Make You a RISC-V Simulator

Alex Chi

July, 2019

在过去的两个星期里，我用 C++ 实现了一个 RISC-V 模拟器。模拟器支持 RV32I 指令集。我先后完成了串行 (Sequential)、并行 (Pipelined) 以及乱序执行 (Out-of-order Execution) 的模拟器。本文将介绍实现的细节和过程。

1 电路、FP 与串行模拟器

在设计之初，我希望模拟器的实现可以尽可能贴近硬件电路。其中的每一个操作，都是能够通过电路方便实现的。

> Aside **什么样的实现是贴近电路的?**
> 电路不受外界条件影响，在寄存器的值和内存给定的情况下，不论仿真多少次，都能得到同一个值 (pure)。电路中也不宜设计一些复杂的数据结构（比如队列）。为了达成这个目标，我设计了第 1.3 章所述的寄存器类，并且在编程的时候增加了几个限制，这一部分参见第 1.2 章。

在 CSAPP HCL 的启发下，我从函数式编程的角度考虑电路仿真，从而实现一个尽可能贴近硬件实现的模拟器。

> Aside **一个 CSAPP HCL 的样例**
>
> ```
> int dstE = [
>     icode in { IRRMOVL } : rB;
>     icode in { IIRMOVL, IOPL} : rB;
>     icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
>     : RNONE;
> ];
> ```

1.1 如何求解电路

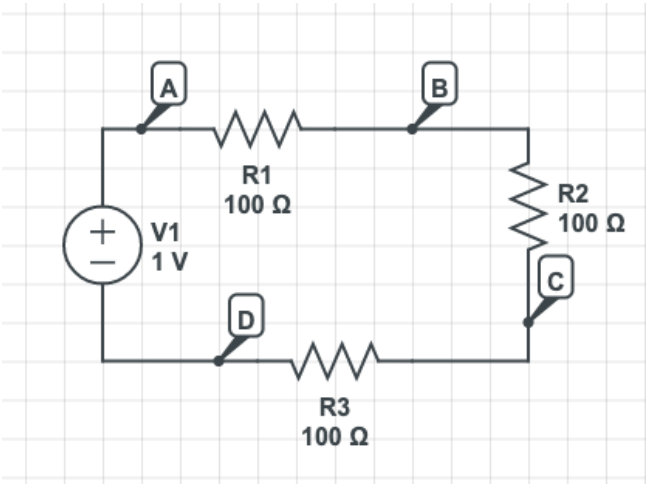

图 1: 一个电路

如图 1 所示，若要求得节点 C 处的电压，则必要先依次求出 A、B 处的电压。而当初始条件确定（电压源确定）时，不论求多少次 C 处电压，它永远都是一个值。由此，我联想到某个语言中的惰性求值 (Lazy Evaluation)。

```
v_source = 1.0
current = 0.003333333
voltage_at_c :: Double
voltage_at_b :: Double
voltage_at_a :: Double
voltage_at_c = voltage_at_b - 100 * current
voltage_at_b = voltage_at_a - 100 * current
voltage_at_a = v_source

ghci> voltage_at_c
0.33333339999999995
```

在 Haskell 中，我们可以描述函数之间的关系。在定义函数的时候，这些值都不会被立刻求出来。只有值被用到的时候，ghc 才会开始执行函数内容求值。这一思想可以被应用在电路仿真中。

1.2 如何设计电路

在 CPU 电路中，我们大体可以把元件分为两大类：时间控制和非时间控制元件。

时间控制的元件包括寄存器、内存等。只有时钟发出信号，它们的值才会被更新，输出端的电平才会有变化。这些元件决定了电路的初始条件，是电路的信号源。非时间控制的元件

包括各种选择器、运算器。这些元件端口的输出值可以通过信号源的电平决定。一旦电路达到稳态，它们的电平就不会再改变。

因而，我们可以在程序中描述一组电路中的关系，就像之前的 Haskell 代码一样。而后，再用 C++ 求得电路电平的关系，更新寄存器，开始下一个周期。

设计电路的原则之一是：数据向前传播。在同一个周期内，这些电路元件之间的求值关系不应该出现互相依赖的关系。对于图 1 的电路，我们也可以用下面这段代码描述。

```
voltage_at_c = voltage_at_b - 100 * current
voltage_at_b = voltage_at_c + 100 * current
```

但这种描述就存在数据间的依赖关系。在设计 CPU 电路时，我们应当保证数据是单向传输的，对于每一根导线，我们可以人为规定它数据传播的方向（从寄存器输出到另一个寄存器输入）。在这一限制下，一个时钟周期内电路总能达到稳态。

1.3 寄存器模型

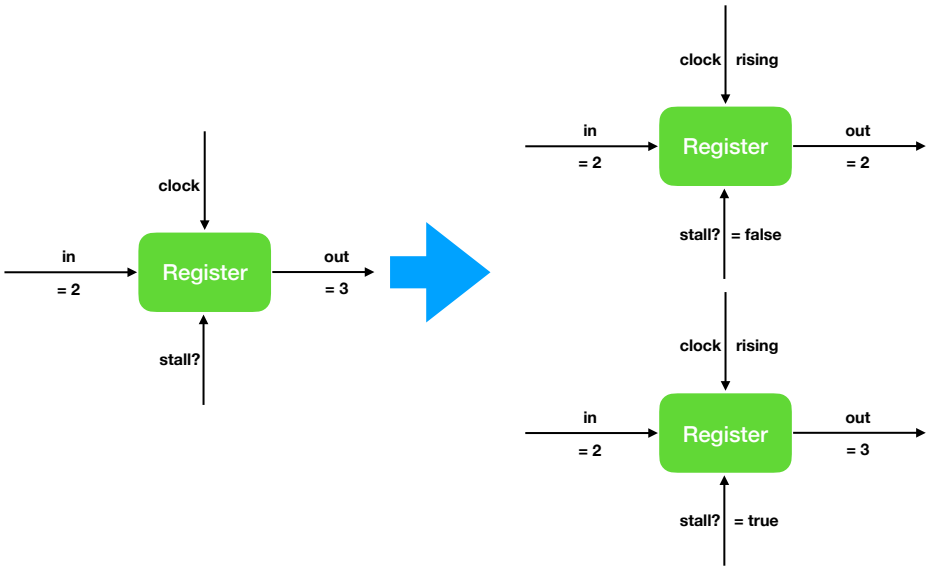

图 2: 寄存器模型

如图 2 所示，寄存器有一个数据输入端和一个数据输出端。寄存器由时钟控制。收到时钟信号 (tick)，如果数据不被搁置 (stall)，就将输出端设置为输入端的值。这一模型可以由下面的 C++ 代码所描述。

```
template<typename T>
class Register {
```

```
public:
    T prev, next;
    bool _stall;
    Register() : prev((T) 0), next((T) 0), _stall(false) {}
    T read() { return prev; }
    T current() { return next; }
    void write(const T &t) { next = t; }
    void tick() { if (!_stall) prev = next; }
    void stall(bool stall) { _stall = stall; }
};
```

> 在第 3.1 章中，我将会介绍一种可读性更高的寄存器类设计。

1.4 惰性求值在 C++ 中的实现

对于 CPU 中的每一个电路端口，我都确定了一个唯一的标签。通过标签即可得到端口的数据。比如，如果我们要在 Write-back 阶段完成 load 指令，将内存读取的值写入寄存器：

```
// 获取寄存器写入目标
auto rd = session->d->get(Decode::rd);
// 获取内存阶段读到的值
auto m_val = session->m->get(MemoryAccess::m_val);
// 写入寄存器
session->rf.write(rd, m_val);
```

在 Memory Access 阶段，我们定义求 `m_val` 的方式。

```
unsigned int addr = session->e->get(Execute::e_val);
switch (session->d->get(Decode::funct3)) {
    case 0b000: // LB
        return (char) session->memory[addr];
    case 0b001: // LH
        return (short) session->memory.read_ushort(addr);
    ...
}
```

求 `m_val` 的过程还会自动求 Execute 阶段所计算的内存地址。因此，通过惰性求值，程序能自动解决电路元件间的依赖关系。

与此同时，由于一个时钟周期中，每个端口的值不论求多少遍都是一样的，我们可以通过 map 或者其他方式将该周期里求过的端口值保存下来。在这个实现中，每个阶段的端口都由一个枚举类型定义。枚举类型的元素是一个从 0 开始编号的整型数。因此，我们可以直接用数组缓存端口的值。

> Aside　**用 CLion 管理工程**
> 在很久以前（大概是四年前），我对 CMake 最头大的事情是它没法用一条指令直接把整个目录的源文件都一起编译。在更久以前（大概是五年前），我对 C++ 最头痛的事情就是写它就要把声明与定义分离。把定义和实现拷来拷去，是一件令人头大的事情。后来，我发现了 CLion。我沉迷一键生成定义 (Definition)，自动加载 CMakeLists.txt。自此以后，写 C++ 变成了一件不那么麻烦的事情。后来，随着 vcpkg 这类靠谱的 C++ 包管理器的出现，在 CMakeLists 中增加依赖也变得简单了许多。在这个项目中我就用到了 vcpkg 安装的 gtest。

1.5　串行模拟器的实现[1]

在串行的 RISC-V CPU 中，电路中的时间控制元件只有内存、32 个寄存器和 PC。每个时钟周期中，各个电路元件从 PC 开始依次读出指令、获取操作数、执行计算、写内存、写寄存器，并将新的 PC 写入 PC 寄存器中。写入寄存器的数据在这一个时钟周期中无法被读出。在数据更新完成后，对所有时间控制元件执行 “tick” 操作，开始下一个时钟周期后，这些写入的数据才能被电路元件读取到。

> Aside　**一些常见的坑**
> 第 0 个寄存器永远是 0。
> `JAL`、`JALR` 指令要把 `PC + 4` 写入 `rd`。

因而，从程序模拟的角度来说，我们只需要求得电路输出的值就可以完成一个周期的模拟。在串行 CPU 中，整个电路有三个输出：到 PC 寄存器的输出，向内存写入，以及向 32 个寄存器写入。这就是每次更新所需要做的事情。

```
PC.write(w→get(WriteBack::w_pc)); // 更新 PC 寄存器
m→hook();                         // 写入内存
w→hook();                         // 写入寄存器
```

由于我们在程序中通过函数的调用关系确定了求值顺序，三句语句可以任意调换而不影响最后的结果。

[1] 串行模拟器的实现在 seq 分支。https://github.com/skyzh/RISCV-Simulator/tree/seq

Aside　**写点单元测试**
在完成 PPCA 的几个项目的过程中，我用 Google Test 写了一些单元测试。虽然单元测试通过并不能保证整个程序可以跑起来，甚至单元测试也没法覆盖所有样例，但如果单元测试挂了，那必然是改程序改错了。比如在写 Parser 的过程中，我发现有些时候内存里读到的指令是 `0xffffffff`。于是，我加上了 hex dump 中行末有空格、有各种换行的 Parser 单元测试，发现了问题所在。

2　更快的电路仿真：从串行到并行

在上一章中，我介绍了串行模拟器设计的过程和其中的一些问题。由于从项目一开始我就采用电路仿真的思路，所以从串行改并行的过程中，我几乎没有进行大的重构。但是后来我发现，惰性求值效率很低。因此，在这一章中，我提出了另外一个电路仿真的方法，并且验证了这种方法依然是 pure 的。

在这一部分，我完成了一个通过数据转发 (Forwarding) 解决数据冲突 (Data Hazard) 的 CPU 模拟器。模拟器还采用了两位自适应分支预测 (Two-level Adaptive Predictor)，在部分样例中达到了 98% 的准确率。

2.1　手动指定求值顺序

观察串行模拟器使用的电路，我们可以发现，CPU 的电路设计在程序编写时就已经设计好了。因而，我们可以手动指定求解电路的顺序。[2]

比如图 1 所示的电路，我们完全可以依次求出 A, B, C, D 点所对应的电平。因为在求解之前，我们就已经知道电路端口与信号源的远近关系（即拓扑序）。这种按顺序一次求值，就能求出当前时钟周期里每个端口的值的方法，和之前的惰性求解是等价的。

以串行模拟器为例。要求出 Write-back 阶段写入的值，就要求出 Memory 阶段读取的值，以及 Execute 阶段运算的值。求 Execute 阶段运算的值，就要求出 Decode 阶段、Fetch 阶段的指令和寄存器值。因而，我们很快就能发现五级模块间的依赖关系。所以，我们可以以 Fetch, Decode, Execute, Memory, Write-back 的顺序传播电路信号，在传播的过程中写入寄存器和内存，并获取下一条指令的地址。

2.2　五级流水线设计与实现[3]

在改为五级流水线后，情况就不一样了。由于每两级之间都有寄存器隔离，所以每个阶段的电路信号仅和两级之间的寄存器有关。在这种情况下，不论以什么顺序求值都可以一次得到电路每一个端口的值。

[2] 手动指定求值顺序的串行模拟器实现在 feed-forward 分支。https://github.com/skyzh/RISCV-Simulator/tree/feed-forward

[3] 并行模拟器的实现在 pipeline 分支。https://github.com/skyzh/RISCV-Simulator/tree/pipeline

在通过暂停流水处理数据冲突时，Decode 阶段需要知道后面三个阶段正在执行的指令和需要写入的寄存器。因而，必然有一组导线从后面三个阶段接到 Decode 阶段。在这种情况下，电路求值就有了依赖关系。Decode 阶段必须要等到 Execute, Memory 和 Write-back 的电路端口达到稳态后，才能求值。

在处理 Control Hazard 的时候，Fetch 阶段需要知道 Execute 阶段所处理的分支指令的跳转目标是否和先前预测的一致。因而，Fetch 阶段必然要在 Execute 阶段之后求值。

所以，我们可以构建出一个求值顺序，保证一次传播电路信号，就能正确求出当前周期中每一个端口的值。其中一种正确的顺序如下。

```
w→hook(); // Write-back
m→hook(); // Memory
e→hook(); // Execute
d→hook(); // Decode
f→hook(); // Fetch
```

在电路信号传播完成后，更新时钟，开始下一个时钟周期，就能完成并行 CPU 的模拟。

2.3 验证并行实现确实并行

在 Session.cpp 中随机选取五个阶段的求值顺序。这个循环执行十遍后，电路必然可以达到稳态，它和手动指定求值顺序的结果一致。因此，模拟器的运行和五个阶段的求值顺序无关。这个模拟器是并行的。验证的代码如下。

```
int seq[5] = { 0 };
for (int j = 0; j < 5; j++) seq[j] = j;
for (int i = 0; i < 10; i++) {
    std::random_shuffle(seq, seq + 5);
    for (int j = 0; j < 5; j++)
        switch(seq[j]) {
            case 0: f→hook(); break;
            case 1: d→hook(); break;
            case 2: e→hook(); break;
            case 3: m→hook(); break;
            case 4: w→hook(); break;
        }
}
```

当然也有其他方法验证。比如开五个线程分别更新五个阶段，每隔 1ms 更新一次时钟。这种验证方法似乎就更加贴近电路中电压的传播了。

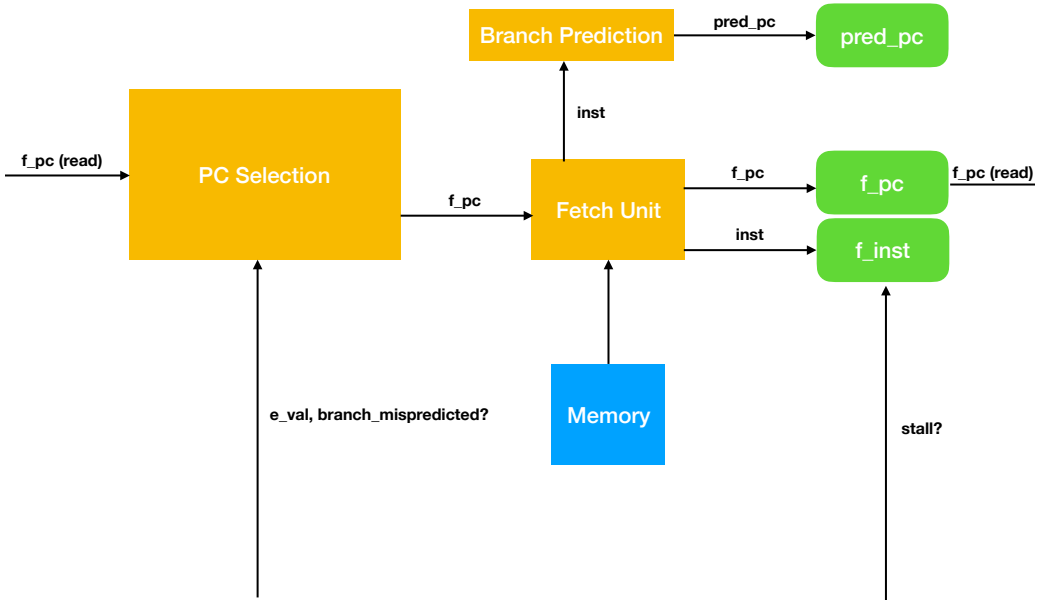

图 3: Fetch 阶段

理论上，我们也可以跑五个线程分别刷新五个阶段的电路。但事实是，这么做会导致各种异常。这可能是因为对寄存器的操作存在 data race。

2.4 模拟器使用的 CPU 电路

模拟器所使用的电路如图 3 4 5 6 7 所示。

Aside **自己造数据**
如果直接用现有的样例测试模拟器的正确性，似乎有些难度。因而，我通过 RARS 编写 RISC-V 汇编，造了几组测试数据，用于检查模拟器面对 Data Hazard 和 Control Hazard 时的行为。这些测试样例都可以在 `tests/` 目录下找到。

2.5 分支预测

在完成五级流水线后，我开始探究分支预测的实现。原本的实现对于分支指令，一律预测紧接的一句 (Always not taken)。对于 `JAL`、`JALR` 之类的指令，也在 Execute 阶段才进行处理。在此基础上，我实现了两位自适应分支预测 (Two-level Adaptive Predictor)。结果如表 1 所示。

分支预测在循环较多的程序中，取得了比较好的效果。

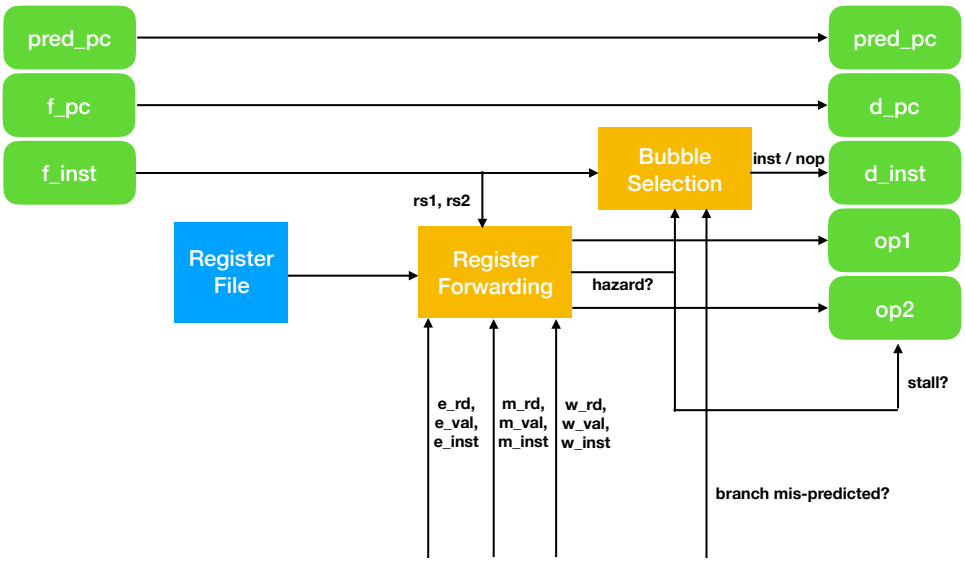

图 4: Decode 阶段

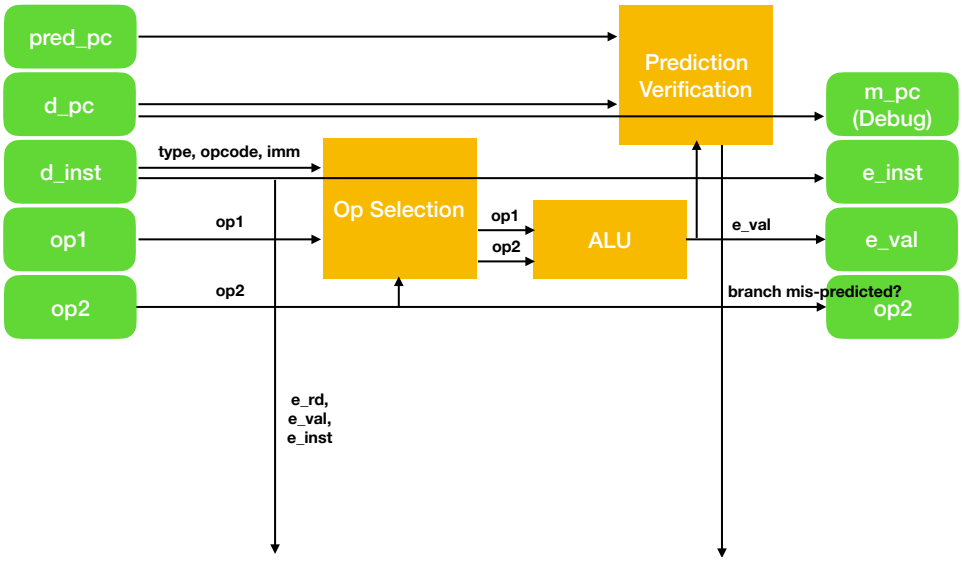

图 5: Execute 阶段

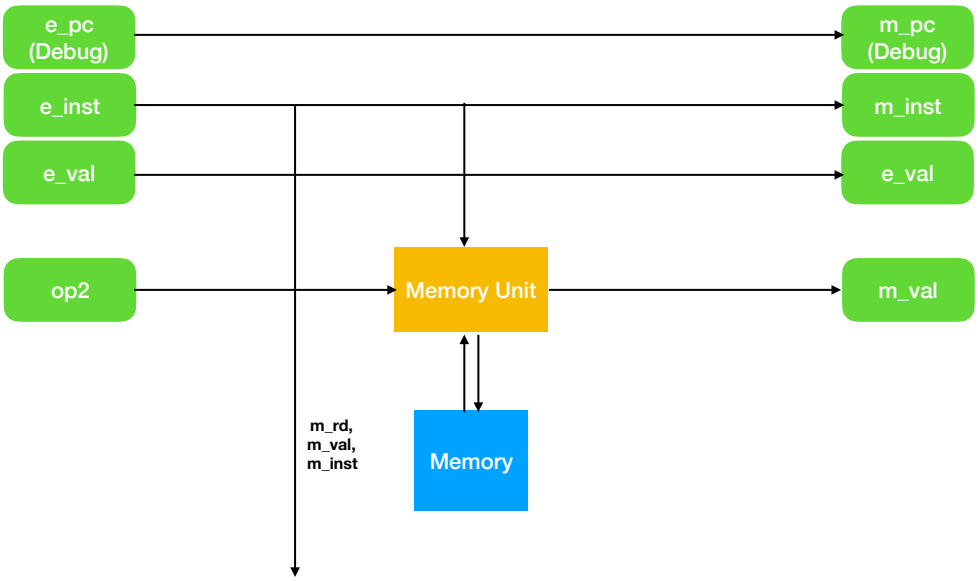

图 6: Memory Access 阶段

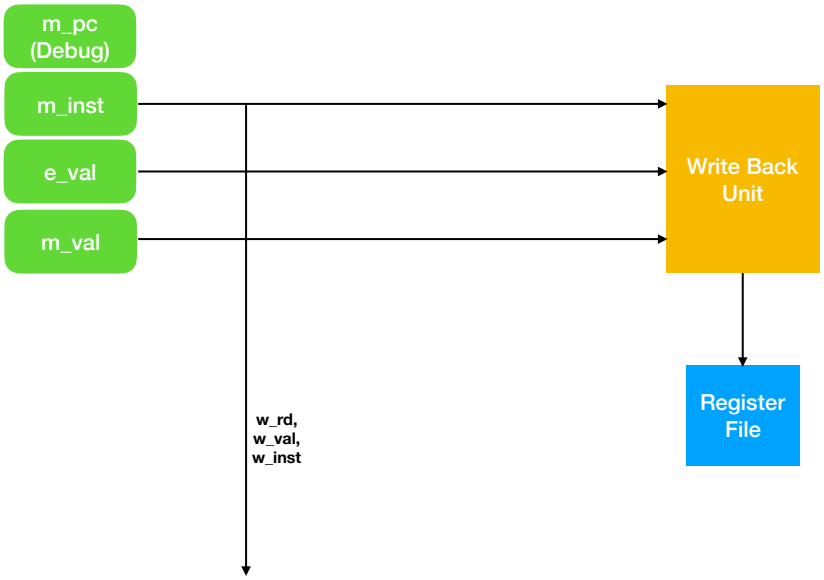

图 7: Write-back 阶段

表 1: 分支预测结果

数据	预测前周期	预测后周期	周期减少率	预测成功率
array_test1.data	218	209	4.13%	54.5455%
array_test2.data	247	237	4.05%	50%
basicopt1.data	633149	559226	11.68%	98.8887%
bulgarian.data	419969	377506	10.11%	94.4862%
expr.data	632	586	7.28%	75.6757%
gcd.data	546	534	2.20%	61.6667%
hanoi.data	201773	190794	5.44%	86.4868%
lvalue2.data	61	59	3.28%	66.6667%
magic.data	582014	553419	4.91%	82.2054%
manyarguments.data	71	69	2.82%	80%
multiarray.data	1777	1650	7.15%	66.0494%
qsort.data	1590883	1430582	10.08%	91.7384%
queens.data	607530	582242	4.16%	81.0156%
statement_test.data	1115	1067	4.30%	66.8317%
superloop.data	577259	532844	7.69%	95.1932%
tak.data	1546200	1530613	1.01%	75.7038%
pi.data	127443492	109722503	13.90%	84.5072%

Aside　**使用 git 管理工程**
说到分支，就不得不提 git 的分支（虽然此分支非彼分支）。在这个项目中，我通过 git 管理工程，开了许多分支。它们包括：seq (Sequential, 串行实现)、feedforward (手动指定求值顺序的串行实现)、pipeline (并行实现)、out-of-order (乱序实现)。我时常在不同分支间切换，以修改某个版本的 bug。偶尔也会在大改项目之后，再切换分支参考之前写过的代码。

3　基于 Tomasulo 算法的乱序执行

在乱序执行中，原有的五级流水线最后只剩下两个大部分：发布指令 (Issue) 阶段和乱序执行 (OoOExecute) 阶段。在乱序执行阶段中，原有 Execute 阶段的 ALU 变成了一个小单元，Memory Access 变成了 Load-Store 单元。

由于我并不知道改如何设计基于 Tomasulo 算法的硬件，在这一章节中的程序仅仅是一个展现乱序执行的纯算法模拟。[4]

[4]乱序执行模拟器的实现在 out-of-order 分支。https://github.com/skyzh/RISCV-Simulator/tree/out-of-order

在这一章中，我将依次介绍仅基于 Tomasulo 算法的实现，以及支持推测执行的实现。最后，我将展示一个支持 Tomasulo 算法、推测执行、分支预测 (Speculation) 的 CPU 模拟器实现。这个模拟器中，重排缓冲 (Reorder Buffer) 有 12 个空位。在 Execute 阶段，有 3 个 Load 缓冲、3 个 Store 缓冲和 4 个 ALU 单元。

3.1 新的寄存器类设计

```
template<typename T>
class Register {
    ... // 这一部分和之前的 Register 类一致
    operator T() { return read(); }
    void operator=(T next) { write(next); }
};
```

在这一设计下，整个程序的可读性将大大提高。比如在 Tomasulo 算法的放操作数阶段：

```
void Issue::issue_rs_to_Vk(unsigned reg_id, RS *rs, RSID unit_id) {
    if (reg_id != 0 && session->e->should_rename_register(reg_id)) {
        rs->Vk = 0;
        rs->Qk = session->e->get_renamed_register(reg_id);
    } else {
        rs->Vk = session->rf.read(reg_id);
        rs->Qk = NONE;
    }
}
```

在这里，Vk, Qk 都是寄存器。对 Vk 和 Qk 的操作，只有在下一个周期才会生效。因而，这种寄存器类的设计在不影响理解的同时，增加了代码的可读性。

当然，在某些情况下，这种写法或许会让人有些误解。比如在发出指令阶段：

```
pc = pc + 4
```

在这里，我们把当前时钟周期 PC 读出，通过某个硬件单元加了 4，并把结果接回了 PC 寄存器上。因而，在同一个时钟周期里，不论这句话被执行几次，下一周期的 PC 永远是这一周期的 PC 加上 4。

3.2 仅基于 Tomasulo 算法的 CPU 设计

在 *Computer Architecture: A Quantitative Approach* 的第 3.5 节中, 作者介绍了 Tomasulo 算法。但是在这一章节中，缺少很多实现的细节，以至于我无法直接参照书本实现一个 CPU。因而，我对书上的设计进行了一定的补充和修改，以便于模拟器的设计。

模拟器中 Load Buffer 和 Store Buffer 分别只有一个。只有在两者均空闲的时候，Issue 阶段才会发出内存相关的指令。内存控制器单元每个时钟周期仅对一个 Buffer 进行操作。由此，可以保证不出现在内存上的 Data Hazard。

模拟器碰到分支时会停止发出指令，直到下一条指令的地址被求出。*CAAQA* 以某个 CPU 的浮点单元设计为例阐述了 Tomasulo 算法，因而，书上并没有提到如何处理分支的情况。所以，我在模拟器设计中做了这个修改。为了实现这个功能，我首先增加了 Branch 预留单元 (Reservation Station)。它的处理和其他 ALU 预留单元一致。所以，分支指令会被发送到这个预留单元上。与此同时，我在 RISC-V 32 个寄存器的基础上又新增了一个寄存器，称为 Branch 寄存器，专门用来保存分支处理的结果。一旦 Branch 单元空闲，就说明分支的下一条语句已经被求出。Issue 阶段只需要读出 Branch 寄存器上，`rs1` 和 `rs2` 两个寄存器的比较结果，结合当前分支指令和 PC，就可以求出下一条语句的地址，从而完成整个分支语句的执行过程。

3.3 RV32I 中较难处理的指令

在实现 Tomasulo 算法的过程中，我发现 RV32I 的部分指令需要多个硬件单元或者特殊处理才能完成。

JALR 指令需要将指令中 `rs1 + imm` 计算出来作为跳转的目标，而后将 `PC + 4` 存入 `rd` 之中。在 Tomasulo 算法的实现中，我将这一指令发送到两个计算单元。由 BRANCH1 计算 `rs1 + imm`，并且用 ADD1 计算 `PC + 4`。当 Issue 阶段发现这个指令时，流水暂停，直到 BRANCH1 计算出下一个指令的地址才继续发送指令。

在后文第 3.4 章中实现推测执行时，JALR 指令需要更加复杂的处理。由于使用了两个运算单元，它需要分配两个重排缓冲中的空位。两个空位中，第一个为 ADD1 的输出，并且在向第一个空位发送指令的时候，将指令当作 JAL 处理。（JAL 指令在发出的时候就能确定跳转目标，在最后提交的时候就只需要把 `PC + 4` 存入 `rd`，提交的过程和 JALR 拆分的第一个操作一致）。第二个操作和分支指令类似，只不过是无条件跳转。当提交 JALR 的第二个操作时，就相当于分支预测失败，要清除重排缓存。

分支指令的实现也比较困难。我给分支指令分配了一个运算单元 BRANCH1，它的功能和其他的 ALU 单元一致。当发出分支指令时，流水线停止，将分支按照跳转方法转换为 ALU 可以处理的运算，并将运算结果存入新增的第 33 个寄存器中。当这个专用寄存器被写入、BRANCH1 运算单元空闲时，发出阶段就能确定下一条指令的位置。这个过程会导致大量的时钟周期浪费在处理分支上，流水线在处理分支的时候不能发出新的指令。这一问题也在后文第 3.4 章中通过硬件推测解决了。

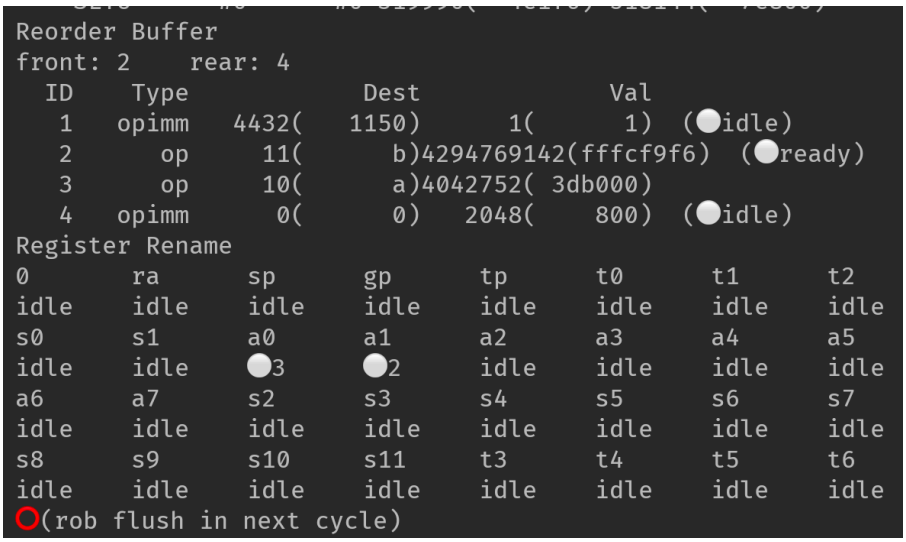

图 8: Debug 实况

> Aside **调整 Debug 输出，或许会有好心情**
> 在实现乱序执行的过程中，我时常要观赏一个时钟周期里的所有硬件数据，来判定问题发生的地方。这些数据包括：发出的指令、PC、寄存器的值、寄存器重命名情况、Reservation Station 的情况。一个 Cycle 的信息可以打一整屏幕。此时，对 Debug 输出排版就显得尤为重要。对于 32 个寄存器，我在调试信息中打出了它们的名字，并同时输出十进制值和十六进制值。对于正在处理数据 (busy) 的单元，我会用 Emoji 符号 (比如红色的圈圈) 突出显示。

3.4 推测执行与重排缓冲

在完成基于 Tomasulo 算法的乱序执行后, 根据 *Computer Architecture: A Quantitative Approach* 的介绍，我对设计进行改进，添加了重排缓冲 (Reorder Buffer)，实现了推测执行 (Hardware Speculation)。

在这里，我完全按照 *CAAQA* 设计了整个模拟器。在其中，我也碰到了书上没有讲明白的地方，并自己解决。

当分支预测出错的时候，我在清除 ROB 的同时，还将所有的运算单元恢复到初始状态。比如，内存单元需要记录自己 Load 或者 Store 到哪一步。这个单元在清除 ROB 的同时需要被重置。

与此同时，我在设计中保留了 Store Buffer。我并没有按照书上实现 store 指令直接发送到 ROB 中的功能，因而保留 Store Buffer 简化了我的模拟器设计。

在完成分支预测时，我在 ROB 中新增了一个寄存器，用来存储当前分支指令在 Issue 阶段预测的下一条指令的地址。由此，在提交指令的时候，模拟器才能判断分支预测是否正确，并请求清空 ROB。

至此，乱序执行的 RISC-V 模拟器基本完成。