计算机组成原理

Homework Solution 7

Made by TA

2023 年 5 月 30 日

题目 1. 处理器!

在学习了单周期、多周期以及流水线处理器的设计之后，我们来系统分析一下这三种设计在性能上的差异。

假定访问 IM、DM 的延迟均是 3ns，ALU 的延迟为 2ns，寄存器堆的读写延迟均为 1ns，其他所有延迟忽略不计。某程序包含 1000 条指令，其中 lw 占 20%，sw 占 20%，R-type ALU 指令占 40%，beq 指令占 20%。

对于流水线处理器，50% 的 lw 指令之后紧跟与之相关的 R-type ALU 指令，因此需要停顿一个时钟周期。此外，流水线对于 beq 的预测准确率为 70%，预测失败时要浪费两个时钟周期冲刷流水线。

考虑 RISC-V 32I 指令集架构的单周期 CPU、五段多周期 CPU 以及五级流水线 CPU，根据以上信息回答下面的问题。

(1) 在单周期 CPU 上执行这 1000 条指令需要多长时间？

(2) 对于多周期 CPU，其最小时钟周期是多少？在多周期 CPU 上执行这 1000 条指令需要多长时间？

(3) 对于流水线 CPU，其最小时钟周期是多少？在流水线 CPU 上执行这 1000 条指令需要多长时间？

(4) 结合以上内容，讨论影响流水线 CPU 性能的因素有哪些？

解答:

(1) 我们先来计算各指令的延迟：

- R-type ALU 指令：IF(3ns)+ID(1ns)+EX(2ns)+MEM(0ns)+WB(1ns)=7ns；
- sw 指令：IF(3ns)+ID(1ns)+EX(2ns)+MEM(3ns)+WB(0ns)=9ns；
- lw 指令：IF(3ns)+ID(1ns)+EX(2ns)+MEM(3ns)+WB(1ns)=10ns；
- beq 指令：IF(3ns)+ID(1ns)+EX(2ns)+MEM(0ns)+WB(0ns)=6ns。

单周期 CPU 需要支持所有指令在一个周期内完成，因此时钟周期最小为 10ns。执行完这 1000 条指令需要 $1000 \times 10 = 10000$ns。

本题的最多误解为 7800ns，认为一条指令执行完成后即可执行下一条指令。实际上单周期 CPU 的时钟周期是固定的，即使指令已经执行完成（例如 beq），也依然需要等待下个时钟周期到来后才能执行下一条指令。

(2) 五段中，最长延迟为 IF(3ns) 和 MEM(3ns)，因此多周期 CPU 的时钟周期最小为 3ns。各指令所需要的时钟周期数如下：

- R-type ALU 指令：4 时钟周期；
- sw 指令：4 时钟周期；
- lw 指令：5 时钟周期；
- beq 指令：3 时钟周期。

因此总耗时为

$$\begin{aligned} T &= 3 \times (40\% \times 4 + 20\% \times 4 + 20\% \times 5 + 20\% \times 3) \times 1000 \\ &= 3 \times 4 \times 1000 \\ &= 12000\text{ns} \end{aligned}$$

本题的最多误解为 15000ns，认为每条指令都需要经历完整的五个阶段。emmm 只能说没有理解好多周期的设计理念。多周期的目标就是为了解决单周期中指令提前结束而时钟周期未结束的问题，因此设计了不同的阶段。

当然，由于一次只有一个阶段在工作，因此多周期 CPU 并不一定能提升效率。唯一的优点或许在于可以将指令与数据存储器合并而不会出现结构冒险了。

(3) 五段中，最长延迟为 IF(3ns) 和 MEM(3ns)，因此流水线 CPU 的时钟周期最小为 3ns。对于理想流水线（没有停顿），总耗时为

$$T_{ideal} = 3 \times (1000 + 4) = 3012\text{ns}$$

lw 指令引起的延迟为

$$T_{lw} = 3 \times 1000 \times 20\% \times 50\% = 300\text{ns}$$

分支预测失败引起的延迟为

$$T_{beq} = 3 \times 1000 \times 20\% \times (1 - 70\%) \times 2 = 360\text{ns}$$

因此总耗时为

$$T = T_{ideal} + T_{lw} + T_{beq} = 3672\text{ns}$$

本小题的正确率甚至比前两题更高.....

(4) 影响流水线 CPU 性能的因素有如下几个方面：

- 流水线时钟频率与阶段数。对于理想流水线来说，其总耗时计算公式为

  $$T = \text{指令数目} \times \text{时钟周期} \times (\text{流水线阶段数} - 1)$$

  因此，对于同样的一段程序，时钟频率越低、流水线深度越深，则耗时越长；
- 指令的相关性与指令顺序。以 lw 指令为例，当流水线中频繁出现 Load-Use Hazard 时，流水线便会频繁停顿，进而影响整体性能；
- 控制相关的处理策略。对于分支指令，采用等待的策略相当于 100% 预测失败，采用分支预测则可以保证一定的预测成功概率。预测成功概率越高，带来的延迟就越小。理想流水线可以看作是 100% 预测成功。

以上三点答到两点即可得到满分。

题目 2. 乘法!

某一天，神秘的外星生命体降临了 3C102 教室，在惊呆了全班同学后迅速离开，并留下了一份神秘的文件。经过翻译，这份文件竟是一份用 Verilog 编写的乘法运算模块!

COD 助教们围着这个神秘的模块仔细研究。他们发现：这个模块**可以时钟同步地花费两个周期（两次上升沿）计算 32bits 整数的有符号乘法**（因为注释里是这么写的）。助教们大喜过望，计划将其接入实验框架之中，满足同学们长期以来对于乘法指令的渴望。十分幸运地，正在写作业的你被助教拉来一起讨论相关事宜。我们的故事便从这里开始了……

(1) 两个 32bits 整数相乘，结果至多是多少位？

(2) 可以在单周期 CPU 中接入该模块，以实现乘法指令吗？请分析原因。

助教们计划将该乘法模块接入流水线 CPU 之中。该模块在 EX 段接收到来自 ID/EX 段间寄存器的数据，经过两次上升沿的驱动，在 WB 段与 MEM/WB 段间寄存器同步输出计算结果。**注意：乘法模块同一时间只能进行一次乘法运算，其内部无法进行流水化处理。** 此外，助教们编写了如下的测试程序来检测乘法模块的工作情况：

```
addi t1 x0 -1
addi t2 x0 3
mul t0 t1 t2    // -1 × 3 = -3
add t0 t0 t0    // -6
```

请根据以上内容继续回答下面的问题。

(3) 当 mul 指令运行到 ID 段时，t1、t2 的正确结果位于什么位置（精确到某一段的连线或模块）？

(4) 当 add 指令运行到 ID 段时，t0 的正确结果位于什么位置（精确到某一段的连线或模块）？此时流水线应当进行怎样的操作以保证指令的正确执行？

解答:

(1) 2 个 32bits 整数相乘，结果至多是 64bits。例如

$$\text{0xFFFF\_FFFF} \times \text{0xFFFF\_FFFF} = \text{0xFFFF\_FFFE\_0000\_0001}$$

注意：无符号、有符号对于本题的结果没有影响。

(2) 单周期 CPU 每条指令都需要在一个时钟周期内执行完成，而我们的乘法器需要两个时钟周期才能计算出结果，因此无法接入单周期数据通路。

有的同学提出可以停顿 PC，等待乘法模块的结果计算出并写回之后再继续执行后续的指令。这样虽然可以解决问题，但并不符合单周期 CPU 一个时钟周期执行一条指令的设计原则。

(3) 如图 1 所示，此时 t1 位于 MEM 段 alu_ans 中， t2 位于 EX 段 alu_ans 中。

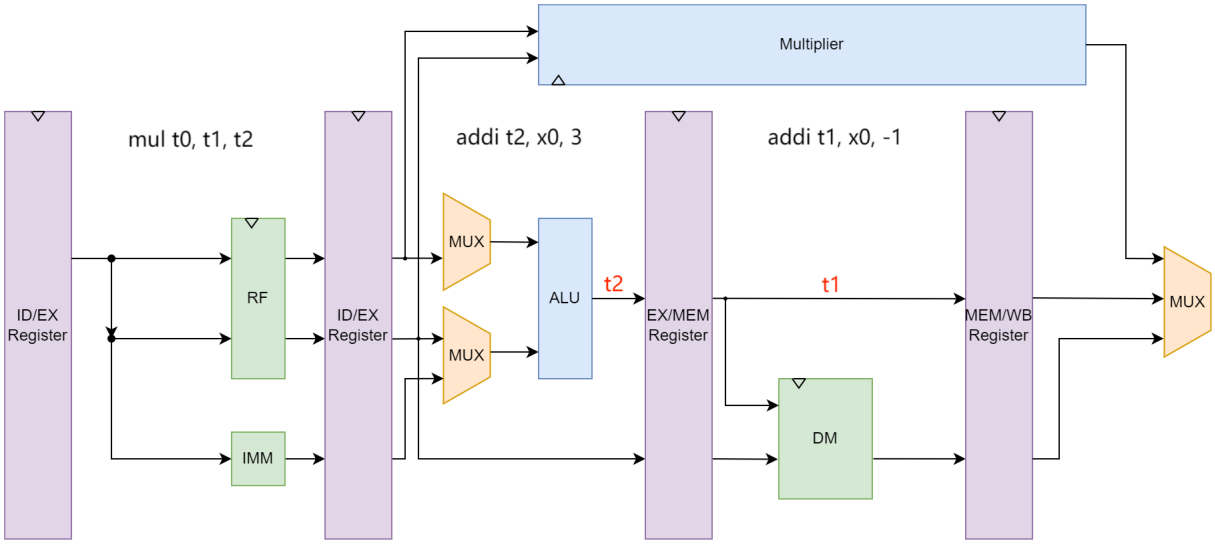

图 1: 第三小问数据通路示意图

(4) 如图 2 所示，此时 t0 位于乘法器模块中，且尚未算出。

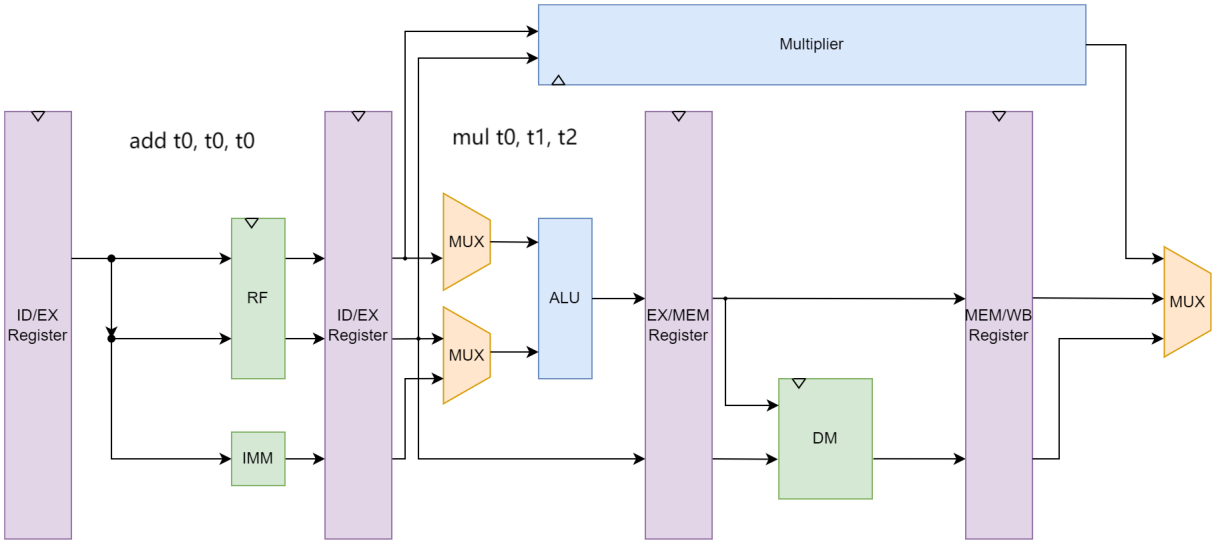

图 2: 第四小问数据通路示意图

由于 add 依赖 mul 的运算结果，假定我们在 EX - MEM&WB 阶段处理前递，则需要在

add 指令运行到 EX 段（此时结果依然未算出）时插入一个周期的气泡，等下个周期乘法器结果输出后再前递到 EX 段。

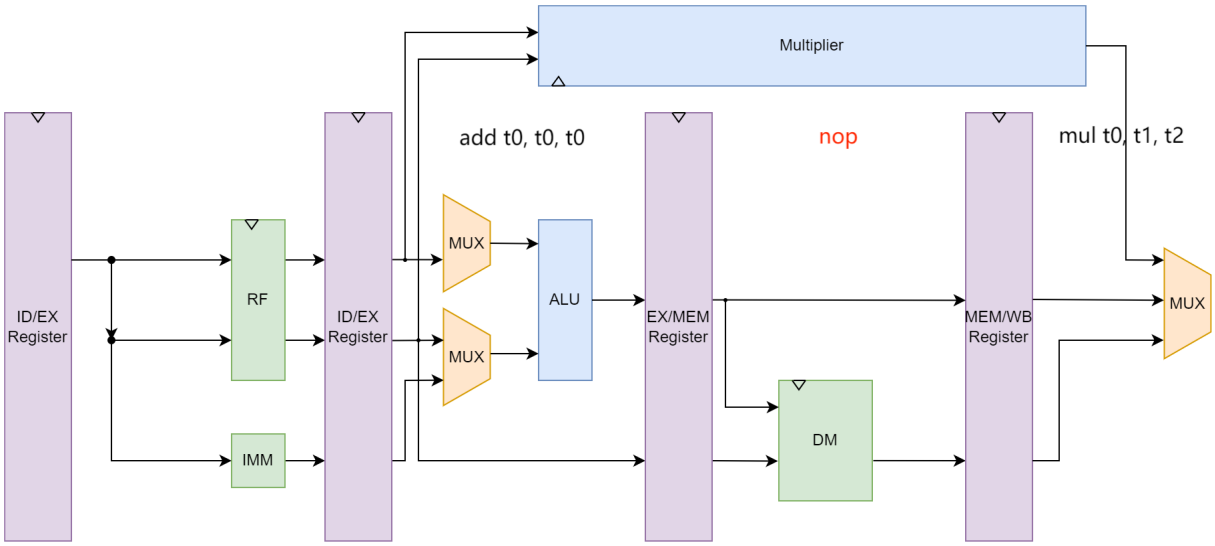

图 3: 插入气泡的示意图

题目 3. RISC-VI!

注：本题中增加的部分指令均不是 RV-32I 标准指令，请大家注意甄别！

又是某一天，WJD 助教突发奇想：能不能自己添加一些指令，得到属于自己的 RISC-VI 指令集呢？于是他设计了如下的几条指令：

- addmi：存储器-立即数加法指令。

  指令格式为 addmi rd, rs, imm，操作为 x[rd]←MEM[x[rs]]+sext(imm)。

- addmr：存储器-寄存器加法指令。

  指令格式为 addmr rd, rs1, rs2，操作为 x[rd]←MEM[x[rs1]]+x[rs2]。

- addmm：存储器-存储器加法指令。

  指令格式为 addmm rd, rs1, rs2，操作为 x[rd]←MEM[x[rs1]]+MEM[x[rs2]]。

为了保证指令能够正常执行，我们的数据存储器有多个只读端口，但仅有一个读写端口。假定我们的程序可以保证地址都是字节对齐的。请根据以上内容回答下面的问题。

(1) 将下面的 RISC-VI 指令序列用 RISC-V 32I 指令改写。你可以自己指定程序所需要的临时寄存器。

addi x5, x0, 0x14

addi x6, x0, 0x18

addmm x6, x6, x5

(2) 修改 RISC-V 32I 五级流水线的数据通路，使其支持这三条指令的正确运行（不考虑这三条指令涉及的冒险，支持指令功能即可）。提示：三条指令的处理流程可以是一致的，你可以以 addmr 为例介绍通路中的改动，以及该指令在各个阶段所需要进行的操作。

(3) 考虑如下的指令序列：

addi x5, x0, 1

addi x6, x0, 0

sw x5, 0(x6)

addmi x6, x0, 1

为了让流水线 CPU 能够正确执行该程序，在上一问数据通路的基础上应当增加怎样的设计？

(4) 相较 CISC 指令集，RISC 指令集通常采用 LOAD-STORE 体系结构，即存储器和寄存器之间的数据交互由专门的 load 和 store 指令负责，其他指令均不能访问存储器，所有的算数与逻辑操作均在寄存器中进行。请根据以上内容分析 RISC 指令集这种操作模式的优缺点。

解答:

(1) 改写后的程序如下。注意 addmm 最终写回的是寄存器堆。

addi x5, x0, 0x14

addi x6, x0, 0x18

lw x7, 0(x5)

lw x8, 0(x6)

add x6, x7, x8

(2) 在不增加流水线级数时，一种可行的方法是在 EX 段读取存储器，在 MEM 段进行算数运算。如图 4 所示：

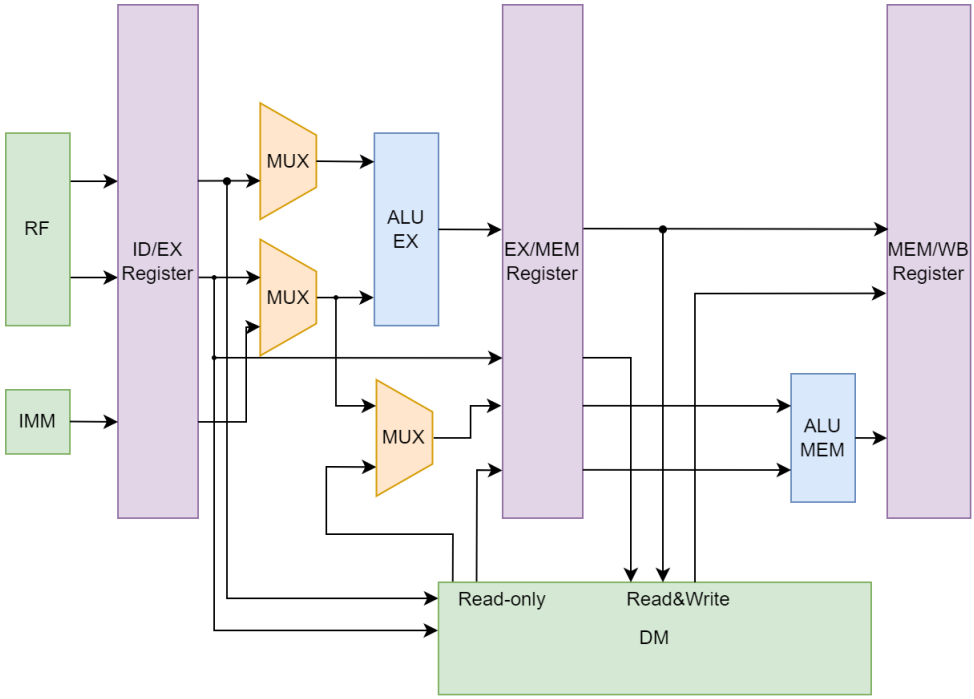

图 4: 题 3-2 数据通路示意图

addmr 指令在 EX 段通过只读端口，将 MEM[x[rs1]] 的值读出，并与 x[rs2] 一起送入 EX/MEM 段间寄存器。在 MEM 段，指令在另一个 ALU 中进行算数运算，并送入 MEM/WB 段间寄存器。

这里很多同学的设计都是正确的，但要注意一个阶段中的延迟大小。一般来说，读取寄存器堆、读写存储器、ALU 运算需要位于三个阶段，从而控制每个阶段的延迟规模。

当然，本题还有其他的实现方式。例如在 WB 段增加 ALU、在 MEM 和 WB 之间增加一个阶段变成六段流水线等等，不同的实现方式会对第三小题带来不同的影响。

(3) 对于指令序列

```
addi x5, x0, 1
addi x6, x0, 0
sw x5, 0(x6)
addmi x6, x0, 1
```

我们注意到在原有流水线相关的基础上，addmi 指令引入了访问存储器的相关性，即 EX 段 addmi 指令读取的 0x0000 地址与 MEM 段 sw 指令即将写入的 0x0000 地址相同。此

时有两种处理策略，一种是直接将 MEM 段的 rd1（dm_din）前递到 EX 段，交给 addmi 作为源操作数，另一种则是直接让数据存储器为写优先模式，保证只读端口读出的是即将写入的新值。

为什么会有这种相关性呢？实际上这是因为我们在 EX 和 MEM 两个阶段都可能读取存储器，这样就可能导致读取出的结果并不是存储器中的最新结果。如果自己的实现中依然只有 MEM 一个阶段访问存储器，则不会有本小题所说的相关性。

(4) RISC 指令集的 LOAD-STORE 体系结构优缺点可概括如下：

优点

- 执行效率高。对于大多数指令，其只需要从寄存器堆中进行算数与逻辑操作，而无需访问内存。因此指令的平均耗时更少（因为寄存器访问延迟远低于内存访问延迟）；
- 结构简单。大多数指令仅涉及寄存器堆与 ALU 部件，无需考虑内存因素，因此通路设计更为简单；
- 指令集简单。运算指令与访存指令均可以独立设计，而不需要针对每一种指令考虑其对内存的操作，降低了 CPU 的成本和功耗。

缺点

- 代码实现复杂。RISC 指令集需要额外的指令将数据从内存转移到寄存器中，因此增加了指令数目与相应的开销；
- 寄存器成本增加。由于所有操作均需要在寄存器中进行，RISC 指令集往往会需要更多的寄存器来暂存中间结果。