Lab10 流水线 CPU 设计实验

单周期处理器的指令执行采用串行方式，CPU 总是在执行完一条指令后才取出下条指令执行。这种串行方式没有充分利用执行部件的并行性，因而指令执行效率低。指令的执行也可以采用流水线方式，将指令执行过程分成多个功能模块，各个模块运行实现流水线化，重叠执行多条指令，以提高 CPU 执行指令的效率，从而达到高性能的计算。

本实验在 RV32I 单周期处理器的基础上进行改造，实现经典的五段流水线式 CPU 设计。通过五段流水线设计熟悉数字电路中常见的流水线设计，理解流水线处理器中存在的三类冒险及冒险的解决方案。

一、实验目的

1) 掌握流水线CPU的设计方法。
2) 掌握流水线CPU冒险处理方案。
3) 掌握流水线CPU的指令仿真验证方法。
4) 学会使用RISC-V GNU工具链和官方测试集。

二、实验环境

1. Vivado 开发环境
2. Xilinx A7-100T 实验板
3. RARS 汇编和 RISC-V 实时模拟器
4. Ripes 汇编编辑器和 RISC-V 模拟器
5. Verilator 仿真工具

三、实验原理

单周期处理器架构存在下列一些问题：

1、每条指令的所有任务均需要在一个周期内完成，较复杂的指令需要经过多个步骤和多个逻辑部件，且这些步骤无法并行。例如，在单周期 CPU 设计中采用了下降沿开始执行，上升沿读取数据存储器

的操作。在下降沿到上升沿的半个周期时间内，需要顺序完成取指、指令译码，读取寄存器及 ALU 计算地址等一系列操作。为了能够满足电路时序要求，只能通过降低主频，延长时钟周期长度来解决。

2、单周期处理器的各个部件采用紧耦合的方式实现，修改一个部件的设计将会影响许多关联的部件。这违背了复杂系统设计中需要遵循的模块化、松耦合的指导思想。同时，给替换 CPU 中的单独模块或者对单个模块进行优化带来了一定的困难。

为了解决上述问题，在设计 CPU 的时候可以采用多周期或流水线架构来实现。在流水线架构中，单条指令的操作被划分为多个相对独立的步骤。处理器中含多个流水段，每个流水段只完成指令中相对独立的一项操作。在流水线上就可以有多条指令同时执行，每个指令所处的执行阶段不同，在同一个时刻由不同的流水段进行处理。当一条指令顺序通过所有的流水段之后就可以完成该指令的执行。流水线的主要优势在于：单个流水段的操作简单，需要的执行时间较短，CPU 可以在较高的主频上运行。同时，每个时钟周期近乎完成一条指令，指令执行的吞吐量提高。

流水线处理器中单个指令需要顺序经过多个流水段，指令的执行时长不一定比单周期处理器短。但是，由于多条指令在流水线上实现了空间并行，流水线处理器可以每个周期完成一条指令的执行，且时钟周期较单周期处理器短。所以，流水线处理器的指令吞吐率较单周期处理器高。

1、流水线 CPU 的分析

流水线设计的原则是：指令流水段个数以最复杂指令所用的功能段个数为准；流水段的时序长度以最复杂的操作所用时间为准。一条指令的执行过程可被分成若干个阶段，每个阶段由相应的功能部件完成。首先要对每条指令的执行过程进行分析，以确定流水线每个功能段的功能和执行时间。通常 RISC-V 指令的执行过程可以划分为如下 5 个功能段。

取指 IF (Instruction Fetch)：主要完成从指令存储中读取指令的工作， 并同时计算下一个周期的 PC。

译码 ID (Instruction Decode)：主要完成指令译码、立即数扩展、读取寄存器堆等的操作以及生成控制信号。

执行 EX (Execution)：主要完成 ALU 运算，判断分支转移指令是否跳转等操作。

访存 MEM (Memory access)：主要完成对内存的读写操作。

写回 WB (Write Back)：主要完成将计算结果写回到寄存器中的操作。

每条指令前两个功能段都一样，后面的功能段随不同类型指令而不同，有些指令需要通过加入“空”功能段来构成 5 个功能段。在插入“空”段时，应遵循以下两个原则：①每个功能部件每条指令只能用一次（如寄存器写口不能用两次或以上）；②每个功能部件必须在相同的阶段被使用（如寄存器写口总是在第 5 阶段被使用）。

因此，R-型指令、I-型运算类指令、U-型指令和 J-型指令需在 WB 之前加一个空的 M 段，使得其 WB 段和 lw 指令的 WB 对齐，都在第 5 段；S-型指令和 B-型指令在第 4 个功能段后加一个空的 WB 段。这

样，所有指令都有 5 个功能段。因此，该处理器的指令流水线可以设计成 5 个流水段。插入“空”段后，有些指令的某些功能可能会在插入的“空”段内完成。例如，J-型指令插入 M 段后，则转移目标地址可以在 M 功能段内写入 PC，而不必等到 WB 功能段才写 PC。

指令执行经历 5 个流水段：IF、ID、EX、MEM 和 WB，每个流水段都在不同的功能部件中执行。为了保存指令执行的中间结果，流水段之间有一个流水段寄存器，例如，IF/ID 寄存器是介于 IF 段和 ID 段之间的寄存器。每个流水段寄存器用来存放从当前流水段传到后面所有流水段的信息。因为每个段间传递的信息不一样，所以各流水段寄存器的长度也不一样。五个流水段的间隔设计了四个流水段寄存器，对应简称为 IF/ID、ID/EX、EX/MEM、MEM/WB 寄存器。

流水线的时序设计相对于单周期处理器要简单，在本实验的设计中将时钟的下降沿作为周期的起点和终点。即每个阶段在时钟下降沿开始，利用流水段寄存器中的信息执行本流水段的操作，然后在下一个时钟下降沿将结果写入下一阶段的流水段寄存器，将工作交给下一个流水段。

在单周期 CPU 的主要部件中，只有 PC 寄存器、寄存器堆、指令存储器和数据存储器是时序逻辑部件，需要时钟驱动，其他部件均是组合逻辑。

在流水线处理器中，PC 寄存器在 IF 流水段中，使用时钟下降沿来更新，即每次时钟下降沿时 PC 刷新，准备取下一条指令。指令存储器包含在 IF 流水段中，采用异步读取指令的方式，即 IF 阶段在下降沿更新 PC 之后，随后读取指令，并在下一个下降沿写入 IF/ID 流水段寄存器。

寄存器堆在 ID 流水段中，寄存器的读取不受时钟沿控制，只要读端口编号改变，读取结果就立即改变。寄存器的写入采用上升沿写入的方式，写入的寄存器号、数据及写使能信号均由 WB 阶段提供。这里写入采用提前半周期的上升沿写入的主要原因是允许 WB 阶段向 ID 阶段转发数据，主要考虑数据冒险的处理。

数据存储器在 MEM 阶段中，采用上升沿读取数据，下降沿写入数据的方式。在上升沿读取数据的主要原因是要保证在 MEM 阶段执行的相关数据信号已经保持稳定。写入时机选择在下降沿主要是为了数据存储器中实现先读后写。

在流水线处理器中使用的控制信号与单周期处理器类似，在流水线中，控制信号在 ID 阶段生成，各个阶段的控制信号可以跟随指令数据一步一步流过整个流水线。PC 和各个流水段寄存器都没有列出写使能信号。这是因为每个时钟都会改变 PC 的值，每个流水段寄存器在每个时钟都会写入一次，因此，PC 和流水段寄存器写使能信号始终有效。此外，IF 段的功能每条指令功能都相同，是公共流水段，没有数据通路所需的控制信号。

需要注意的是相同的数据信号很可能在不同流水段均会要使用到，建议在信号命名时加上流水段的前缀。例如，用 rd 来表示指令的目的寄存器时，该信息有可能随着指令流过不同的流水段，而且当多条指令同时在不同阶段时，每个流水段对应的 rd 很可能不同。可以将 ID 阶段的 rd 命名为 ID_rd，EX 阶段的 rd 命名为 EX_rd，这样代码会相对更加清晰可读。

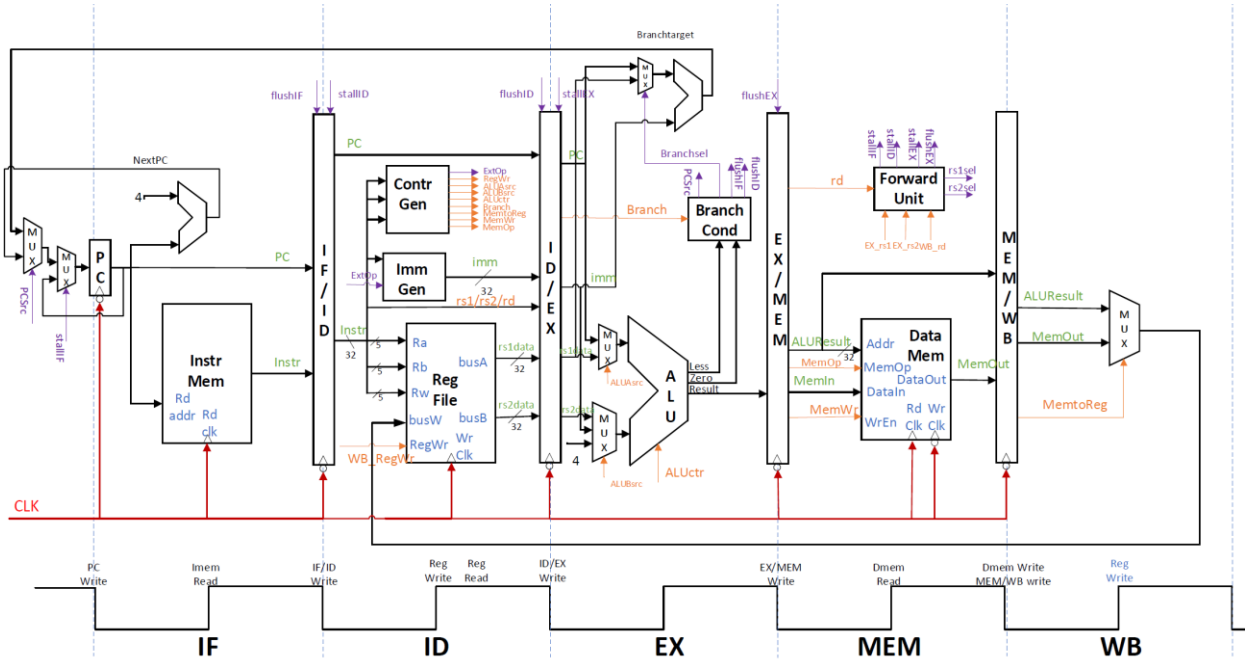

图 10.1 五段流水线原理图

2、流水线 CPU 的设计

根据对 RV32I 中 37 条目标指令的分析，可以得到相应的 5 段流水线 CPU 原理图如图 10.1 所示，需要提醒的是部分控制信号和部件设计与单周期 CPU 稍有不同，如 Branch、less 信号等。

各流水段的功能及控制信号取值设计如下。

1） 取指（IF）段

取指阶段（IF）的主要任务是确定 PC，并从指令存储器中取出 PC 地址存储的对应指令。IF 阶段将 PC 存放在 PC 寄存器中，时钟下降沿更新 PC。PC 的来源有三种：

正常顺序执行：下降沿用 PC+4 更新 PC

阻塞 IF 执行：当 stallIF 信号生效时，暂停 IF 阶段的执行，PC 保持原值不变

跳转执行：当需要进行 branch 或跳转时，用 Branchtarget 中指定的地址更新 PC，在本设计中使用 PCsrc 信号来指示是否需要跳转，为 1 时表示需要跳转。

在下降沿到来后，PC 更新到新值，新的 PC 送给指令存储器的读地址，在上升沿读取指令 Instr。

在下一个下降沿，IF 阶段执行结束后需要将 IF 的执行结果存入 IF/ID 流水段寄存器。由于本指令在 IF 阶段获取的信息只有 PC 和 32 位的指令 Instr，所以在 IF/ID 流水段寄存器中只需要保存 PC 和 Instr，共 64 位。流水段寄存器的更新需要两个额外的控制信号，一个是 flushIF—冲刷 IF 阶段执行结果，另一个是 stallID—阻塞 ID 执行阶段。这两个信号从其他流水段中直接连线过来，不需要经过段寄存器。

冲刷和阻塞

冲刷是指执行过程中取到或执行了错误的指令，需要从流水线中把相关的指令清除掉，变成空操作或者气泡（NOP 或 Bubble）。例如，在跳转执行确定正确跳转地址之前，IF 和 ID 阶段会提前顺序读取后续指令并部分执行。当确定要跳到新的地址时，这些错误的执行信息应当被清除。当 flushIF 为 1 时，可以将 IF/ID 流水段寄存器清除为全 0，这样等价于取了一条空指令 NOP，对处理器的操作不产生后续影响。

值得注意的是，这里是简化处理，将 Instr 直接清除为 32 位全零。实际 RISC-V 的空指令一般是 addi zero,zero,0，对应二进制是 0x00000013。请在实现过程中将全零指令也译码为空指令，避免出错。

阻塞是指暂停某个流水段的执行。当流水线中某个阶段暂时无法执行时，该阶段前面的所有流水段都应该停下来等待其执行完毕再继续执行。暂停流水段的执行通常可以通过保持流水段前的段寄存器不变来实现。例如，当要暂停 ID 阶段时，可以在下降沿到达时将 IF/ID 段寄存器现在结果原封不动地再写回 IF/ID 段寄存器，这样下一周期 ID 阶段还是执行之前的操作，等价于等待了一个周期。同样的，当 stallIF 信号有效时，可以通过保持 PC 寄存器不变，重新执行 IF 阶段来起到阻塞 IF 阶段的作用。

PCsrc、stallIF、flushIF 及 stallID 这些控制信号均由其他流水段提供，采用跨越流水段的方式以连线方式连接。不可在这些信号上添加额外的寄存器或者将这些信号通过段寄存器传递。

2) **指令译码（ID）段**

译码阶段 ID 流水段的功能是：对指令中的操作码 op 字段和 funct3 字段进行译码，生成相应的控制信号，同时生成运算用的操作数。这些操作数包括根据指令中的 Rs1 和 Rs2 的值到通用寄存器组中取出的相应寄存器内容，以及通过扩展器对指令中的立即数字段进行扩展后生成的完整 32 位立即数。

解析指令的方式与单周期基本相同，从指令中获取出立即数 imm、rs1、rs2、rd 等信息。控制信号的定义也可以沿用之前单周期的方式，其中 EXop 用于确定立即数扩展方式，在 ID 阶段使用。在 EX 阶段使用的控制信号是 branch、ALUAsrc、ALUBsrc 及 ALUctr。在 M 阶段使用的控制信号包括 MemOp 及 MemWr。在 WB 阶段使用的控制信号是 RegWr 和 MemtoReg。这些控制信号将随着指令执行一起流过各个流水段，在对应的流水阶段控制执行部件完成指令对应的操作。所以每条指令在每一个阶段执行过程中所需要的数据及控制信息均可以通过流水段寄存器中对应字段直接获取。

可以将各个流水段所需的控制信号打包在一起，便于传输和管理。这样 EX 阶段的控制信号包括 branch、ALUAsrc、ALUBsrc 及 ALUctr，共 10 位。ID 阶段的控制信号包括 MemOp 及 MemWr，共 4 位。WB 阶段为 RegWr 和 MemtoReg，2 位。

ID 阶段译码完成后，将 rs1 和 rs2 送入寄存器的 Ra 和 Rb 进行寄存器读取。注意寄存器读取仍然是不受时钟控制，只要输入的寄存器号改变就立即输出对应寄存器数据。为了区分寄存器号及寄存器中的数据，在后续描述中将 rs1 中的 32 位数据写作 rs1data。

在完成了译码阶段的工作后，ID 阶段需要将指令执行的所有信息写入 ID/EX 流水段寄存器。具体写入内容包括数据和控制信号两部分。其中数据需要将 PC、imm、rs1data、rs2data，共 32×4=128 位写入段

寄存器。控制部分需要将 rs1、rs2、rd、EX 阶段控制信号、MEM 阶段控制信号和 WB 阶段控制信号，共 15+10+4+2=31 位写入段寄存器。因此，共需要长度为 159 位的段寄存器来存放 ID 阶段需要向后传递的所有数据及控制信息。

ID 阶段结束时的下降沿将根据 ID 阶段的执行结果更新段寄存器。同样，在 ID/EX 段寄存器更新时，需要用 flushID 及 stallEX 这两个信号来控制是否需要冲刷和阻塞。冲刷和阻塞的方式类似 IF 阶段，即冲刷时段寄存器清零，阻塞时段寄存器保持不变。

3) 执行（EX）段

执行阶段（EX）的主要任务是利用 ALU 进行实际运算，通过 ALU 运算结果确定 branch 是否需要跳转，并计算跳转地址 Branchtarget 等等。不同指令经 ID 段译码后得到不同的控制信号，用来控制执行部件进行不同的操作。

ALU 运算过程类似单周期处理器，先根据 ALUAsrc、ALUBsrc 来确定 ALU 的两个输入，然后根据 ALUctr 执行相应的运算获取运算结果 ALUResult 及 Less 和 Zero 两个标志位。跳转地址的运算也类似单周期，这里的主要不同点是流水线处理器在 IF 阶段就直接算好了 PC+4，因此在执行阶段主要是计算 branch 和跳转的目的地址，分为 PC+imm 和 rs1data+imm 两种情况。这里使用一个单比特的控制信号 Branchsel 来选择具体用哪种方式来计算 Branchtarget。跳转的控制与单周期类似，由一个小的逻辑单元 Branch Cond 来完成，该单元的输入是 Branch 控制信号及 ALU 的 Less 和 Zero 标识位。根据 Branch 判断跳转的类型，然后根据 ALU 输出的标志位来确定是否需要跳转，最后输出 Branchsel 来控制跳转目标的计算方式，并通过 PCsrc，flushIF 和 flushID 等信号来对 IF 阶段进行跳转控制。具体的跳转控制设计将在后面的控制冒险部分详细描述。

在执行阶段完成后，需要传递给下一阶段的数据主要包括 ALUResult 和 rs2data 共 64 位。需要传递的控制信号包括 rd、MEM 阶段控制信号及 WB 阶段控制信号，共 5+4+2=11 位。所以，EX/MEM 流水段寄存器需要保存一共 75 位的数据和控制信息。

在下降沿到达时，同样根据 EX 阶段的执行结果来更新 EX/MEM 流水段。同时，在 flushEX 信号有效时，将流水段寄存器内容清零。

4) 访存（M）段

访存阶段（MEM）的主要任务是对数据存储进行读写。基本的读写流程与单周期类似，都是将地址信号放在数据存储的 addr 端口并将需要写入的数据放在存储器的 DataIn 端口，根据 MEM 阶段的控制信号执行读写操作即可。为了保证读取的数据能够在下降沿到来之前准备好，需要将数据存储的读取时间安排在时钟的上升沿。另一方面，为了允许测试存储以先读后写的方式来进行字节写入操作，则需要将写入时间安排在时钟的下降沿。

MEM 阶段还将进行数据冒险的检查，通过 Forward Unit 来实现将后续阶段的数据转发给 EX 阶段，以及检测 Load Use 类冒险。

MEM 阶段结束的下降沿上，MEM 阶段需要将 ALUResult 和数据存储的输出 Memout，共 64 位传给下一阶段。同时，MEM 阶段也要将 WB 阶段的 2 位控制信号以及目的寄存器号 rd 传给 WB 阶段。所以，MEM/WB 流水段寄存器中一共包含 71 位信息。MEM 阶段不存在 flush 和 stall 操作。

5) 写回（WB）段

写回阶段（WB）负责将最终运算结果写回到目的寄存器中，其执行的动作比较简单。首先，根据 MemtoReg 确定是将 ALUResult 还是 Memout 写回。然后根据 RegWr 来决定是否要写回。将寄存器写回的时间点定在时钟的上升沿。这样可以实现将 WB 阶段需要写入寄存器的数据直接转发到 ID 阶段，具体描述参见数据冒险部分。写回之后指令执行结束。

综上所述，每个流水段寄存器中保存的信息包括两类：一类是后面阶段需要用到的所有数据信息，包括 PC、立即数、目的寄存器、ALU 运算结果、标志信息等，它们是前面阶段在数据通路中执行的结果；还有一类是前面传递过来的后面各阶段要用到的所有控制信号。

在流水线处理器中，控制信号一旦在 ID 段由控制器生成就不会改变，并和数据信息同步地依次传递到后面的流水段中。流水线控制器的设计可以完全按照单周期控制器设计的思路进行。

3、流水线冒险处理

在 CPU 中指令的执行是有依赖关系的，后续指令的执行依赖于前序指令的执行结果。因此，在 CPU 流水线中需要处理指令之间的三大类依赖关系，对应流水线中的三类冒险。

结构冒险：结构冒险是指不同执行阶段中的指令需要使用同一个硬件资源，造成硬件资源冲突。例如，当两条指令同时要写入寄存器，且寄存器一个周期只允许一个写入时，就存在结构冒险。在设计的五段流水线结构中，通过插入空流水段，使得每个阶段的硬件资源都只有一个指令在使用；在寄存器堆中读写操作分时序执行，相当于两个独立部件；设计独立的下地址计算加法器等方式解决了结构冒险。

控制冒险：控制冒险是指分支和跳转指令有可能改变指令执行顺序。因此，在指令执行过程中如果前序指令中的分支和跳转不能在 IF 阶段就确定下一指令的具体地址，则 IF 阶段会继续顺序取 PC+4 的指令，造成指令取指错误，在真正进行跳转时需要将流水线中错误的指令冲刷掉。

控制冒险的处理方法：在 IF 阶段缺省使用 PC+4 来作为下一个周期的取指地址。这等价于在所有的指令中均预测下一条指令是顺序执行的。但是，这种预测显然是会有出错的情况的。当遇到一个跳转 jal 或者条件分支 bne 这样指令，PC 有可能会跳转到一个新的地方。这时，问题在于在上述的设计中，CPU 要等待到 EX 阶段才能确定条件分支是否会发生跳转。如果发生了跳转，等价于之前预测 PC+4 的猜测发生了错误，前面两级 IF 和 ID 获取的指令是不正确的，必须要冲刷掉。同时，在跳转发生是，需要将 IF 阶段的 PC 更新为正确的跳转目标地址。

为了实现这样的功能，将分支和跳转的处理集中在 EX 阶段进行，即在 EX 阶段结束的下降沿完成指令冲刷和 PC 重置的工作。对于指令冲刷，可以在时钟下降沿根据 Branch Cond 模块的输出，将 flushIF 及 flushID 两个信号置位。在此条件下，下降沿到达时，IF/ID 及 ID/EX 两个流水段寄存器被清零，原先在 IF 阶段和 ID 阶段的两条指令被刷成了空指令 NOP，但是后续 EX、MEM 和 WB 的指令不受影响，继续执行。对于 PC 重置，利用 Branch Cond 输出的 Branchsel 信号选择计算 PC+imm 或者 PC+rs1data 获取跳转目标地址 Branchtarget。然后，将 IF 阶段的 PCsel 信号置位，则在下降沿到达时 PC 将更新为 Branchtarget。下一周期 IF 阶段将从跳转目标地址开始取指令。

在本实验的设计中，为了简化处理，将跳转确定（branch resolution）的时间点放在了 EX 阶段。由于 EX 阶段与 IF 阶段间隔两个流水段，因此每次预测错误将会冲刷 2 个流水段的指令，引入两个空操作。这会降低流水线的整体效率。有统计数据表明一般软件的汇编代码中大约有 20%的指令是跳转和分支指令，其中又有 70%会跳转。这样，如果预测每条指令都继续取 PC+4 会有大约 14%的预测错误。每次预测错误增加两个周期的空操作，这样系统整体会增加 2×14%=28%的周期数。为了提高性能，存在有两个优化方向：一是提升预测准确率，即设法降低 IF 阶段取错指令的概率。在课程中介绍了利用分支预测与 Branch Target Buffer（BTB)部件来实现高效跳转的方案。另一种优化思路是将 branch resolution 的时间点提前到 ID 阶段，减少预测错误需要带来的代价。例如，对于 jal 这样的跳转指令，在译码时就能够确定是否跳转并计算跳转目标，所以可以直接在 ID 阶段确定跳转。对于 Branch 或者 jalr 这类指令，需要先读取寄存器，然后在 ID 阶段增加比较器确定是否跳转。这一方面增加了 ID 阶段的工作量，有可能会导致主频降低，另一方面，由于在有数据冒险的条件下，需要将后续指令的结果转发到 ID 阶段，相对比较复杂。同时，如果 ID 或 IF 均有可能发起跳转和冲刷时，如果确保各种情况下指令均正常执行是一个比较大的挑战。因此，有能力的同学可以挑战在 ID 阶段实现跳转的方式来设计自己的流水线。

数据冒险：数据冒险是指指令执行过程中，前后指令对寄存器的读写产生冲突，造成需要阻塞部分指令等待数据冲突消失。数据冒险又被细分为 Read After Write (RAW)、Write After Read（WAR）和 Write After Write（WAW）三种类型。其中 RAW 类冒险是指后续指令需要使用前面指令的计算结果，但是前序指令还未将数据写入寄存器时，后续指令就需要读取寄存器了，此时后续指令需要阻塞并等待前序指令完成。WAR 是指前序指令需要某个寄存器的内容，但是后续指令抢先更新了寄存器，造成数据读取错误。WAW 是指两个指令均需要写入某个寄存器，但是由于更新次序问题，后面的指令先写入，前面的指令后写入，造成寄存器中的数据不是最新的结果。在这三种数据冒险中，简单五段流水线只会出现 RAW 冒险，所以在本实验中仅需解决 RAW 冒险，其他两类冒险暂不考虑。

RAW 型数据冒险处理方法：考虑以下代码：

```
1    addi t1,zero,100
2    addi t2,t1,20
3    add  t3,t1,t2
```

4　　add t4,t3,t1

其中第 1 条指令的计算结果 t1 会被第 2、第 3、第 4 条指令使用，当第 1 条指令处于 WB 阶段时，第 2、第 3、第 4 条指令分别在流水线中处于 MEM、EX 和 ID 阶段。理论上 WB 阶段完成后，寄存器 t1 才会被更新，所以作为数据消费者的第 2、3、4 条指令都已经错过了读取寄存器的 ID 阶段，无法获取 t1 的最新结果。这种情况下，就会出现 RAW 类型的冒险。

上述 RAW 类型的根本原因是流水线中数据生成者的指令需要等到 WB 才写回寄存器，而数据消费者指令会在 ID 阶段读取数据。这时，生产者写回的阶段与消费者读取的阶段相差 3 个周期。因此，一条运算指令后续的 3 条指令均有可能潜在与该指令发生 RAW 型数据冒险。在五段流水线中，解决数据冒险有两种基本方案：数据转发和阻塞。

数据转发：对于运算类指令，实际的运算结果在 EX 阶段的结尾就已经确定了，但是该结果需要流过 MEM 和 WB 两个阶段才能写入寄存器。因此，可以考虑将流水线后面的指令的运算结果直接送给前面的流水段，让前面的流水段能立刻用上最新的结果。

下面讨论数据转发的三条路径，路径的分类取决于数据生产者指令和数据消费者指令之间的距离，即消费者指令是生产者指令后面第几条指令。

1）生产者和消费者间隔为 3 条指令：这种情况对应上述代码从第 1 条指令传递 t1 给第 4 条指令的场景。第 1 条指令处于 WB 阶段时，第 4 条指令正好处于 ID 阶段。在传统设计中读取和写入寄存器均在时钟周期的末尾，ID 阶段无法获取最新的数据。不过，可以通过改变写入的时间点来完成 WB→ID 的转发。将写入寄存器的时间点设置在周期中间，即时钟上升沿，同时寄存器读取是不受时钟控制的。因此，新的数据可以在 WB 阶段中间写入，立刻被 ID 读取，并在周期结束的下降沿直接写入 ID/EX 流水段寄存器。这样就完成了第一种情况下的转发。

思考：提前在周期中间写入寄存器会不会有副作用？会不会造成某些指令读取了错误的新数据？

2）生产者和消费者间隔为 2 条指令：这种情况对应上述代码中从第 1 条指令传递 t1 给第 3 条指令的场景。当第 1 条指令处于 WB 阶段时，第 3 条指令处于 EX 阶段。此时可以考虑将 WB 阶段的数据转发到 EX 阶段。WB→EX 的转发可以使用对比 WB 阶段的写入寄存器号 WB_rd 与 EX 阶段的两个源寄存器号 EX_rs1 及 EX_rs2，如果两者能够匹配，说明 WB 的结果需要被 EX 阶段使用。在需要转发时，可以利用选择器来选择 EX 阶段的 EX_rs1data 和 EX_rs2data 是从 WB 的数据中获取还是从 ID/EX 段寄存器中获取。这样就可以将 EX 阶段的寄存器数据替换为最新的结果。

思考下列问题：

（1）WB 阶段的转发是否需要考虑 WB 阶段的 WB_RegWr 信号？

（2）应该用 WB 阶段的 WB_ALUResult 还是 WB_MemOut 来替换 EX 阶段的数据？

（3）如果 EX 阶段的 EX_rs1 及 EX_rs2 都能匹配上 WB_rd，即第 3 条指令换成了 addi t3,t1,t1，此时应该如何处理？

（4）如果写入的寄存器是 0 号寄存器 zero，应该如何处理？

3）生产者和消费者间隔为 1 条指令：这种情况对应上述代码中从第 1 条指令传递 t1 给第 2 条指令的场景。当第 1 条指令处于 MEM 阶段时，第 2 条指令处于 EX 阶段，可以借用这个机会来将数据转发到 EX 阶段。此时，进行 MEM→EX 转发。这种转发类似 WB→EX 转发，是通过匹配 MEM_rd 与 EX_rs1 及 EX_rs2 来确定是否需要转发。同时要保证 MEM_RegWr 是有效的，即该数据确实要写入寄存器。

思考：应该用 MEM 阶段的 MEM_ALUResult 还是 MEM_MemOut 来替换 EX 阶段的数据？

数据转发的优先级：如果 EX 阶段的寄存器号与多个后续阶段的目的寄存器均发生匹配，应该如何选择转发哪一个数据？可以观察到，离数据消费者指令越近的数据是越新的，因此，越靠近的指令优先级越高。即，MEM→EX 转发的优先级>WB→EX 转发> B→ID 转发。其中，优先级最低的 WB→ID 转发无需额外处理，这是由于 WB→ID 转发是将转发的数据写入 ID/EX 段寄存器，如果后续指令有更新的数据，自然会在 EX 阶段覆盖掉 WB→ID 转发的结果。另一方面，MEM→EX 转发与 WB→EX 转发的次序是需要额外处理的，即当这两种转发都要刷新 EX 的某个寄存器数据时，应优先选择 MEM 阶段提供的数据。

阻塞：阻塞是在新的数据还没有准备好之前，将需要使用该数据的指令及其后续指令先暂留在当前流水段，暂时不向前流动的数据冒险解决方案。在理论课程中学习过，Load 指令需要等待到 MEM 阶段才能从数据存储器中读取出结果，如果 Load 指令后紧跟一条需要使用 Load 结果的指令，如下情况：

```
1 lw      t1,0(a0)
2 addi    t2,t1,20
```

就会发生 Load-use 冒险。在这种情况下，当第 1 条 Load 指令处于 MEM 阶段读取数据存储器的时候，第 2 条 addi 已经在 EX 阶段了，但是数据还没准备好。虽然可能可以在半个周期，即时钟上升沿就读到存储器，但是余下半个周期留给 ALU 和跳转地址计算还是时间略紧。此时，可以考虑阻塞流水线的执行，让 EX 阶段及前面 2 个流水段的指令暂停执行，等待第 1 条 Load 指令进入 WB 阶段后，通过 WB→EX 转发来解决数据冒险的问题。

阻塞条件的检测也可以通过转发单元进行判断，需要阻塞的情况是 EX_rs1 或 EX_rs2 与 MEM_rd 一致，且当前 MEM 中是在进行数据存储器读取操作。当需要进行阻塞时，首先要将 IF，ID 及 EX 阶段的指令暂停执行，即使得 stallIF，stallID 和 stallEX 这三个信号有效，在本周期保持 PC、IF/ID 及 ID/EX 三个段寄存器不更新。这样可以重新执行 IF，ID 和 EX 阶段的三条指令的对应操作。同时，阻塞时还应进行 flushEX 操作，这主要是由于 EX 阶段的执行是采用旧的数据，执行结果不一定正确，所以应该将 EX/MEM 段寄存器在本周期冲刷为全零，等价于插入一个空操作 NOP。

思考：EX 阶段的指令在发生 load-use 冒险时，对 EX 阶段进行阻塞会不会产生其他副作用？

这里需要注意，选择在 EX 阶段确定是否进行跳转，并将跳转信息以 PCsrc 及 BranchTarget 的形式直接连回了 IF 阶段。如果 EX 阶段用旧数据进行判断，可能会造成 IF 阶段错误选择跳转地址进行跳转。在前述设计中对这个问题的解决方案是，在 IF 阶段将 stallIF 的选择器的优先级设置为高于跳转选择 PCsrc。

因此，当进行 stallIF 操作时，即使 EX 阶段执行不正确，PCsrc 有效也不会对 PC 产生影响，PC 仍然保持不变等到阻塞结束再继续执行。同样的，对于 IF 和 ID 阶段的 flush 信号也应在 stall 时被屏蔽，即 stallID 信号有效时应该不进行 flushIF 操作。

如果同学自行设计了控制冒险的处理方案，需要考虑其他情况，例如跳转在 ID 阶段就请求 PC 更新，ID 阶段阻塞的情况下，如何确定阻塞和 flush 之间的优先级。这类问题请同学们自行考虑。

其它潜在情况：虽然在上述多个思考题中提示了流水线处理器实现中常见的各类“坑”，但是在具体实现中同学们的方案可能有所不同，需要仔细反复确认自己的方案在各类情况下是否都可以正确工作，避免出现 CPU 隐藏各种 Bug。此类 Bug 在实际上板调试时很难定位和解决。

例如一种常见的情形及其可能导致的 bug。考虑下面三条代码：

```
1       addi   a1,a0,16
2       lw     t1,0(a2)
3       sw     t1,0(a1)
```

这三条指令在 CPU 设计中是否会出问题？当指令 3 在 EX 阶段时，指令 2 在 MEM 阶段，指令 1 在 WB 阶段。观察到指令 2 和指令 1 均和指令 3 发生 RAW 型数据冒险，指令 3 中 sw 的执行需要指令 1 提供 a1 用于计算地址，同时需要指令 2 中的 t1 来写入数据存储器。这时，由于指令 2 和指令 3 之间是 load-use 冒险，所以指令 2 会 stall 指令 3 在 EX 阶段的执行，同时冲刷 EX/MEM 阶段的流水段寄存器。但是，指令 1 此时在 WB 阶段需要向 EX 阶段的指令 3 转发数据，在转发完毕后，指令 3 的 EX 阶段被 stall，且冲刷掉了，ID/EX 段寄存器中的数据仍然是之前在 ID 阶段读取的旧数据。到了下一周期，指令 2 进入 WB 阶段，指令 1 已经完成写回寄存器。此时指令 3 继续 EX 阶段，但已经没有机会再从寄存器读取 a1 最新值了，这时指令 3 执行仍然用的是 ID/EX 段寄存器中的旧数据。因此，上述的 CPU 设计在执行这段指令时会出错。虽然解决方案很简单，就是将 WB→EX 转发的结果在 EX 阶段 stall 的情况下写入 ID/EX 段寄存器，但是要发现此类特殊代码组合才会出现的问题并定位是非常困难的。所以建议大家在设计流水线时应仔细考虑各种特殊情况，避免流水线在实现之后仍然有隐藏 Bug。

4、其他设计

访存通路，直接使用 FPGA 芯片提供存储器件来模拟指令存储器 IMem 和数据存储器 DMem，即读请求一个周期后 Mem 能够返回数据，比较简单，可以不连接 Cache。

对访存通路有兴趣的同学可以通过学习 AXI 接口和 DDR IP 核的内容，实现真正的内存到 CPU 的访存通路。

外设能帮助系统有更强的表现力。实验板上最直接可以调用的外设有 LED 灯、七段数码管、开关等，可以自行分配一块内存地址给外设，然后通过 Load 和 Store 来访问以及控制他们。也可以外接键盘和显示器来和 CPU 进行交互。

四、实验内容

和单周期 CPU 设计一样，流水线 CPU 仍然需要单步仿真 Test Bench 和官方测试两种方式验证。

在 CPU 单步测试代码中，测试脚本会在时钟下降沿之后观察本周期的操作结果，并要求以五段流水线的时序来执行测试代码。测试代码会严格按照五段流水线的周期数来判断 CPU 执行过程是否符合要求。测试数据冒险的过程中，要求完成 WB→ID,WB→EX,MEM→EX 三类数据冒险转发，Load-use 冒险阻塞一个周期。控制冒险要求统一在 EX 阶段进行跳转，冲刷两条指令。注意，测试代码中的数据存储使用先读后写的方式来实现按字节写入，所以请在实现的时候按照提前半个周期上升沿读取，下降沿再写入的方式实现对数据存储器的访问。如果实现的 CPU 时序与上述时序不一致，无法通过在线测试。

对于官方测试，将检查点推后了 4 个周期，插入了 4 条全零指令以允许 CPU 完成测试点之前所有代码的执行。请注意将全零指令译码为空指令 NOP。

在设计过程中建议在顶层 CPU 测试模块中预留测试数据接口，上板测试前可以将对应接口接至板上的 LED 或七段数码管，显示 CPU 的内部状态。具体选择测试接口输出哪些内容可以自行决定，可以考虑 PC、寄存器结果，控制信号等等。在初次上板测试时，可以将 CPU 时钟连接到板载的 BTNC 按钮上，每按一下单步执行一个周期，方便进行调试。

自行完成流水线 CPU 的实现，并通过在线测试中流水线 CPU 的单步测试及官方测试部分。

在线测试仅针对讲义中提供的五段流水线设计，且要求进行数据冒险转发、Load use 时阻塞一个时钟周期并且 branch 和跳转指令在 EX 阶段执行。同学们自己实现的流水线可能会在时序上与在线测试结果不一致。可以自行参考课程网站上提供的 test bench 自行编写测试代码，需要完成 RISC-V 官方测试集中 rv32ui-p 开头的所有指令的测试，在现场验收后可以通过。

测试集测试通过后，需要加载实际程序上实验开发板进行验证，在顶层模块设计中，通过拨档开关来输入参数，通过七段数码管来显示输出，支持单步执行，可在 LED 上显示 PC 低 16 位。例如：验证累加和程序、生成第 n 个斐波那契数列程序、判断闰年程序、判断素数程序等。

假设流水线 CPU 模块接口定义如下：

```
module rv32ip(
    input     clock,
    input     reset,
    output [31:0] imemaddr,
    input   [31:0] imemdataout,
    output   imemclk,
```

```
    output [31:0] dmemaddr,
    input  [31:0] dmemdataout,
    output [31:0] dmemdatain,
    output  dmemrdclk,
    output  dmemwrclk,
    output [2:0] dmemop,
    output  dmemwe,
    output [31:0] dbgdata);
//add your code here

endmodule
```

请根据上述描述，按照下列步骤完成实验。

1、 使用 Vivado 创建一个新工程 lab10。

2、 点击添加设计源码文件 lab10.zip。

3、 添加测试文件。将 testcase 目录下的文件拷贝到 XXX.sim/sim_1/behav/xsim 目录下。

4、 添加约束文件。

5、 根据实验要求，完成源码文件的设计。

6、 对工程进行仿真测试。

7、 仿真通过后，进行综合、实现并生成比特流文件。

8、 生成位流文件后，加载到实验开发板，进行调试验证，并记录验证过程。

五、思考题

1、如何添加中断与 CSR 寄存器支持。

2、简述高级流水线设计方法。

3、如何设计更多流水段的 CPU。