计算机组成原理实验

Lab 5 实验手册

流水线 CPU 设计

Made by TA

2023 年 5 月 8 日

目录

在阅读实验手册之前，你需要了解的内容包括：

1. **本次实验包括实验 PPT、实验手册、演示视频以及附件文件（包括 PDU）等内容。我们不幸地发现：即使我们在实验 PPT、实验手册、群聊中多次说明这些内容的存在，依然有同学在最后时刻才发现自己错过了很多。现在我们再次强调：实验手册最后的附录部分包括了本次实验提供的相关资料，请点击睿客网盘连接进行下载。从 Lab5 开始，我们将不再强调这些问题。** 实验手册（也就是你正在看的这个文档）是实验 PPT 讲解的额外补充，用于明确实验细节。 Lab5 的实验附件中包括了数据通路图、PDU 源文件以及我们提供的测试程序。
2. 实验手册的每一部分内容都有着对应的作用。当你遇到困难无法继续时，请确保你已经认真查阅了实验手册中的全部内容！如果你依然对实验内容有所疑问，欢迎你在群聊或私聊中提出你的问题，我们会在许可的范围内进行解答。
3. 请保证实验内容为自己独立完成。我们将对重复率过高的实验结果进行严肃处理。
4. 为了保证区分度，实验的部分内容难度较大，请量力而行，不要在超出自身能力范围外的部分投入过多的精力。

祝大家实验顺利！

本次实验已开通 FAQ 文档！

FAQ 是 Frequently Asked Questions 的缩写，中文释义为常见问题解答，或者是帮助中心。你也可以将其理解为一份针对大家提出问题的统一解答。本次实验的公开 FAQ 文档地址为 https://cscourse.ustc.edu.cn/vdir/Gitlab/PB20020586/lab-of-cod-faq/-/blob/master/Lab5FAQ/lab5.md。当你遇到疑惑的地方时，可以先看看这里，如果还是没有解决你的问题，可以在群聊或私聊中提问。 **我们会根据大家的问题不定期更新 FAQ 文档，请大家随时保持关注。**

本次实验已开通 PDU bug 反馈渠道！（PDU 源文件内容请参考文档末尾的附件链接）

PDU 为助教针对本学期课程需求重新编写的板上调试工具，由于时间紧张，难免会出现 bug。为了保障大家实验的顺利进行，我们为大家开通了 bug 反馈通道。如果你在实验中遇到了难以解决的问题，或影响正常使用的恶性 bug，可以在下面的链接中反馈 https://www.wenjuan.com/s/UZBZJv96IMN/。我们会及时据此更新 PDU 的相关内容，并

在群聊以及 FAQ 中及时告知大家。感谢大家对我们的理解与支持!

1. 前言

在完成了单周期 CPU 的搭建任务后，相信大家已经从作业里感受到了其延迟之巨大。由于一条指令要在一个时钟周期内完成，每个部分的延迟都会进行叠加。最小时钟周期必须要满足通路中最耗时的路径的需求。作业题 4.7 的例子中，即使耗时最长的部件只有 250ns，CPU 的最小指令周期也达到了 920ns。

作为资本家的我们自然希望处理器可以有着更高的工作速度，也就是更高的核心时钟频率。要提高时钟频率，自然就需要降低电路中的延迟。一个自然的思路是，将完整的数据通路**拆分**为各个阶段。让每条指令从第一段开始依次走到最后一段，全部完成后再依次执行下一条指令的各个阶段。这就得到了多周期 CPU。

助教的碎碎念：

如何将数据通路划分成多个阶段呢？按元器件划分？这样可能导致同一位置上不同来源数据的阶段不同，也就无法同时在这里汇合。那如果按照工作内容对汇合点处的通路进行分段呢？例如：ALU_ans 是一个汇合点，因为不同来源的数据经过 ALU 后统一得到一个结果； RF_WB_out 也是一个汇合点，因为不同来源的数据经过该选择器之后得到即将写回的结果。这样分段可以保证结果的正确性，但依然不是最优的。

一直以来，我们希望大家能够意识到不同操作带来的延迟是有差异的。访问内存往往是最为耗时的步骤，访问寄存器堆同理。除此之外，大型组合电路的延迟也不容小觑（例如 ALU）。我们希望划分后各段之间的延迟差异不会太大，避免出现某一段占据了极高的延迟这种情况。因此，一个优秀的分段方式应当将长延迟路径进行拆分，短延迟路径进行整合。

回到我们课上讲到的多周期 CPU：IF 段需要访问指令存储器，延迟较高，所以进行拆分； ID 段需要访问寄存器堆，延迟较高，所以进行拆分，而 Control、IMM 等模块的延迟较小，可以在此阶段并行处理；EX 段的延迟主要来自 ALU 模块，其他的选择器延迟则相对较小，我们进行合并分段； MEM 段的延迟来自于数据存储器的读写这也是五段中最高的延迟），因此单独分段；最后 WB 段只包含选择器与写入寄存器堆的延迟，单独成段。

> 这样的分段方式是合理的。当然，针对 IF 以及 MEM 段的高延迟，现代处理器也会选择将这两段再次拆分，得到七段、十段甚至更多的分段方式。

多周期 CPU 的时钟周期至少需要是其中最长一段的时间。因此，一条指令的执行时间至多为段数 × 时钟周期。这样算下来时钟频率确实上去了，但指令的耗时并没有减少多少。那如何减少指令执行的总时间呢？考虑这样的设计：在一条指令进入第二段时，便让下一条指令进入第一段，每一段中同时处理不同指令。 CPU 的时钟周期并没有改变，但这样即可实现平均一条指令一个时钟周期，总执行时间大幅减小，如下图——这就称作**流水线 CPU**。

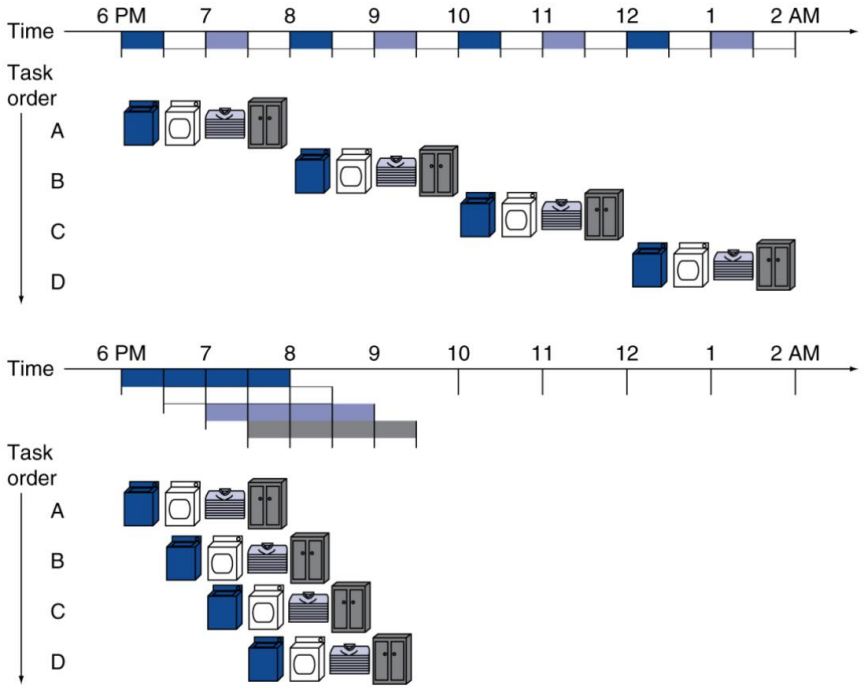

图 1: 多周期（上）与流水线（下）效率对比

很遗憾的是，上述只是考虑了最简单的情况。稍加考虑即能发现可能存在的问题：控制指令会导致之后执行的指令发生变化，而在流水线前面的段落无法识别，载入了错误的指令；部分写指令需要写回寄存器堆，而如果下一条指令需要使用写回的值，在写回未进行时只能读取到旧的值…… 这类问题被称作**冒险**(hazard)，也就是在下一个时钟周期无法直接执行下一条指令的情况。

本次实验中，我们将完成一个完整的流水线CPU，并且通过前递与停顿解决这些冒险，使得汇编程序可以正常运行。

2. 主要内容

本文档主要介绍的内容如下：

- 对流水线 CPU 的分段与段间寄存器介绍；
- 对流水线 CPU 的每段组件与单周期区分进行介绍；
- 对流水线 CPU 的冒险处理单元进行介绍；
- 对于外设与调试单元 PDU 的介绍。

你需要完成的内容概括如下：

> 详细的实验内容见文档结尾，此处给出大致内容以方便同学们带着目的去阅读文档。

> **温馨提示：**
> **不要被看起来复杂的数据通路吓到**，很多代码可以复用，只需要大家耐心看图连线就好了（笑）。

- 根据我们提供的数据通路，在单周期 CPU 基础上完成流水线 CPU 的硬件设计，并完成仿真。
- 将 PDU 接入到 CPU 上，完成上板测试。
- 运行给定的汇编程序，并检查运行结果。

3. 多周期设计与段间寄存器

3.1 段落划分

为了将单周期拆分为多个段落，最经济的拆分方式即将耗时的主要部分（如寄存器堆、ALU、存储器等）分开。参考前言部分中的分析，结合实际逻辑上的要求后，我们的 RISC-V 流水线最终分为了五个段落：

- IF（Instruction Fetch，取指令），核心耗时为指令存储器的读取。
- ID（Instruction Decode，译码），包含将指令翻译为各个控制信号并读取寄存器堆，核心耗时为寄存器堆的读取。
- EX（Execution，执行），由算术逻辑单元 ALU 进行运算，得到指令的计算结果，同时计算可能需要的跳转地址，核心耗时为 ALU 计算。
- MEM（Memory，访存），对数据存储器进行读取或写入，核心耗时为数据存储器的读写。
- WB（Write Back，回写），将需要写回寄存器堆的数据写入。注意此处耗时与数据存储器写入一样，只考虑准备的时间，因为实际写入是在时钟上升沿进行的。

各个段落的简单示意图如下：

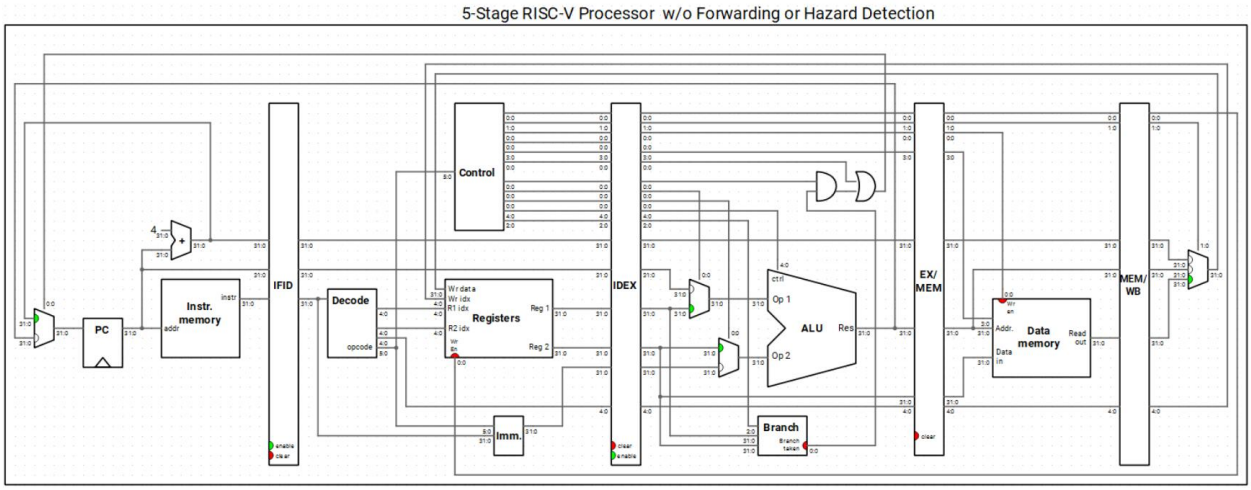

图 2: Ripes 中五段划分示意图

你可能会有这样的感觉：流水线 CPU 的数据通路仅仅是在单周期的数据通路之间插入了段间寄存器而已。事实上也确实如此。当然，单周期 CPU 模块之间的信号已经发生变化。例如，从指令存储器中读出的指令需要先等待流水线轮转，才能进行译码得出控制信号，而并非与后续模块直接相连。为了实现各个段内部的信号同步，我们采用**寄存器**进行控制。出于流水线的设计，信号同步只会发生在各段之间，因此每段可以合为一个整体的**段间寄存器**模块统一处理。

3.2 段间寄存器

先给大家展示一下完成版的 ~~73 腿蜈蚣~~ 段间寄存器（以 ID/EX 为例）：

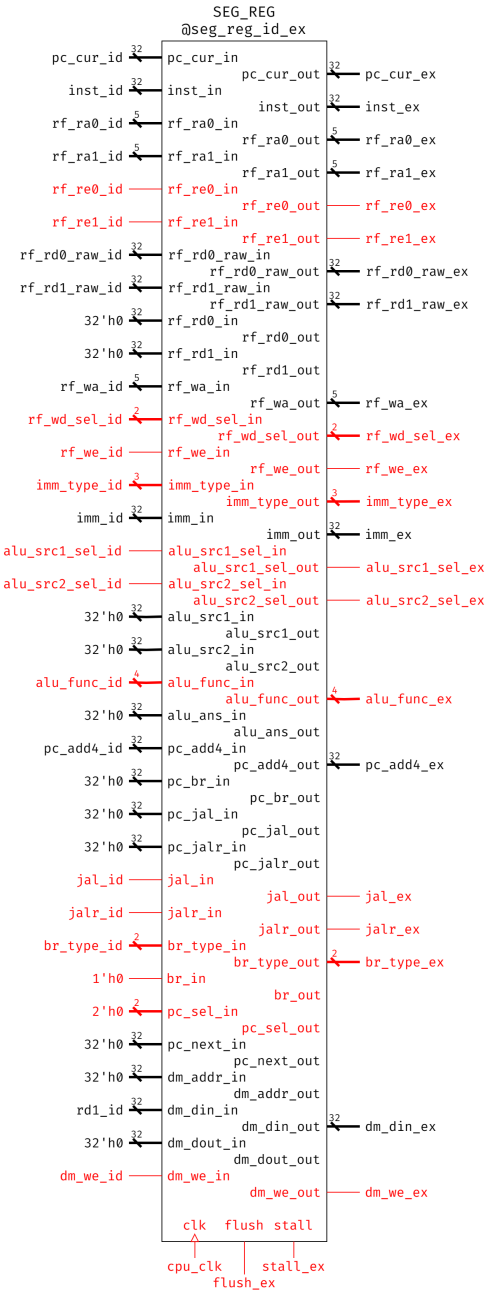

图 3: ~~73 腿蜈蚣~~ ID/EX 段间寄存器数据通路示意图

仔细观察数据通路图可以发现，每段的段间寄存器事实上是**例化了同一个模块**，但连接不同的输入、输出。从原则上来说，每个段间寄存器承担的职责是将该段需要传递的信号传递给下一段。例如，ID 段生成的 alu_func 信号通过 ID/EX 段间寄存器传到 ALU 的控制

接口上，这一信号在其他段中则不应该出现。然而，这么做会引起两个问题：

其一，每段的段间寄存器需要重新设计。但有些信号（如寄存器堆写使能 rf_we）事实上需要多次传递，而有些信号（如寄存器堆的读地址 rf_ra1）则在最初设计时看似不要传递，在处理数据相关时却发现需要传递。设计过程中对模块接口不断的复制、修改容易引起混乱。

其二，如果信号只保留有限的生命周期，则调试时需要反复查看不同段落正在执行的指令，这加大了调试的难度。

因此，我们选择使用统一的段间寄存器模块，让信号一经产生就传递到最后，从而简化了段间寄存器的设计。对于在上一阶段尚未产生的信号，直接在输入处接 0，忽略其输出，即只保留有效端口的使用。当然，这样也就导致我们的段间寄存器十分庞大。

> 这里需要强调的一点是：模块的输出端口允许空置，但输入端口不允许，否则会发生错误。因此对于无用的输入，可以将其端口置 0。例如

```
1     .useless_input(32'b0)   // A 32-bits useless input port
2
```

在数据通路中，我们一般利用信号名称的最后一段标识信号所在的段落，从而区分不同段落的同一信号，也方便检测段间寄存器的连线是否正确。

在控制信号部分，除了 clk 以外，段间寄存器还预留了 stall 与 flush 接口，分别用于停驻（指令内容不随时钟前进）与清空（插入气泡）。我们将在冒险处理部分对它们的作用进行介绍。

> 段间寄存器有两种实现方式，一种是输入、输出的每个信号分离开，设置 73 个端口，并为每一项输入信号设置专门的寄存器。另一种则是只保留一个输入端口、一个输出端口与三个控制端口，通过位拼接的方式进行输入输出，在内部仅使用一个超长位宽的寄存器。这两种实现方式各有优劣优劣，你可以根据自己的需要与习惯作出选择。

3.3 附加-控制提前

> 文档标明附加的部分关系到选做，可以考虑读完其他部分后再回头阅读。

在刚才的介绍中，我们把所有的跳转信号放到了 EX 阶段生成。**目前，流水线 CPU 的跳转逻辑是假定每一条指令都不跳转**，也就是说在跳转指令执行到 EX 段之前，ID、IF 段的 PC 都是由跳转指令之前的指令决定的。假如跳转指令的结果是需要跳转，那么当前在 IF 和 ID 阶段的指令只能被废弃（猜错了）。

在理想情况下，CPU 在 IF 段读取到一条跳转指令后，就应该得出下一时刻的跳转情况。然而，跳转指令的结果需要等到 EX 段才能得到，我们并没有未卜先知的能力。因此，如何尽快准确“猜测” 出当前指令的跳转结果也是优化 CPU 的一个方向。

回到我们的设计上。如果能在 ID 阶段就得到跳转指令的结果，则只会造成 IF 阶段的指令废弃，相当于提升了跳转指令一个周期的性能（猜错时只需要废弃一个阶段的内容）。这种提前跳转的方式在实际 CPU 设计中也是常用的。

> 另一个减小控制指令延迟的常用方式是**分支预测**，会在综合实验文档中介绍。思考：为什么不能干脆将跳转提前到在 IF 阶段？（提示：考虑延迟与最小时钟周期）

一个将 jal 提前到 ID 阶段的数据通路如下：

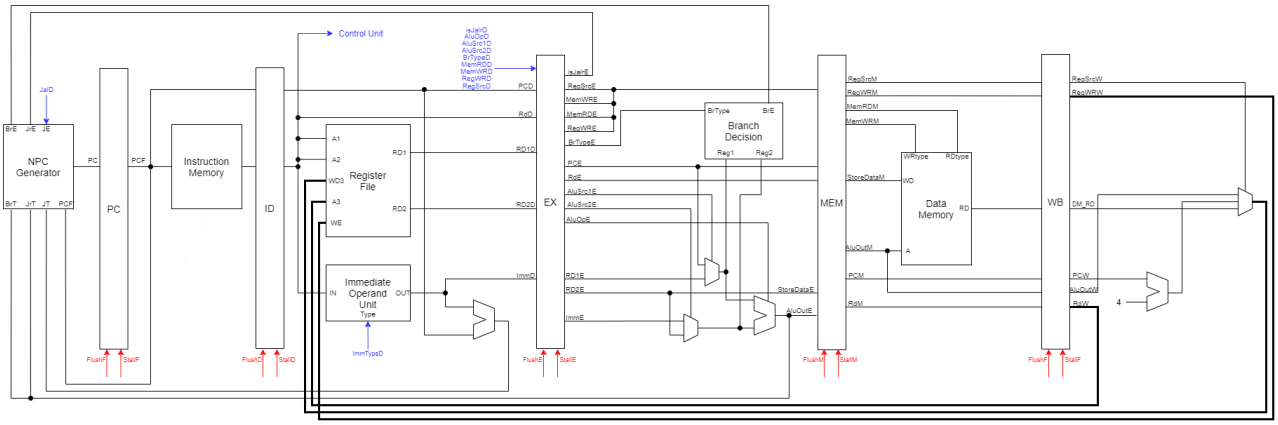

图 4: jal 提前的数据通路示意图

在这个数据通路中，我们直接在 ID 阶段计算 pc 的跳转地址，并触发 jal 的跳转。值得注意的是，当跳转指令在不同阶段出现时，我们需要对其进行优先级的区分。例如在 EX 段执行的指令是 jalr 的同时，在 ID 段执行的指令是 jal，请思考此时如何正确跳转。

除了 jal 指令以外，分支模块 Branch Desicion 和 jalr 也可以前移到 ID 段完成。不过，由于其需要读取寄存器堆的值，并且计算较复杂，将其前移到 ID 段会大幅增加 ID 阶段的延

迟，几乎相当于原本 ID 与 EX 段延迟的和，也会带来更复杂的数据冒险。虽然在我们的模型下，存储器的读写延迟事实上一般大于这个和，不会有太大影响，但如此会导致 EX 段和 ID 段基本合并，不符合五级流水线设计原则。

3.4 附加-中断与异常

在之前的实验中，有不少同学产生了疑惑，我们的 CPU 并没有提供程序结束的指令。在计算机系统概论的学习中，程序结束是依靠 TRAP 完成的，而这事实上是一种**中断**。它的释义是：计算机在执行程序的过程中，当出现异常情况或特殊请求时，计算机便停止现行程序的运行，转向对这些异常情况或特殊请求的处理。处理结束后，再返回到现行程序的间断处，继续执行原程序。处理中断的系统称为**中断服务程序**(Interrupt Service Routines, ISR)。

常见的中断类型有如下几种：

- 异常(Exception)，响应软硬件的故障而产生的中断，如指令访存未对齐、整数除以 0 等应引发此类中断。
- 输入输出中断(I/O)，由外设（如串口）引起。由于我们的实验通过地址映射(MMIO)的方式实现输入输出，因此不会产生这类中断。
- 陷入(Trap)，由一些专用指令引发，进行特殊处理。

> 这一段在各个教材上有着非常复杂的说法，例如，有的将这种对异常情况或特殊请求的处理总称为异常，有的则总称为陷入，这时中断一词就特指外部（如 I/O）引起的中断，而原本的细分类则又会被冠以其他名词。当在其他地方见到时，请根据实际情况确定名词的含义。

在 RISC-V 指令集架构中，ebreak 与 ecall 都是较为典型的陷入指令（其对应的二进制表示见 RISC-V 指令集手册）。 ebreak 相当于在程序中打了一个断点，ecall 则代表引发一个环境调用异常，二者都是在收到特殊信号后再继续运行。更多相关内容详见 https://zhuanlan.zhihu.com/p/461722132，下方是一个简单的处理流程示意：

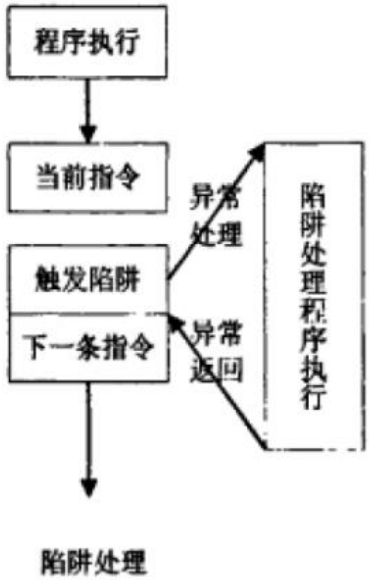

图 5: 陷入处理流程示意图

在这个实验中，我们不需要模拟完整的中断处理流程，只需要实现 ebreak 指令的断点功能即可，也即要求：若代码中运行了一行 ebreak，CPU 应当在其前的指令都正常执行完成后停下，并等待来自 PDU 的后续控制。该过程与 PDU 的断点调试过程基本一致，但是二者停下时指令的位置不同。 PDU 的断点对应下一条即将执行的指令，而 ebreak 的断点对应上一条恰好执行完成的指令。

4. 重点单元介绍

4.1 作为段间寄存器的 PC

流水线 CPU 的 PC 模块如下：

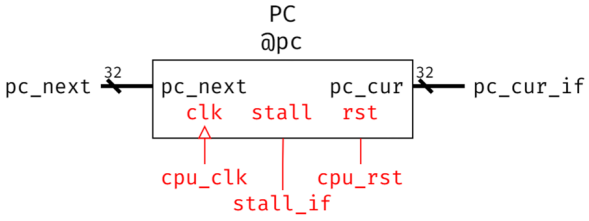

图 6: PC 模块示意图

可以发现，其比单周期加入了一个 stall 接口，就像我们的段间寄存器一样。由于在流水线 CPU 中 PC 寄存器与段间寄存器有相同的控制方式，因此可以将其看作一个特殊的段间寄存器，也即进入 IF 前的段间寄存器。这个段间寄存器只有一个寄存信号，即 PC。

从对称性的角度，PC 寄存器也应该有一个 flush 信号，用于冲刷其内容。但由于其唯一作用是产生地址用于从 ROM 中读出指令，需要清空时只要将读出的指令清空即可，无需将 PC 寄存器置于特定值，因此我们将 flush 信号移到了读出指令的选择器上。

> 输出选择器的 flush 信号并不是必须的，这取决于 flush 信号的工作逻辑。我们将在后面的部分中说明这一点。

4.2 寄存器堆写优先

流水线寄存器堆的接口与单周期是相同的，但是有一个非常重要的调整，也即**写优先**。考虑如下指令序列，其中初始时 x1 为 0：

addi **x1**, x0, 1

addi x2, x0, 0

addi x3, x0, 0

addi x4, **x1**, 1

根据流水线的分段结构，当第一行代码进行到 WB 阶段，即 x1 写入时，第四行代码恰好进行到 ID 阶段的 x1 读取。若仍用之前的直接**异步读取同步写入**策略，由于写入只在上升沿进行，此时 ID 段读取出的结果依然为 x1 的旧值 0（尽管此时写使能 we 与写数据 wd 均已到达）。这样 x4 的计算结果就会出错。因此，寄存器堆的异步读取必须改为写优先，也即当读取和写入同时进行的时候，寄存器堆能直接传出正在写入的值。

> 特别提醒：不允许通过将寄存器堆的上升沿写入改为下降沿写入来达成写优先的效果，因为这会导致各种复杂的时序问题。我们将严格此项内容的检查过程。

> 不要忘记考虑 x0！即使下一小节的控制信号中已对 x0 进行了特殊处理，这里仍需保证写优先自身的正确性。

4.3 控制单元的调整

流水线 CPU 的控制单元如下：

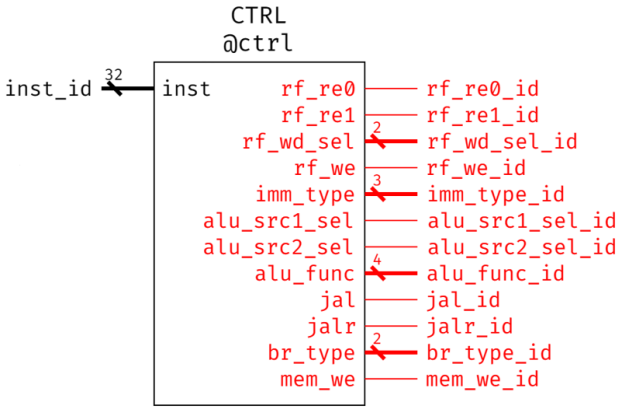

图 7: 控制单元数据通路图

比起单周期，控制单元加入了两个特殊的控制信号——寄存器读使能(rf_re)。其含义正如字面意思，当需要读寄存器时，读使能有效（高电平）。观察寄存器堆数据通路可以发现，这两个信号事实上并未传到寄存器堆中，而是在冒险处理(Hazard)中使用。我们将在介绍前递的时候进行介绍。

此外，我们非常推荐进行如下的控制信号设计：**在读寄存器 rs 为 x0 时，将 rf_re 设置为 0；在写寄存器 rd 为 x0 时，将 rf_we 设置为 0**。这么做可以规避大部分前递中可能出现的错误。

4.4 nop 指令与段间寄存器的清空

在课上我们已经学过，有时候需要在流水线中塞入“气泡”。在实际情况下，这是通过清空段间寄存器来实现的——但是，何为清空？全是0就是清空吗？事实上，清空段间寄存器意味着将该段变为执行一条“什么都不做”的指令，也就是 nop 指令。

在我们的流水线 CPU 设计里，“什么都不做”是通过 add x0, x0, x0，即 0x00000033 来实现的：

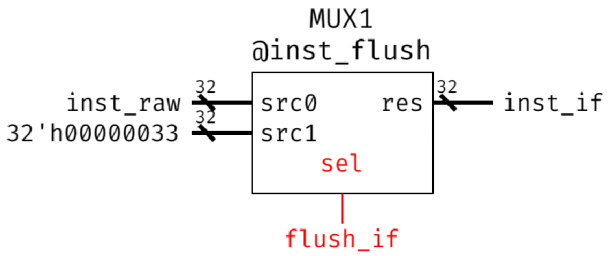

图 8: 指令寄存器的清空

若按照上一小节所述进行控制信号设计，这条指令会传出全为 0 的控制信号，这样容易验证之后段间寄存器除了指令部分恰好都全为 0，就能让“清空”变得直观。

另一个常用的 nop 指令是 addi x0, x0, 0，即 0x00000013。RARS 中也是如此实现的。但是，在我们的数据通路中，这会导致 alu_src_2 变为 1，因此出于清空段间寄存器的简便考虑，我们使用 add x0, x0, x0 作为 nop 指令。

4.5 附加-存储器读写处理

若想实现诸如 lh、lb、lhu、lbu 与 sh、sb 这类能做到未必对齐的指令，需要额外增加读取地址处理与写入地址处理的模块。

例如，若希望实现 lb x5, 0x13(x0) 这样的指令，实际流程是：

- 将读地址的 [9:2]，即 0x04 作为读地址读出数据存储器的值 mem_rd；
- 送入 mem 输出处理单元（可以考虑放在 WB 段，因为 MEM 段的延迟已经很大），用生成的控制信号进行处理。具体来说，将新的结果的 [7:0] 设置为 mem_rd 的 [23:16]，并将新结果高 24 位设置为 mem_rd[23] 的符号扩展（可用拼接实现）；
- 将处理后的新的结果写回 x5。

sb、sh 指令同理，其数据处理可以考虑在 EX 阶段完成，保证送入 MEM 段的直接为可以写入的数据。

这里给出一个 Logisim 实现的存储器读写处理作为参考：

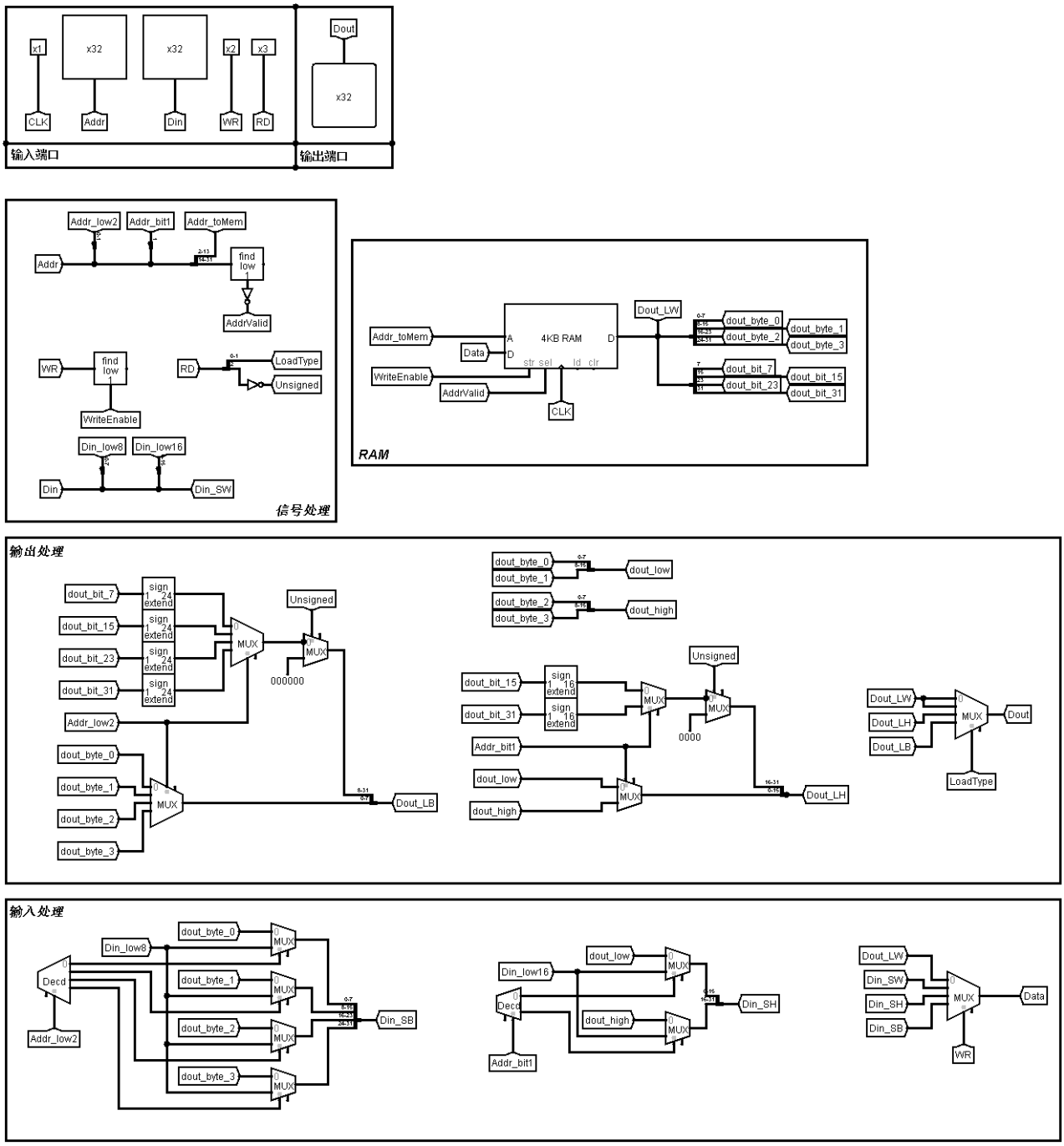

图 9: 数据存储器读写处理示意图

5.

三种冒险及其处理

在本节，我们终于要开始解决前言中提到的最大问题，也就是“冒险”了。根据冒险出现的原因，冒险一般分为三类：结构冒险、数据冒险与控制冒险。

5.1 结构冒险

在 LC3 中，数据和指令是可以共用存储器的，因此如果在写入数据的同时读取指令，就会产生存储器需要同时读写的问题，这就是**结构冒险**，这样的架构称为冯诺依曼架构。

而在我们的流水线 CPU 中，这一问题并不会出现，原因是我们采用了**哈佛结构**，将数据存储器与指令存储器分开，于是，结构冒险得以自然解决。

5.2 数据冒险-前递

考虑如下的指令序列：

addi **x1**, x0, 1

addi x2, x0, 0

addi x3, **x1**, 1

理论上，在第三条指令进入 ID 阶段，需要读取 x1 的数据时，第一条指令进入 MEM 阶段，还没有到写回的时间。类似的，对如下的指令序列，第一条指令甚至只进入了 EX 阶段就面临第二条指令在 ID 阶段需要读取 x1 的问题：

addi **x1**, x0, 1

addi x3, **x1**, 1

不过，更进一步地思考可以发现，这个问题并不像想象中一样不可解决。在 EX 阶段和 MEM 阶段结束后，x1 的新值已经在 ALU 中被算出，只是尚未写回寄存器堆而已。而实际使用寄存器堆的值进行计算的是在 ID 结束后的 EX 阶段。如果这时直接把计算出的新值传递给 ALU 作为操作数，程序完全可以正常运行。这就叫做**前递**。

上面的两种情况即对应两种前递方式，分别是从 MEM 段（EX 段结束后）前递到 EX 段与从 WB 段（MEM 段结束后）前递到 EX 段。实际判别条件也即（注意可能 EX 段读取的两个寄存器都是 MEM/WB 段正需要写入的）：

- MEM/WB 段写使能为 1；
- EX 段某寄存器读使能非零；（思考：一定需要这个判断吗？）
- 上述寄存器读地址等于 MEM/WB 段的写地址；
- 若为 MEM 段，写回的数并非数据存储器读取结果。（这种情况的处理见下一小节）

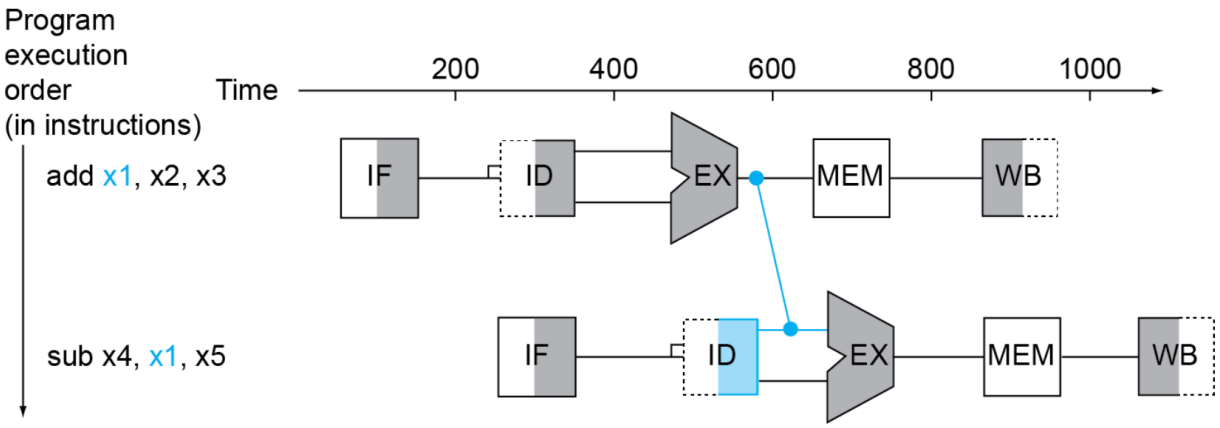

图 10: MEM 段前递示意图

值得注意的是，如下的指令序列会出现 MEM 与 WB 同时检测到前递，请根据这个例子思考此时应当如何前递：

addi **x1**, x0, 1

addi **x1**, **x1**, 1

addi x2, **x1**, 1

> 若写入的对象是 x0，自然不需要任何前递操作，这就是为什么生成控制信号时最好考虑读写 x0 的情况。正常的程序中一般不会出现写入 x0 的奇怪情况，但我们的测试样例可能会！

5.3 数据冒险-气泡

利用前递处理数据冒险的好处是，流水线 CPU 不需要进行任何停顿。但接下来的数据冒险例子就没有那么友善了：

lw **x1**, 0(x0)

addi x2, **x1**, 1

这种冒险被称为读取-使用冒险(Load-Use Hazard)，由于读取在 MEM 段结束才能完成，这时无法直接前递，而是必须等待一个周期。如果直接前递，则 EX 段的最大延迟就会再加上存储器的读取延迟，这是我们不能接受的。

> 按照书本上所介绍的，EX 段正在执行时，lw 指令的结果还没有从存储器中读出。由于时光不能倒流，我们便无法通过前递的方式将尚未产生的结果向前传递。当然，我们可以等一段时间，直到存储器的结果读出后再前递到 EX 段，并完成 EX 段的后续计算。这样虽然实现了前递，但是 EX 段的延迟就会变得十分巨大，从而影响整条流水线的性能。

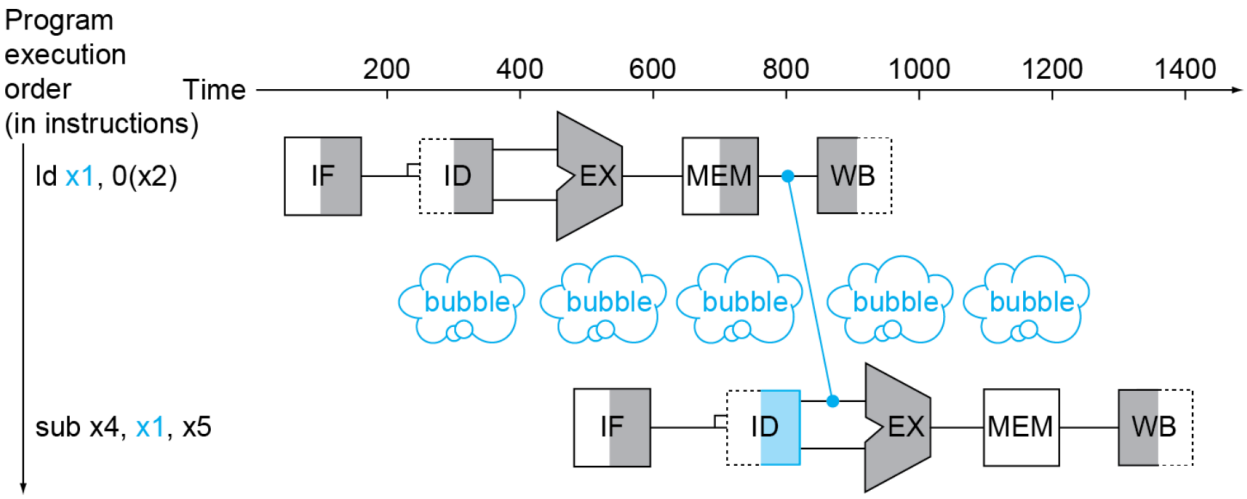

图 11: 读取-使用冒险的处理示意图

这种冒险的判断方式前三步与上一小节相同，但最后一个条件为 MEM 段的写回数据选择为数据存储器的结果。

为了达到插入气泡的效果，已经读取到过时数据、计算出错误结果的 EX 阶段指令将被忽略，也即在下一时钟上升沿**清空** EX/MEM 段间寄存器；而还未执行的 IF 前(PC)、IF/ID、ID/EX 段中的指令需要**停驻**一个周期，让气泡在它们之前通过。

在下一个时钟周期，由于 ID/EX 停驻了，在 EX 阶段的还是刚才的指令，而这次在 WB 阶段检测出需要前递，此时前递就不存在问题了。

5.4 控制冒险

第三种冒险是由跳转指令产生的控制冒险。值得一提的是，由于我们采用了默认不跳转的原则，当 B 类指令不发生跳转时，并不会产生控制冒险，只有跳转发生时才会出现问题。按照给出的数据通路，所有跳转都是在 EX 阶段进行检测的，检测后下一次的 PC 会变

为正确的跳转地址。因此当前按照错误地址读取到的 IF/ID 段间寄存器与 ID/EX 段间寄存器都应清空。

5.5 冒险处理模块

冒险处理模块如下：

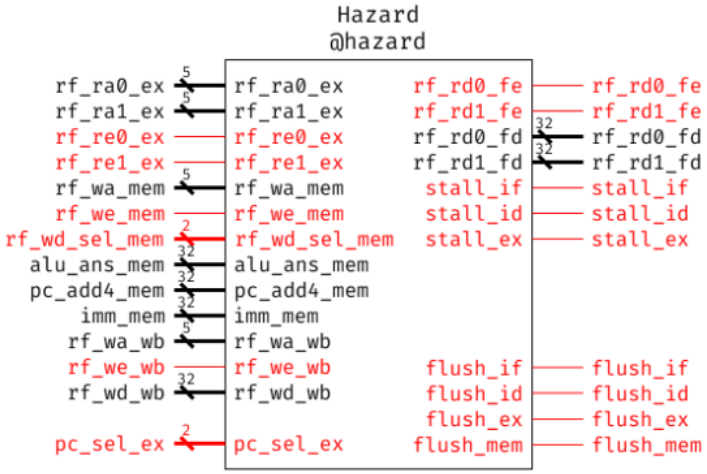

图 12: 冒险处理模块示意图

它将所有可能用于判断、生成的信号作为输入，并将前递使能、前递、停驻与清空信号作为输出。将它与 ALU 前的选择器、各段间寄存器配合后，即可实现对各种冒险的处理。

一点说明：

在给出的数据通路中，我们对冒险处理模块的 stall 信号的约定是：接收到 stall 信号的寄存器（PC 或段间寄存器）在下个时钟上升沿将保持原值不变。这实际上是寄存器的反向写使能信号。

关于 flush 信号，我们有两种约定。其一是接收到 flush 信号的段间寄存器将其输出信号改变（异步 flush，寄存器中存储的结果不变，但是输出增加一级选择器），这时我们需要的是 flush_if, flush_id 和 flush_ex；其二是接收到 flush 信号的段间寄存器在下个时钟上升沿到来时将寄存器存储的值改变（同步 flush，与同步复位类似，寄存器中存储的结果改变），这时我们需要的是 flush_id, flush_ex 和 flush_mem。这两种处理方式各有其好处和缺点：使用第一种方式的优点在于直观，flush 信号到来时即清空当前流水段的当前指令，但使用这种方式时，我们不能清空 rf_ra0, rf_ra1, rf_re0, rf_re1，否则会导致处

理前递时 Hazard 模块的输入改变（乱套了）；使用第二种方式时则无需考虑这些，但其含义变为清空当前流水段即将到来的指令。

助教在这里比较推荐大家使用第二种方式，实验文档也是按照第二种方式书写的，而数据通路保留了两种方式的实现可能。

6. PDU 与顶层模块

6.1 PDU 调试流程

在 Lab4 中，我们已经为大家提供了具有完整功能的 PDU，本次 Lab5 的 PDU 功能、操作方式与 Lab4 相同，相关的基础知识请参考 Lab4 的文档，本部分主要介绍差异。

注意：在使用 PDU 进行调试之前，你需要确保：

- CPU 内部的所有模块都已经实现（正确性可以不用验证），PC 寄存器可以被正确写入与读出；
- CPU 内部的 cpu_check_addr 和 cpu_check_data 已根据提供的流水线 CPU 数据通路图正确连接。

  如果 cpu_check_addr 为 0x0XXX，则代表查询的是 CPU 内部通路上的信息，如果 cpu_check_addr 为 0x1XXX，则代表查询的是寄存器堆的信息。因此，可以通过检查 cpu_check_addr[12] 来判断 Debug 数据的来源。

  若检查的是 CPU 内部通路的信息，cpu_check_addr[7:5] 代表查看的阶段， 0 到 5 分别表示 IF(0)、ID(1)、EX(2)、MEM(3)、WB(4) 与 Hazard(5) 部分的信号。

  本部分数据通路见下图：

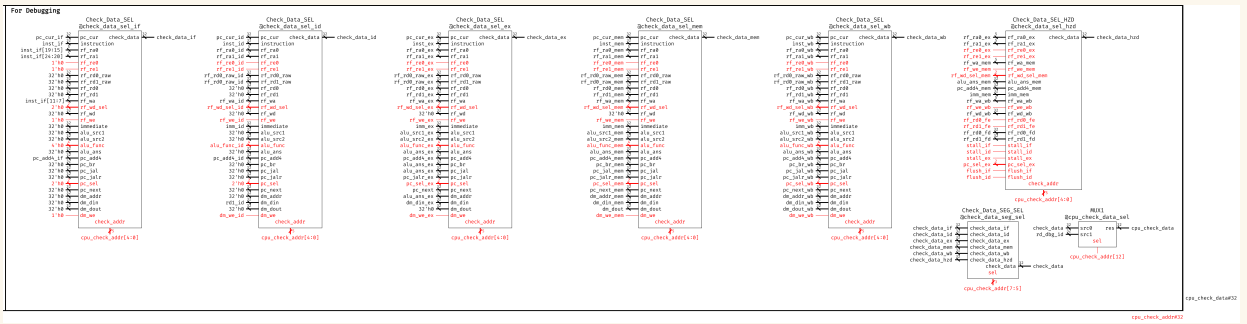

图 13: 调试部分数据通路

- MEM 内部的指令与数据存储器已被正确例化并初始化。上板后，我们需要 PDU控制 CPU 执行相应的汇编程序，因此需要保证此时 MEM 模块中已经保存了待检测的汇编程序。

流水线 CPU 一共提供了 4 个测试程序，详见附件部分的介绍与各文件的注释。对前三个测试建议通过仿真进行测试，最后一个测试 pipeline test 则可以上板操作，具体结果与上一部分类似，leb[0] 亮起代表成功，led[1] 亮起代表失败，检测 x25 的值可以判断具体失败的位置，具体流程为：

(1) 拨动 sw7，该开关对应着系统 reset 信号。开关拨下后，PDU 会中断 CPU 时钟信号，将状态由 CPU 运行状态（led7 亮起）转变为调试状态（led7 熄灭，led4 亮起）。

注意：由于 FPGAOL 的问题，部分板子在烧写后 led 均为熄灭状态，输入 ck 指令后串口回显功能也不正常。这个时候可以检查一下 ck0 00，即 IF 段 PC 的内容是否为 0x2ffc 或者 0x3000。如果数码管显示正常，则可以继续进行测试，否则应当检查 debug 部分数据通路的连接情况。

(2) 使用调试指令 ck1 **19**（注意进制）; 查看寄存器 x25 中存储的数值，该数值具体含义见 pipeline test 的注释。你可能需要进一步查找，以从我们提供的测试程序中找到相应的测试部分，查看该部分的起始指令地址（不妨假定为 0x3XXX）。

(3) 重新烧写 bit 流，设置断点 bp 3XXX; 随后输入 run; ，让 CPU 恰好运行到该测试样例的开始处。**注意：我们的断点地址对应的是 IF 段的 PC**。实际调试时，请根据指令所处的阶段考虑断点地址应当如何设置。

(4) 接下来，通过 step; 指令单步运行 CPU，并使用 ck 系列指令设置相应的 check_addr 查看 CPU 内部的信息，并于自己在 Rars 或 Ripes 上的结果进行对比，直至找到执行结果不一致的指令，即可定位 CPU 的 bug。我们建议你关注以下内容： ALU 的输入操作数、段间寄存器的行为、寄存器堆的信号等。

6.2 Debug 地址映射规则

这一部分内容本来不应该被放到实验手册中的，但是奈何其他文件大家总是看不见（摊手），因此我们在这里进行说明。

为了节约大家的时间，本次实验的 Debug 模块已经由助教为大家写好（在附件文件中）。大家可以直接在自己的 CPU 模块中将其例化，并正确连线。这一部分的逻辑如下：

```
1   module Check_Data_SEL (...);
2       ...
3       always @(*) begin
4           check_data = 0;      // Default value
5           case (check_addr)
6               5'd0: check_data = pc_cur;
7               5'd1: check_data = instruction;
8               5'd2: check_data = rf_ra0;
9               5'd3: check_data = rf_ra1;
10              5'd4: check_data = rf_re0;
11              5'd5: check_data = rf_re1;
12              5'd6: check_data = rf_rd0_raw;
13              5'd7: check_data = rf_rd1_raw;
14              5'd8: check_data = rf_rd0;
15              5'd9: check_data = rf_rd1;
16              5'd10: check_data = rf_wa;
17              5'd11: check_data = rf_wd_sel;
18              5'd12: check_data = rf_wd;
19              5'd13: check_data = rf_we;
20              5'd14: check_data = immediate;
21              5'd15: check_data = alu_sr1;
22              5'd16: check_data = alu_sr2;
23              5'd17: check_data = alu_func;
24              5'd18: check_data = alu_ans;
25              5'd19: check_data = pc_add4;
```

```
26                  5'd20: check_data = pc_br;
27                  5'd21: check_data = pc_jal;
28                  5'd22: check_data = pc_jalr;
29                  5'd23: check_data = pc_sel;
30                  5'd24: check_data = pc_next;
31                  5'd25: check_data = dm_addr;
32                  5'd26: check_data = dm_din;
33                  5'd27: check_data = dm_dout;
34                  5'd28: check_data = dm_we;
35              endcase
36          end
37      endmodule
38
39      module Check_Data_SEL_HZD (...);
40          ...
41          always @(*) begin
42              check_data = 0;      // Default value
43              case (check_addr)
44                  5'd0: check_data = rf_ra0_ex;
45                  5'd1: check_data = rf_ra1_ex;
46                  5'd2: check_data = rf_re0_ex;
47                  5'd3: check_data = rf_re1_ex;
48                  5'd4: check_data = pc_sel_ex;
49                  5'd5: check_data = rf_wa_mem;
50                  5'd6: check_data = rf_we_mem;
51                  5'd7: check_data = rf_wd_sel_mem;
52                  5'd8: check_data = alu_ans_mem;
53                  5'd9: check_data = pc_add4_mem;
54                  5'd10: check_data = imm_mem;
55                  5'd11: check_data = rf_wa_wb;
56                  5'd12: check_data = rf_we_wb;
57                  5'd13: check_data = rf_wd_wb;
58                  5'd14: check_data = rf_rd0_fe;
59                  5'd15: check_data = rf_rd1_fe;
60                  5'd16: check_data = rf_rd0_fd;
61                  5'd17: check_data = rf_rd1_fd;
62                  5'd18: check_data = stall_if;
```

```
63                  5'd19: check_data = stall_id;
64                  5'd20: check_data = stall_ex;
65                  5'd21: check_data = flush_if;
66                  5'd22: check_data = flush_id;
67                  5'd23: check_data = flush_ex;
68                  5'd24: check_data = flush_mem;
69              endcase
70          end
71      endmodule
72
73      module Check_Data_SEG_SEL (...);
74          ...
75          always @(*) begin
76              check_data = 0;      // Default value
77              case (check_addr):
78                  3'd0: check_data = check_data_if;
79                  3'd1: check_data = check_data_id;
80                  3'd2: check_data = check_data_ex;
81                  3'd3: check_data = check_data_mem;
82                  3'd4: check_data = check_data_wb;
83                  3'd5: check_data = check_data_hzd;
84              endcase
85          end
86      endmodule
```

你也可以根据需要自行修改内部的相关端口或编号。

6.3 PDU 指令扩充

本次实验中，我们增加了 Check_addr[7:5] 作为段间寄存器的选择信号。为了便于大家输入，PDU 的 ck0 指令在原有指令的基础上进行了扩充：

- ck00 XX; 该指令会将 Check_addr 设置为 0x0000_0000 + XX，对应查看 IF 段的内容；
- ck01 XX; 该指令会将 Check_addr 设置为 0x0000_0020 + XX，对应查看 ID 段的内容；
- ck02 XX; 该指令会将 Check_addr 设置为 0x0000_0040 + XX，对应查看 EX 段的内容；

- ck03 XX; 该指令会将 Check_addr 设置为 0x0000_0060 + XX，对应查看 MEM 段的内容；
- ck04 XX; 该指令会将 Check_addr 设置为 0x0000_0080 + XX，对应查看 WB 段的内容；
- ck05 XX; 该指令会将 Check_addr 设置为 0x0000_00a0 + XX，对应查看 Hazard 部分的内容；

例如，如果我现在想要查询 EX 段正在执行的指令是什么，通过查表可以得到其模块内部编号为 1，在原来的 ck0 指令中，我们需要输入 ck0 41 进行设置（注意到 41[7:5]=010，对应着 EX 段）；而现在则可以通过输入 ck02 01 进行设置。当然，原来的 ck0 41 命令依然可以正常使用。

6.4 顶层模块

本次实验中顶层模块中的端口、连线与通信关系和上一个实验中相同，同样通过地址映射实现到 led 外设的连接，详见 Lab4 的文档，这里附上完整的流水线 CPU 数据通路（此图片也在我们提供的 figs 文件夹中）：

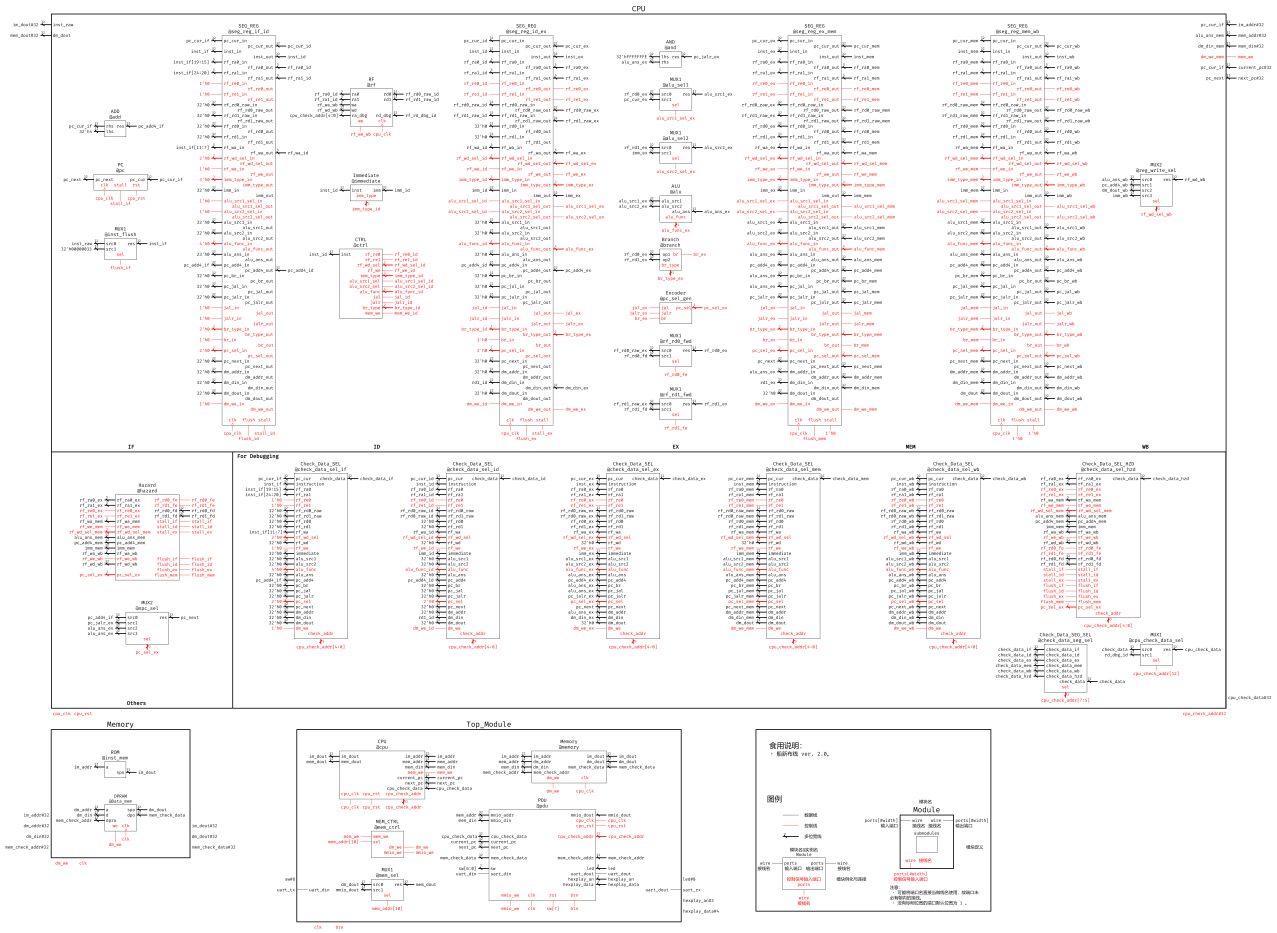

图 14: 流水线 CPU 数据通路

再次强调，不要被看似复杂的数据通路吓到。只要正确完成单周期 CPU，按照数据通路连线后，几乎只有冒险处理单元需要仔细考虑设计。

7. 实验任务

本次实验所需完成的各项工作介绍如下：

【必做部分】

1. **根据流水线 CPU 数据通路，结构化描述流水线 CPU，并进行功能仿真。** 请结合我们提供的数据通路图与所学知识，在单周期 CPU 的基础上搭建流水线 CPU。

   本次实验的流水线 CPU 需要支持以下 10 条指令的功能：add、addi、lui、auipc、beq、blt、jal、jalr、lw、sw。在内存单元方面，你需要例化 ROM 作为指令存储器，例化 DRAM 作为数据存储器。两个存储器的容量均为 256x32bits。

关于功能仿真，你可以自行设计一些简单的汇编程序，导出 COE 文件进行初步验证。我们为你准备了包含 CPU 和 MEM 的仿真文件 CPU_tb.v，你可以通过该文件检查自己的 CPU 运行情况是否正常。（仿真时记得用 COE 文件初始化存储器）当仿真部分通过后，再使用我们提供的汇编程序进行上板测试。

> 为了与 PDU 相连，寄存器堆与存储器的基本要求与上一个实验中相同，而 CPU 中需要进行的连接可以参考 6.1 小节的内容进行实现。

2. **使用本次实验提供的测试程序生成的 COE 文件作为指令存储器的初始化文件，验证 CPU 的功能正确性。** 本次实验中，我们为你提供了几个正确性验证程序（Testcase），你可以在附件中找到它们。这些测试程序将会检测你的 CPU 设计是否存在漏洞，并给出相应的检测结果。将包含 PDU 的整个项目下载至 FPGA 中测试，采用串口调试功能查看测试程序的运行结果。 **pipeline_test.asm 包含前三个测试，因此最终检查时只需检查该测试通过即可，无需重复烧写。**

> **推荐大家以如下顺序完成实验：**
>
> - 将单周期 CPU 拆分为各段，并添加段间寄存器。
> - 对单周期 CPU 的模块（如寄存器堆）进行调整，以通过基础测试 simple_test.asm。
> - 将冒险处理模块的控制部分完成，以通过控制冒险测试 control_test.asm。
> - 进一步完成冒险处理模块，以通过数据冒险测试 data_test.asm。
> - 用 pipeline_test.asm 进行必做部分最终测试。
> - 阅读附加部分文档并量力完成选做。

【选做部分】

1. **提前控制指令。** 参考 3.3 小节的介绍，将控制指令 jal 提前到 ID 阶段完成，并对应调整冒险处理模块，使得其顺利完成控制冒险测试。

2. **程序结束的实现。** 参考 3.4 小节的介绍，正确实现 ebreak 指令的处理。

   若将 ebreak 作为程序结束的标志，我们期望看到的结果是：CPU 运行到 ebreak 指令后，PDU 能够将状态跳转到调试状态。此时 led4 亮起，数码管显示 Check_addr 所对应的 Check_data 的结果。

你也可以将 ebreak 设置为真正的断点指令。当 CPU 运行至 ebreak 指令、PDU 状态正确跳转后，如果此时设置断点并连续运行，则 CPU 会在下一次 ebreak 指令处或断点地址处再次停下。

这两种实现方式在分数上没有差异。为了实现本项选做，你需要修改 CPU、PDU 中的部分代码，以实现 PDU 状态机的正确跳转。理论上，在正确修改状态跳转的代码后，程序应当能够直接实现 ebreak 的断点功能。提示：与此相关的部分代码在 PDU_Ctrl.v 文件中。

大家或许注意到了，扩展指令并没有算作流水线部分的选做得分，这是由于最后的**综合实验**即为对流水线进行改进，实现更多功能，因此较复杂的选做均算作了综合实验的内容。下面给大家一些参考，具体要求、讲解与评分标准均以 Lab6 实验文档为准：

- 在原有的 10 条指令的基础上，扩展完成 RV32I 的其他指令。
- **手动**搭建乘法/除法器并实现乘/除法（不允许直接利用 Vivado 的乘除）。
- **手动**搭建浮点数运算器并支持浮点运算（同上）。
- 对更多中断、异常处理的支持。
- 对显示器显示的支持。
- 运用分支预测技术提升 CPU 性能。
- 运用高速缓存技术提升 CPU 性能。

本次实验需要大家在实验平台上在线提交相关内容。你提交的文件结构应当满足下面的文件树格式：

```
/
└── lab5_[姓名]_[学号]_ver[尝试编号]
    ├── figs ..................................... 图片文件夹，如果没有可以无此文件夹
    ├── lab5_[姓名]_[学号].pdf ..................... 实验报告文件
    ├── src ..................................... 需要提交的相关程序文件夹
    │   ├── Module_name.v ..................... 非仿真 .v 文件
    │   ├── Program_name.asm ..................... 汇编源程序文件
    │   └── ...
    └── others ..................... 其他你打算提交的文件，如果没有可以无此文件夹
```

请将全部文件按照上面的格式压缩成一个文件，提交到实验平台上。

请确保你的实验报告至少包含以下的内容：

- 实验原理。请根据自己的理解描述本次实验的实验内容以及设计流程，包括部分模块的设计思路，如 Hazard Unit 等，若实现了选做部分，请简述选做的实现思路；
- 估算你实现的流水线 CPU 的最小需要时钟周期与其相对单周期的指令平均所需时间的改进（自选数据大致估算即可，无需精确计算）；
- 实验过程中遇到的一些问题，或者难以解决的内容。你可以记录自己试错的过程，也可以展开自己的心路历程（本项内容不作为评分依据）。

实验手册中有一些我们为大家列出的思考点。这部分内容无需在实验报告上列出。此外，我们也欢迎大家在实验报告中给出对于本次实验的反馈。

> 实验检查与报告提交的 DDL 按照各班各组的约定设置。超出 DDL 的检查与提交将按照规定扣除部分分数。请保证个人实验的独立完成!

8. 附件

本次实验所提供的相关文件如下：

```
/
└── lab5_files
    ├── figs ..... 图片文件夹
    │   ├── Datapath.png ..... 我们为你绘制的流水线 CPU 数据通路图
    │   └── （顶层模块示意图同Lab 4，不再给出）
    ├── TOP ..... 项目文件，包含了 PDU 以及其他框架文件
    │   └── ...
    ├── CPU_src ..... CPU 的部分代码
    │   ├── 仿真
    │   │   └── CPU_tb.v ..... 包含 CPU 和 MEM 模块的仿真文件
    │   └── 模块文件 ..... 一些你可能会用到的模块文件
    │       └── ...
    ├── Testcase ..... 测试程序文件夹
    │   ├── simple_test.asm ..... 无冒险的测试
    │   ├── control_test.asm ..... 只含控制冒险的测试
    │   ├── data_test.asm ..... 只含数据冒险的测试
    │   ├── pipeline_test.asm ..... 完整的汇编测试
    │   └── ebreak_test.asm ..... 程序结束测试（选做 2 用）
    └── PDU 指令手册.xlsx ..... 详细的 PDU 指令手册
```

睿客网盘链接：https://rec.ustc.edu.cn/share/70f74240-ebda-11ed-99ef-63e4c539b0ec

考虑到我们可能会更新附件内容，请大家定期关注群聊中的消息。我们会在此链接中进行文件更新。