

实验名称：五级流水线 CPU 设计

实验人：谢志康

学号：22307110187

实验时间：2024. 5. 3

参考学习——

[关于流水线的三种冒险 - 知乎 \(zhihu.com\)](#)

[手把手代码实现五级流水线 CPU——第三篇：流水线控制逻辑 fwb-CSDN 博客](#)

[从零开始写 riscv 处理器（五）数据冒险：停顿与前递 load use 数据冒险-CSDN 博客](#)

流水线三种冒险情况——

1. **数据冒险**：当指令在流水线中重叠执行时，后面的指令需要用到前面的指令的执行结果，而前面的指令尚未写回导致的冲突，称为数据冒险（也称为数据相关性）。
2. **结构冒险**：当一条指令需要的硬件部件还在为之前的指令工作，而无法为这条指令提供服务，那就导致了结构冒险。（这里结构是指硬件当中的某个部件、也称为资源冲突）。（我们架构 ibus 和 dbus 分开，不存在此问题）
3. **控制冒险**：如果现在想要执行哪条指令，是由之前指令的运行结果决定，而现在那条之前指令的结果还没产生，就导致了控制冒险（实际上就是 riscv 的跳转指令引起的）。

Div_rem_sig

整体在之前多周期 CPU 代码上完成，基本功能实现代码与之前相同，将 CPU 执行方式改成五级流水线架构。参考以下结构设计——

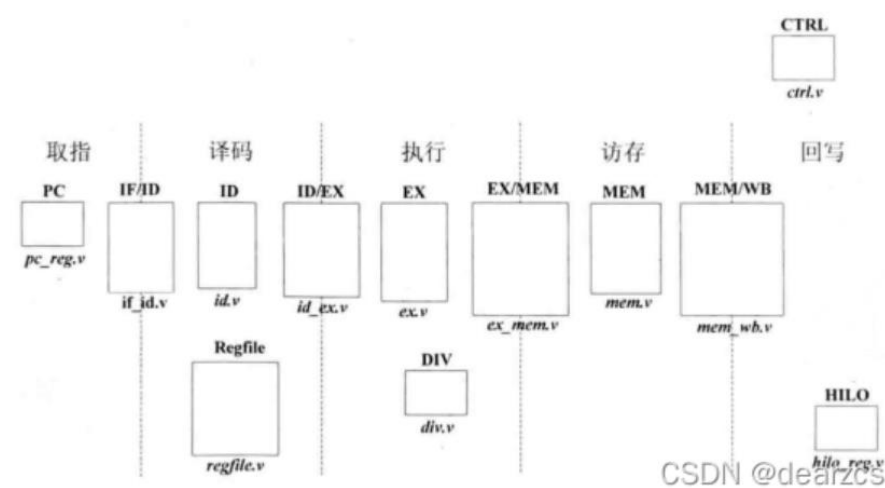


Figure 1

在我原本的 CPU 中，ex 阶段的乘除余数等指令是多周期，使用暂停处理（没算完不取下一条 instr），mem 阶段访存也是多周期，同样使用暂停处理（访存未结束 stall 住，不执行 wb 操作，等访存结束整个 CPU 继续跑），将这样的多周期 CPU 改成流水线，本质上和单周期到流水线相同。如下图所示——

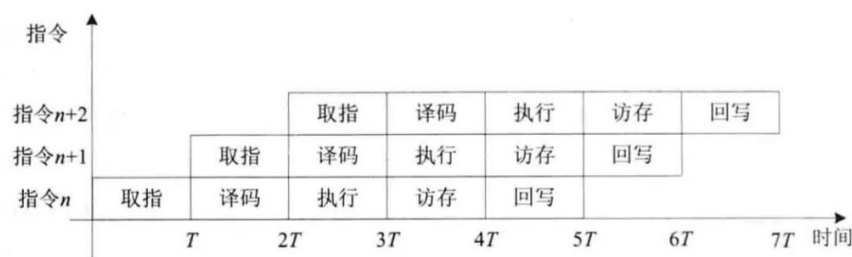


图 3-3 五级流水线示意图 知乎 @wataru030

Figure 2

执行时间和访存时间不止一个 T ，但每一次操作都取总周期数最大的一个作为当前操作的周期数。譬如，译码、执行、访存同时进行，译码所需 1 周期，执行除法所需 64 周期，访存所需 10 周期，总的这一次操作，CPU 会在 `stall_this_alu_div` 信号下停止 64 个周期（代码实现上就是空转），等计算完成后继续往后跑，到下一个流水线时间段。

模仿 figure1 的方式，在原本的架构上新增了 `IF_ID_PIPE`，`ID_EX_PIPE`，`EX_MEM_PIPE`，`MEM_WB_PIPE` 四个中间过渡模块，即用来实现流水寄存器。新增 `hazard` 模块处理各种冒险情况。其余如 `fetch.sv`，`control.sv` 进行了微小改动（新增接口用于初始化 `if_id` 的某些信号），其余模块无需改动。

IF 阶段: `Fetch.sv` `NPC.sv` `PC.sv`

`Fetch.sv` 取指，若一切准备就绪（指当前周期访存完成，下周期指令已经准备好），从 `ibus` 中取出下一周期的指令 `if_instr`。同时，对 `id_instr` 进行初始化，取 `if_instr` 时将 `id_instr` 链接过去（这里是 debug 试错试出来的），如果在本周期 `if_instr` 取指成功，下个周期就应该执行本条 `instr`，而若没有初始化，`id_instr` 为空值（起始状态），整个 CPU 没法跑起来，因此将他们先行链接，保证 CPU 往下跑，而 `id_instr` 并不会一直同步为 `if_instr` 的值，在 `IF_ID_PIPE` 模块会根据 `flush` 以及 `stall_hazard` 传入值情况将其修改为正确的要执行的 `id_instr`。达到了 `instr` 在 `IF_ID` 过渡阶段的正确传递功能。

`NPC.sv` 利用 `ex` 阶段算得的各个值决定下一个 PC。新增了一个接口，`output` 一个信号 `HD_br`，告诉 `hazard` 模块遇到了分支指令要进行处理（不止 b 型指令，还有 j 型）

`PC.sv` 根据 `NPC` 值给出下一个周期的 PC 值。

IF_ID 阶段: `IF_ID_PIPE.sv`

`IF_ID_PIPE` 作用将取指阶段的结果（`if_pc`，`if_instr`）在下个时钟周期传到译码阶段，特别的，输入由 `hazard` 模块判断出的 `stall_hazard` 和 `if_id_flush` 信号（后面 `hazard` 模块详述）

```

always_ff @(posedge clk) begin
    if(reset == `RstEnable || flush) begin
        id_pc = 64'h00000000_80000000;
        id_pc4 = 64'h00000000_80000004;
        id_instr = 32'b0;
    end else if(stall) begin
        id_pc = id_pc;
        id_pc4 = id_pc4;
        id_instr = id_instr;
    end else begin
        id_pc = if_pc;
        id_pc4 = if_pc4;
        id_instr = if_instr;
    end
end
end

```

Flush 信号表示遇到了分支指令。即，上一个周期解析的 id_instr 为分支指令，上一个周期 flush 被置为 1，此时应该清空 IF_ID 流水线寄存器，达到正确处理跳转指令的目的。具体功能是，将 id_instr 设置为 0，即相当于插入了一个周期的 bubble，其它器件先跑空指令（啥都不做），等到本周期处理完是否跳转再取到正确的 NPC。

2.控制冒险

• 问题:

由于需要每一次取值操作后，必须马上确定下一条指令的地址

- | | |
|---|---|
| 1 | 当取出指令为ret时，下一条指令需要从栈中读出，因此必须等到访存操作结束后才能确定下一条指令的地址 |
| 2 | 当取到的指令是分支条件指令时，流水线无法立即进行立即判断是否进行跳转，通过执行阶段后才能确定跳转 |

• 方式:

暂停执行新指令

Stall 指令表示出现数据冒险（相关性），采用暂停的策略，空跑 CPU。

ID 阶段: SEXT.sv Control.sv ALU_input_MUX.sv ALU_SIGN.sv

这几个模块较之前多周期架构没有改变。唯一 ALU_input_MUX 中先一步将 ex_operand 与 id_operand 进行了链接，也是为了处理初始化的问题。即，第一次 id_operand 已经有值，ex_operand 还未定义，下一个周期 ALU 开始计算 ex_operand 会出现问题，因此提前将其链接，若链接错误，下一个周期 ex_operand 会根据前递信号、顺序执行信号和 rst 信号重新得到正确的值，若链接正确则就是直接进行计算（选择顺序执行信号），不会产生问题。

SIGN 模块有一个地方 bug 搞了很久，是 ex_signed_A ex_signed_AW ex_signed_B ex_signed_BW 四个信号的阻塞赋值或是非阻塞赋值的问题，若非阻塞，他们会在下个周期才被修改，而下个周期已经要靠这四个信号计算 divu, mulu 等指令了，就会乱掉。因此最后采用阻塞赋值，在当前周期就被顺序执行掉，相当于下个周期 ex 要执行时，所有原操作数已经准备好，就可以直接开始计算不会出错。

ID_EX 阶段: ID_EX_PIPE.sv

ID_EX 阶段的流水寄存器。Id_ex_flush 则清空流水线寄存器（相当于加入 bubble，该信号由 hazard 模块产生），特别的，两个源操作数由 forward 信号给出（hazard 模块产生），用于处理数据冒险（采用前递的方式处理数据冒险，具体在 hazard 模块解释）

EX 阶段: RegFile_wD_MUX.sv ALU.sv ALU_DIV.sv ALU_DIVW.sv ALU_MUL.sv

ALU_output_MUX.sv

与多周期架构相比没啥变化，算不完的还是 stall，不会影响啥。

EX_MEM 阶段: EX_MEM_PIPE.sv

与之前相同。

MEM 阶段: MEM.sv

访存模块，与之前相同。

MEM_WB 阶段: MEM_WB_PIPE.sv

与之前相同。

WB 阶段: RegFile.sv

寄存器写回模块，与之前相同。内存写回在 MEM 阶段做了。

冲突冒险处理: Hazard.sv

数据冒险处理——

- 识别:

指令在译码阶段读取寄存器时，通过读取寄存器的id值来分别与执行阶段、访存阶段以及写回阶段所执行指令的目的寄存器进行对比

1 | 如果存在寄存器id值相等的情况，就说明指令之间存在数据相关，那么该指令就要在译码阶段等待

通过取寄存器 id 值分别于 ex mem wb 阶段的 write_register 相比来依次检测相邻间隔、间隔一条指令、间隔两条指令的冒险。

```
assign RAW_A_rD1 = (ex_wR == id_rR1) && ex_rf_we && rf1_occupied && (ex_wR != 0);
assign RAW_A_rD2 = (ex_wR == id_rR2) && ex_rf_we && rf2_occupied && (ex_wR != 0);

assign RAW_B_rD1 = (mem_wR == id_rR1) && mem_rf_we && rf1_occupied && (mem_wR != 0);
assign RAW_B_rD2 = (mem_wR == id_rR2) && mem_rf_we && rf2_occupied && (mem_wR != 0);

assign RAW_C_rD1 = (wb_wR == id_rR1) && wb_rf_we && rf1_occupied && (wb_wR != 0);
assign RAW_C_rD2 = (wb_wR == id_rR2) && wb_rf_we && rf2_occupied && (wb_wR != 0);
```

使用前递的方式处理冒险，若有任意冒险，将前递使能置为 1。

```
assign forward_A_sig = RAW_A_rD1 || RAW_B_rD1 || RAW_C_rD1;
assign forward_B_sig = RAW_A_rD2 || RAW_B_rD2 || RAW_C_rD2;
```

接下来确定前递 operand 值，考虑到可能同时发生多次冒险，按照 A>B>C 的优先级将操作数前递（前递具有优先性，应先处理 EX 冒险）。详见 hazard.sv 代码。

最后处理特殊的 load_use 型冒险：

在数据冒险的情况中，ld类指令的前递方式相较复杂，因为load指令回写的数据是从data memory中取出，ALU结果并不是最终回写结果。所以，当一条指令试图在加载指令写入一个寄存器之后读取这个寄存器时，前递不能解决此处的冒险。此种情况的数据冒险称为 **Load-use型冒险**。

当加载指令后跟着一条需要读取加载指令结果的指令时，流水线必须被阻塞以消除这种指令组合带来的冒险。将流水线暂停一个周期后，前递逻辑就可以处理这个相关并继续执行程序了。

因此，除了一个前递单元外，还需要一个**冒险检测单元**。该单元在 ID 流水线阶段操作，从而可以在加载指令和相关加载指令结果的指令之间加入一个流水线阻塞。这个单元检测加载指令，冒险控制单元的控制逻辑满足如下条件：

if (ID/EX.MemRead

and((ID/EX.RegisterRd = IF/ID.RegisterRs1) or (ID/EX.RegisterRd = IF/ID.RegisterRs2)))

stall the pipeline

表明如果IF/ID寄存器堆中的rs1、rs2与ID/EX寄存器堆中的rd相同，且检测为load指令(只有load指令的MemRead=1)，指令会停顿一个时钟周期。

解决方法对应代码——

```
logic load_use = (RAW_A_rD1 || RAW_A_rD2) && (ex_rf_wsel == `RF_WSEL_DBUS);
```

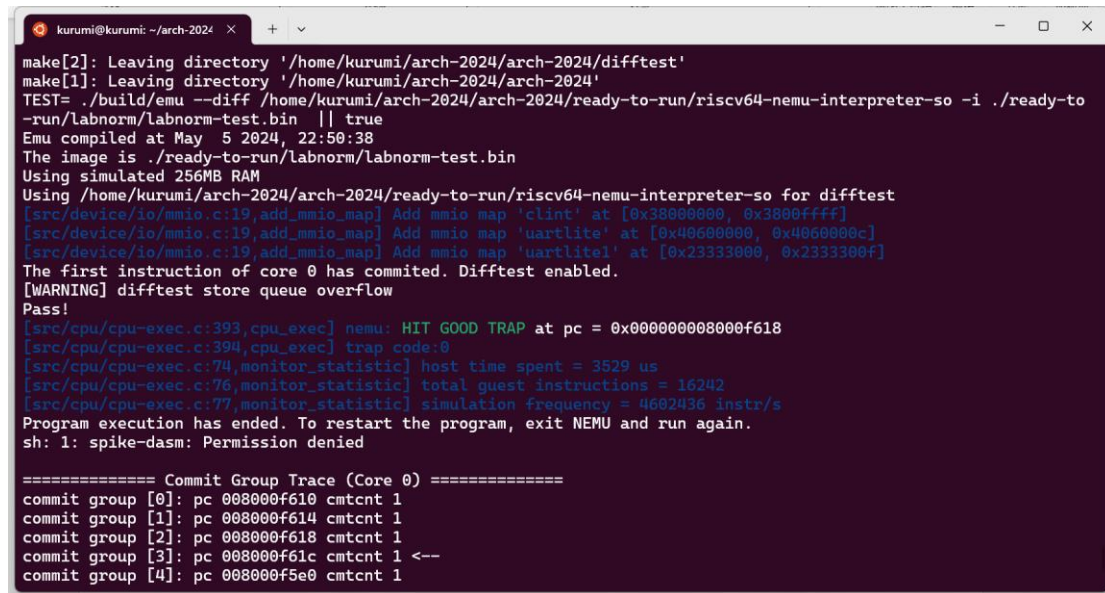
如果 load_use 型数据冒险存在，那么 PC 和 IF_ID 流水线寄存器停顿一个周期

(stall_hazard), 清空 ID_EX 流水线寄存器(id_ex_flush)。

在静态分支预测部分中, 接受 NPC 传来的跳转信号, 如果发生跳转则清空 if_id id_ex 这两个流水线寄存器。

整体五级流水线 CPU 架构及各组件解释如上

通过运行截图——



```
kurumi@kurumi: ~/arch-2024
make[2]: Leaving directory '/home/kurumi/arch-2024/arch-2024/difftest'
make[1]: Leaving directory '/home/kurumi/arch-2024/arch-2024'
TEST= ./build/emu --diff /home/kurumi/arch-2024/arch-2024/ready-to-run/riscv64-nemu-interpretor-so -i ./ready-to-run/labnorm/labnorm-test.bin || true
Emu compiled at May 5 2024, 22:50:38
The image is ./ready-to-run/labnorm/labnorm-test.bin
Using simulated 256MB RAM
Using /home/kurumi/arch-2024/arch-2024/ready-to-run/riscv64-nemu-interpretor-so for difftest
[src/device/io/mmio.c:19,add_mmio_map] Add mmio map 'clint' at [0x38000000, 0x3800ffff]
[src/device/io/mmio.c:19,add_mmio_map] Add mmio map 'uartlite' at [0x40600000, 0x4060000c]
[src/device/io/mmio.c:19,add_mmio_map] Add mmio map 'uartlite1' at [0x23333000, 0x2333300f]
The first instruction of core 0 has committed. Difftest enabled.
[WARNING] difftest store queue overflow
Pass!
[src/cpu/cpu-exec.c:393,cpu_exec] nemu: HIT GOOD TRAP at pc = 0x0000000008000f618
[src/cpu/cpu-exec.c:394,cpu_exec] trap code:0
[src/cpu/cpu-exec.c:74,monitor_statistic] host time spent = 3529 us
[src/cpu/cpu-exec.c:76,monitor_statistic] total guest instructions = 16242
[src/cpu/cpu-exec.c:77,monitor_statistic] simulation frequency = 4602436 instr/s
Program execution has ended. To restart the program, exit NEMU and run again.
sh: 1: spike-dasm: Permission denied

===== Commit Group Trace (Core 0) =====
commit group [0]: pc 008000f610 cmtcnt 1
commit group [1]: pc 008000f614 cmtcnt 1
commit group [2]: pc 008000f618 cmtcnt 1
commit group [3]: pc 008000f61c cmtcnt 1 <--
commit group [4]: pc 008000f5e0 cmtcnt 1
```