更新数据时用的 always@(posedge clk)，一旦写错很容易污染大量数据，最好只

剩下 enable（用在 if 上）和 data（写在  $\leq$  右端），对应 Logisim 中 D 触发器（等价与 verilog 中的寄存器）的相应位置。这样写可以在极大程度上避免玄学错误，也使得逻辑更为清晰，方便调试。

4. 对于某些算法，注意合理安排“策略额外信息”所需要的空间，避免发生溢出错误。

最后我们讨论一下历史信息的更新策略。所谓的历史信息本质上是两位状态，对应如下的状态机：



再根据当前分支是否成功（注意不是“是否成功预测”！），按照状态图获得新的历史信息状态，写回到缓存中即可。注意对于新加进来的缓存的初值，不要直接使用被淘汰缓存的当前值来计算，而应该给定新值。

## 多级中断流水上板

### 难点

多级流水中断是一个难点，但只要捋清思路，在重定向或者动态分支预测的代码基础上进行修改实现起来也不算特别困难。下面是多级流水中断的某种实现方式的要点整理和说明，代码改写大致分为下面四个步骤，实现思路的细节和需要注意的地方会在后文详细给出。本节的目的在于提供一种可能且非最优的思路，实际上多级中断流水的实现方式有很多，希望读者能在阅读完本节后设计出自己的实现。

1. 添加模块实现 CP0 寄存器组，本课设中只需要实现简化的（对自己完成本任务而言最方便的）CP0 寄存器组即可，不必遵从 MIPS 的规范；
2. 根据 CP0 寄存器组给出的信号以及流水中当前的具体情况，编写逻辑判断是否应进入中断隐指令阶段，若进入中断隐指令阶段则进一步产生关中断、保存断点至 EPC，以及要求将对应中断处理程序的地址送入 PC 的信号；在控制模块中增加指令 MFC、MTC 以及 ERET 的实现，MFC 以及 MTC 的指令格式设计和电路实现（不必遵从 MIPS 的指令格式，可在其之上修改来方便自己的 MFC 和 MTC 电路实现）和 CP0 寄存器组的实现方式有关；
3. 在多级流水线的 CPU 中，嵌套中断会产生许多冲突和需要处理的事件，这主要是因为关中断的指令从进入流水到被实际执行（写 CP0 的 IE 寄存器）之间可能有多个时钟周期的延迟，这段时间是一个危险期，因为中断处理程序/隐指令硬件假设这段时间已经不会再被中断。为解决这一问题，需要设计机制（比如在流水中传递信号）使得高优先级中断不能在低优先级中断隐指令阶段以及保护现场阶段时立刻得到响应和处理（应推迟到阶段结束），后文会详细介绍各种需要处理和注意的事项。在这个过程中需要修改流水接口，因为有些信号的产生需要用到流水中传来的值通过组合逻辑进行判断。
4. PC 程序计数器的处理，在中断隐指令阶段需要设置为中断处理程序的入口地址，在中断返回时需要设置为上一次被中断的地址。

## CP0 寄存器组

### 内部寄存器

中断产生的识别放在在 ID 段，中断处理放在 EX 段，本指南将会基于这两个前提条件进行编写，但中断处理也可以选择放在 WB 段。

**IR(3 bit)**: 中断寄存器，每个位代表一个中断，为 1 表示该位对应的中断还没被响应或者中断还未返回；

**IRS(3 bit)**: 标识 CPU 正在处理的中断，若 CPU 正在处理某个中断，则该中断对应的位为 1；

**IE(1 bit)**: 中断使能寄存器，通过将该寄存器清零的方式关中断；

**EPC(32 bit)**: 保存断点，保存中断发生时第一条没有执行完毕的指令地址。

以上四个必要的寄存器，还可以根据个人的具体设计编写其他的寄存器实现相应的逻辑，以下介绍一下可以借鉴的操作。

**INT(1 bit)**: 输出端口，在有比当前正在处理的中断优先级高的中断来临(或者当前没有正在处理的中断)并且中断使能寄存器为 1 的情况下，输出为 1。

可以根据 IR 的记录来设置，这样的话，在优先级低的中断保护完现场开中断的时候，可以及时产生 INT 信号打断当前中断服务程序。如果该输出为 1，并且流水中没有写 CP0 寄存器组的操作或者正在执行中断隐指令的情况下，表明可以处理该中断，否则不进行中断处理。

**INT\_NUM(2 bit)**: 中断号，由于 IRS 会在 WB 段才更新，但是产生 INT 信号并且判断该中断需要处理的时候，就需要判断即将处理的中断类别。

### 内部寄存器的设置与清零

如果在 CP0 寄存器组模块部分正确的处理了多级中断的处理顺序和相应的输出信号，那么后面的工作会比较轻松。CP0 寄存器的写使能简单的分为两个：IE 写使能，EPC 写使能。

1. 在中断产生时就需要设置 IR；
2. 在 EX 段开始处理中断，直到 WB 段开始写 CP0 寄存器组。将 IE 置零，以防止中断处理程序在保护现场的过程中被打断；设置 IRS；设置 EPC，保存断点。
3. 其余的写 CP0 操作都根据分析 MFC、MTC 以及 ERET 指令产生的信号来执行。

下面是写入 EPC 和 IE 中的值的数据的可能来源，实际的来源与 CP0 寄存器以及中断相关的指令格式设计相关，不一定与下列来源完全相同。

1. 写到 EPC 中的值有两个数据来源

- a) 随流水传来的 PC 的值即断点，这是在中断进入时 PC 的值。（在 ID 端识别 EX 段处理中断的前提下）请自己分析确定此时应当储存 PC 而不是 PC+4 的值。
  - b) 来自寄存器文件。
2. 写入 IE 中的值有两个数据来源，一个是中断产生开始执行中断隐指令的时候，写入 IE 的值就是一个常数 0 即关中断。还有可能是解析指令的时候某个寄存器中的值(通常为 0 或者 1)用于开中断或者关中断。

## 控制模块

处理中断的过程中最重要的是能处理各种冲突，从而使得中断能正常的进行，为了方便可以将这一部分的处理在 ID 段的控制模块中。还有对 MFC、MTC 以及 ERET 指令分析产生的信号，这些指令自然应放在控制模块处理。

### 中断隐指令阶段

上文说过在 CP0 的 INT 输出为 1 的情况下也仅仅表明有优先级高的中断来临并且 CPU 处于开中断状态，但是并不表明该中断一定能被立即处理。这是因为可能优先级低的中断正处于中断隐指令阶段，或者某个中断程序在恢复现场之前先执行关中断指令，但是该指令还在流水中未被执行等等情况。因此在 INT 输出为 1 的情况下，该中断只能在满足两个条件的情况下处理：1.CPU 没有处于中断隐指令阶段；2.流水中没有未执行的关中断的操作。一种比较简单的方法是，一旦流水中有写 CP0 寄存器组的使能控制信号，就进行插气泡操作，等待当前流水中写 CP0 的指令全部执行完毕，再判断当前中断是否能够处理。

进入中断隐指令阶段后，保证 CPU 不能执行后续的其他指令，只有写 CP0 的操作，包括关中断，保存断点，以及更新 IRS。

### 中断服务程序

写控制器的指令分析之前，首先需要写好自己的中断服务程序。中断服务程序在中断隐指令执行完毕之后执行，此时已经关中断不会响应其他任何中断，开始进行现场保护，通过压栈的方式保存可能会使用的寄存器，考虑到更高优先级的中断可能打断当前中断处理程序并修改 EPC，因此保存的寄存器也包括 EPC，完成操作后通过 MTC 指令将 IE 置为 1 即开中断；接下来执行的是中断服务程序的功能代码；执行完毕之后开始进行恢复现场的操作恢复现场后 ERET 返回，本次中断处理完毕。

### 指令分析和信号产生

MFC 和 MTC 指令的实现原则是尽可能让代码编写时比较方便，以下是一种可能的思路。



1. MFC 指令，格式为 MFC0 \$R1,\$R2(R1 表示寄存器文件中的寄存器，R2 表示 CP0 中的寄存器，可以用不同编号代表不同寄存器，比如 \$1 表示 EPC，\$2 表示 IE)，表示将 R1 中的值读出写入 R2 中。这个指令只用于将 EPC 寄存器中的值压栈。首先将 EPC 寄存器中的值读到寄存器文件中的寄存器 R1 中，然后执行 sw 指令压栈。因此只要通过 op 和 func 的值确定是 MFC 指令，那么就可以直接断定这是从 EPC 中读数据写入寄存器 R2 中。自行分析 MFC 指令需要产生哪些控制信号。
2. MTC 指令的格式和 MFC 类似，表示将 R2 中的值写入 R1 中。这个指令用于开中断或者关中断，以及在恢复现场的时候写 EPC。具体做法是将需要写入 R1 的值写入 R2 中，然后再执行 MTC 指令。
3. ERET 指令需要进行的操作是：开中断，将 EPC 中的值送入 PC 中，并且设置 IR 与 IRS。将当前返回的中断处理程序对应的中断位置为 0 表示已经处理完毕，而 IRS 显示当前继续要处理的中断或者没有中断要处理。
4. 控制模块应当分辨当前是分支指令，还是中断来临的跳转或者是中断返回时的跳转，给出相应的输出，在 EX 段处理中断的时候可以给 PC 一个正确的跳转地址。

## 与 CP0 相关的数据冲突处理

在流水 CPU 中，读写寄存器会产生数据冲突，而中断涉及到 CP0 寄存器组，所以也不可避免的会出现数据冲突，比如在需要根据 IE 判断是否为开中断时，流水上还有对 IE 写的操作。。处理思路与之前流水章节说明的思路一致，选择气泡或者重定向，但由于一般情况设计出的 CP0 寄存器组以及其在 CPU 中的运作方式不一定合理，重定向会较为困难，故可以偷懒选择插入气泡消除冲突。

## 程序计数器

中断还有一个最重要的环节是程序计数器（PC）能在中断相应和返回时跳转到正确的地址。如果在 ID 段解码发现当前执行的指令为中断返回，那么需要将 EPC 的最新值写入 PC。而如果是中断相应，那么需要将对应中断程序的入口地址写入 PC。

中断响应时中断处理程序的入口地址获取有两种方式，一种通过硬编码实现，另一种通过中断向量表实现。硬编码实现比较简单，要求先完成中断处理程序和主程序的编写，，在 MARS 中汇编后便可确定三个中断处理程序的入口地址，在中断处理时，依据中断的类型即可以确定应写入 PC 寄存器的入口地址。中断向量表实现超出本版本指南的范围，留待日后补充。



## 附录-常见 warning 列表以及对应解决方法

1. [Synth 8-992] bubble is already implicitly declared earlier [SynLajIntelKnightsLanding.v:85]



```

    en(en && 'bubble),
    .load_pc(load_pc),
    .pc_new(pc_new); Note: undeclared symbol bubble, assumed default net type wire
endmodule

wire bubble;
wire [ 32:0 ] pc;
wire [ A ] pc_new;

```

由于 vivado 会在你使用未声明变量时隐式帮你声明，故此情况下可能碰到该 warning。

注意当被隐式声明的变量的位宽实际上不是 1 时，可能导致问题。

调整声明的位置，移动到使用之前即可解决。

2. [Synth 8-4445] could not open \$readmem data file 'F:/OneDrive/azure\_projs/computer\_organization\_lab/proj/idealPipeline.hex'; please make sure the file is added to project and has read permission, ignoring [CmbInstMem.v:12]

检查.hex 文件路径是否正确，文件的格式对不对（比如有没有删除“v2.0 raw”）。

当出现这个 warning 时，你的 cpu 肯定没有正确加载要运行的汇编代码，所以仿真时发现 cpu 好像一直什么都没做的时候务必先检查有没有这个 warning。

3. [Synth 8-3848] Net prog in module/entity CmbInstMem does not have driver. [CmbInstMem.v:11] (99 more like this)

检查此模块里面的对应变量是不是没有在任何地方赋值。

4. [Synth 8-350] instance 'vDCD' of module 'SynDataCollisionDetector' requires 10 connections, but only 8 given [SynLajIntelKnightsLanding.v:145]

检查实例化该模块的上层模块是不是有一部分端口并没有指定输入/输出。

5. [Synth 8-3331] design SynDataMem has unconnected port rst\_n (8 more like this)

确定这些 port 是否没有在任何地方使用，以及是否是你故意不使用的。

6. [Synth 8-6014] Unused sequential element cnt\_reg was removed. [AuxCounter.v:16] (16 more like this)  
[Synth 8-3332] Sequential element (ext\_out\_zero\_reg[17]) is unused and will be removed from module Pipeline\_ID\_EX. (22 more like this)

出现这些 warning 的原因一般是在使用行为建模方式描述组合电路时，纯组合逻辑有关的变量被声明为 reg 类型。这些 warning 一般可以忽略。

## 附录-关于流水接口自动化生成

---

感谢本组苟桂霖同学提供的自动生成脚本，github repo 链接如下：

<https://github.com/FluorineDog/Pipelined-CPU-Generator>