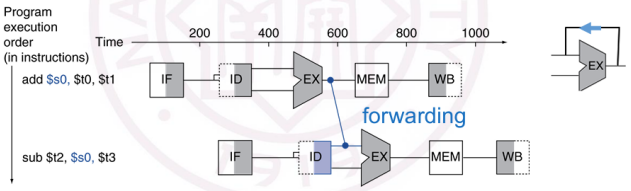
期末知识点总结梳理（chapter4）

- 如何设计一个计算机的处理器
  - PC
  - Register file
  - ALU
  - Data Memory
  - Instruction Memory
- 如何建立处理器的数据通路
- 单周期设计的关键问题
  - 1. 最长延迟决定时钟周期
  - 2. 不同指令的延迟差异很大
    - 指令类型对比：
    - ├── 简单运算：寄存器 → ALU → 寄存器 (短路径)
    - ├── 分支指令：寄存器 → ALU → PC更新 (中等路径)
    - └── load指令：内存 → 寄存器 → ALU → 内存 → 寄存器 (最长路径)
  - 解决方案：流水线
- 流水线是如何设计的
  - IF: Instruction fetch from memory
  - ID: Instruction decode & register read
  - EX: Execute operation or calculate address
  - MEM: Access memory operand
  - WB: Write result back to register
- 流水线的数据通路
- CPU的流水线中的异常
  - 结构冒险
  - 数据冒险
  - 控制冒险
- 数据冒险（可以直接加旁路和必须加一个气泡）

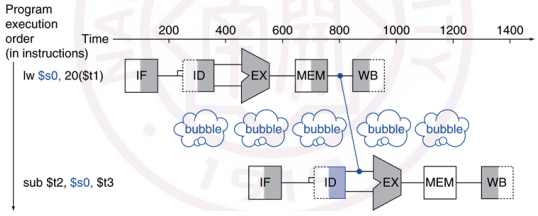
R4-06 CPU的流水线中的异常？——数据阻塞/冒险

- An instruction depends on completion of data access by a previous instruction
  - add $s0, $t0, $t1
    sub $t2, $s0, $t3



R4-06 CPU的流水线中的异常？——数据阻塞/冒险

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



- 数据冒险检测和解决的机制
  - 编译器代码重排序优化
- 如何检测到数据相关的发生
  - 核心思想：在流水线各阶段传递寄存器编号
  - 检测方法：比较当前指令需要的源寄存器与前面指令的目标寄存器
  - 例如：ID/EX.RegisterRs 与 EX/MEM.RegisterRd 进行比较

1a. EX冒险:

```
EX/MEM.RegisterRd = ID/EX.RegisterRs
EX/MEM.RegisterRd = ID/EX.RegisterRt
```

2a. MEM冒险:

```
MEM/WB.RegisterRd = ID/EX.RegisterRs
MEM/WB.RegisterRd = ID/EX.RegisterRt
```

  - EX冒险

- **EX hazard**
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

- MEM冒险

Consider the sequence:

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4

      vs.

add $1,$1,$2
add $5,$1,$3
add $1,$1,$4
```

Revise MEM hazard

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01
- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

- Load—Use又该如何检测
  - Load指令需要到MEM阶段才能从内存中获取数据
  - 即使有转发机制，也无法在EX阶段提供数据
  - 必须插入停顿（stall），无法通过转发解决

检测条件:

```
Load-use hazard when:
ID/EX.MemRead and
((ID/EX.RegisterRt = IF/ID.RegisterRs) or
 (ID/EX.RegisterRt = IF/ID.RegisterRt))
```

  - Load-Use冒险的处理方法
    - 插入一个气泡
    - 停止PC的前进，使其保持在ID阶段
- 控制冒险
  - 更长的流水线无法及早确定分支结果
  - 停顿代价变得不可接受
- 解决控制冒险：
  - 预测
  - 静态分支预测（Static branch prediction）
    - 基于典型分支行为：
    - 例子：循环和if语句分支
    - 策略：
      - 向后分支：预测为跳转（taken）
      - 向前分支：预测为不跳转（not taken）
  - 2. 动态分支预测（Dynamic branch prediction）

        - 分支预测缓冲区（Branch prediction buffer）：
        - 也称为分支历史表（branch history table）
        - 1位存储预测信息
        - 索引方式：使用最近分支指令的地址
        - 存储内容：分支结果（taken/not taken）
        - 执行流程：
            - 执行分支：检查预测表
        - 开始取指：根据预测获取指令
        - 如果预测错误：冲刷流水线并翻转预测
- 双重循环问题
    - 1位预测器的问题：
        - 内循环分支会被预测错误两次！
    - 具体分析：
        - 内循环最后一次迭代：
            - 之前一直预测为"taken"（继续内循环）
            - 最后一次应该"not taken"（退出内循环）
            - 第一次预测错误：预测为taken，实际为not taken
        - 下次进入内循环时：
            - 由于上次预测错误，预测器状态变为"not taken"
            - 但第一次迭代应该是"taken"（继续内循环）
            - 第二次预测错误：预测为not taken，实际为taken
    - 2位分支预测器解决方案
        - 四个状态：
            - 强taken（Predict taken）
            - 弱taken（Predict taken）
            - 弱not taken（Predict not taken）
            - 强not taken（Predict not taken）
        - 状态转换规则：
            - 连续两次预测错误才会改变预测结果
            - 单次预测错误只改变"强度"，不改变预测方向
- 数据和控制相关同时发生
    - R型遇上Beq的情况
        - 当R型指令的目标寄存器与紧接着的beq指令的源寄存器相同时：
        - 问题：beq需要在ID阶段读取寄存器值进行比较，但前面的R型指令要到WB阶段才写回结果

        - 结果：无法通过转发解决，因为数据还没准备好
    - 2. Load遇上Beq的情况
        - 当load指令的目标寄存器是紧接着beq指令的比较寄存器时：
        - 问题：load指令在MEM阶段才能获得数据，而beq在ID阶段就需要进行比较
        - 解决方案：需要2个停顿周期（stall cycles）
        - 第一个beq被标记为"stalled"，等待数据准备
    - 3. Load+R型遇上Beq的情况
        - 当比较寄存器是前面第二条ALU指令或第二条load指令的目标时：
        - 优势：只需要1个停顿周期
        - 原因：有更多时间让数据在流水线中传播
- CPU的流水线中的异常和中断区别
    - xception（异常）
        - 来源：CPU内部产生
        - 示例：
            - 未定义的操作码（undefined opcode）
            - 数值溢出（overflow）
            - 系统调用（syscall）
            - 等等
    - Interrupt（中断）
        - 来源：外部I/O控制器产生
        - 示例：
            - 键盘输入
            - 网络数据到达
            - 定时器超时
            - 硬盘读写完成
- MIPS异常和中断如何处理
    - 关键寄存器：Exception Program Counter (EPC)
        - 作用：保存发生异常的指令地址（或被中断的指令地址）
        - 目的：异常处理完成后能够返回到正确位置继续执行
- 流水线的优化
    - 更长流水线
        - 从测试来看，Cypress Cove 的等效流水线深度大约是14-22 级
    - 超长指令字/静态多发射处理器
        - 超长指令字（VLIW）（静态多发射处理器）是由美国Yale大学教授Fisher提出的，是一条指令来实现多个操作的并行执行，之所以放到一条指令是为了减少内存访问。

- 将多条指令组合成"issue packets"（一个时钟周期内发射的指令包）
- 将这些包装入"issue slots"（发射槽）
- 编译时检测并避免冒险（hazards）

- 动态多发射处理器（超标量处理器）
  - CPU内部有多条流水线，可以并行处理多条指令
  - 在单个时钟周期内，指令可以乱序执行，但按序退休（retire）
  - 目标：让CPU的IPC (Instructions Per Clock) > 1
  - 特性　　VLIW (静态)　　超标量 (动态)
  - 调度责任　编译器　　CPU硬件
  - 冒险检测　编译时　　运行时
  - 指令选择　预先确定　　动态选择
- 数据优化的角度
  - 向量计算机
  - 软件优化（超线程）
-