支持数据转发的 RISC-V 流水线

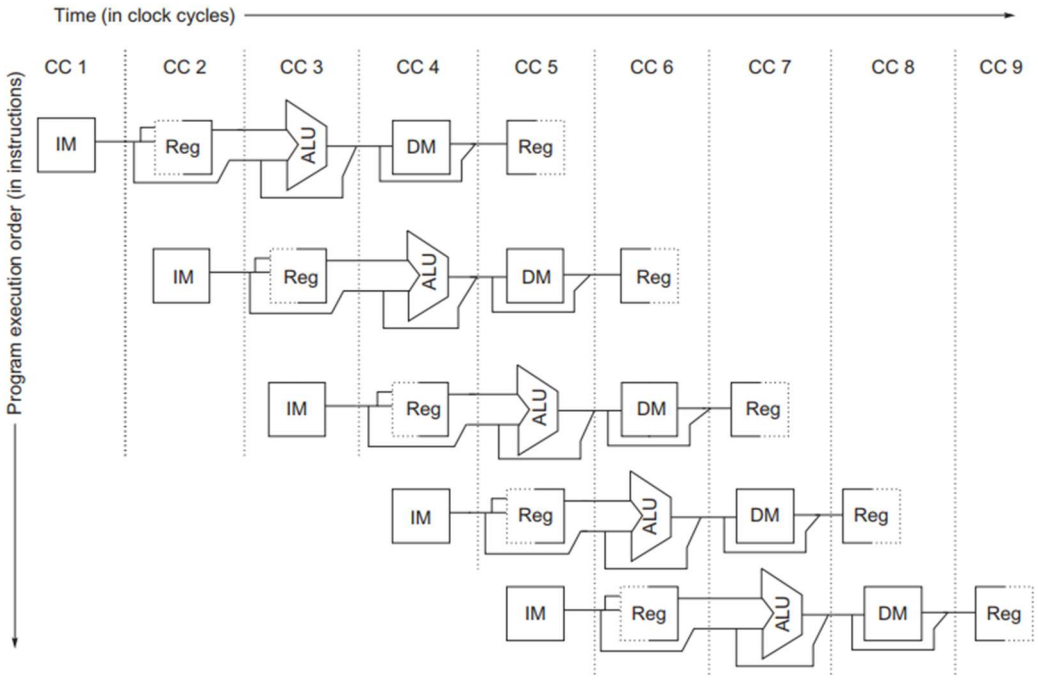
童敢 黄家腾 赵琛然

摘要：本次实验我们小组实现了一个经典的五级 RISC-V 流水线，基于哈佛结构，数据和指令分开存放，使用数据转发和 Stall 消除了数据相关，数据、堆栈、输入输出等通过逻辑隔离的方式在数据内存中实现，能够顺利运行给定的测试程序。

一、设计实现

（一）流水线结构

因为是计算机体系结构实验课程，所以流水线设计参照了教材《Computer Architecture A Quantitative Approach (6th Edition)》中的附录 C，沿用 MIPS 的经典五级流水线：取指——译码——执行——访存——写回，如下图所示：



最终版本的流水线各级的任务分工如下：

1. 取指：生成 PC 值，将值发送到指令内存，然后将获得的指令发送到译码阶段；
2. 译码：将指令进行译码，获取操作类型、操作数、目的地址等信息，同时分支跳转在这一阶段进行计算，将目标地址发回取指阶段，将译码得到的操作、操作数等信息发送到执行阶段；
3. 执行：包括各种算数逻辑运算单元，加减乘运算、逻辑运算、移位运算执行为 1 个时钟周期，除法单元为 33 个始终周期（使用试商法实现），在根据操作类型将相应的运算结果发送到访存阶段；
4. 访存：Load 和 Store 指令的执行，访问数据内存，将 Load 的结果发

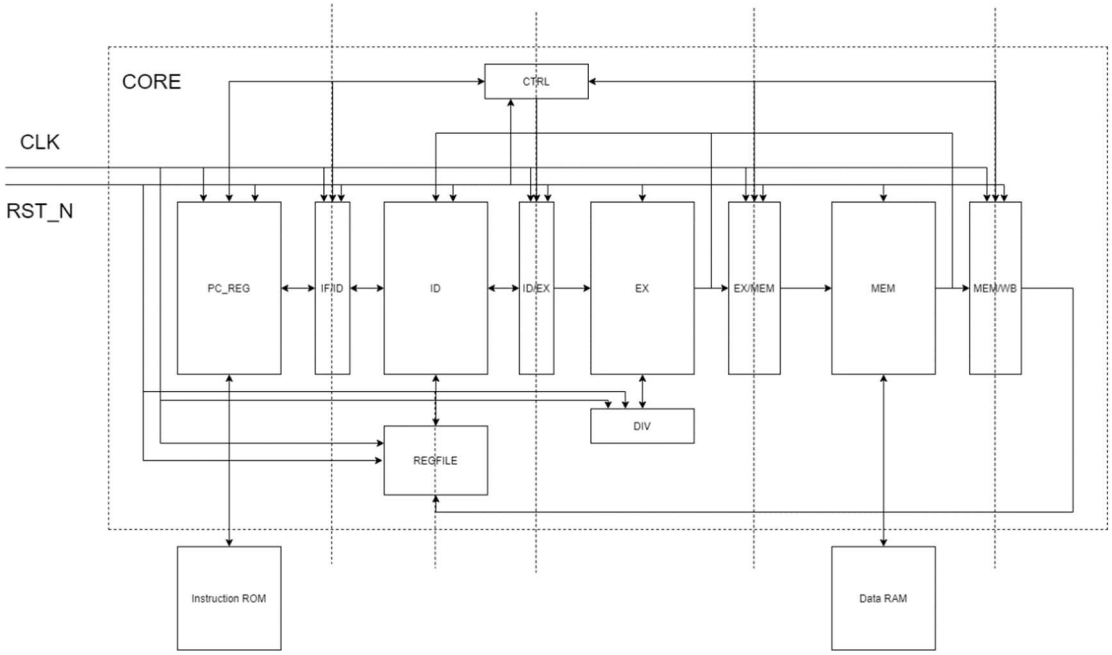
送到写回阶段；

5. 写回：将指令的运算结果写入寄存器，指令的生命周期结束。

（二）处理器结构

流水线的每级都为组合逻辑实现，级与级之间为时序逻辑实现，通过始终信号驱动级与级之间的模块，使流水线得以工作。

整体的处理器结构如下图所示：



1. 流水线

流水线分别由 IF/ID、ID/EX、EX/MEM、MEM/WB 和 REGFILE 控制。WB 阶段集成在了 REGISTERS 中，写逻辑为时钟驱动的，读逻辑为组合逻辑。PC_REG 即为 IF 阶段。

2. 数据转发

可以存在的数据转发为从 EX 阶段到 ID 阶段、MEM 阶段到 ID 阶段、WB（REGFILE）阶段到 ID 阶段，直接向 ID 阶段发送要结果和目的地址，如果和 ID 阶段需要的操作数地址一致，则使用转发的数据。考虑到数据的有效性，使用转发结果的优先级为 EX>MEM>WB。

3. LOAD 相关

如果 LOAD 指令的后一条指令将使用该数据，但 LOAD 的结果只能在 MEM 阶段结束后才能得到，存在 LOAD 相关。因此需要检测这种相关，在上一条指令是 LOAD 且加载的值本条指令需要使用时，则 STALL 一个周期。

4. 分支指令

分支指令包括有条件分支和无条件分支，除了 JAL 指令外，其他的指令都需要读取寄存器，因此这些指令分支目标的获取都放在了这一阶段。对于 JAL 和 JALR 而言，下一条指令一定会送到 ID 阶段但是不应该被执行，分支成功的有条件分支也是如此，因此此处也需要进行判断，如果分支成功，则 STALL 一个周期。

5. STALL

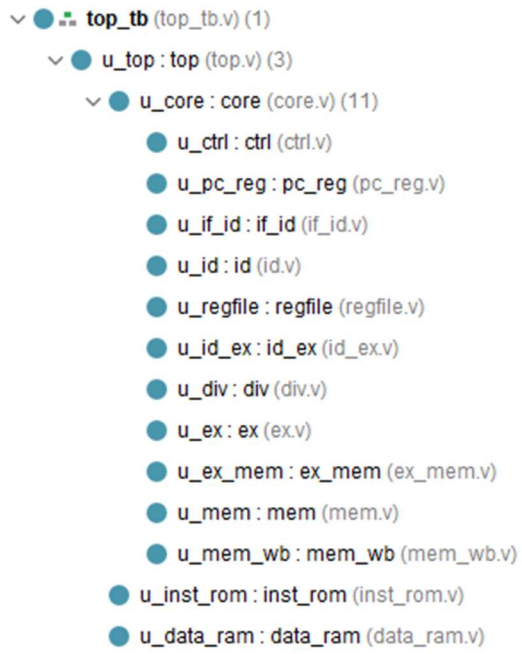
STALL 由 CTRL 模块控制，由各模块向 CTRL 模块发送请求，CTRL 再根据请求的模块位置，暂停在其之前的流水线。

（三）具体实现

1. 模块结构

核心模块为 CORE，包括除了指令 ROM、数据 RAM 之外的所有其他模块。CORE 相应接口与指令 ROM、数据 RAM 连接，将三者封装为 TOP 模块，再连接 TOP_TB 模块进行模拟。

模块文件结构如下：



2. 预定义头文件

“defines.vh”为头文件，包括涉及到的各种预定义的常量，方便编程使用。

头文件里面编码了全局常量、寄存器文件属性、数据内存属性、指令内存属性、指令功能码编码、运算资源编码等编码，需要调整时可以直接修改，而无需每个文件都找到对应的位置再取修改。

3. Memory 文件

指令 ROM 在仿真开始时会从 inst_rom.mem 文件中读取指令，起始地址为 0，一行代表四个字节的指令，每个周期 PC_REG 根据 PC 值指示的地址从中取数值。

数据 RAM 没有 Memory 文件，因为目前的程序还不涉及，默认内容全部为 x，执行 STORE 指令之后才会向数据 RAM 中写入，才能被 LOAD。

4. 堆栈和输入输出

这两部分都放在了数据 RAM 里，因为对于 RV 处理器而言，都是地址空间中的设备，只需要在逻辑上将数据 RAM 划分成为多个逻辑上独立的地址空间。

在我们小组的实现中，32 位地址前 27 位为 0 时表示 I/O 的地址空间，向该位置 STORE 数据即表示输出数值。但为了方便观察（内存太大了），在该模块中定义了小的 I/O 模块内存，用于在地址满足条件时也会向其中写入数据。

5. 实现顺序

实现的顺序参照了《自己动手写 CPU（雷思磊）》一书中关于 MIPS 处理器的实现，具体为先实现流水线，再实现数据转发，然后实现所有算数指令，再依次实现分支指令和访存操作，五个阶段的代码文件随报告一并提交。最终版本的代

码实现文件路径为：

04.MemoryAccess\MemoryAccess.srcs\sources_1\new

二、运行测试程序

（一）机器指令的生成

1. 修改源代码

由于提供的源代码文件 main.cpp 中包含了输入输出指令，因此我们做了简化：固定了 n 的值，其值可以在汇编指令中方便地修改；将输出语句删除，更改为直接返回 func1+func2 的数值，此处不只返回其中之一是因为经过测试不知何故不会计算另一个值，可能是被编译器优化掉了，但是猜测没有根据。

修改后的源代码见 main.c 文件。

2. 生成汇编指令

首先用开源的 riscv-gnu-toolchain 生成相应的编译工具，以支持当前已经实现的指令集，用到的二进制编译程序为 riscv32-unknown-elf-gcc，参考了 Dhrystone 中的 build 文件，编译生成汇编指令的命令如下所示：

```
riscv32-unknown-elf-gcc -march=rv32im -fno-stack-protector -fno-zero-initialized-in-bss -ffreestanding -fno-builtin -nostdlib -nodefaultlibs -nostartfiles -mstrict-align -S main.c
```

接着对生成的 main.s 进行修改，将返回 main 函数的指令更改为：

```
sw  a4,0(zero)
sw  a5,4(zero)
```

这两条语句表示将 func1 和 func2 运行的结果保存到输入输出地址空间的地址 0 和 4 处，即各 4 个字节。

修改后的源代码见 main.s 文件。

3. 生成机器码

依然可以使用工具链生成，但为了验证处理器运行的正确性，我们使用了在线的 RV 汇编模拟器（https://www.kvakil.me/venus/），将汇编代码输入即可模拟运行、下断点、单步运行等，同时也可以 dump 出机器指令，将机器指令复制到 inst_rom.mem 文件中进行仿真即可。

n=10 时生成的机器指令见 main.mem 文件。

（二）运行测试程序

1. 确定程序入口

由于没有实现 BootLoader 之类的东西，因此需要手动数出 main 的入口，将其地址作为 RstDisable 时（从复位状态开始运行时）生成的第一条指令地址，此时程序将从 main 开始执行，而不是从 0 开始执行。

2. 堆栈指针位置

堆栈指针设置为数据内存中间的位置，由于压栈、弹栈都会根据 sp 寄存器来访问堆栈，因此需要在开始时设置堆栈指针的位置。根据 RV 的汇编手册，sp 对应的寄存器为 x2，所以初始化时将 x2 寄存器的值设置为了 32'h00007ff0。

3. 运行测试程序

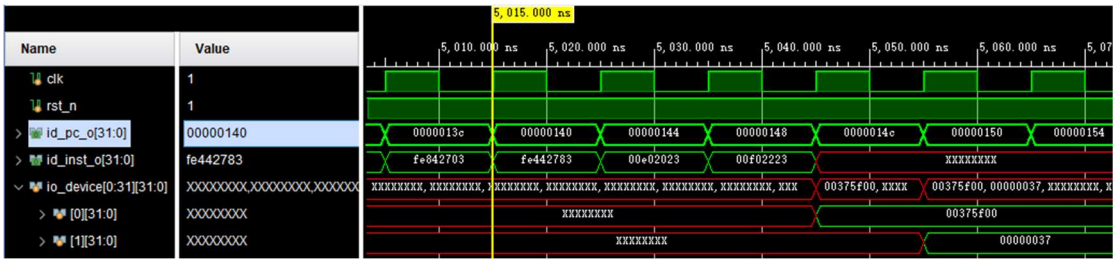
当 n=13 时，阶乘的结果为 6227020800，已经超过了 2^{32}，所以对于阶乘而言，堆栈的深度不是问题（只要内存足够大），而 n 超过 13 之后的结果溢出才是

问题。但我认为，这种溢出不应是处理器设计人员考虑的事情，而是程序开发人员需要处理的事情。

当 n>12 时并不意味着程序无法正常运行，只是阶乘的结果会溢出，只保留低若干位。

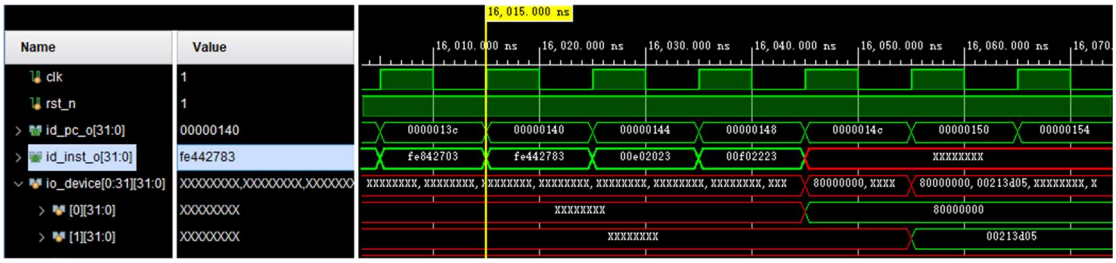
我们分别测试了 n=10 和 n=32 的情况，其中 n=10 时，阶乘结果应为 3628800（0x375f00），斐波那契数列结果应为 55（0x37），n=32 时，阶乘结果（不考虑溢出的正常结果）应为 263130836933693530167218012160000000，斐波那契数列结果应为 2178309（0x213d05）。

通过 vivado 进行仿真运行，当 n=10 时结果如下：



共运行 500 多个时钟周期，io_device[0]表示输出的阶乘的结果，io_device[1]表示输出的斐波那契数列的结果，二者结果均正确。

当 n=32 时结果如下：



共运行 1600 多个时钟周期，阶乘结果为 0x80000000，为溢出的结果，不正确；斐波那契数列结果为 0x213d05，结果正确。