1. 双发射流水线

1.1 双发射流水线原理

最初的处理器要完成一条指令之后才处理下一指令，每个时钟周期都只有一个阶段（此处共 5 个阶段，包含**取指 IF、译码 ID、执行 EX、访存 MEM、写回 WB**）的电路在有效工作，其它电路都处于闲置状态。很容易想到，只要把闲置的部分也利用起来，处理器的运行效率就能提高数倍。

时钟周期	1	2	3	4	5	6	7	8	9	10
指令顺序	第 1 条指令					第 2 条指令				
指令阶段	取指	译码	执行	访存	写回	取指	译码	执行	访存	写回

图 1 无流水线的多周期处理器

像下图中示意的这样，在第 1 条指令的第一个阶段完成后，就立即启动第 2 条指令。在第 6 个时钟周期时启动第 6 条指令，此时第 1 条指令运行完成，还有 4 条指令在流水线中次第前进。以此类推，n 个时钟周期启动 n 条指令，所有阶段的电路都被充分利用，这就是**指令流水线**。分成五个阶段的流水线，就有 5 倍的指令吞吐量。

时钟周期	1	2	3	4	5	6	7	8	9	10
第 1 条指令	取指	译码	执行	访存	写回					
第 2 条指令		取指	译码	执行	访存	写回				
第 3 条指令			取指	译码	执行	访存	写回			
第 4 条指令				取指	译码	执行	访存	写回		
第 5 条指令					取指	译码	执行	访存	写回	
第 6 条指令						取指	译码	执行	访存	写回

图 2 单发射流水线时空图

多发射技术旨在允许处理器在一个时钟周期内译码并发射多条指令到待执行单元中，使得每个周期可以产生多条准备好进入执行阶段的指令。多发射处理器大致包括以下几种:

（1） **超长指令字处理器：**每个周期发射固定数目的指令，静态调度;

（2） **静态调度超标量处理器：**每个周期发射一条至多条指令，静态调度（由编译器或程序本身安排指令执行顺序，硬件不会动态调整），顺序执行;

（3） **动态调度超标量处理器：**每个周期发射一至多条指令，动态调度（由 CPU 硬件动态决定指令执行顺序，允许乱序执行），乱序执行。

所谓**超标量技术**，是指处理器拥有多个并行的流水线执行单元，从而使得处理器每个周期可以完成多条指令的执行阶段。超标量处理器采用动态发射的结构，根据数据相关性不同，每个周期可以发射不同数量的指令，并要求硬件进行流水线冲突的

检测。

本次课程基本要求为实现双发射流水线（五级流水，包含**取指 IF、译码 ID、执行 EX、访存 MEM、写回 WB**），取指、译码、发射、写回宽度都为二或以上，且在执行级包含两个及以上的 ALU。这样可以相比于单发射处理器再提高一倍吞吐率。乱序执行为进阶要求。

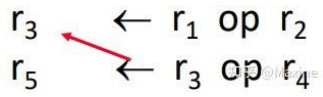
时钟周期	1	2	3	4	5	6	7	8	9	10
第 1 条指令	取指	译码	执行	访存	写回					
第 2 条指令	取指	译码	执行	访存	写回					
第 3 条指令		取指	译码	执行	访存	写回				
第 4 条指令		取指	译码	执行	访存	写回				
第 5 条指令			取指	译码	执行	访存	写回			
第 6 条指令			取指	译码	执行	访存	写回			
第 7 条指令				取指	译码	执行	访存	写回		
第 8 条指令				取指	译码	执行	访存	写回		
第 9 条指令					取指	译码	执行	访存	写回	
第 10 条指令					取指	译码	执行	访存	写回	
第 11 条指令						取指	译码	执行	访存	写回
第 12 条指令						取指	译码	执行	访存	写回

图 3 双发射流水线时空图

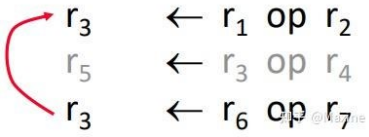
1.2 数据依赖与前馈/旁路

数据冲突包括：写后读（read after write，**RAW**）、写后写（write after write，**WAW**）、读后写（write after read，**WAR**）依赖。

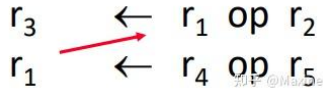
（1）写后读 RAW（真正的数据依赖性）：



（2）写后写 WAW：



（3）读后写 WAR：



上述数据三种冲突中，**只有写后读 RAW 是“真冲突”**，因为后续指令的源操作数真实地依赖于前序指令的写回结果。而写后写和读后写本质上是由于不相关的指令指定了同名寄存器引起的，可以通过寄存器重命名技术解决。

解决 RAW 依赖性的方法：

- 检测并等待（Stall）
- 检测并**前馈/旁路**数据**（Bypass/Forward）**
- 检测并将其移动到独立的指令之后**（指令重排序，乱序执行）**

……

考虑如下的指令序列：

```
add a1, a2, a3
sub a4, a1, a5
```

发生 RAW 冲突时，被阻塞在 ID 级的指令要等到前导指令完成写回（WB）后才能继续执行。但实际上在 WB 阶段之前，后续指令所依赖的数据结果就已经产生。

图 4 **通过停顿（Stall）解决依赖**

因此，在第 4 个周期 ALU 可以直接利用第 3 个周期的计算结果来执行 sub 指令的操作，即让其进入 EX 阶段。硬件实现上，需要增加一条从 ALU 输出端前馈到 ALU 输入端的**数据通路**（更准确地说，是从 EX/MEM 级流水线寄存器前馈到 ALU 输入），此即为**前馈或旁路技术**。 前馈使得流水线停顿被消除。此为 EXE → EXE Bypass，另外还有 MEM → EXE Bypass、WB → EXE Bypass 等，可按需设计。

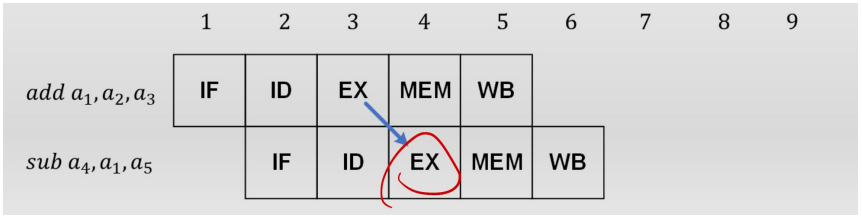


图 5 **前馈机制**

1.3 双发射与单发射的实现区别

（1） 取指 & 译码

取指宽度 = 2，意味着每周期要取两条指令，并分别译码。译码后需要判断：两条指令是否可以并行执行？是否存在数据相关性（RAW、WAR、WAW）？

（2） 寄存器 & 访存

本次提供的存储组件为**单端口**，因此若先后两条指令同时读取，虽然在单发射时不会产生冲突，但因端口有限，也会产生冲突。而如果涉及写入操作，还可能在同时发射的两条指令间出现上述 RAW、WAW、WAR 等冲突。可考虑在寄存器与存储中实现**双端口读写（仅作为进阶要求，基本要求单端口即可）。**

另外，需要注意在译码等读寄存器的情形时，发送读寄存器请求与寄存器给出结果间存在一拍的延迟。如译码时发出请求，可能在执行时才得到结果。因此读寄存器要单独占一拍，不能读完寄存器立刻由译码转入执行。

（3） 执行

双发射流水线的执行操作需要**两个独立的执行单元**，如双整数 ALU 或整数 ALU + 浮点 ALU。

（4） 旁路

由于双发射有两个执行单元，数据旁路需要处理**同时发射的两条指令之间的互相依赖**。单发射时只在单套硬件的不同位置之间添加数据通路，而此时则需要在**两套硬件之间**添加旁路。如仍然考虑 1.2 中的两条指令，若两条指令同时发射，则此时需要在 add 指令使用的 **EX1 单元**的输出端和 sub 指令使用的 **EX2 单元**的输入端间添加数据通路，而 EX1 输出到 EX1 输入、EX2 输出到 EX2 输入的旁路和单发射时一致。另外，同时发射的两条指令间若存在，即使添加旁路机制也需要**一周期的停顿**。

2. CSR

2.1 RISC-V 特权架构

为了加强对操作系统和信息安全的支持，RISC-V 定义了 3 种工作模式：**机器模式、超级用户模式（Supervisor Mode，S-Mode）和普通用户模式（User Mode，U-Mode）。** 每种模式分别对应一个特权层级（Privilege Levels）。其中机器模式的特权层级最高， 而普通用户模式的特权层级最低。在高特权层级运行的代码比在低特权层级的代码 拥有更多的权限，受到的约束也比低特权层级的代码要少。

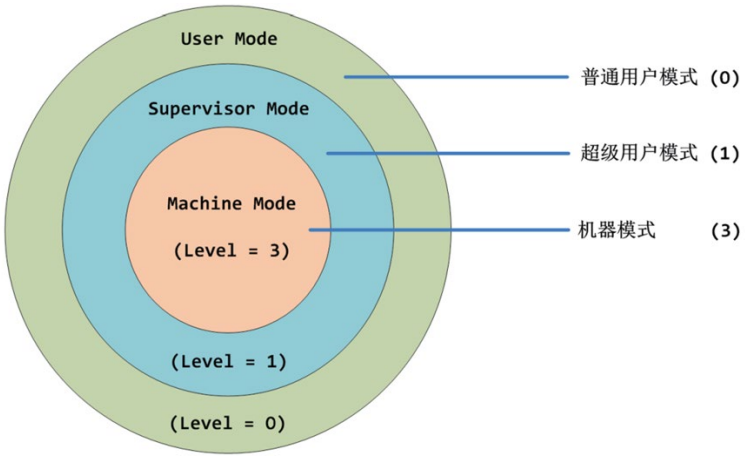


图 6 RV 特权架构

2.2 CSR 控制与状态寄存器

CSR，即 Control and Status Register，**控制与状态寄存器**，反映和控制 CPU 当前的状态和执行机制。

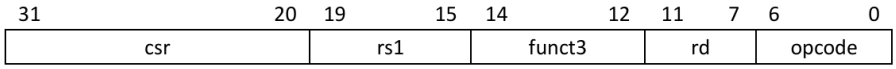
RISC-V 在特权架构部分单独定义了一个控制状态寄存器的地址空间，并分配了 12 位地址来做索引。在这 12 位地址当中，最高的两位 [11:10] 被用来指示寄存器是可读写（00，01 或 10），还是只读（11）。地址位 [9 :8] 表示有权访问该寄存器的最低特权层级。对本项目涉及的的机器模式 CSR，这两位都是 2'b11。

有关 CSR 的具体信息可在 RISC-V Spec 特权指令集部分中查阅。本项目需要实现的 mcycle、mtime 两个 CSR 位于机器级部分。

（ComputerArchitectureLab/5_DetailDocuments/资料/RISC-V 指令集卷 2-特权级指令-中文版.pdf at master · Summer-Summer/ComputerArchitectureLab · GitHub）

2.3 CSR 访问指令

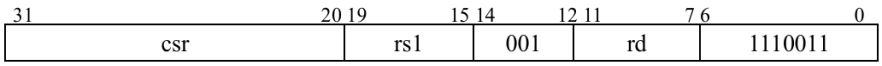
针对 CSR 寄存器的读写，有相应的特殊指令，这些特殊指令都被定义在 RV32I 中，如下所示：



source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	zimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	zimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	zimm[4:0]	CSRRCI	dest	SYSTEM

RISC-V Spec 一共定义了 6 种 CSR 指令，三个不需要立即数；而另外 3 个则需要，这些立即数只有 5bit, 且是 0 扩展的。所有的 CSR 都需要将原值写回到数据寄存器。上述六种 CSR 指令配合特殊的源操作数索引和目的操作数索引(等于 0 或者不等于 0)，就可以变化成众多的 CSR 伪指令。

本项目要求实现其中的 **csrrw 指令**。可在 RISC-V Spec 主体的指令列表部分找到（RISC-V 手册）。



指令形式：`csrrw rd, csr, rs1`

指令功能：CSR 寄存器读后写。记 csr 值为 t，将 rs1 写入 csr，再将 t 写入 rd。

例：`csrrw t0, mstatus, t0`

将 mstatus 的值与 t0 的值交换。

2.4 mtime 与 mcycle

在机器模式（Machine Mode，本项目只支持该模式）下这些存储器主要包括以下六类：

- 处理器信息相关：例如处理器的厂商信息，架构信息，核心数等等，是一个芯片自身的 I 固有信息。
- 中断配置相关：例如中断开关以及中断入口等信息。

- 中断响应相关：例如中断原因，中断返回地址等信息。
- 存储器保护相关：设置不同地址空间的存储器的访问属性，例如可读可写可执行等等。
- 性能统计相关和调试接口相关

本项目需要实现的 **mtime、mcycle** 则属于最后一类。作为一种硬件性能监测的手段，RISC-V 在其特权架构部分定义了一系列**计数器或定时器**，用来记录从某一时间点开始后处理器已运行的时钟周期数和时间。

（1） mcycle

RISC-V 中为机器模式定义了一个 **64 位**的 mcycle 寄存器，用来记录机器已经运行的时钟周期数，CPU 时钟周期计数器，每个时钟周期自动+1。可以通过 csrrw 指令访问。

这个寄存器的低 32 位和高 32 位分别存放在 **mcycle 和 mcycleh** 中。对 RV32 来说，由于无法一次性地将 mcycle 和 mcycleh 同时读取出来，为了保证 64 位数据的完整性，需要在寄存器的读取方式上做一些处理。一种解法是要求软件总是先读取 mcycle，紧接着再读取 mcycleh。软件读取 mcycle 时，硬件同时将当时的 mcycleh 值保存下来，并在下次读取时提供该值。

（2） mtime

RISC-V 在设计时也对 RTC（RealTimeClock，实时时钟）的实现做了考虑。为此，RISC-V 在其特权架构部分为机器模式定义了一个 **64 位**的 mtime 寄存器。

mtime 是时钟定时器。一般来说，它应该以比较精确的石英晶体振荡器为时钟源，并以固定的频率做计数。然而，这个固定的频率具体是多少，RISC-V 中并没有作出明确规定。许多系统将该频率设置为 32.768kHz，因为 32.768kHz 的晶振非常容易获得，而且 32.768kHz 频率较低，适合做休眠时钟。另外，32768 又是 2 的整数次幂，很容易由 32.768kHz 产生周期为 1s 的时钟。

提示：mtime **不是直接由 CPU 驱动，而是由外部定时器（通常是 CLINT）驱动**。它通常**不会随 CPU 频率变化**，而是由固定频率的时钟递增。

另外，需要注意： RV32I/RV32E 访问 mtime 需要把这两个寄存器拆成两个 32 位寄存器来访问，这需要两条指令。如果遇到低 32 位要进位的情况，那么就会产生问题，访问的结果会产生巨大的偏差。

Bi-Mode 分支预测器

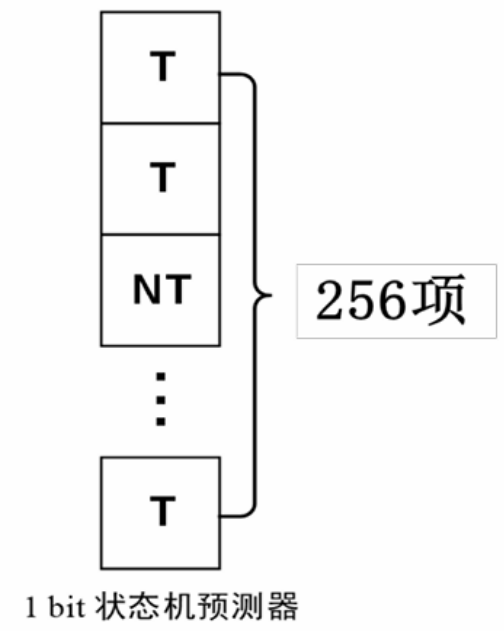
建议提前自学理论课 ppt《Chapter3-处理器流水线结构》P138 开始的分支预测部分

分支预测简介

控制依赖：分支指令的结果决定了后续指令的路径，处理器必须等待分支执行完成后才能确定下一条指令，进而造成**流水线停顿**，流水线后续阶段会因等待分支结果而空闲。

因此，需要通过分支预测器**预测分支方向**，提前预取可能的指令，避免流水线停顿。

分支预测器基础结构



模式历史表（PHT，Pattern History Table）用于记录分支预测器的状态，通过指令 PC 值来索引表中的条目。

一般 PC 是 32 位，如果使用 PC 的全部位来索引，整个寻址范围达到了 2 的 32 次方，存储开销将会非常大，因此一般取 PC 的一部分。一般而言，在程序中位置相近的指令，其 PC 的高位都是相同的，为了避免这些临近的分支指向 PHT 的同一个条目，通常使用 PC 的低位作为 PHT 索引。例如上图的一个具有 256 表项的 PHT，可以使用 PC 的低 8 位进行索引。

分支预测可以分为两大类：

1. **静态分支预测器**

静态预测器基于固定规则预测方向。常见的策略包括：

- 总是预测不跳转（Always Not-Taken）：

假设所有分支均不跳转，顺序执行下一条指令。

- 总是预测跳转（Always Taken）：

假设所有分支均跳转，直接预取目标地址的指令。

2. **动态分支预测器**

动态预测器根据分支历史行为调整预测策略。

1-bit 预测器是最简单的动态分支预测器，基于分支的**最近一次行为**进行预测。

1-bit 预测器使用宽度为 1bit 的 PHT 记录预测器状态。

- 如果本次发生了跳转，则预测下一次也跳转
- 如果本次没有跳转，则预测下一次也不跳转

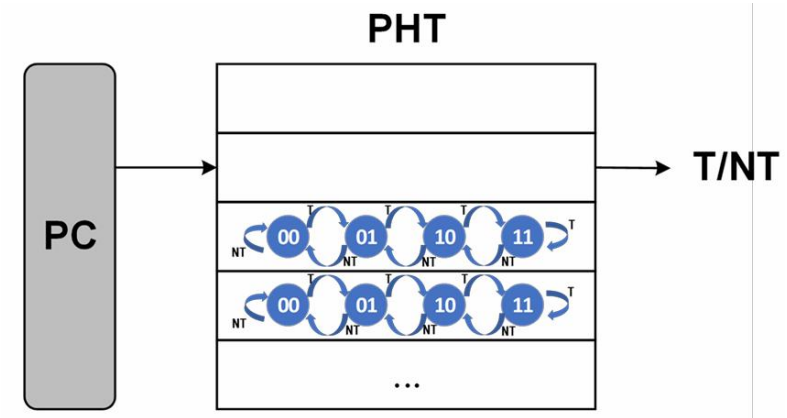
缺点：对偶发变化敏感，单次行为变化会立即改变预测，导致误判。

例如某个分支几乎总是不跳转，但偶尔会有一次跳转，那么使用 1-bit 预测器进行预测时会连续发生两次错误预测。

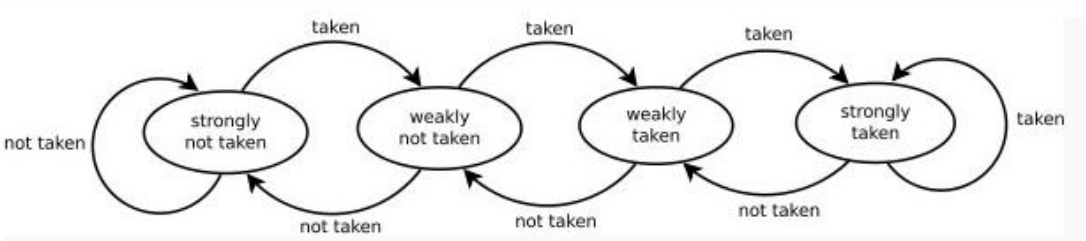
因此引入 2-bit 预测器。

2-bit 分支预测器（两位饱和计数器，Bi-Modal 预测器）

2-bit 分支预测器实际上是一种有 **4 个状态的状态机**。



PHT（Pattern History Table，模式历史表）



更新规则：

- 若实际结果与预测一致，**强化当前状态**：
  - 预测跳转且实际跳转 → 计数器递增（如 `10` → `11`）。
  - 预测不跳转且实际未跳转 → 计数器递减（如 `01` → `00`）。
- 若实际结果与预测不一致，**削弱当前状态**：
  - 预测跳转但实际未跳转 → 计数器递减（如 `11` → `10`）。
  - 预测不跳转但实际跳转 → 计数器递增（如 `00` → `01`）。

因此，2-bit 预测器**必须连续失败两次**才会改变预测结果，增强了预测的偏向性，使得预测器在面对偶发行为变化时仍能保持对主要趋势的稳定性。

以如下循环代码为例：

```
C
int sum = 0;
while (ture) {
    for (int i = 0; i < 100; ++i) {
        //branch not taken
        sum += a[i];
    }
    //branch taken
}
```

假设 1bit 和 2bit 预测器的初始状态分别是 1 和 11，则两种预测器的工作状态如下表（红色为错误预测）：

i	0	1	2	…	98	99	100	0	1	…
1 bit预测器	1	0	0	…	0	0	0	1	0	…
2 bit预测器	11	10	00	…	00	00	00	01	00	…

- 在该例子中，在循环第一次完整执行时，2bit 预测器会比 1bit 预测器多一次预测错误
- 但此后循环代码每次重复执行时，2bit 预测器都会比 1bit 预测器少发生一次预测错误

基于以上内容可以推广到 N-bit 预测器，但实际上过度增加 N 的数值在带来更大存储开销的同时并不能显著提高预测准确率。因此 2-bit 是最常用的预测器。

相关预测器（或全局预测器）

以上结构的预测器通常被称为局部预测器，仅考虑了每一条指令各自的局部历史，而没有考虑分支指令之间的相关性。对于某些情况下，分支是否跳转依赖于其他分支的跳转情况，比如下面的 C 代码，第二条分支是否跳转依赖于前一条分支的跳转情况。

```C
if(b){
    a = 0;
    ...
}
if(a==0){
    ...
}
```

全局历史寄存器（Global History Register, GHR）是一个 N 位的移位寄存器（例如 4 位、8 位、12 位），用来存储所有分支的历史信息，每一位表示一个分支的历史结果。

记录分支历史结果时，将结果左移入 GHR，移除最旧的历史位。

例如，初始 GHR = 0000

- 分支 1 结果：Taken → GHR = 0001
- 分支 2 结果：Not-Taken → GHR = 0010
- 分支 3 结果：Taken → GHR = 0101
- 分支 4 结果：Taken → GHR = 1011

在全局预测器（如 Gshare）中，**GHR 与 PC 的一部分结合生成索引**，用于查询分支历史表（PHT），一般采用将 GHR 与 PC 的一部分做异或运算的方法得到 PHT 索引。

别名冲突

由于 PHT 的容量有限，可能会有**多个不同的分支指令映射到相同的索引位置**。

当两个不相关的分支共用同一预测条目时，会相互干扰，导致错误预测，这种现象称为**分支别名问题**。

例如，两个有分支指令 A 和 B，其地址经哈希后均映射到索引 0 的预测条目上。

分支 A 的行为：90%概率为 Taken（例如循环条件判断）；

分支 B 的行为：90%概率为 Not-Taken（例如错误处理分支）。

预测器可能会在 01 和 10 之间震荡，就**很难收敛**到正确的预测模式。

Bi-Mode

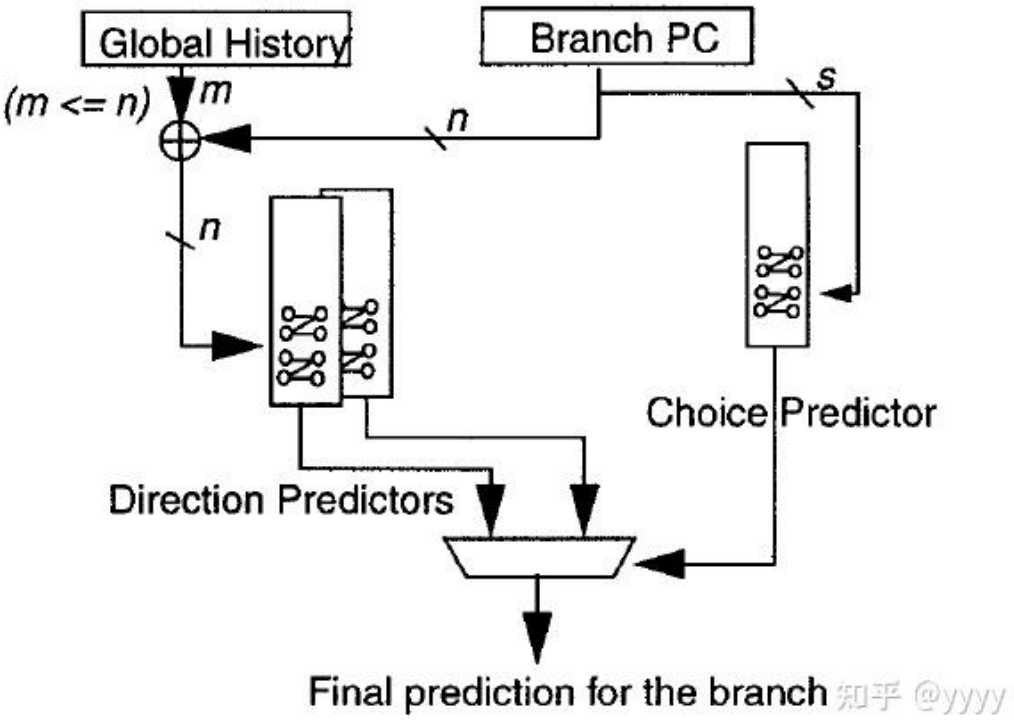
为解决分支别名冲突的问题，采用 Bi-Mode 分支预测方法。

结构

将单一预测表 PHT 分为**两个预测表**：

- Taken 表（**T 表**）：偏向预测分支跳转（Taken）。
- Not-Taken 表（**NT 表**）：偏向预测分支不跳转（Not Taken）。

同时配备**一个选择器**。选择器实际上是一个独立的预测表，同样由 2-bit 计数器构成。它会根据分支指令的历史行为，决定当前分支应使用 T 表还是 NT 表的预测结果。需要注意的是，选择器的索引函数需要与预测表的函数不同。例如，图中选取 PC 的一部分 n 位与 GHR 的值异或作为预测表的索引，选取 PC 的一部分 s 位作为选择器表的索引。



工作原理

沿用上文别名冲突的例子。

假设选择器 A 初始值均为 10，T 表初始值为 11，NT 表初始值为 00。

步骤 1：初次预测

- 分支 A（偏向 Taken）：
  - 选择器 A 为 10，选择 T 表 → T 表中索引 0 的计数器为 11 → 预测 Taken（正确）。
- 分支 B（偏向 Not-Taken）：

- 选择器 B 为 10，选择 T 表 → T 表中索引 0 的计数器为 11 → 预测 Taken（错误）。

步骤 2：更新选择器

- 分支 A 预测正确：
  - 选择器 A 计数器递增，10 → 11。
  - 更新 T 表中索引 0 的计数器（因实际结果为 Taken，T 表计数器从初始值 11 递增至 11）
- 分支 B 预测错误（实际为 Not-Taken）：
  - **选择器 B 计数器递减，10 → 01**。
  - 更新 NT 表中索引 0 的计数器（因实际结果为 Not-Taken，NT 表计数器从初始值 00 递减至 00）

步骤 3：选择器切换

- 分支 A：
  - 选择器 A 计数器为 11，依旧选择 T 表。
- 分支 B：
  - **选择器 B 计数器为 01，切换至 NT 表**。
  - NT 表中索引 0 的计数器为 00，预测 Not-Taken，正确。

后续分支 A 稳定使用 T 表，分支 B 稳定使用 NT 表，虽然两个分支的索引相同，但实际上**不再共享相同的预测条目**，则别名冲突被隔离。

分支目标缓存

分支预测器只预测分支指令是否需要跳转，但跳转地址的预测也很重要。如果即知道这条这条分支指令的跳转方向，也知道这条分支指令的跳转目标，那么我们就有机会将分支代价降为零。

分支目标缓存（BTB，Branch Target Buffer）会存储历史分支指令的目标地址，一般由 data array、tag array 和有效位构成。

Tag	Target PC	Valid
0x66EE38	0x847CE94D	1

标志（tag）和索引（index）组合用于判断 BTB 是否命中，一般而言，tag 可以取 PC

高位，index 可以取 PC 低位。

通常采用 BTB 和 BHT（Branch History Table，分支历史表，即上述的预测表）结合的方式进行分支预测。

在**取指阶段**利用 PC 寻址 BTB，如果命中且有效，则说明这是一条跳转指令，利用从 BTB 中获取到的地址去取后续指令。若未命中，则按顺序预取下一条指令。

再在**后续流水级**辅以 BHT 用于捕捉被 BTB 遗漏的分支跳转，并根据最终程序执行的结果修正 BTB 和 BHT 中的表项。

例如，如果预测错误，实际“Not-Taken”或者计算出来的目标地址与 BTB 提供的目标地址不同，则需要**刷新流水线**，即清除所有在错误分支后已进入流水线的指令，恢复正确的 PC，并更新 BTB 和 BHT。

参考链接：

https://zhuanlan.zhihu.com/p/460942331

https://zhuanlan.zhihu.com/p/579747343

CPU 与外部 ROM、RAM 接口

本课程提供现有的存储模块 mem.scala，以下是详细介绍。

功能

该存储模块为**单端口同步读写寄存器**，支持**指令读取**和**数据读写**，可加载外部文件初始化存储内容。

参数

参数名	类型	默认值	描述
memDepth	Int	用户自定义	存储器的深度，即存储单元的数量
instWidth	Int	用户自定义	每次取指的指令宽度（通常为 1 或更大，表示一次取多条指令），双发射即为 2

接口

接口名称	说明
reset	复位信号，等于 1 时输出空指令（NOP），即 addi x0, x0, 0
if_mem	指令取指（IF）阶段和存储器（MEM）之间的接口
ex_mem(lsu_mem_c)	指令执行（EX）阶段和存储器（MEM）之间的接口，即访存单元（LSU）到存储器（MEM）的接口
mem_id	存储器（MEM）和指令译码（ID）阶段之间的接口
mem_lsu(mem_lsu_c)	存储器（MEM）到访存单元（LSU）的接口

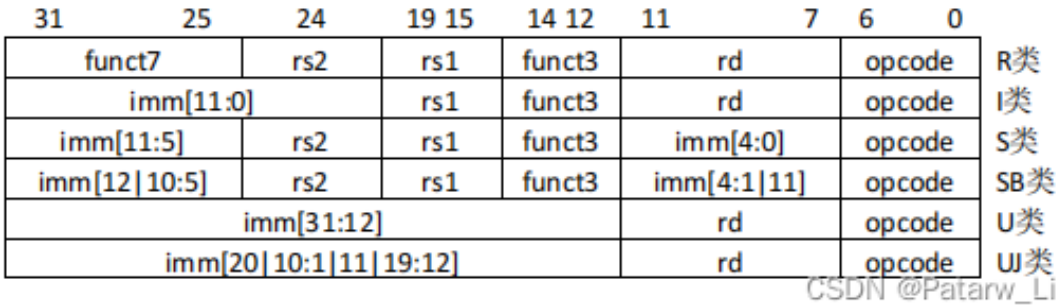
具体信号

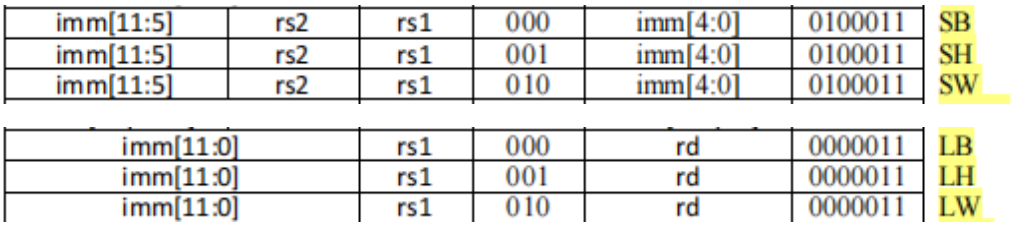
信号名	方向	位宽	描述
if_mem.instAddr	输入	64bit	取指单元提供的指令地址
ex_mem.atomFlag	输入	1bit	标记是否为原子操作（本课程不需要，置 false 或忽略即可）
ex_mem.dataAddr	输入	64bit	访存指令的目标地址
ex_mem.writeEn	输入	1bit	写使能信号，1 有效表示写入数据
ex_mem.writeData	输入	64bit	需要写入存储器的数据
ex_mem.func3	输入	3bit	访存指令的 funct3 字段，决定数据宽度（Byte, Halfword, Word, Doubleword）
mem_id.inst	输出	Vec(instWidth, UInt(32.W))	取出的 instwidth 条 32-bit 指令，供 ID 译码使用

mem_lsu.data	输出	64bit	从存储器读取的数据，传回给访存单元（LSU）

使用说明

关于 funct3





funct3 是指令编码的[14 : 12]部分，在访存指令（load, store）中，funct3 是数据的宽度的标志。

如图，"000"标志着这个访存指令的数据宽度为一个字节，"001"表示数据宽度为半字（两个字节），"010"表示数据宽度为单字（四个字节）。

初始化存储器

该模块支持从外部文件加载内存数据，示例如下：

```
Scala
loadMemoryFromFile(memInside, "drystone.data",
MemoryLoadFileType.Hex)
```

这样可以在仿真时使用外部数据初始化存储器，用于测试和调试。

该语句**只会在仿真测试时生效，不会影响硬件设计**。

.data 文件在后续测试环节由助教提供。

单端口/双端口

本模块为单端口存储模块，需要考虑**同时发射的两条指令均为访存指令**的情况。

如有需要，可自行修改为双端口，但还需要考虑**双端口同时访问存储器时的数据依赖问题**，类似寄存器的数据依赖问题。

关于位宽

本课程项目要求为 32 位 cpu，而本 mem 模块为 64 位。

调用本模块时可通过**高位补 0** 和**低位截取**操作解决位宽问题。

printf 语句

为方便后续的仿真测试，需要在 mem 模块中添加 printf 的功能。

功能描述

当 mem 模块检测到指定地址 0x10001ff1 的写入时，将写入的数据以字符的形式打印。后续的测试程序会将需要打印的字符以 ASCII 码的形式存储到这个地址中，从而实现 printf 的功能。

代码参考

```Verilog
initial begin
    forever begin
        if (u_top0.lsu0.lsuEnable == 1'b1 &&
            u_top0.lsu0.io_ex_lsu_0_opcode == 5'h8 &&
            u_top0.lsu0.io_ex_lsu_0_dataAddr == 'h10001ff1) begin
            $write("%c", u_top0.lsu0.io_ex_lsu_0_writeData[7:0]);
        end
    end
end
```

这是一段 Verilog 的测试代码，可以参照这部分用 chisel 进行实现。

推荐分工

1. 分支预测 BTB
2. 分支预测 BHT
3. 取指

4. 译码
5. 执行
6. 访存
7. 寄存器堆
8. top+性能测试
9. 机动

建议使用 git 进行同步管理