lab1-instruction

Lab1：基于 stall 的五级流水线

实验目的

- 理解流水线的基本概念与思想
- 理解数据竞争、控制竞争、结构竞争的原理和解决方法
- 基于在单周期 CPU 中已经实现的模块，实现 5 级流水线框架
- 加入冲突检测模块和 stall 执行模块解决竞争问题
- 理解流水线设计在提高 CPU 的吞吐率，提升整体性能上的作用与优越性

实验环境

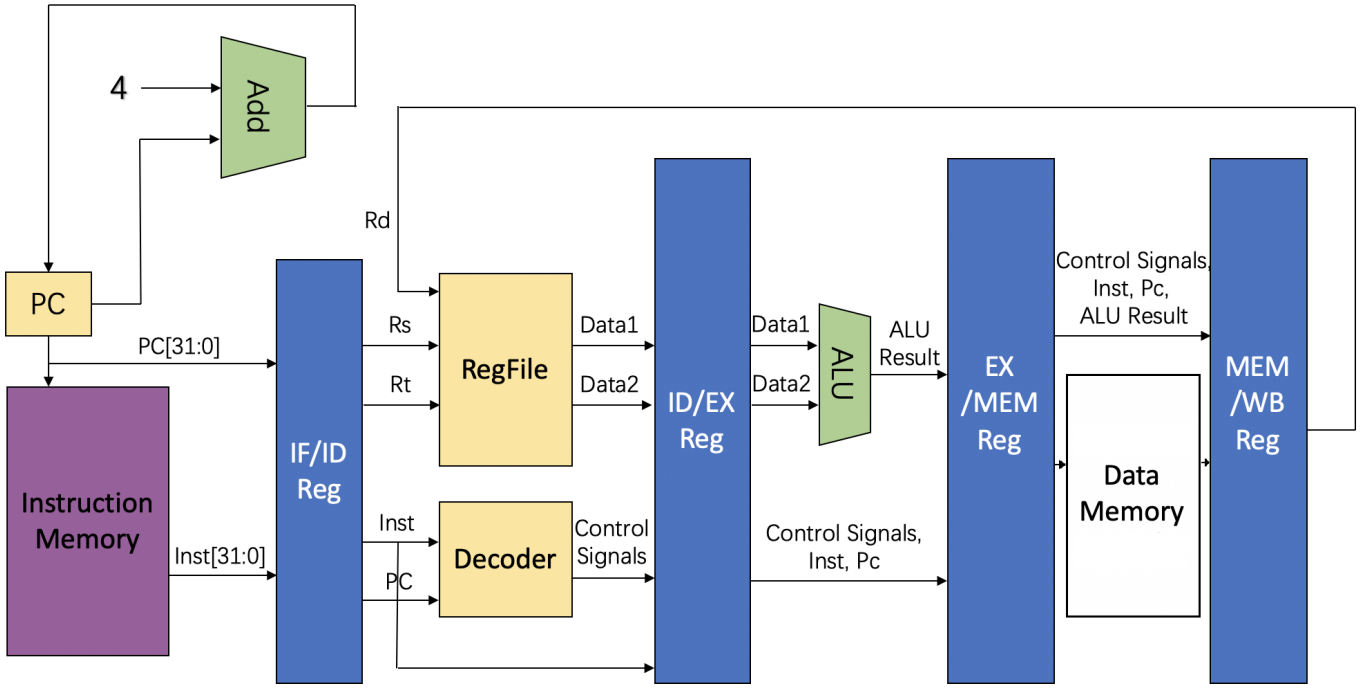
- **HDL**：Verilog
- **IDE**：Vivado
- **开发板**：Nexys A7
- 提供测试程序和测试框架

实验背景

流水线与单周期 CPU 设计对比

关于流水线的基本设计思想已经在计算机体系结构课程的理论部分中提及，这里不再赘述。
但实际落实到电路设计，与此前计算机组成中设计的单周期 CPU 相比还是有很多不同之处。

在同一时刻下，流水线的五个流水级运行的指令实际各不相同。
因此，我们一般将一条指令所需的控制信号一起存在流水段间寄存器中，随指令其他信息一起顺着流水线传递，这里的段间寄存器是指：IF/ID 寄存器、ID/EX 寄存器、EX/MEM 寄存器、MEM/WB 寄存器这四个用于流水段间传递指令信息的寄存器组。
例如 add 指令，在 ID 阶段对指令解码，得到所使用的寄存器 rd, rs1, rs2 需要等待到 WB 阶段写回，就需要由段间寄存器存储 rd, rs1, rs2 的值，一级一级地传递到 WB 阶段。



原则上每一拍过后，流水线都会向前流动一段。
第 x 拍 IF 段内取出指令，第 x+1 拍该指令流入 ID 段进行译码，例如译码得到的是 lw 指令，那么第 x+2 拍流入 EX 段内指令 lw 完成地址计算，第 x+3 拍 lw 流入 MEM 段完成访存的读取，第 x+4 拍流入 WB 阶段将访存中的值写回寄存器。
一般而言段内任务需要当拍完成，因为在下一拍时，段内的信号就会变为下一条流入指令需要的信号，在使用段间寄存器以外的寄存器时，务必要注意可能产生的时序问题。下表展示了一个简单的顺序执行的指令序列是如何在流水线的各个流水级中流淌的。

time	IF	ID	EXE	MEM	WB
cycle1	inst1	→	→	→	→
cycle2	inst2	inst1	→	→	→
cycle3	inst3	inst2	inst1	→	→
cycle4	inst4	inst3	inst2	inst1	→
cycle5	inst5	inst4	inst3	inst2	inst1
cycle6	inst6	inst5	inst4	inst3	inst2
cycle7	...	...	...	...	...

流水级划分

流水级划分的一般经验准则：

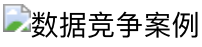
1. **按功能划分**：不同的功能最好属于不同的流水级，因为一般情况下，单个流水级功能越少，组合逻辑越简单，延迟越低，时钟频率越高；
2. **按时间划分**：在有些组件中（例如译码器），可能会出现较长连续的组合电路；它们往往会成为系统性能的瓶颈，但拆分组合电路、在中间插入阶段寄存器，并不会对电路的功能造成影响。由于流水线时钟频率取决于最长阶段，一般情况下，阶段越平均，CPU 的整体利用率越高，性能越高；
3. **按空间划分**：空间上远离的组件应该尽可能属于不同的流水级，使用寄存器隔断往返的长距离通信带来的时间开销。在 FPGA 板上，由于各类硬件资源已经固定而无法任意挪动，按空间划分显得更为重要。

根据经验准则，我们可以构造最基础的 5 段流水线设计（取指、译码、执行、访存、写回），在此基础上，我们可以根据实际布线中观察到的时间开销有针对性地进行优化。（本质上就是根据时延来划分流水级）

stall 介绍

在流水线的运行过程中，同一时刻内不同流水段运行的指令各不相同，如果说指令之间不存在关联性和依赖性，那么我们只需要在流水线中加入中间寄存器组，就可以保证流水线高效、持续地运行。

但这种情况总是理想的，在实际的运行过程中，指令间的关联性和依赖性往往无法避免，下图给出的便是一组数据冲突的案例。



可以看到 sub 指令的 x5 寄存器依赖于 add 指令 x5 寄存器的写回，所以 sub 指令必须等待 add 指令完成 WB 阶段写回再执行 ID 操作才可以得到正确结果。

静态处理的方法是编译器在 sub 和 add 之间加入两条 nop 指令，这样等 sub 执行到 ID 的时候， add 正好执行完了 WB，可以解决上述的数据竞争问题。

但是在实际的系统中，这相当于将流水线暂停的工作交给编译器去处理，但这一方面会加重编译器的负担，另一方面也不利于处理器的后续优化。

因此，我们需要在处理器中实现 stall 机制，当流水线检测到数据冲突或者控制跳转指令后通过 stall 机制处理，保证执行结果的正确性。

stall 实现分析

以数据冲突为例进行分析，常见的数据相关有 WAW/WAR/RAW，由于我们的指令都是顺序执行的，这里只考虑 RAW 一种情况即可。

当指令流入 ID 阶段时，我们对指令进行译码，根据译码的结果读取对应寄存器的值，此时如果发现待读取的寄存器发生了 RAW 冲突，就要触发 stall 机制进行处理。

我们不妨假设 ID 段要读取的寄存器设与 EX 段将要写入的寄存器发生了冲突（与 MEM, WB 段冲突的情况请同学们自行考虑）。
那么从逻辑上我们的 stall 机制应当使得 ID 段的指令被锁住， 直到寄存器的值可以被正确读取前都不应当向后流入 EX 段，否则读取的寄存器值错误，会影响指令执行的正确性。

> **Example**
>
> 例如，相邻的两条指令为 `addi x1, x0, 1; add x2, x1, x0`。
> 第二条指令执行时 x1 寄存器的值应该为 1，而在流水线中上一条指令还并没有执行完，如果不添加 stall 机制，读取到的值为第一条指令尚未执行时 x1 寄存器的值。

也就是说接下来的一个周期，该指令仍然锁住在 ID 阶段，直到与之冲突的 EX 阶段指令流入到 WB 段完成写回，使得该指令可以读取到正确的寄存器的值时，该指令才可以流入到 EX 段。

那么我们以 ID 阶段来看，向前看 IF, ID 两阶段都应当锁住当前的指令和状态，等待若干周期后继续执行，向后看 EX/MEM/WB 阶段应当继续执行段内的指令，正常的让指令流入下一阶段。

再考虑分支跳转指令的情况，当执行到分支跳转指令时，由于可能发生跳转，在 pc 置位前已经流入 IF 段的指令可能并不是我们将要执行的指令， stall 机制需要考虑流水线内指令的锁存和刷新两方面的问题，对于分支跳转指令，根据同学们实现的流水线不同，也需要 stall 不同的周期数。

BRAM 和 DRAM 的区别

光刻过FPGA的同学都知道 FPGA 上面有两种 RAM 资源，分别是 Distributed Memory 和 Block Memory。
在下文中，我们将前者简称为 DRAM，后者简称为 BRAM。
两者的写行为相同，主要区别在读行为：DRAM 是异步读，也就是说能够在单拍返回读取结果，而 BRAM 是同步读，也就是说必须要下一拍才能拿到读取结果，即 BRAM 有一拍读延时。

> 注意，DRAM 在正常语义下是指 Dynamic Random Access Memory。

DRAM 也被叫做 LUTRAM，是使用普通的 LUT 资源拼成的 RAM。
由于 DRAM 在物理上是不连续的，因此默认配置下可能会带来较大的延时。
除了直接使用 Distribute Register Generator 来显式地生成 DRAM 之外，我们平时编程使用的`reg`在绝大数情况下也会被综合成 DRAM。

```
// 7:0 和 0:511 次序是有讲究的，不可以随意改为 0:7 和 511:0
reg [7:0] mem [0:511];
always @(posedge clk) begin
    if (we) mem[addr] <= wdata;
end
assign rdata = mem[addr];
```

上面例子中`mem`的写法和我们在处理器中的寄存器组的写法是一样的，它本质上就是一个大号的寄存器数组。
因为 LUT 本身是一个简单的时序单元，所以读取 Distribute RAM 的值只需要等待一个多路选择器的时延即可。
对于只读的 memory，vivado 也可能采用 LUT 构造一个大号的查找表来充当 ROM，它的读操作也只需要等待一个多路选择器的时延。

但是凡事都是有代价的，首先 LUT 是 FPGA 上珍贵的资源，用来构造大块的 memory 容易导致 FPGA 的资源耗尽。
此外，如果大量 LUT 被 memory 占据，就会导致 core 的其他部件在布线的时候不得不去更远的地方分配 register 和 LUT，这会导致逻辑电路的布线时延大大增长，电路时序整体变差。
所以如果对读时序有严格的要求，比如寄存器组必须在单拍内返回数据，对于大块的存储，通常考虑使用 Block RAM 来改善布线时延。

Block RAM 存储单元数量数倍于 Distribute RAM，主要集中在 FPGA 的特定区域。
构造 Block RAM 同样有两种方法：

- 使用 Block Memory Generator 构造 IP 核
- 对寄存器数组使用特定的代码模板

在综合时 vivado 会根据 memory 的行为自动推导该使用什么类型的 RAM 资源来实现它，你可以对寄存器数组使用`(* ram_style = block *)`的修饰符来强制指定实现方式。
下面的代码给出了能够被 vivado 识别的行为模型，主要特征为：同步读和访问端口数量小于 2。

```
(* ram_style=block *) reg [7:0] mem [0:511]; //法二
always @(posedge clk) begin
    if (we) mem[addr] <= wdata;
    rdata <= mem[addr]; //法三
end
```

Block RAM 的读写都依赖于时钟，下图为写优先的 BRAM 的时序图，可以看到，无论是都还是写，在给出输入地址、数据之后，都要等一个时钟周期才可以得到稳定的输出：

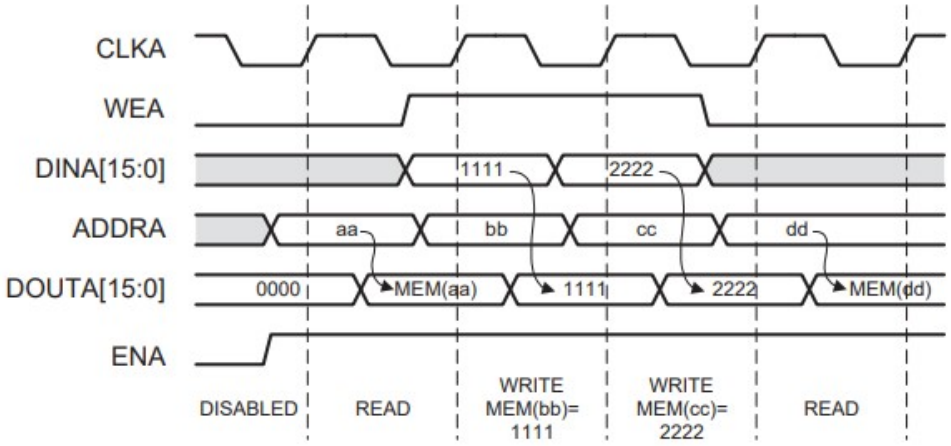
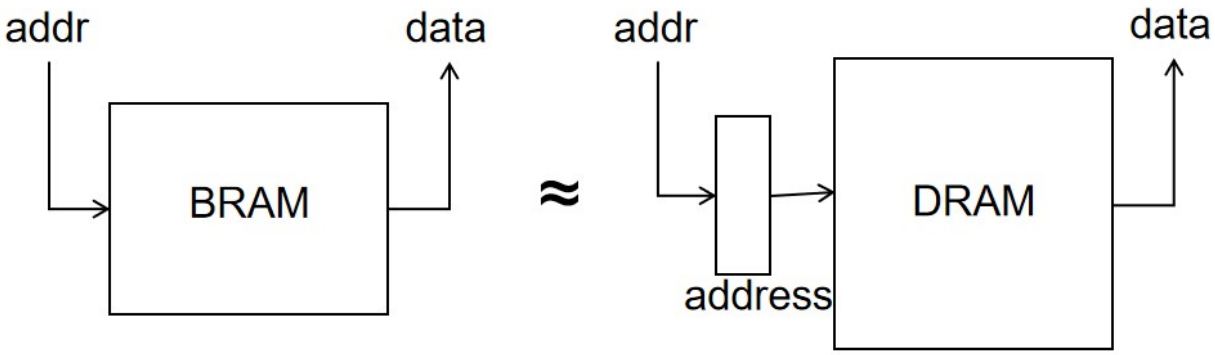


Figure 3-9: **Write First Mode Example**

vivado 的文档表明，实际得到的 BRAM 类似于如下模型：存储单元 mem 数组是使用 BRAM 存储单元实现的，在存储单元的输入位置有一组寄存器。在时钟上升沿，我们的 address 等输入数据被存入寄存器，然后寄存器的值影响 BRAM 内部的逻辑电路输出对应的数据。

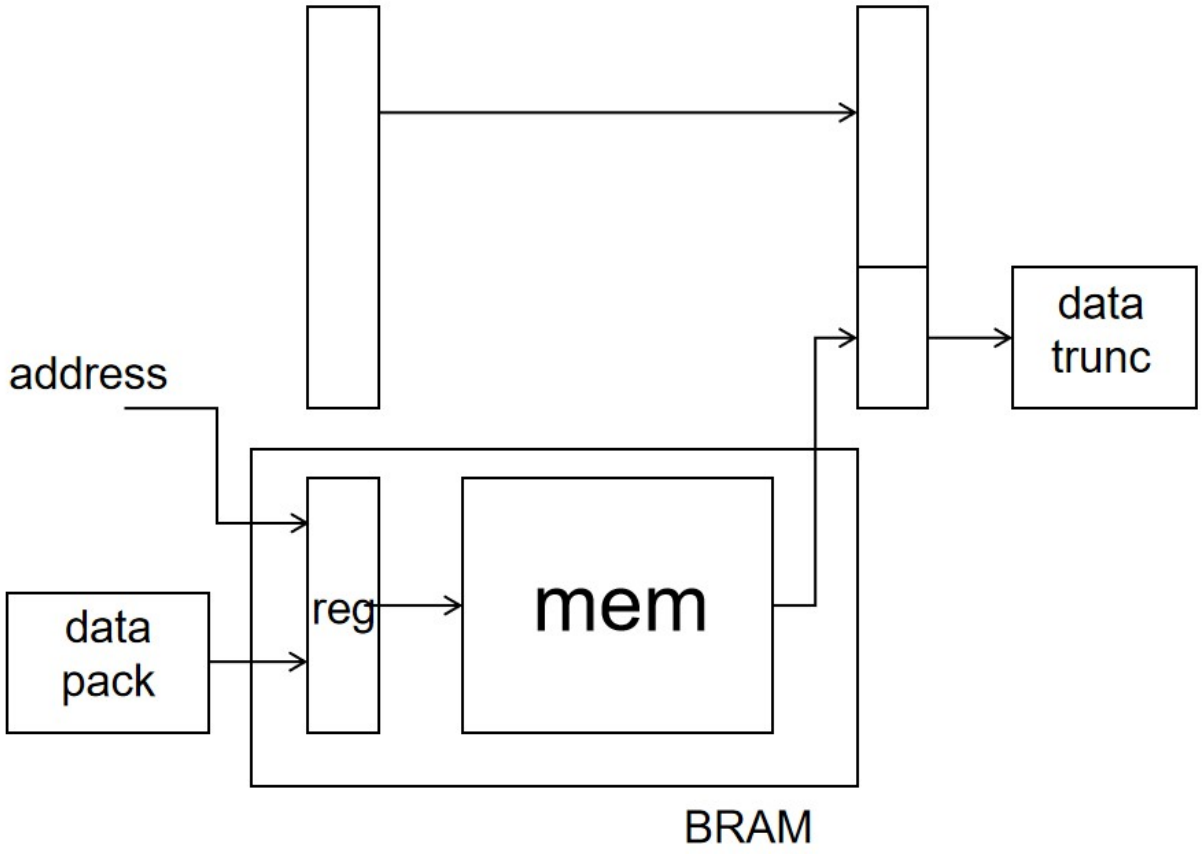


如果我们仅仅将 RAM.v 的 BRAM 当作一个整体，不在意内部细节的话，其实只需要知道它在输入 address 之后，下一个周期才可以得到输出得 data。因此如果我们希望在 MEM 阶段得到 RAM.v 的输出，只要在 EXE 阶段就送入对应的 address 即可。

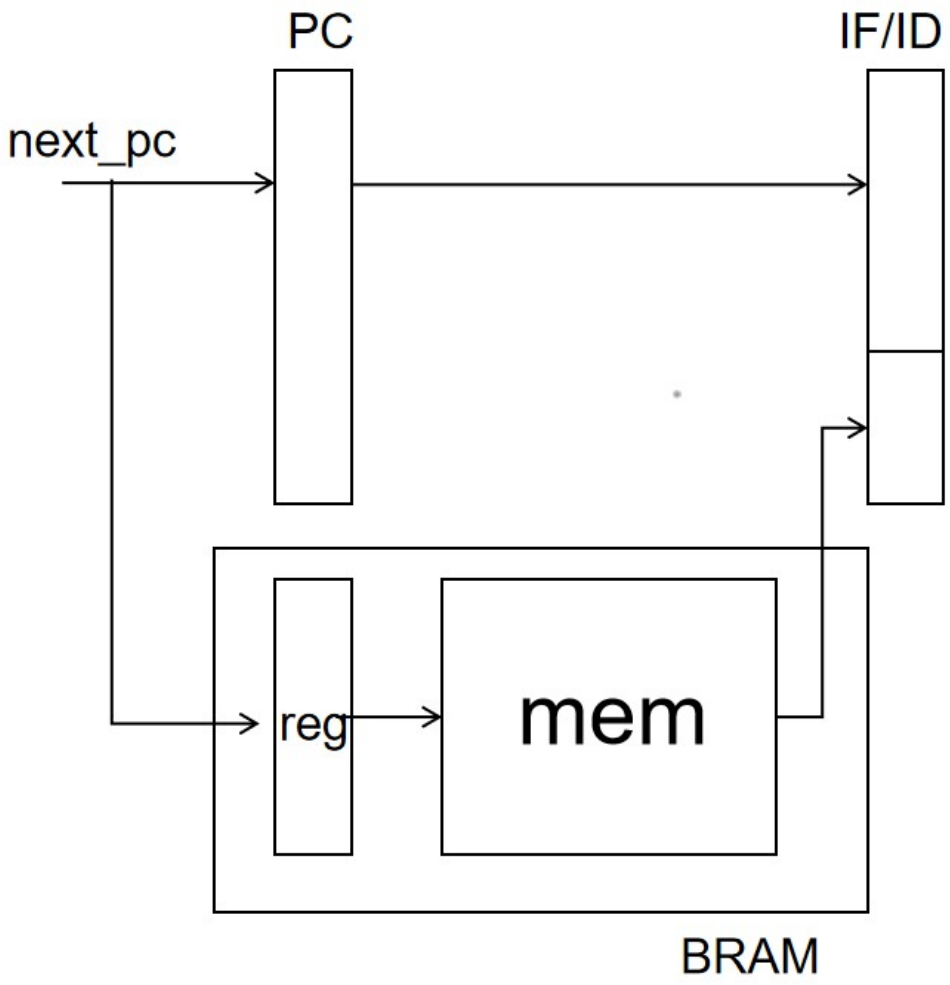
BRAM 在流水线中如何使用

我们在 SCPU 实验中使用的是 DRAM 实现的 memory，因为如果 memory 的访问本身就需要一个时钟周期的话，我们无法在一个周期执行完一条 load 指令。但是对于 pipeline 的流水线，我们则改用了 BRAM 的 memory。现在介绍一下 pipeline 的调整方法，至于琐碎的细节，交给大家自行完善。

对于 load 指令，我们的目标电路图大致如下图，相当于用 BRAM 内部的 reg 替代了 EXE/MEM 寄存器组中的 reg：



对于 IF 阶段，我们的目标电路图则如下图：



余下的细节就靠大家自己努力啦。

实验要求

在 SCPU 的基础上加入中间寄存器实现五级流水线 CPU，处理器要求支持 RV64I 指令集并加入竞争检测机制和 stall 执行机制，解决三种竞争。此外，使用 BRAM 代替原先的 DRAM。

RISC-V 相关的指令可以在官网上找到相应的编码信息和介绍。
也可以查询RISC-V 非特权指令集手册。

要求实现的指令列表如下：

- ○ lui, auipc
- ○ jal, jalr
- ○ beq, bne, blt, bge, bltu, bgeu
- ○ lb, lh, lw, lbu, lhu, lwu, ld, sb, sh, sw, sd
- ○ addiw, slliw, srliw, sraiw
- ○ addw, subw, sllw, srlw, sraw
- ○ addi, slti, sltiu, xori, ori, andi, slli, srli, srai

- ◯ add, sub, sll, slt, sltu, xor, srl, sra, or, and

实验步骤

1. 根据参考设计图或自己设计的设计图搭建完整的流水线加法机，继续使用 Nexys7 对应的外设进行实验，实现 lab1 的带 stall 的五级流水线替换 lab0 中的 CPU 模块。Memory 使用提供的 RAM.v 模块，**但请不要修改除了初始化文件以外的代码部分**；
2. 在本实验中不要求对数据通路进行专门的封装，推荐直接将 Control Unit (Decoder) 视为译码模块放在 ID 段中；
3. 进行仿真测试，以检验 CPU 基本功能；
4. 调整时钟频率，确保时钟周期长度不小于最长流水段的信号延迟；
5. 进行上板测试，以检验 CPU 设计规范，上板测试调试工具相关内容请参考 lab0。

实验建议

1. **RAM.v 模块的使用**：建议大家在完成 PipelineCPU 内部连线的时候，将 RAM.v 模块以外的所有模块封装为单独的 Core.v 模块，然后 Core.v 和 RAM.v 模块在进行连线，这样在我们后续 lab2 将 RAM.v 模块替换为更复杂的总线连接的内存模块的时候的时候，可以不需要做过多的调整。
2. **stall-flush 的接口问题**：建议每个中间寄存器 IF/ID、ID/EXE、EXE/MEM、MEM/WB 都有面向冲突处理的接口 stall 和 flush，前者负责处理停顿，后者负责处理刷新；对于发出 stall 和 flush 控制信号的模块建议对每个中间寄存器都发出独立的 stall 和 flush 信号，例如下面的接口样式：

```
RaceController(
    ....
    output wire stall_PC,
    output wire stall_IFID,
    output wire stall_IDEXE,
    output wire stall_EXEMEM,
    output wire stall_MEMWB,
    output wire flush_IFID,
    output wire flush_IDEXE,
    output wire flush_EXEMEM,
    output wire flush_MEMWB
);
```

   这些预留的接口有利于后续冲突处理的扩展，大家在 lab2 马上就可以体会到了。

3. **Verilator 测试相关**。不同于 SCPU 每个周期只运行一条指令， PipelineCPU 是 5 条指令一起运行的，但是只有 WB 阶段输出的结果才是最后指令运行完毕提交的结果，所以在进行 testbench.v 测试的时候需要把 WB 的 PC、inst、rd_id、rd_we、rd_data 输出出去。考虑到 stall 会插入无效的中间指令，所以建议每个中间寄存器加一个 valid 状态，只有指令有效的时候 valid=1 ，然后将 valid 一并发送到 cosim.v 中去，这样只有有效合法的指令才会被 cosim.v 模拟检测，而 stall 引入的无效指令不会影响 cosim.v 的检测工作；
4. **命名与接线的小技巧**：Verilog 连线是一件费时费力的机械劳动，掌握一些命名和接线的技巧有利于提高连线的效率和准确率。建议每个流水级内部的线都加一个对应流水级的后缀，比如 ID 阶段的 PC 命名为 PC_ID，EXE 阶段的 PC 命名为 PC_EXE，可以防止接线的时候名字记混。模块连线的时候建议沿着流水级的拓扑顺序进行连线，比如以 PC-IFID-{Ctrl,RegFile,ImmGen}-IDEXE-...这样的顺序开始声明模块和连线，这样大多数模块只需要在连线之前声明自己输出的线即可，而输入的线在声明上游模块输出线的时候已

经声明好了，声明的线也可以快速连接自己的输入输出，防止错连、漏连、线的重复定义和未定义。最后注意定义的线宽，不然忘记定义线宽。

思考题

1. 对于 test1（4-15 行），请计算你的 CPU 的 CPI，再用 lab0 的单周期 CPU 运行 test1，对比二者的 CPI；
2. 对于 test2（17-52 行），请计算你的 CPU 的 CPI，再用 lab0 的单周期 CPU 运行 test2，对比二者的 CPI；
3. 请你对数据冲突,控制冲突情况进行分析归纳，试着将他们分类列出；
4. 如果 EX/MEM/WB 段中不止一个段的写寄存器与 ID 段的读寄存器发生了冲突，该如何处理?
5. 如果数据冲突和控制冲突同时发生应该如何处理呢?
6. 能否使用 BRAM 来实现寄存器组?
7. 尝试分析整体设计的关键路径。