10 (50) 条指令的流水线 CPU 设计报告

潘俊达 2021201626

10 条指令的流水线 CPU 设计报告

数据冒险中的转发和阻塞设计

为了给出统一的数据冒险（阻塞）解决方案，我们先总结一下数据冒险的情况：

数据冒险关注的是寄存器最新值是否滞留在 X_REG 内的情况，我们通过比较寄存器编号 readRegister1, readRegister2, writeRegister 来判断是否存在数据冒险。

例如指令在 EX 阶段发现**所需**寄存器的**最新值**位于 MEM_WB_REG 中并已**准备好**，那么就需要将这个值转发过来覆盖之前在 ID 阶段取得并逐阶段传递到这的值；然而如果我们不能马上操作这个值，我们还需要存储下来防止随着流水线的推进这个值离开流水线寄存器(X_REG) 写入 GPR，但我们又没有机会重新读 GPR（结构冒险）。

为此定义了几个量：

readyPipelineStage

X_REG 中的 writeRegisterInput（要写到寄存器的值）会在哪个阶段准备好

> 这个值可以通过控制信号 writeRegisterInputSrc 求，例如 writeRegisterInputSrc_aluResult 就说明 writeRegisterInput 是在 EX 阶段计算的，那么值就是在 EX_MEM_REG 中准备好的。

requiredPipelineStagex

X_REG 中的 readRegisterInputx（要读的寄存器的值）在哪个阶段真正需要

> 例如 sw 指令中，两个寄存器的值分别在 EX 和 MEM 阶段需要，并不一致

通用的数据转发与阻塞信号处理

我们用一种统一的结构 ForwardingUnit 处理所有的数据冒险，它接收上一阶段的 readRegisterResultx 和流水线寄存器输出尽量转发更新后的 forwardingResultx 和 stallx，流水线上的 ID, EX, MEM 阶段都各自有两个这个单元，这种单元包括：

- 最新数据依赖判断：从最近的下一个 X_REG 开始遍历所有的 X_REG，判断其 writeRegister 是否和当前的 readRegisterx 相等，最先找到的就是最新的数据，如果没有找到就说明不需要转发。

> 例如 EX 阶段需要遍历 EX_MEM_REG, MEM_WB_REG

- 阻塞判断：如果存在数据依赖，分别判断
  1. 写入的数据是否已经准备好
  2. 读取的数据是否当前需要

  如果没有准备好，而且当前需要，发出阻塞信号，stallx。
- 记忆模块：因为 ForwardingUnit 是组合逻辑，如果我们虽然得到了数据但却不得不阻塞（MDU 忙，另一个操作数还需要阻塞等待），那么之后这个值就可能离开流水线，所以我们需要一个寄存器来存储这个值，以及一个标记位来标记这个值是否有效，当这个值有效时，forwardingUnit 的组合逻辑会直接输出这个值。另一种方法是在 ID 阶段预判并阻塞，但这显然会导致更多的阻塞。

> - 上升沿：当需要阻塞时（stall1 || stall2 || otherStall == true），forwardingUnitx 在上升沿存储当前的 forwardingResultx，有效位为阻塞信号取反 ~stallx

> - 上升沿：当阻塞解除时（或者说只要正常执行完当前阶段），清除有效位
> - 组合逻辑：当需要转发时，判断上周期存下的 memoryForwardingResultx 是否有效，有效则直接输出，否则按照正常流程处理

阻塞的具体实现

通用的阻塞实现，以 EX 需要阻塞为例：

EX 需要的数据还在处理中，那么 IF, ID, EX 需要保持数据不变，同时写信号清零，EX 向 EX_MEM_REG 写入气泡。

MDU 忙也是类似的处理。

> 这保证了之前阶段的行为一直在重放，同时因为不写 GPR 或 DM，所以重放的内容也不变。
>
> 在五级流水里，MEM 和 WB 是不会阻塞的，所以不处理写寄存器的信号和写DM的信号是安全的，仅有写 PC 的信号是需要被清零的。

特殊情况：同时有两个阶段需要阻塞

此时我们需要保证阻塞的阶段是最早的那个，在流水线上看就是最后的阶段，做那个阶段阻塞的行为。

优势

- 完整和统一的转发和阻塞控制
- 精确的阻塞：通过指定 requiredPipelineStage，我们可以在更晚的时候再考虑阻塞，从而减少阻塞的次数，提高性能。在这个问题上本实现已经是最优解。
- 利用记忆模块避免了为了解决特殊情况的明显的性能损失或者结构明显改变。

正确性证明

1. 不需要转发的情况：如果在 ID 阶段时没有发现数据依赖，说明 ID 已经取得了最新的数据，随着流水线的推进，这个值会传递下去，取值正确。
2. 需要转发的情况且不需要阻塞的情况，从 ID 阶段开始，每个阶段都会尝试获取一次最新值，直到需要的那一阶段，所以只要在需要的阶段之前数据准备好了，就一定会被转发到，并存储到 X_REG 中被传递下去，取值正确。
3. 需要转发的情况且需要阻塞的情况，如果在需要的阶段时数据还没有准备好，那么就会发出阻塞信号，直到数据准备好，此时将值取到，取值正确。
4. 以上我们证明了我们必然会在执行前“取到过”最新值，记忆模块保证了我们能一直持有它直到不再需要。

综上，我们解决了所有的数据冒险问题。

实现细节

实际实现时，为了方便，没有把记忆模块写进转发单元，而是把记忆模块写在了 EX_MEM_REG 里，然后组合逻辑传给 ForwardingUnit 处理。这样 ForwardingUnit 仍然是纯组合逻辑元件。

50 条指令的流水线 CPU 设计报告

50 条和 10 条的区别是引入了 MDU 和 非对齐内存读写，非对齐内存读写只需要用组合逻辑，组合出真实要读写的值即可。

MDU 如果不考虑忙，则其行为和 ALU 一致，所以只考虑 MDU 忙的处理：

MDU 相关的功能

MDU 相关的阻塞

首先触发 MDU 忙的乘除操作按照定义直接通过流水线即可，不需要阻塞。

之后到来的要用到 MDU 的指令会触发阻塞，即

```
assign stallEX = stallEX1 | stallEX2 | (mduBusy & controlSignal.mduUse);
```

MDU 相关的数据冒险

统一分析就是，因为读写 MDU 都在EX阶段，一定保序，所以不要考虑转发之类的问题

优势

MDU 相关的阻塞当且仅当 mdu 忙且要用 mdu，在不考虑改变指令顺序的情况下已经是最优阻塞。

其他额外功能

syscall

实现了 syscall 中的 print integer 和 exit (terminate execution)

实现方法是将 syscall 视作正常指令，其需要读取 v0 和 a0 寄存器，然后根据 v0 寄存器判断 syscall 行为，根据 a0 输出。以上内容被实现在 WriteBack 阶段，以避免产生阻塞。

所以在算法测试部分，我的输出能做到和标准输出完全一致（即正确区分 syscall 是停止还是输出）。

我的输出：

```
@000030e8: $ 4 <= 00000001
         1
@000030f0: $ 2 <= 0000000a
./5-WriteBack.sv:71: $finish called at 4305 (1s)
```

Mars 输出：

```
@000030e8: $ 4 <= 00000001
1@000030f0: $ 2 <= 0000000a
```

exception

实现了简单的异常检测（没有异常处理），如果遇到异常（这里假设只有算数异常）：

1. EX 层传递一个类似气泡的东西：写信号清零，但是记录 pcValue 和标记这里有异常。
2. MEM 和 WB 层正常执行操作（即保证异常前的指令执行完毕）
3. EX 前的层冻结（实际上因为没有实现异常处理的相关逻辑，所以这个操作没有什么实际作用）
4. 异常传递到 WB 时终止程序，输出异常信息。

效果：当有算数异常时，我的输出和标准输出的长度完全一致

我的输出：

```
......
@00003120: $17 <= 0000ff0c
@00003124: $ 1 <= 2279ff0c
Runtime exception at 00003128: arithmetic overflow
./5-WriteBack.sv:85: $finish called at 905 (1s)
```

Mars 的输出

```
@00003120: $17 <= 0000ff0c
@00003124: $ 1 <= 2279ff0c
Error in D:\Code\mips\pipeline-tester-py\test.asm line 85: Runtime exception at
0x00003128: arithmetic overflow

Processing terminated due to errors.
```

How to run

Just run

0. install iverilog (and vvp)
1. `cd src/ && make`
2. place `code.txt` in the `src/` directory
3. `make run`

Run with pipeline-tester-py

run.json

```json
{
    "compiler": "iverilog",
    "iverilog_path": "iverilog",
    "vvp_path": "vvp",
    "vivado_path": "C:/Xilinx/Vivado/2018.3/bin/vivado.bat",
    "iverilog_params": [
        "-g2012"
    ],
    "test_bench_only": true,
    "mars_run_params": [
        "nc",
        "db",
        "500000",
        "ae2",
        "mc",
        "CompactDataAtZero"
    ],
    "im_length": 2048
}
```

iverilog 版本问题

测试过的 iverilog 版本为

- `Icarus Verilog version 12.0 (devel) (s20150603-1539-g2693dd32b)` 即 Icarus Verilog for Windows v-12
- `Icarus Verilog version 12.0 (stable) ()` 即 Github 最新 Releases 编译构建的

已知 Ubuntu20.04 下 `apt install iverilog` 的版本是不兼容的

Results with cycle count

通过精准要求阻塞（即 10inst 阶段设计的 requiredPipelineStagex 量），性能可以平均提升 16.2%

code	cycle_decode	cycle_stage	speedup
add-1.asm	50023	50021	1.00004
add-2.asm	35023	30023	1.16654
add-4.asm	32527	25025	1.29978
add-8.asm	31277	22525	1.38855
floyd.asm	14492	11278	1.28498
gcd-hardmul.asm	450	430	1.04651
gcd-softmul.asm	320	299	1.07023
lfsr.asm	10021	9022	1.11073
pointer-chasing.asm	12451	11393	1.09286
mean			1.16225

> 原始输出在 doc/output/decode.txt 和 doc/output/stage.txt 中
>
> 复现用魔改版脚本 https://github.com/panjd123/pipeline-tester-py