****

**《计算组成原理实验》**

**实验报告**

**（实验三）**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学院名称** | **：** | 计算机学院 | | | | | |
| **专业（班级）** | **：** | 21级行政1班 | | | | | |
| **学生姓名** | **：** | 黄鑫 | | | | | |
| **学 号** | **：** | 21307008 | | | | | |
| **时间** | **：** | 2023 | 年 | 1 | 月 | 11 | 日 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | **成绩** | **:** |  |
| **实验三** | **：** | **流水线寄存器（YstCPU Risc-V）** | | | |

**目录**

[第一章 实验目的 4](#_Toc125056955)

[第二章 实验器材 4](#_Toc125056956)

[第三章 实验任务 4](#_Toc125056957)

[第四章 实验原理与分析 4](#_Toc125056958)

[1. 流水线技术 4](#_Toc125056959)

[2. 流水线寄存器 5](#_Toc125056960)

[3. 三级流水线 5](#_Toc125056961)

[4. 五级流水线 6](#_Toc125056962)

[5. 五级流水线与数据冲突与阻塞 7](#_Toc125056963)

[6. 五级流水线与阻塞与旁路 7](#_Toc125056964)

[7. 五级流水线与提前分支（Final） 9](#_Toc125056965)

[第五章 实验过程与结果 10](#_Toc125056966)

[1. PipelineRegister 10](#_Toc125056967)

[2. ThreeStageCPU 10](#_Toc125056968)

[3. FiveStageCPU Stall 11](#_Toc125056969)

[4. FiveStageCPU Forwarding 11](#_Toc125056970)

[5. FiveStageCPU Final 13](#_Toc125056971)

[6. 标准测试全部通过 19](#_Toc125056972)

[第六章 俄罗斯方块 20](#_Toc125056973)

[第七章 实验心得 22](#_Toc125056974)

[1. 细节 22](#_Toc125056975)

[2. 竞争冒险 22](#_Toc125056976)

[3. 使用msys2下载工具的旧版本 22](#_Toc125056977)

[4. ForwardingCPU的研究 22](#_Toc125056978)

# 实验目的

1. 使用流水线设计缩短关键路径
2. 正确处理流水线阻塞与清空
3. 使用转发逻辑减少流水线阻塞
4. 烧板子跑俄罗斯方块

# 实验器材

电脑一台，Intellij IDEA 软件一套，Basys3板一块。

# 实验任务

lab3中的实验任务如所示：

1. 我们定义一个带参数的 PipelineRegister 模块，来实现不同数据位宽的流水线寄存器。
2. 用一个控制单元来检测控制冒险并清空流水线，来解决三级流水线CPU中的冒险。
3. 用一个控制单元来检测并解决控制冒险和数据冒险，来将CPU拓展成五级。
4. 继续拓展，增加旁路功能，减少阻塞的发生。
5. 继续拓展，将分支/跳转指令的执行从 EX 段提前到 ID 段。

PipelineRegister单元的代码文件位于 src/main/scala/riscv/core/PipelineRegister.scala

三级流水线CPU的代码文件位于 src/main/scala/riscv/core/threestage

剩余三个五级流水线CPU 的代码文件分别位于 src/main/scala/riscv/core中的：

fivestage\_stall、fivestage\_forward、fivestage\_final。

在上面提到的单元的相应文件里面，请在各自的注释处填入相应的代码，使其能够通过各自的测试。

# 实验原理与分析

本实验里我们将实现流水线寄存器。相关介绍：

## 流水线技术

在单周期的 CPU 设计中，关键路径太长，频率难以提升，并且每个时钟周期只能执行一条指令，指令吞吐率低。下面，我们将尝试使多条指令重叠执行（即流水线技术）来解决这个问题。

流水线技术即是将指令执行的五个阶段（IF/ID/EX/MEM/WB）各自独立出来，成为五个单元，让每个时钟周期中一条指令只能占用一个单元，这将使得CPU同时最多可并行五条指令。流水线技术的有效之处在于，它将时钟周期缩小为关键阶段的执行时间，然后通过并行执行指令的方式，使得每条指令的理想CPI无限趋近于1（指令数趋近）。

竞争冒险的处理是流水线 CPU 设计的难点和关键所在。流水线 CPU可能的竞争冒险有三种：

1. 结构冒险。即IF和MEM模块同时访问同一处内存地址造成的冲突。本实验默认使用了InsCache和DataCache解决了冲突。
2. 数据冒险。即当前指令在要使用的数据依赖于上条指令的执行结果，称当前指令为依赖指令，另一条为被依赖指令。数据按照来源可分为寄存器与内存。对于后者，由于读写内存都只能发生在MEM模块，因此不可能产生数据冒险。对于前者，本实验已实现同一时钟周期读写寄存器组先写读后写的功能，因此可以避免一部分的RAW数据冲突。其它的数据冲突将在下文分析，并使用旁路与阻塞技术处理。
3. 控制冒险。发生在跳转指令（B指令、jal、jalr），当确认跳转时，需要清空CPU中已经开始执行的指令。

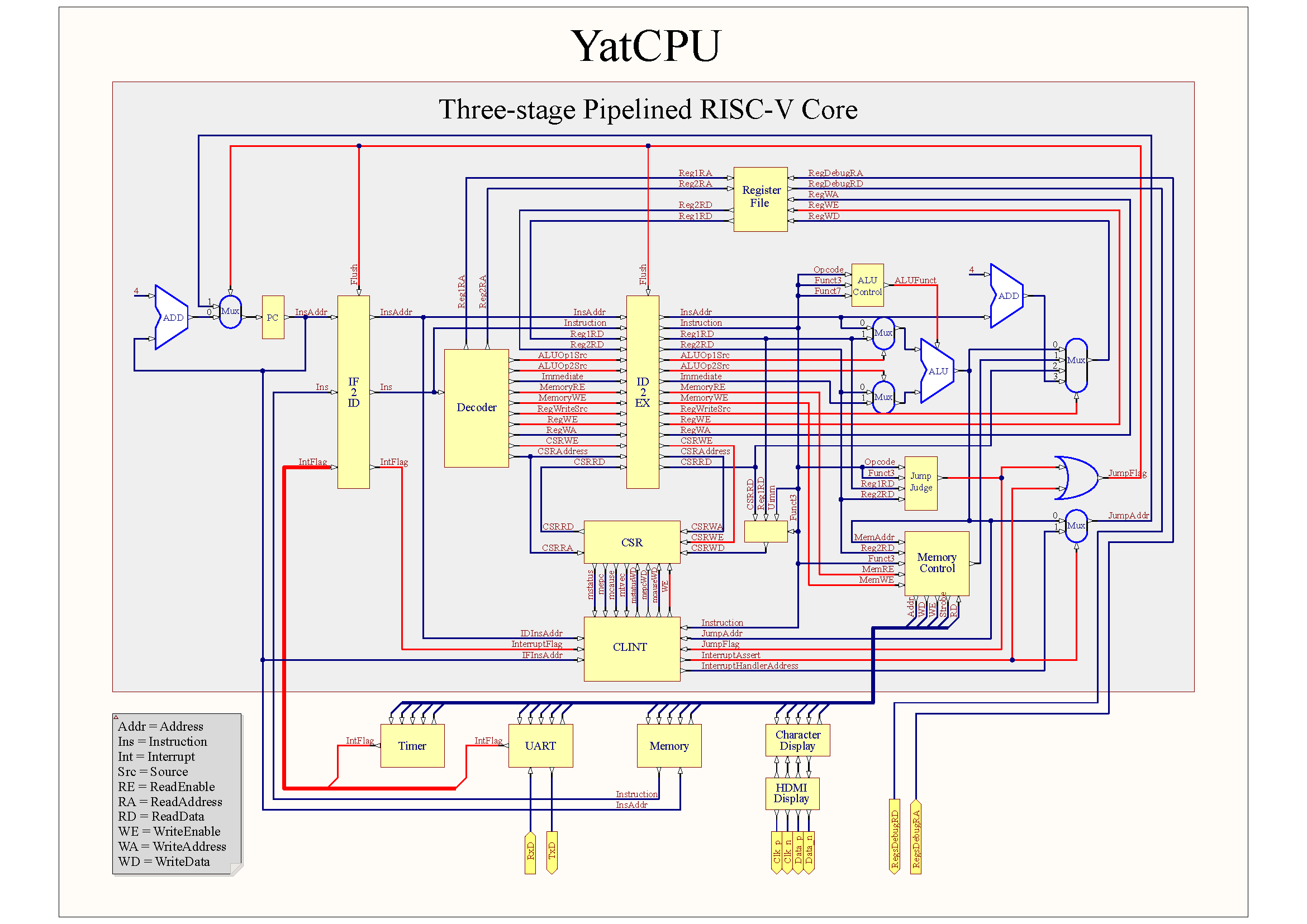
在下面的实验中，我们首先设计一个简单的三级流水线 CPU（IF、ID 和 EX 三级），它只涉及分支和跳转指令带来的控制冒险，处理起来较为简单；然后，我们再将三级流水线 CPU 的 EX 级继续切分为 EX、MEM 和 WB，形成经典的五级流水线，这样做带来的数据冒险需要使用阻塞和转发技术进行处理；最后，我们将分支和跳转提前到 ID 阶段，进一步缩短分支延迟。

## 流水线寄存器

要实现流水线的“五个阶段用五个模块处理且指令以此向后续阶段执行”的功能，我们就需要在每两个阶段之间增加一个寄存器组来存储来自上一阶段且后续阶段需要的数据信息，这样的寄存器组称为流水线寄存器。

除了暂存数据信息的功能外，出于处理竞争冒险的目的，流水线寄存器还需要有阻塞和清空的功能。

## 三级流水线



这是三级流水线的架构图，展示了数据通路与控制信号。虽然看起来比上一个实验的复杂，但其实只是多了两个流水线寄存器。控制信号与模块接口比较易读，此处不再赘述。

lab中已经将连接流水线寄存器各种接口的体力活做好了，任务只有处理竞争冒险，而这正是理解流水线CPU的重中之重。

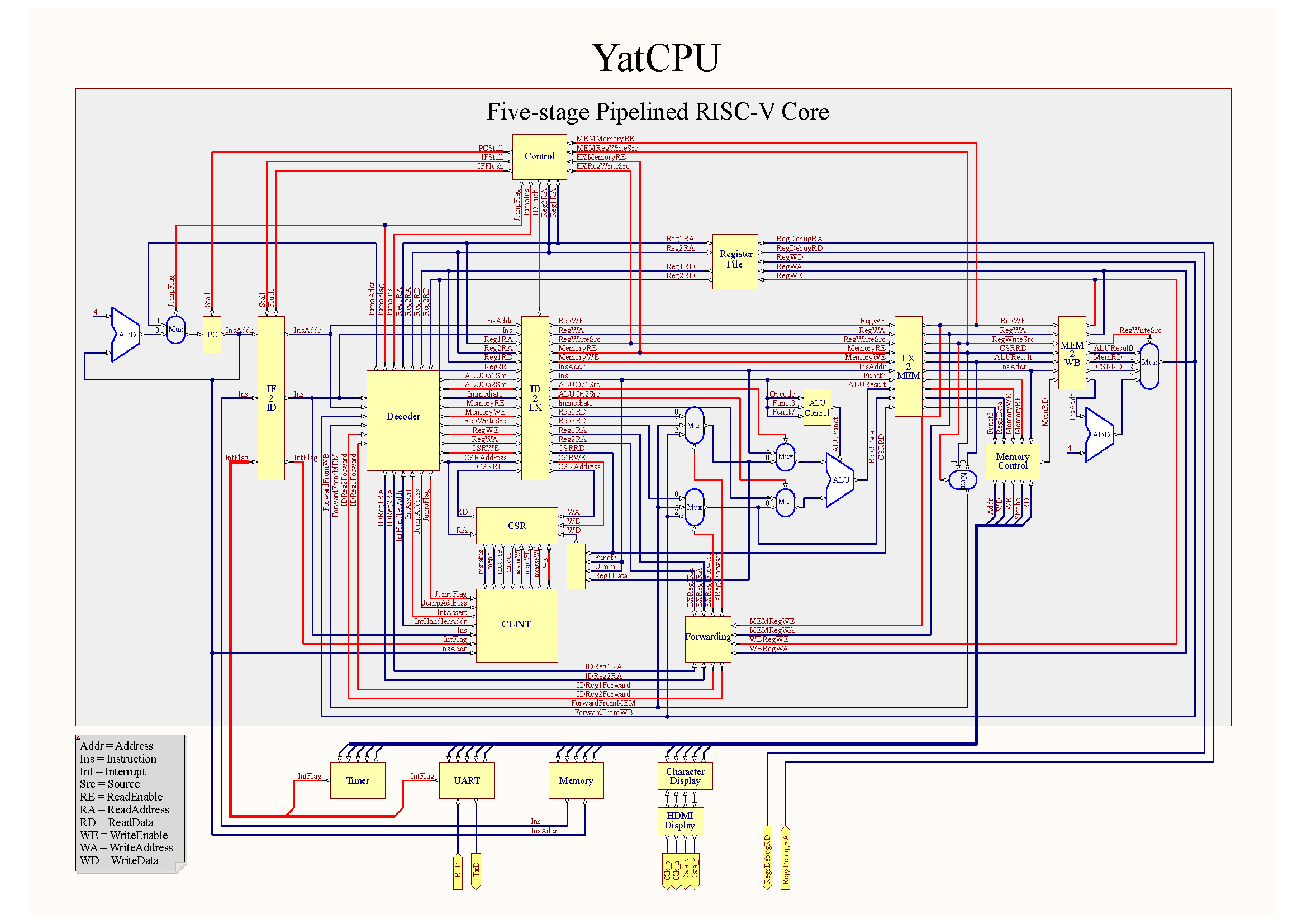
在三级流水线中，由于所有数据处理操作都在 EX 阶段进行，因此不存在数据冒险，我们只需要处理程序跳转带来的控制冒险。有三种情况可能发生程序跳转：

1. EX 段执行到跳转指令
2. EX 段执行到分支指令且分支条件成立
3. 发生中断，EX 段收到 CLINT 发来的 InterruptAssert 信号，这相当于在 EX 段的指令之上叠加了一条跳转指令，EX 段的指令继续执行，IF 段和 ID 段的指令将被丢弃

无论哪种情况，都是由 EX 段向 IF 段发送跳转信号 jump\_flag 和跳转的目标地址 jump\_address，但在 jump\_address 写入 PC 并从该处取出指令前，流水线的 IF 和 ID 段已经各有两条不需要执行的指令，好在这两条指令的结果还没有写回，我们只需要清空对应的流水线寄存器，把它们变成两条空指令即可。

## 五级流水线

在三级流水线中，执行阶段逻辑复杂，仍然可能导致较大的延迟。为了进一步缩短关键路径，我们可以扩展流水线级数，将执行阶段进一步分为 ALU 阶段、访存阶段以及写回阶段，如下图所示。



把三级流水线进一步分割为五级流水线将带来更加复杂的数据冒险，下面我们将尝试使用阻塞的方式（Control单元）解决数据冒险，得到一个功能完整的五级流水线 CPU。接着我们可以使用旁路（Forwarding单元）和将分支跳转提前到 ID 阶段（Control单元）进一步提升 CPU 效率。

## 五级流水线与数据冲突与阻塞

我们先学习阻塞，并使用阻塞来解决数据冲突。

阻塞的理想效果是，ID模块及其前面的内容保持一个时钟周期不变，同时产生EX气泡，后续模块照常执行。具体到控制信号则是，令pc和if2id寄存器stall，令if2ex寄存器flush。这里一定要想清楚，令pc和if2id寄存器stall的效果是，IF和ID模块在下个时钟周期执行的指令与当前周期相同；令if2ex寄存器flush的效果是，EX模块什么都不执行（或者执行无关的缺省行为），我们不在乎气泡中EX模块的执行结果，但我们需要避免EX模块原有的执行结果对程序有影响，因此要清空if2ex寄存器。

接下来是判断何时发生数据冲突。数据冲突分为三种：WAW、WAR、RAW。本实验的指令执行时间都相等，而且没有默认没有编译优化，因此不会出现WAW和WAR。RAW分为三种情况：id\_rs === ex\_rd、id\_rs === mem\_rd、id\_rs === wb\_rd，而最后一种因为RegisterFile已经通过硬件实现了同周期先写后读解决了。

综上，只需要当id\_rs === ex\_rd或id\_rs === mem\_rd发生且rs\_ex不是zero寄存器时，执行一次阻塞即可。

## 五级流水线与阻塞与旁路

直至目前，我们的流水线CPU已经可以正常工作了。在发生数据冲突时，CPU会将ID模块及其之前的指令一直阻塞住，直到被依赖的指令到达WB模块将数据写入寄存器。但是，如果我们使用旁路技术，就可以不用等被依赖指令到达WB模块就拿到所需的数据。

（1） 旁路技术，是指通过硬件技术将已经产生但尚未写入寄存器的值，提前送到依赖指令处。我们规定，只要满足以下条件，旁路就需要将数据运送：

1. rs\_ex不是zero寄存器（写zero寄存器操作相当于不存在数据冒险，需要特判，否则当依赖于load指令时还会出现不需要的阻塞）
2. rd\_mem与rs\_ex相等且RegWE\_mem打开（或rd\_wb与rs\_ex相等且RegWE\_wb打开）。

可以发现，旁路forwarding单元只负责数据的传送，而至于控制信号的产生（阻塞和跳转时的清空）则交给Control单元。这和它们的目标——数据和控制竞争冒险，是相符的。

与旁路技术相关的指令有R-type（I\M\B\S\CSR情况同R）、load类和jal类，我们以add、lw、jal这三条指令为例进行分析。

1. 先看add和lw。研究旁路的源头和目的地，从依赖与被依赖两方面分析。
   * 依赖指令只会在EX模块中使用依赖的数据，因此我们的旁路只需要将数据送至EX模块；
   * 被依赖数据产生的可能位置有EX或MEM。产生于EX例如add指令，数据存进ex2mem寄存器，在MEM段取得；产生于MEM例如lw指令，数据存进mem2wb寄存器，在WB段取得（这是有意义的，因为若ID依赖MEM，则下一周期EX在旁路前用的仍是旧的错误数据）。
     + 如果依赖数据产生于EX模块，指令分布有四种可能：
       - 两条指令紧邻，从MEM取得；
       - 隔一条指令后才出现依赖指令，从WB中取得；
       - 当前指令同时依赖前两条指令，这时我们应该使用最新的依赖数据，从MEM中取得；
       - 隔两条指令及以上，从RegFile取得。
     + 如果依赖数据产生于MEM模块，一律从WB中取得。指令分布有三种可能：
       - 两条指令紧邻，阻塞一个周期直到依赖数据产生；
       - 隔一条指令后才出现依赖指令，无需阻塞；
       - 隔两条指令及以上，从RegFile取得。

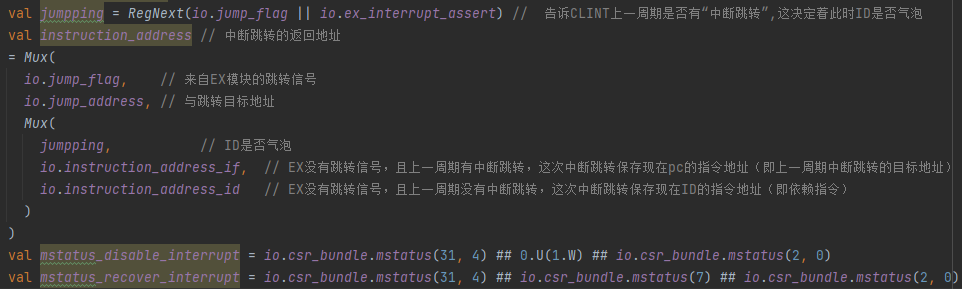
总结出规律：

1. 旁路数据来自MEM或WB，输送给EX模块。
2. 有数据冲突就搭建旁路，有两条旁路时，来自MEM的优先。
3. 被依赖指令为load指令，且两指令紧邻，阻塞一个时钟周期。
4. 再来看jal指令。跳转时我们需要清空CPU，但jal指令有一个写寄存器的操作，为保证指令正常执行，跳转只需要清空前两个流水线寄存器即可。

阻塞和跳转清空流水线寄存器都是Control单元的任务。现在分析在同时满足阻塞和清空的条件的情况下，Control单元应该采取的行为：

1. 清空的条件是EX模块的if\_jump\_flag信号为真，即（clint跳转信号 或 EX模块为跳转指令 或 EX模块为分支指令且分支判断）为真。
2. 阻塞的条件是ID模块的rs等于EX模块的rd，且EX模块为Load指令。

两者同时成立只有一种可能——即此时EX模块为Load指令，ID模块的指令依赖Load指令，CLINT发出跳转信号。那么这种情况Control单元究竟该阻塞，还是清空，还是同时阻塞清空呢？（虽然测试都能过，不过我在CLINT的代码中发现，助教OnewayLab已经考虑到了这个问题。）见源码图：



按照CLINT的实现，中断程序结束后将回到此时ID模块的指令，也就是说，此时Control单元应该将ID模块清空（即清空id2ex）且不阻塞。（不用担心ID指令的rs，中断会保护现场的）

## 五级流水线与提前分支（Final）

截至目前，我们已经把阻塞与旁路技术都运用上了，实现了一个解决了全部竞争冒险的流水线CPU。但你也许对与目前流水线的效率还是不太满意，因为每次跳转都需要浪费两个时钟周期。接下来，我们往 CPU 中加入少量硬件，将分支/跳转指令的执行从 EX 段提前到 ID 段，进而把程序跳转的损失减少到一个时钟周期。

首先，我们需要把跳转的判断从 EX 模块移到 ID 模块。

其次，跳转目标地址的计算是使用 EX 段的 ALU 进行的，因此我们需要给 ID 段增加一个加法器来计算目标地址。

最后，我们需要添加额外的旁路逻辑（毕竟jalr和B-type都有rs），将前面指令的执行结果旁路到 ID 段给分支或跳转指令使用；如果所依赖的结果还没有产生，还需要进行阻塞

下面来重点分析一下**Control单元**的工作。将中断跳转、跳转/分支、依赖（数据冲突）和Load指令之间可能的关系当成自变量，将Control的操作当成因变量：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 中断 | 跳转/分支 | 依赖 | Load | Control |
| 0 | 0 | 1 | 0 | -（旁路即可） |
| 0 | 0 | add紧邻依赖lw | | 阻塞 |
| 0 | 1 | 0 | 0 | 跳转清空 |
| 0 | 1 | 0 | 1 | 跳转清空 |
| 0 | 1 | add紧邻依赖非Load | 0 | 跳转清空 |
| 0 | 跳转/分支指令紧邻依赖非Load | | 0 | 阻塞 |
| 0 | 1 | add紧邻依赖lw | | 阻塞+跳转清空 |
| 0 | 跳转/分支指令紧邻依赖非Load | | 1 | 阻塞+跳转清空 |
| 0 | 跳转/分支指令紧邻依赖Load | | | 第一个周期：阻塞  第二个周期：阻塞+跳转清空 |

注：1.阻塞，阻塞方式是一个EX气泡。

2.Final的跳转清空效果是**下一周期**有ID气泡。

3.中断其实不影响测试。

其实研究到这里测试已经通过了，此处便不再分类讨论有中断的情况……（参见我发起的讨论[lab3中的中断跳转与清空](https://github.com/SYSU-SCC/yatcpu-docs/discussions/15)）但是可以额外讲讲中断发生时的流程。

中断发生时，CLINT接收到中断信号时，首先为进入中断处理程序做准备，准备工作包括确定中断返回地址（Decoder会为CLIINT提供指令跳转信号和指令跳转目标地址）：

1. 若此时ID段有指令跳转信号，则CLINT就会保存ID段的指令跳转目标地址。
2. 若此时ID段没有指令跳转信号，则CLINT就会保存当前的PC值，即此时IF段执行的指令地址。

准备工作完成后，CLINT执行中断跳转。中断跳转的过程和指令跳转是一样的。CLINT会将中断跳转信号和中断跳转地址发给Decoder，然后Decoder将发出综合跳转信号给IF段和Control单元。Control单元的执行效果总是“下个周期一个ID气泡”，不论跳转原因是中断还是指令。

# 实验过程与结果

有了上面的铺垫后，补充代码逐步完善流水线寄存器就很容易了。

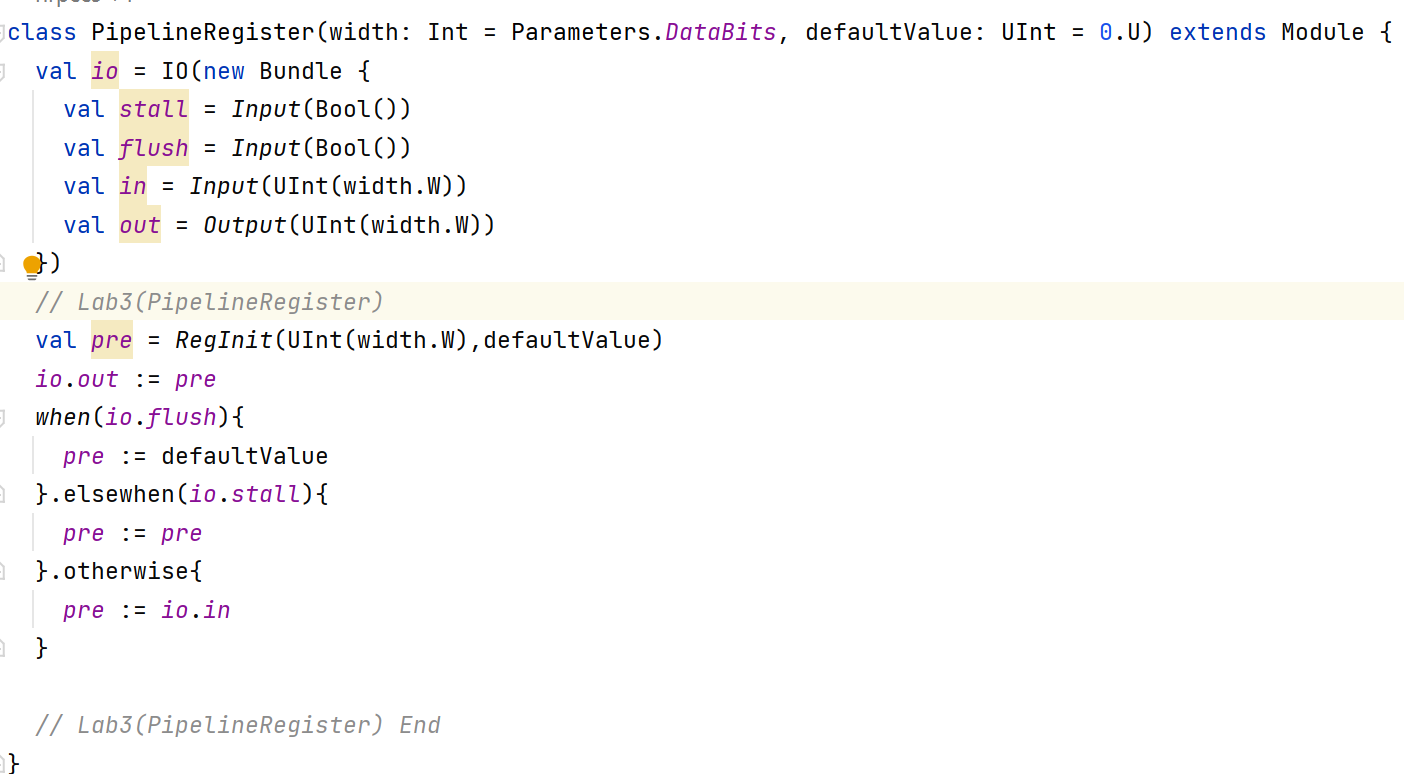
## PipelineRegister

[分析见上文](#_流水线寄存器PipelineRegister)。

清空的效果是将内容恢复为默认值（创建时设置）；阻塞的效果是保留原有内容；正常运行时，先输出存储的值，再存储输入的值。

这使得PipelineRegister就像一个回合制选择器，控制信号只有清空信号flush和阻塞信号stall，而行动有三种。

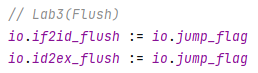
其中两个清空信号是我们在后续的竞争冒险处理中我们自行定义的，而我需要确保它们不会同时为真。



最后，不同模块之间需要过渡的数据不同，但我们又希望PipelineRegister模块能够复用，因此我们将PipelineRegister作为在具体的流水线寄存器的实现中每个单独数据的模板Module。

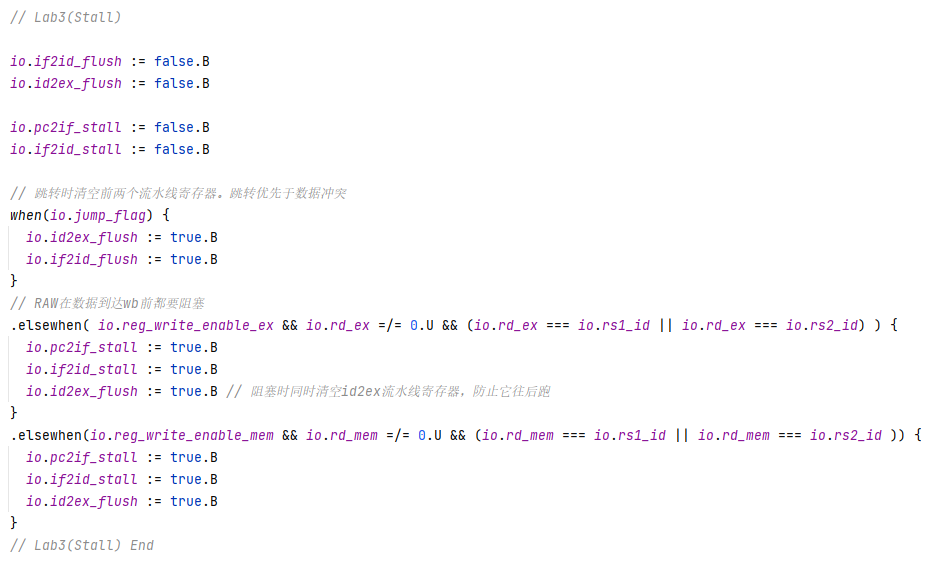
## ThreeStageCPU

按[分析](#_三级流水线)中说的，根据jump\_flag信号来清空两个流水线寄存器：



## FiveStageCPU Stall

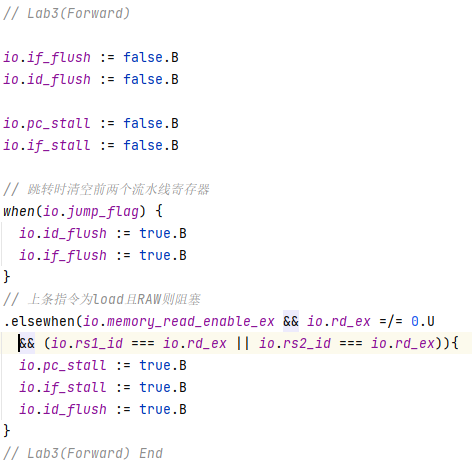
按[分析](#_五级流水线与数据冲突与阻塞)中说的添加功能，当id\_rs === ex\_rd或id\_rs === mem\_rd发生且rs\_ex不是zero寄存器时，执行一次阻塞：



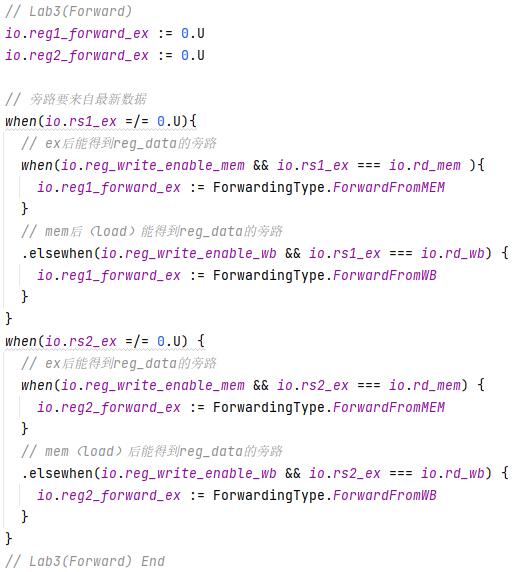
## FiveStageCPU Forwarding

[分析](#_五级流水线与阻塞与旁路)中繁杂思路已经说的很明白的，实现起来倒是简洁。

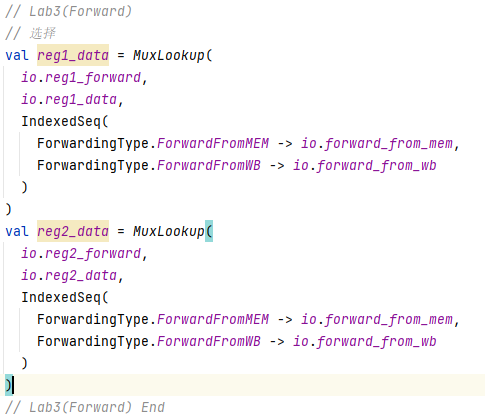
Control完成解决控制冲突的任务实现如下（相比于只有阻塞的CPU，阻塞的清空少了许多）：



而Forwarding完成了解决数据冲突的任务：



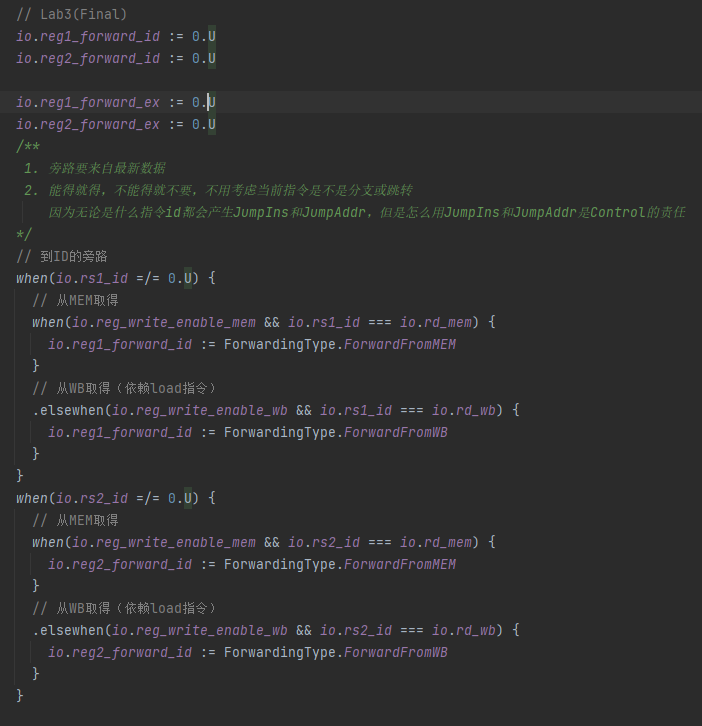
不要忘了在EX模块还要实现旁路的接口，因为每个RegData都有了三种情况：来自ID模块、来自ex2mem的旁路和来自mem2wb的旁路：

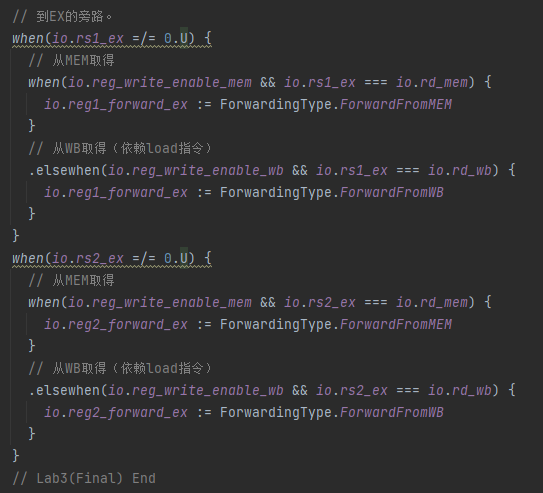


## FiveStageCPU Final

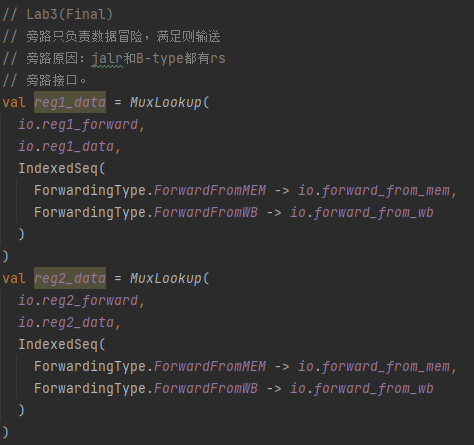
[分析](#_五级流水线与提前分支)中繁杂思路已经说的很明白的，但这次实现起来……。

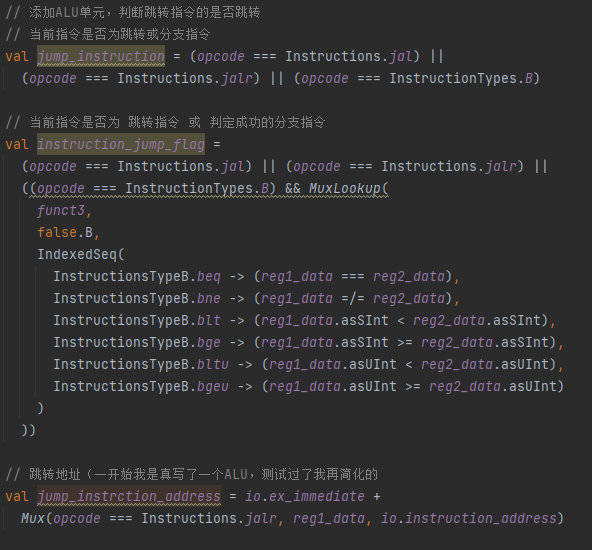
首先是逻辑上最易懂的**Forwarding**旁路单元：

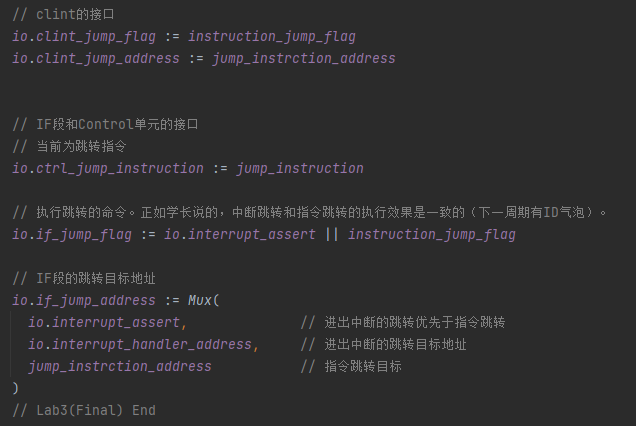




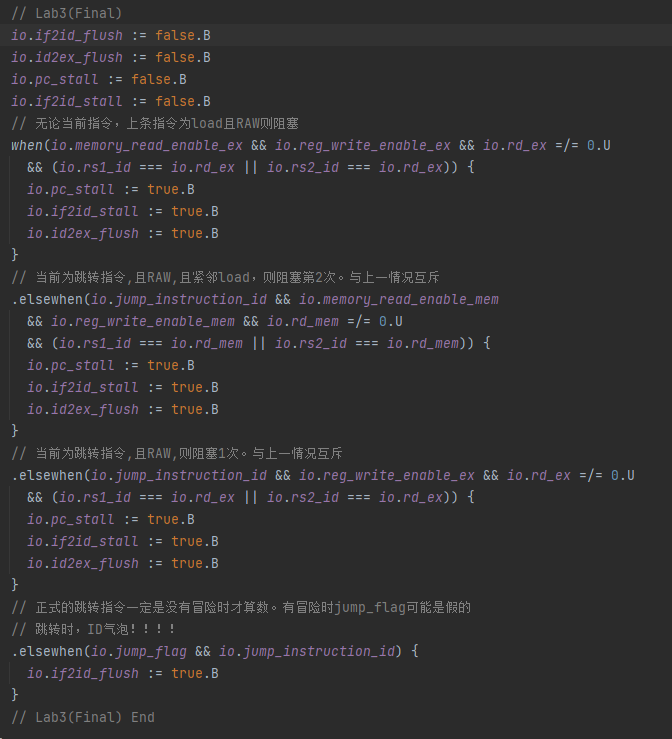
然后是**ID段**。我们将分支指令的判断时机从EX段提前到来ID段，因此ID在上一版CPU的基础上，增加了跳转的判断和对跳转目标地址的计算。而这又要求ID段也有一段旁路，需要增设旁路接口。



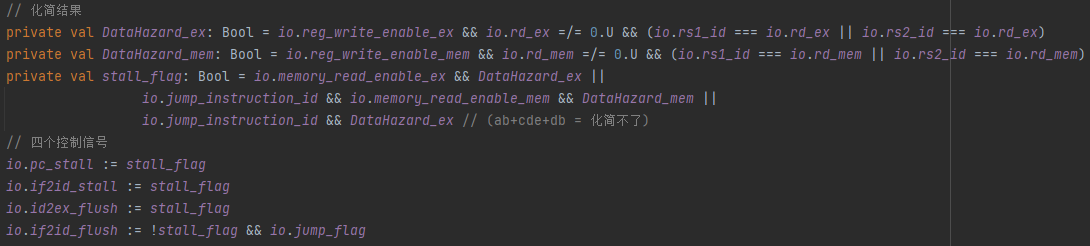


而ID段对中断跳转和指令跳转的流程理解是理解FinalCPU的重点。

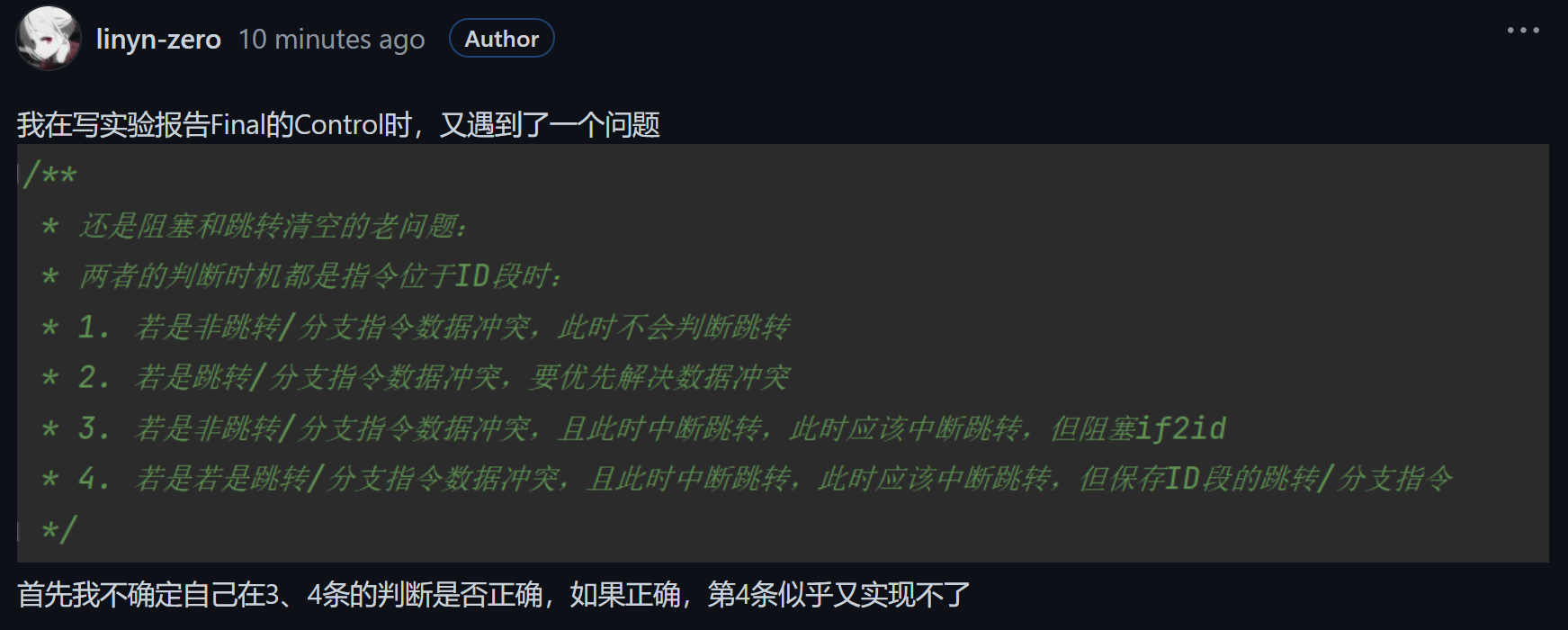
最后是**Control单元**，具体过程见[分析](#_五级流水线与提前分支（Final）)



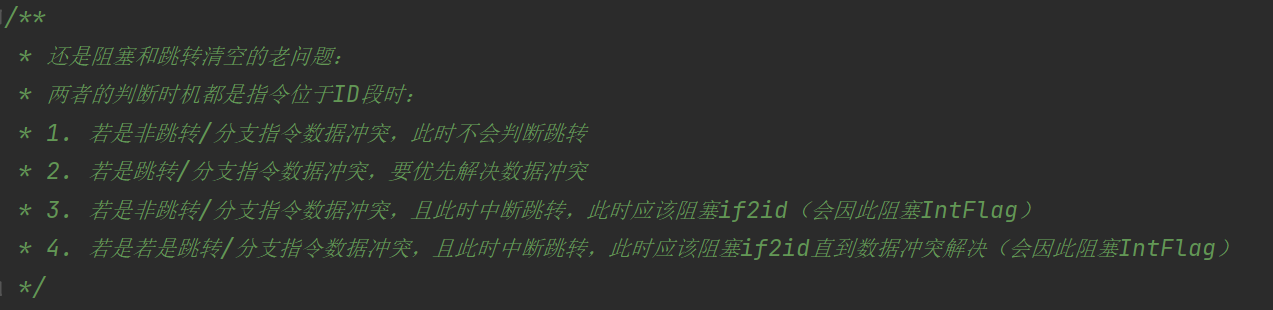
再化简一下：



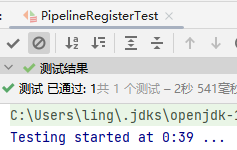
天呀中断越纠越复杂，本想研究一下学长是怎么实现中断的Control的，结果……不过与实验无关，把过程贴在这里：[lab3中的中断跳转与清空 · Discussion #15 · SYSU-SCC/yatcpu-docs (github.com)](https://github.com/SYSU-SCC/yatcpu-docs/discussions/15)。问题：

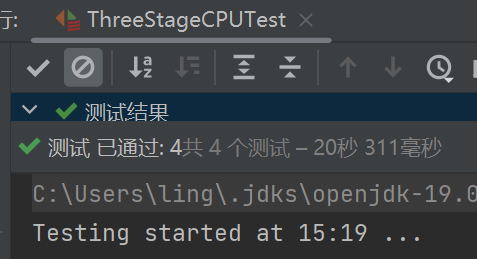


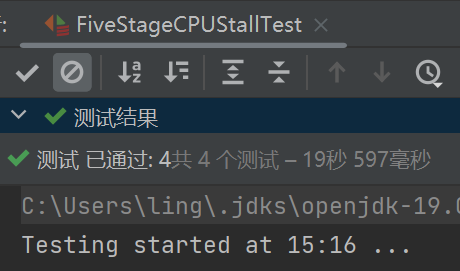
解决后：

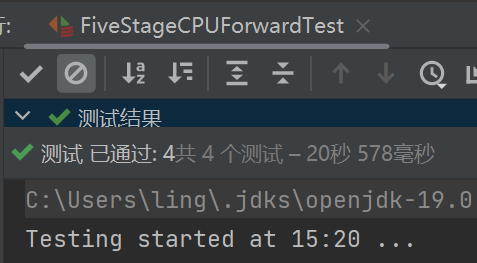


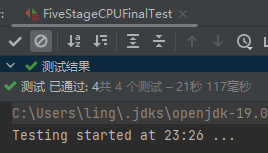
## 标准测试全部通过







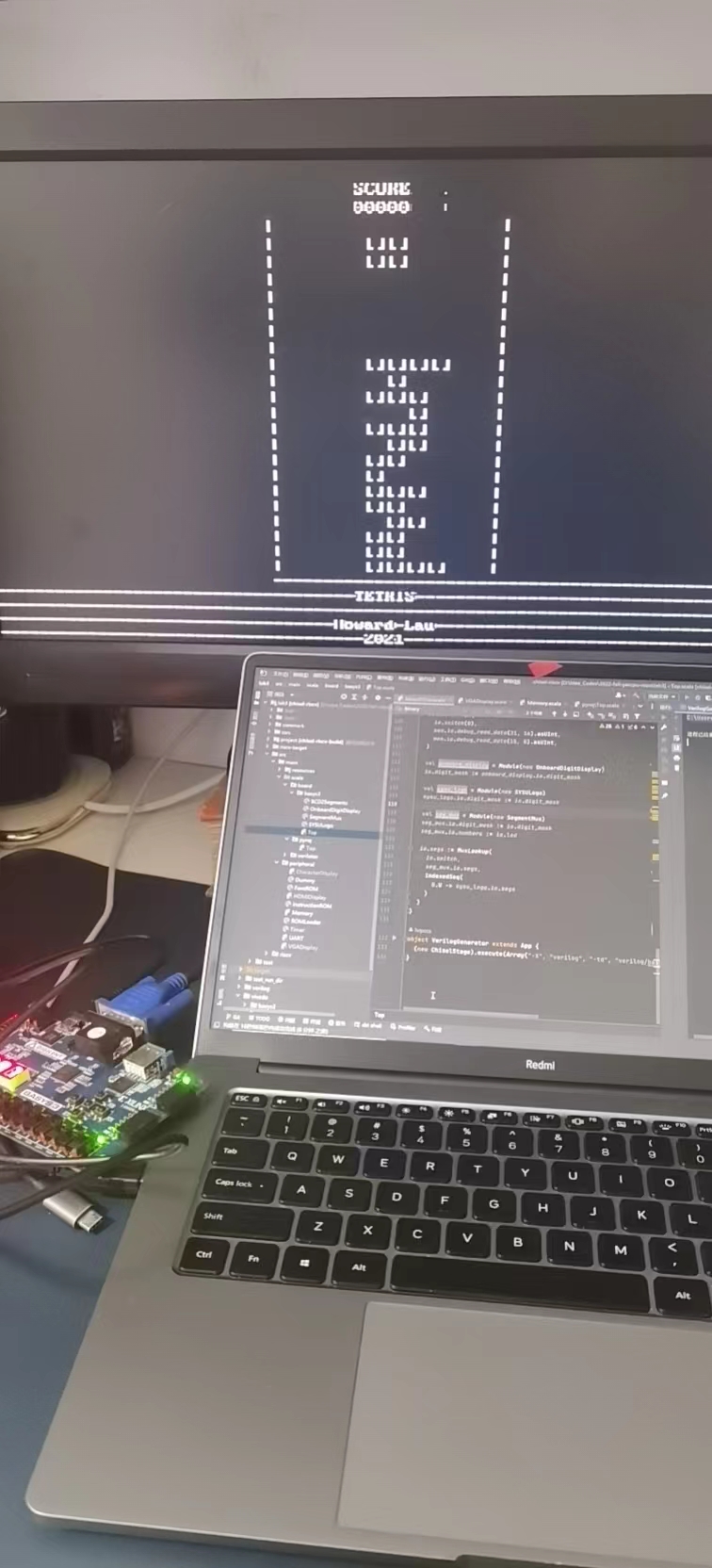




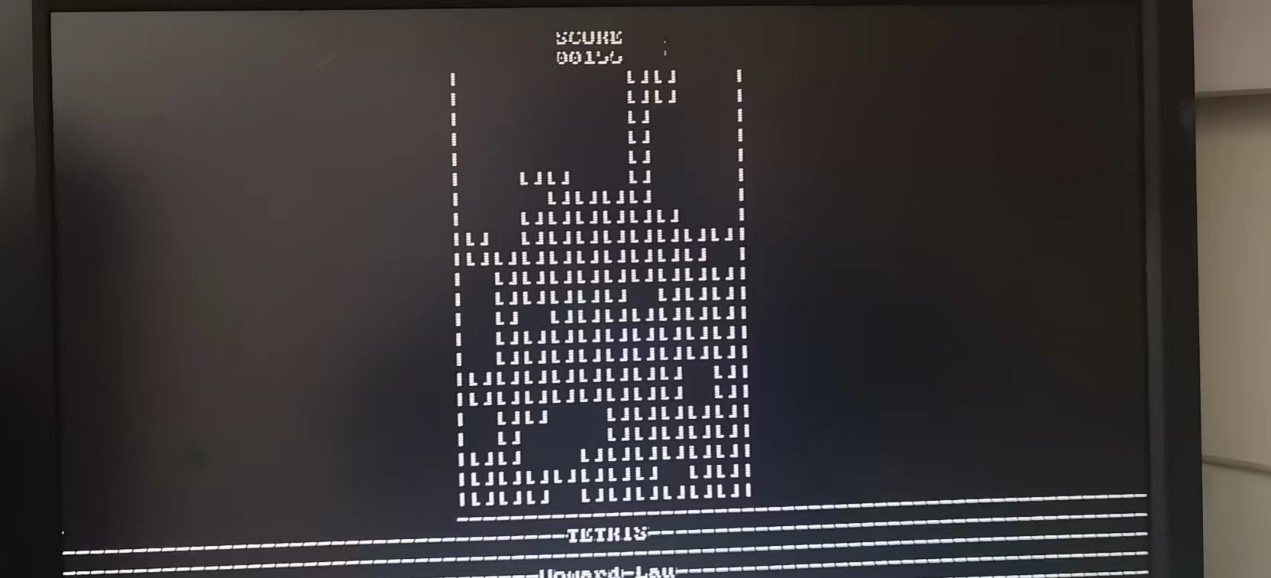
# 俄罗斯方块

终于到我心心念念的俄罗斯方块环节了。

烧板后，显示错误，我用lab的display等模块复刻在lab3后成功：



再用Xshell连接串口，能玩了！



# 实验心得

## 细节

我在通过PipelineRegisterTest后，执行ThreeStageCPUTest却出错了，研究结果是PipelineRegister的错误。不止我一个，身边的同学在这个地方发生的同样的错误。原因在我看来有以下几点：

* 1. 测试不够全面，无法检查PipelineRegister的完全正确时的执行效果；
  2. 我对PipelineRegister的执行流程、效果的理解不够到位。

PipelineRegister是如何工作的，它的出现是为了什么，它是先接收数据，还是先输出？这些问题足以体现一个工程师的底子。

## 竞争冒险

处理竞争冒险的一个很重要工作在于，将各种可能的冲突情况考虑清楚，这能体现一个工程师平日的知识储备，更能体现思考问题，穷尽可能的思维能力！

lab3的final真的卡我好久呀……

终于过了，方法在于排列组合……

## [使用msys2下载工具的旧版本](https://blog.csdn.net/Linyn_Zero/article/details/128716124?spm=1001.2014.3001.5501)

研究烧板的时候不小心把verilator更新了，结果最新版本有bug，折腾半天终于解决了，顺便出了篇博客，嘻嘻。要感谢助教大大howard给的方向。感觉又学到东西了！

## ForwardingCPU的研究

我花了2个小时研究“同时满足阻塞和清空的条件的情况下，Control单元应该采取的行为”，因为涉及了CLINT中断的部分，如risc-v中断指令，中断跳转前后操作等，因此比较深入。中途想要放弃来着，毕竟不影响测试，但后来坚持住分情况分析出来了！不过到了finalCPU中，跳转判断提前到了ID模块，势必又会影响CLINT的接口和实现逻辑，啊啊好烦……欸不对，反而变简单了！

1. final ID段的加法部件

这要感谢助教大大OnewayLab的指点，我才意识到不是测试能过就行的，还要出于实用性上，将实现方式最简化。（意思就是说能一行解决的问题就不要写个ALU在里面）

1. 后续没解决的问题

见[lab3中的中断跳转与清空 · Discussion #15 · SYSU-SCC/yatcpu-docs (github.com)](https://github.com/SYSU-SCC/yatcpu-docs/discussions/15)