# CA Basic pipeline

## 1 基本pipeline

review

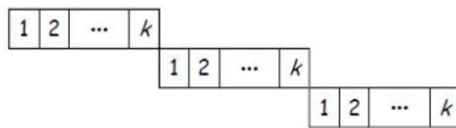
- **ISA的功能设计：任务为确定硬件支持哪些操作。方法是统计的方法。存在CISC和RISC两种设计理念**
  - CISC（Complex Instruction Set Computer）
    - 目标：强化指令功能，减少指令的指令条数，以提高系统性能
    - 基本方法：面向目标程序的优化，面向高级语言和编译器的优化
  - RISC（Reduced Instruction Set Computer）
    - 目标：通过简化指令系统，用最高效的方法实现最常用的指令
    - 主要手段：充分发挥流水线的效率，降低（优化）CPI
- **典型ISA的特色**
  - 在当时比较有特色的设计
    - 如MIPS的不对齐的存储器访问，分支指令后的延迟槽。
    - SPARC的寄存器窗口重叠技术；
    - PA-RISC 的Nullification指令；
    - Alpha仅允许按字访问存储器等
  - 基于目前硬件技术现状，存在一些已经过时的决定



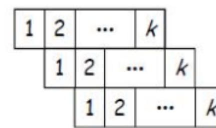
**采用流水线技术实现ISA的思想， 早于RISC**

# 流水线的基本概念

- **一个任务可以分解为$k$个子任务**
  - K个子任务在 K 个不同阶段（使用不同的资源）运行
  - 每个子任务执行需要1个单位时间
  - 整个任务的执行时间为 K倍单位时间
- **流水线执行模式是重叠执行模式**
  - K个流水段并行执行K个不同任务
  - 每个单位时间进入/离开流水线一个任务




**Serial Execution**
One completion every $k$ time units




**Pipelined Execution**
One completion every 1 time unit

# 流水线的性能

- **设 $\tau_i$ = time delay in stage $S_i$, $i=1..k$**
- **时钟周期$\tau$ = max($\tau_i$) 为最长的流水段延迟**
- **时钟频率 f = 1/$\tau$ = 1/max($\tau_i$)**
- **流水线可以在k+n-1个时钟周期内完成n个任务**
  - 完成第一个任务需要 k个时钟周期 ← 进入流水
  - 其他n-1个任务需要n-1个时钟周期完成
- **K-段流水线的理想加速比（相对于串行执行）**

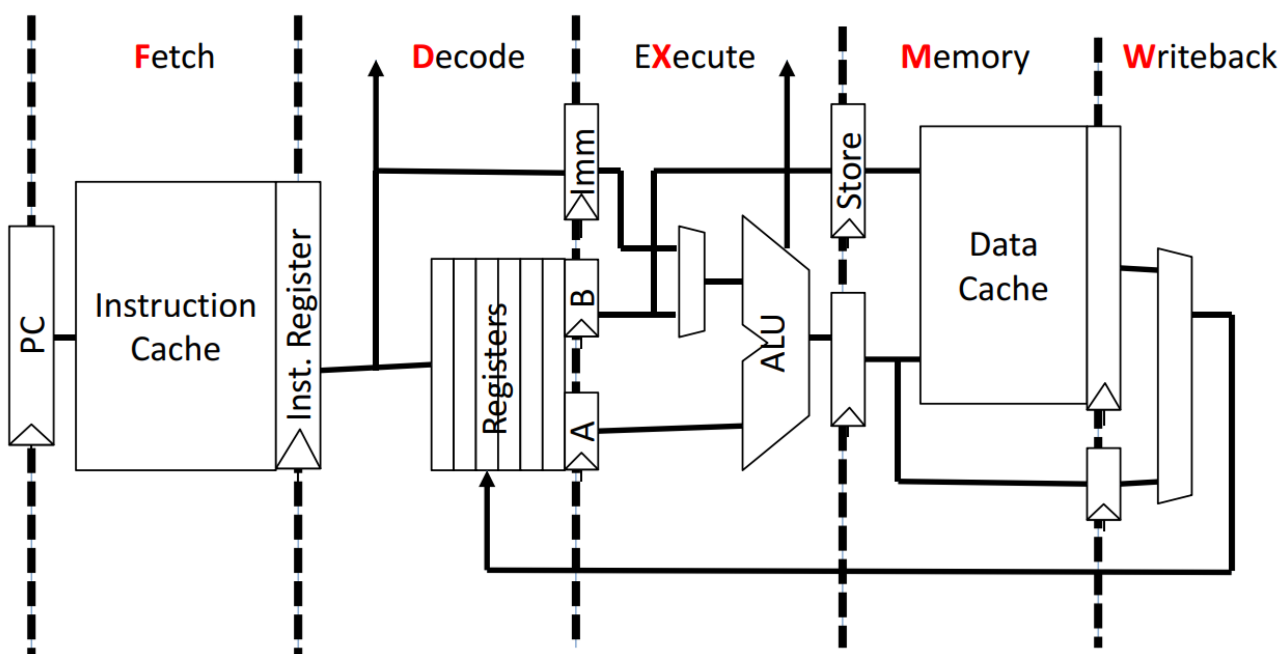
理想 ①无相关 ②pipeline中有reg file，有时间开销

$$S_k = \frac{\text{Serial execution in cycles}}{\text{Pipelined execution in cycles}} = \frac{nk}{k+n-1} \qquad S_k \rightarrow k \text{ for large } n$$

# 典型的RISC　5段指令流水线

- **5个流水段，每段的延迟为1个cycle**
- **IF: 取值阶段**
  - 选择地址：下一条指令地址、转移地址
- **ID: 译码阶段**
  - 确定控制信号 并从寄存器文件中读取寄存器值
- **EX: 执行**
  - Load 、Store：计算有效地址
  - Branch：计算转移地址并确定转移方向
- **MEM: 存储器访问（仅Load和Store)**
- **WB: 结果写回**

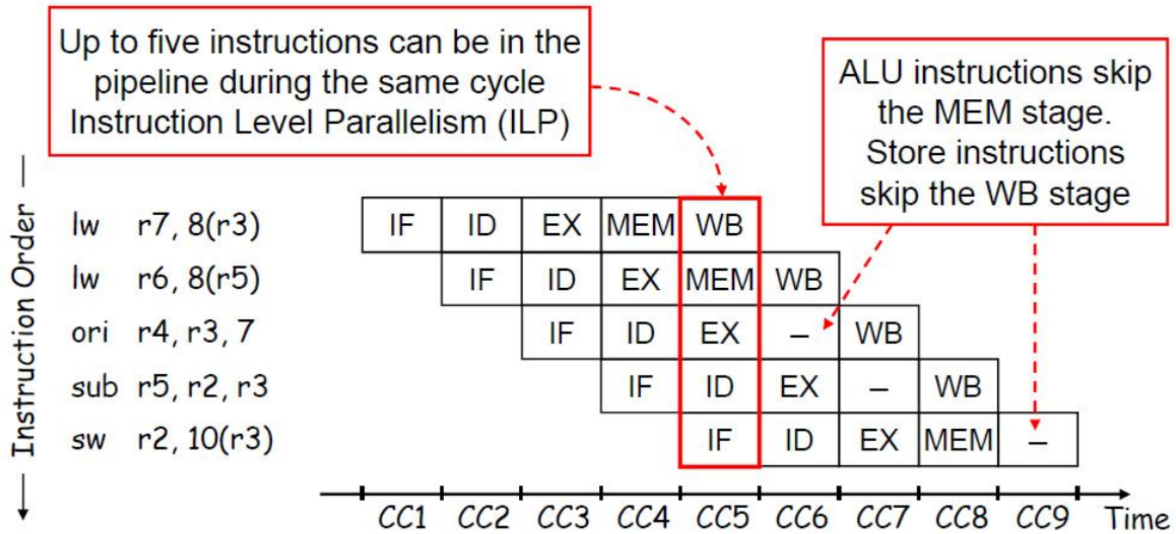
# 典型的 RISC 5段流水线




*This version designed for regfiles/memories with synchronous reads and writes.*

# 指令流时序

- **时序图展示：**
  – 每个时钟周期指令所使用的流水段情况
- **指令流在采用5段流水线执行模式的执行情况**



# 单周期、多周期、流水线控制**性能比较**

- **假设5段指令执行流水线**

| Instruction | Fetch | Reg Read | ALU | Memory | Reg Wr | Time |
|---|---|---|---|---|---|---|
| Load | 350 ps | 250 ps | 350 ps | 350 ps | 250 ps | 1550 ps |
| Store | 350 ps | 250 ps | 350 ps | 350 ps | | 1300 ps |
| ALU | 350 ps | 250 ps | 350 ps | | 250 ps | 1200 ps |
| Branch | 350 ps | 250 ps | 350 ps | | | 950 ps |

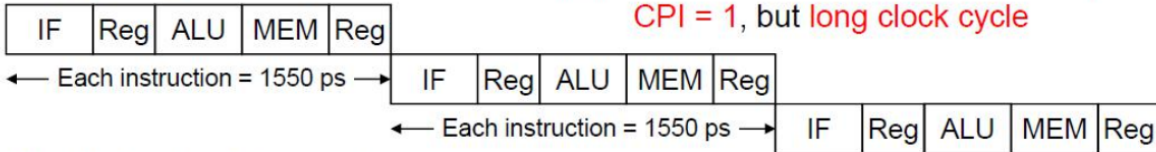
- **某一程序段假设：**
  – 20% load, 10% store, 40% ALU, and 30% branch
- **比较三种执行模式的性能**

# 单周期、多周期、流水线控制性能比较

## Single-Cycle Execution:

$T_{clock}$ = 350+250+350+350+250 = 1550 ps

CPI = 1, but long clock cycle

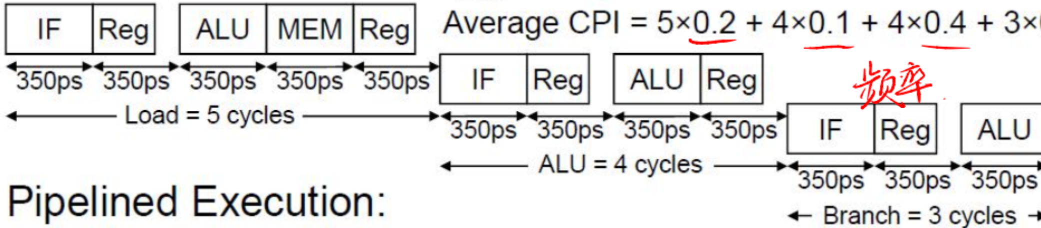



## Multi-Cycle Execution:

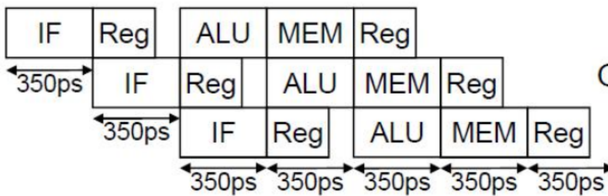
$T_{clock}$ = 350 ps

Average CPI = 5×0.2 + 4×0.1 + 4×0.4 + 3×0.3 = 3.9

频率




## Pipelined Execution:




$T_{clock}$ = 350 ps = max(350, 250)

One instruction completes each cycle

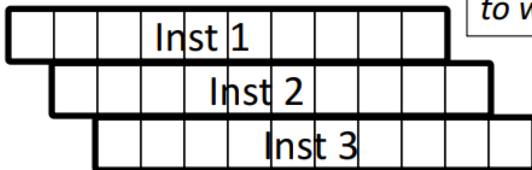
Average CPI = 1

Ignore time to fill pipeline

# Pipeline CPI Examples

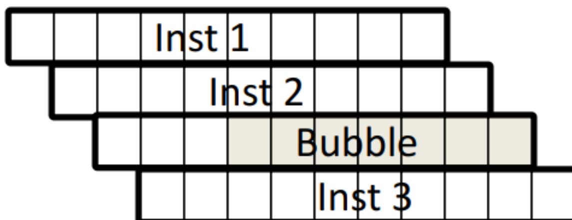
Time →

*Measure from when first instruction finishes to when last instruction in sequence finishes.*



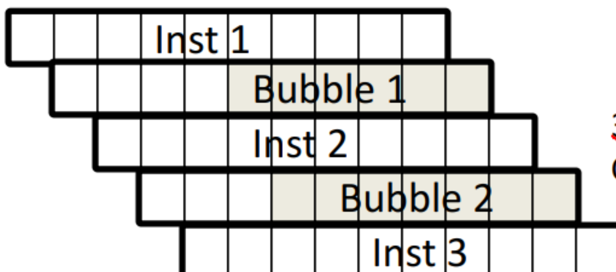

3 instructions finish in 3 cycles

CPI = 3/3 =1




3 instructions finish in 4 cycles

CPI = 4/3 = 1.33




3 instructions finish in 5cycles

CPI = 5/3 = 1.67

# 三种基本的数据相关

- 写后读相关(Read After Write (RAW))
  - 由于实际的数据交换需求而引起的

```
I: add x1,x2,x3
J: sub x4,x1,x3
```

真相关

- 读后写相关（Write After Read (WAR)

```
I: sub x4,x1,x3
J: add x1,x2,x3
```

  - 编译器编写者称之为“anti-dependence”（反相关），是由于重复使用寄存器名“x1”引起的.

假相关

- 写后写相关（Write After Write (WAW))

```
I: sub x1,x4,x3
J: add x1,x2,x3
```

  - 编译器编写者称之为“output dependence”，也是由于重复使用寄存器名“x1”引起的.
  - 在后面的复杂的流水线中我们将会看到 WAR 和WAW 相关

基本流水线中不会出现

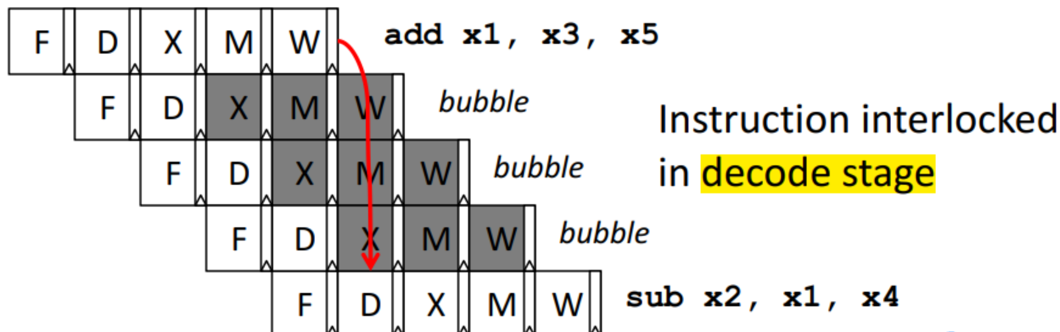
# 消减数据相关的三种策略

- 联锁机制（Interlock) 等
  - 在issue阶段保持当前相关指令，等待相关解除
- 设置旁路定向路径（Bypass or Forwarding)
  - 只要结果可用，通过旁路尽快传递数据
- 投机（Speculate)
  - 猜测一个值继续，如果猜测错了再更正

设置表

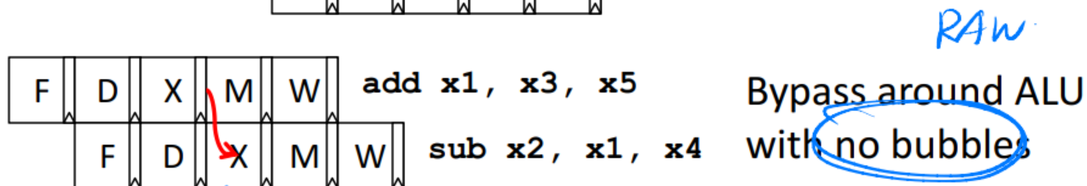
# Interlocking Versus Bypassing

```
add x1, x3, x5
sub x2, x1, x4
```




Instruction interlocked in decode stage

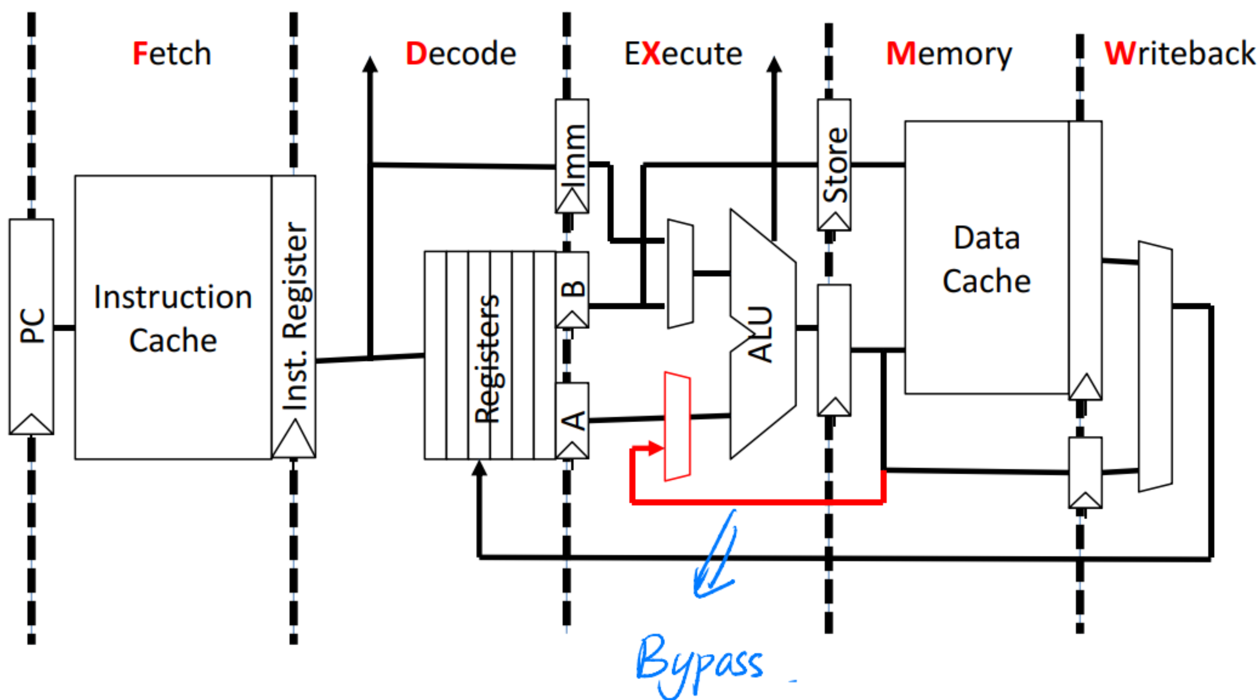
RAW



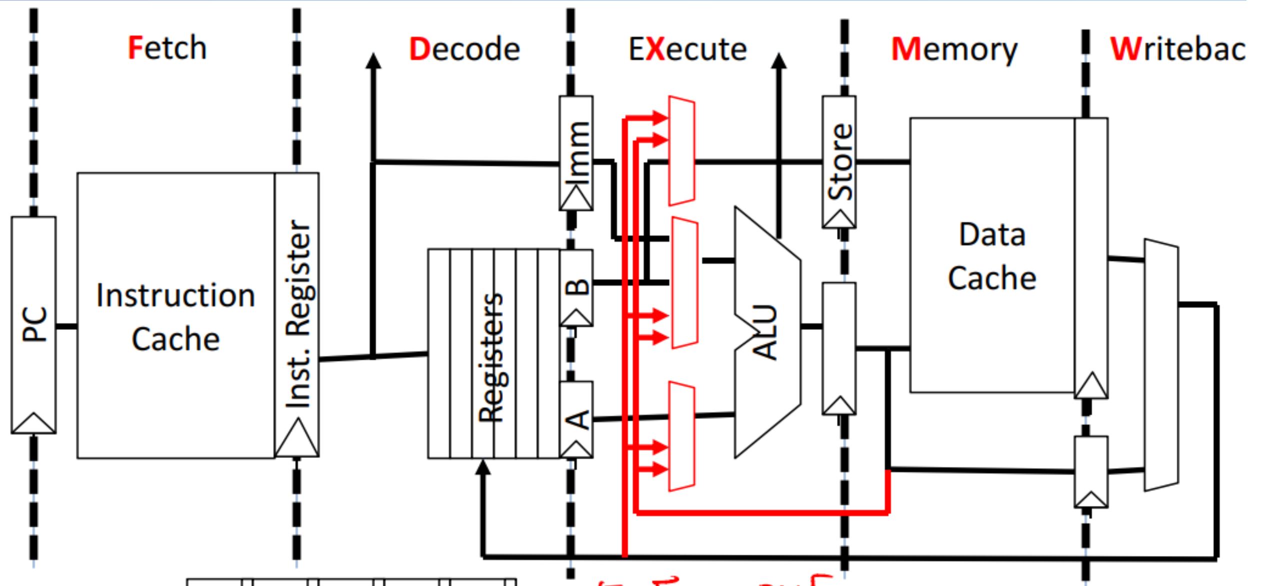

Bypass around ALU with no bubbles

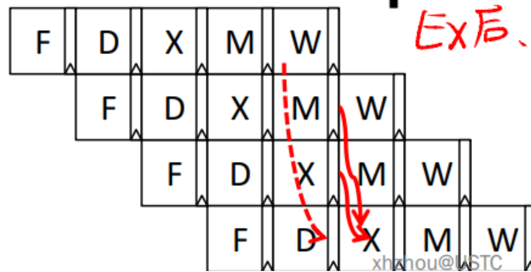
数据旁路：x1 计算出直接写回 ALU 前 Reg

# Example Bypass Path



# Fully Bypassed Data Path




EX后、MEM后

*[ Assumes data written to registers in a W cycle is readable in parallel D cycle (dotted line). Extra write data register and bypass paths required if this is not possible. ]*

# 采用软件方法避免数据相关

**Try producing fast code for**

a = b + c;

d = e – f;

让编译器去做：指令调整

静态调度

**assuming a, b, c, d ,e, and f in memory.**

**Slow code:**

```
LW  Rb,b
LW  Rc,c
ADD     Ra,Rb,Rc
SW      a,Ra
LW  Re,e
LW  Rf,f
SUB     Rd,Re,Rf
SW d,Rd
```

Fast code:

```
LW      Rb,b
LW      Rc,c
LW      Re,e
ADD     Ra,Rb,Rc
LW      Rf,f
SW      a,Ra
SUB     Rd,Re,Rf
SW      d,Rd
```

假设：

1、使用Load的结果中间需要至少1条指令的间隔

2、运算的结果，写入内存至少需要1条指令的间隔



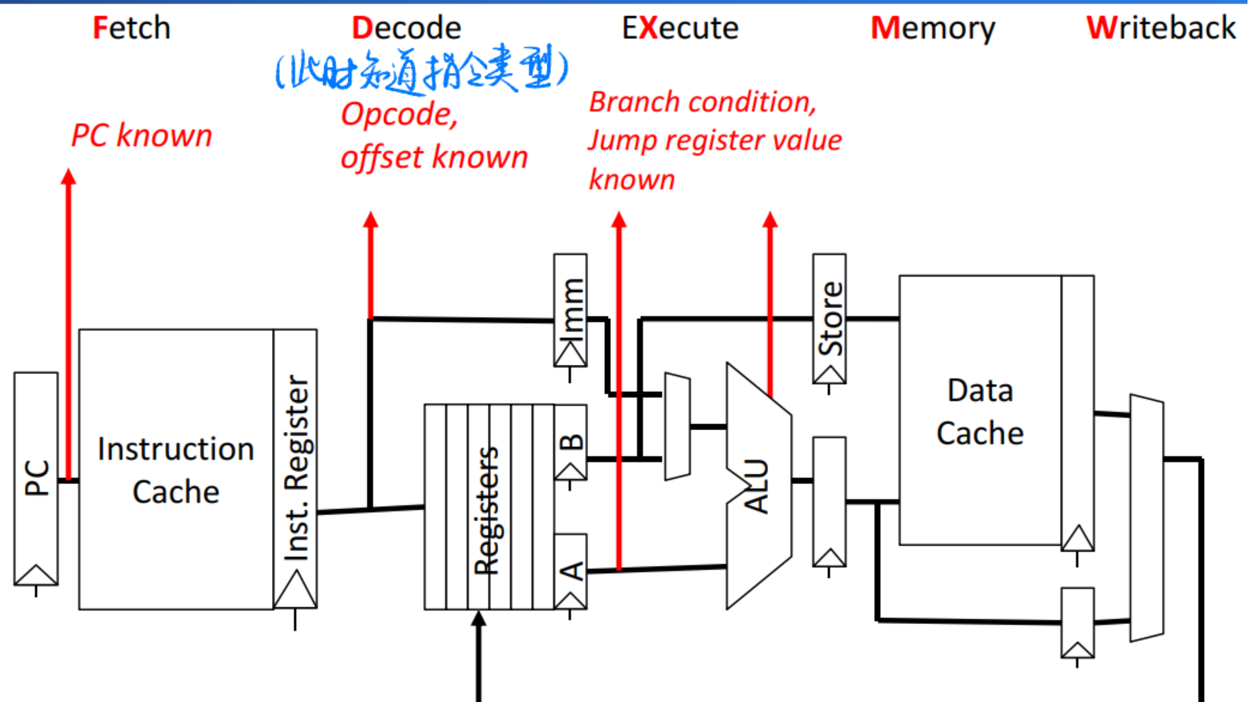
## 1.2 控制相关

# Control Hazards

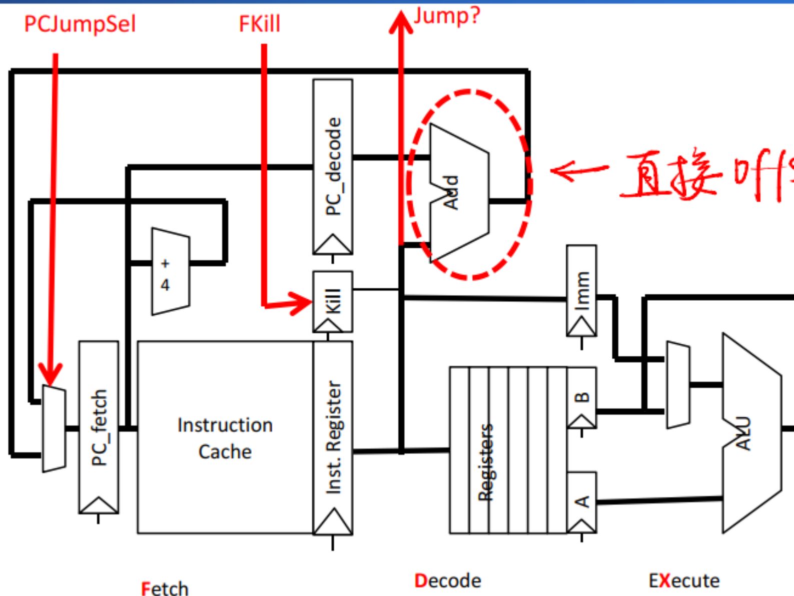
## 如何计算下一条指令地址（next PC)

- **无条件直接转移**
  - Opcode, PC, and offset
- **基于基址寄存器的无条件转移**
  - Opcode, Register value, and offset
- **条件转移**
  - Opcode, Register (for condition), PC and offset
- **其他指令**
  - Opcode and PC ( and have to know it's not one of above )

# Control flow information in pipeline

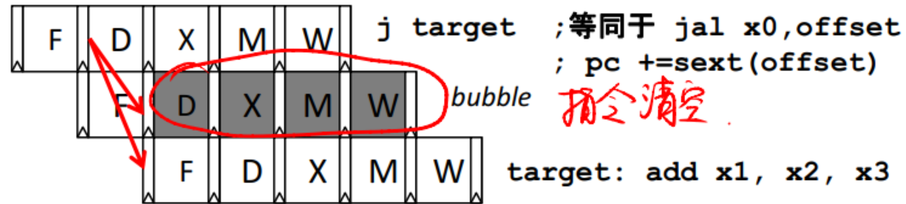


# RISC-V Unconditional PC-Relative Jumps



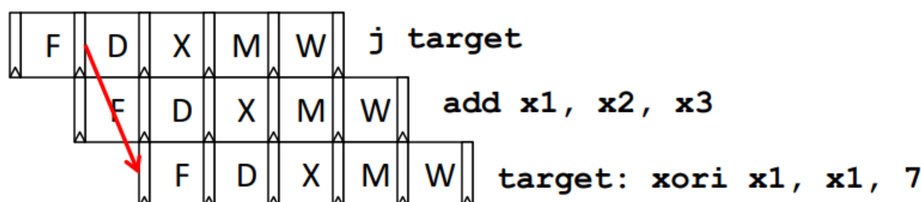

**改进的数据通路**

*[ Kill bit turns instruction into a bubble ]*

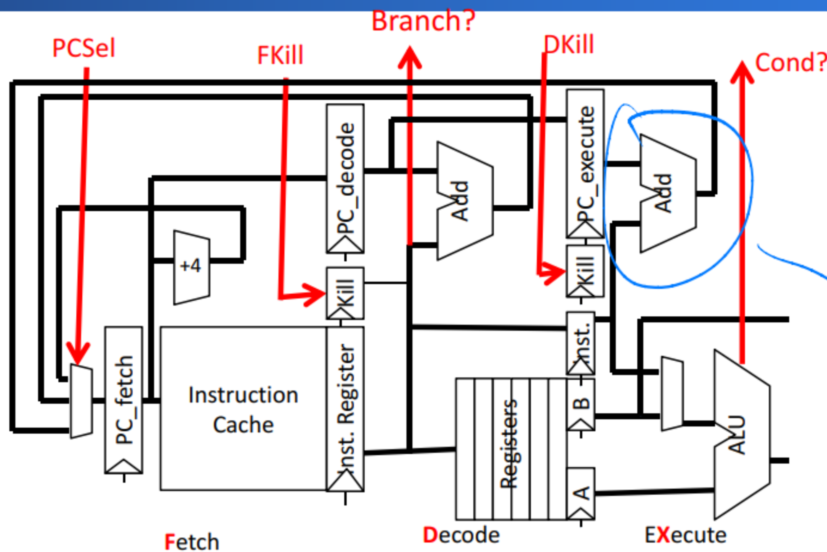


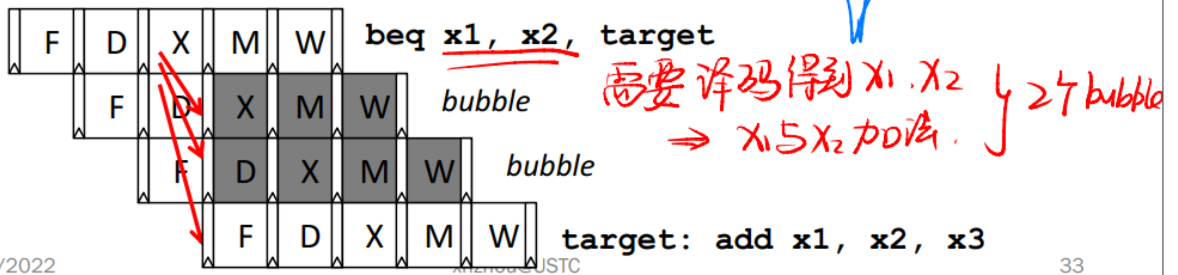
# Branch Delay Slots

- 早期的RISC机器的延迟槽技术—改变ISA语义，在分支/跳转后的**延迟槽**中指令总是在控制流发生变化之前执行：
  - 0x100 j target
  - 0x104 add x1, x2, x3 // Executed before target
  - ...
  - 0x205 target: xori x1, x1, 7
- 软件必须用有用的工作填充延迟槽（delay slots），或者用显式的NOP指令填充延迟槽



# RISC-V Conditional Branches



# Pipelining for Jump Register

- **Register value obtained in execute stage**

# 小结：解决控制相关的方法

- **#1: Stall 直到分支方向确定**
  - **可通过修改数据通路，减少stall**
- **#2: 预测分支失败**
  - 直接执行后继指令
  - 如果分支实际情况为分支成功，则撤销流水线中的指令对流水线状态的更新
  - 要保证：分支结果出来之前不会改变处理机的状态，以便一旦猜错时，处理机能够回退到原先的状态。
- **#3: 预测分支成功**
  - 前提：先知道分支目标地址，后知道分支是否成功
- **#4: 延迟转移技术**

# 为什么在经典的五段流水线中指令不能在每个周期都被分发(CPI>1)

- **采用全定向路径可能代价太高而无法实现**
  - 通常提供经常使用的定向路径
  - 一些不经常使用的定向路径可能会增加时钟周期的长度，从而抵消降低CPI的好处
- **Load操作有两个时钟周期的延迟**
  - Load指令后的指令不能马上使用Load的结果
  - MIPS-I ISA 定义了延迟槽，软件可见的流水线冲突 (由编译器调度无关的指令或插入NOP指令避免冲突），MIPS-II中取消了延迟槽语义（硬件上增加流水线interlocks机制)
    - MIPS:"Microprocessor without Interlocked Pipeline Stages"
- **Jumps/Conditional branches 可能会导致流水线断流 (bubbles)**
  - 如果没有延迟槽，则stall后续的指令

**带有软件可见的延迟槽有可能需要执行大量的由编译器插入的NOP指令**
**NOP指令降低了CPI，增加了程序中执行的指令条数**

# 流水线的性能分析

- **基本度量参数**：①吞吐率，②加速比，③效率
- **吞吐率**：在单位时间内流水线所完成的任务数量或输出结果的数量。

$$TP = \frac{n}{T_K}$$

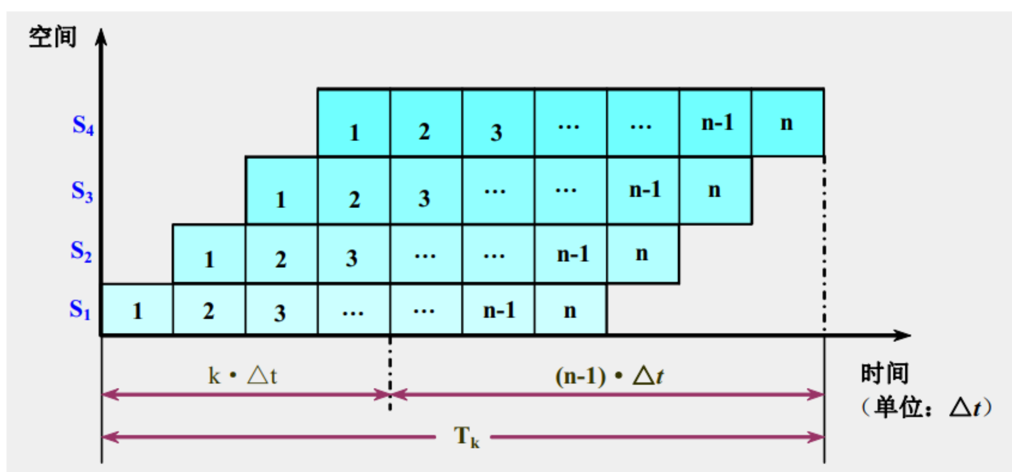
$n$：任务数

$T_k$：处理完成$n$个任务所用的时间

# 流水线技术提高系统的任务吞吐率

## 1. 各段时间均相等的流水线

– 各段时间均相等的流水线时空图



**理想情况（k段流水，n个任务）**

$$T = (k + n - 1)\Delta t_0 \tag{1}$$

## 吞吐率

- 流水线完成n个连续任务所需要的总时间（假设一条k段线性流水线）
  Tk = kΔt + (n-1)Δt = (k + n-1)Δt
- 流水线的实际吞吐率 $TP = \dfrac{n}{(k+n-1)\Delta t}$
- 最大吞吐率: 流水线在连续流动达到稳定状态后所得到的吞吐率。 $TP_{max} = \dfrac{1}{\Delta t}$

$$TP_{\max} = \lim_{n \to \infty} \frac{n}{(k+n-1)\Delta t} = \frac{1}{\Delta t} \qquad TP = \frac{n}{k+n-1} TP_{\max}$$

| S4 | | | | 1 | 2 | 3 | 4 | 5 | .. | .. | .. | n-1 | n |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S3 | | | 1 | 2 | 3 | 4 | 5 | .. | .. | .. | n-1 | n | |
| S2 | | 1 | 2 | 3 | 4 | 5 | .. | .. | .. | n-1 | n | | |
| S1 | 1 | 2 | 3 | 4 | 5 | .. | .. | .. | n-1 | n | | | |

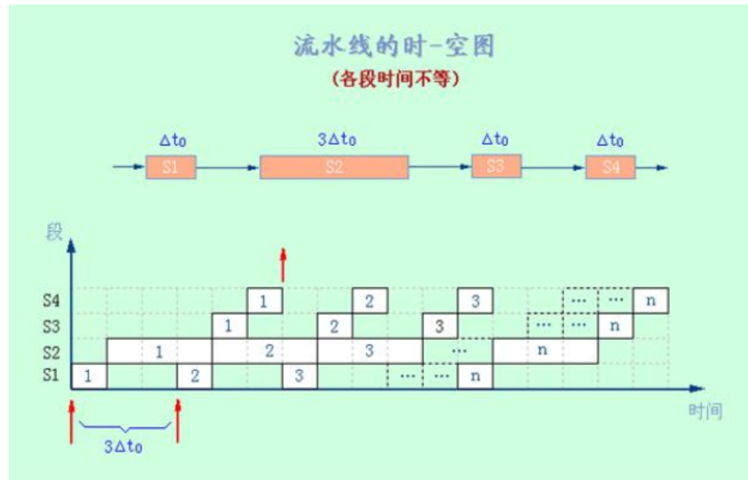


**流水线的实际吞吐率小于最大吞吐率**

**只有当n>>k时， 才有TP≈TPmax**

# 流水线中的瓶颈——最慢的段

## 2. 各段时间不完全相等的流水线




- 各段时间不等的流水线及其时空图
  - 一条4段的流水线
  - S1，S3，S4各段的时间：Δt
  - S2的时间：3Δt （瓶颈段）
- 流水线中这种时间最长的段称为流水线的瓶颈段

- 各段时间不等的流水线的实际吞吐率：
  （Δti为第i段的时间，共有k个段）

$$TP = \frac{n}{\sum_{i=1}^{k} \Delta t_i + (n-1)\max(\Delta t_1, \Delta t_2, \cdots, \Delta t_k)}$$

- 流水线的最大吞吐率为

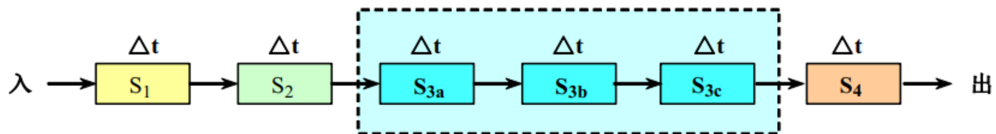
$$TP_{\max} = \frac{1}{\max(\Delta t_1, \Delta t_2, \cdots \Delta t_k)}$$

# 3. 解决流水线瓶颈问题的常用方法

– 细分瓶颈段

例如：对前面的4段流水线

把瓶颈段$S_3$细分为3个子流水线段：$S_{3a}$，$S_{3b}$，$S_{3c}$



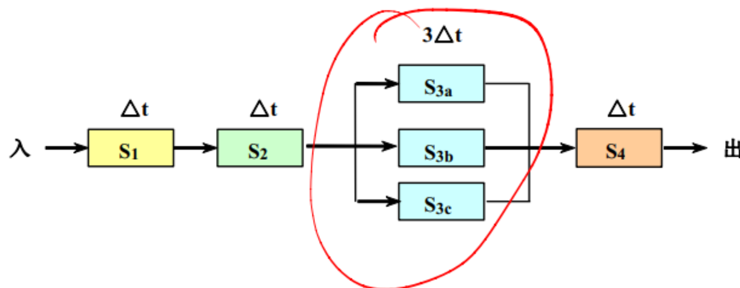

**改进后的流水线的吞吐率：**

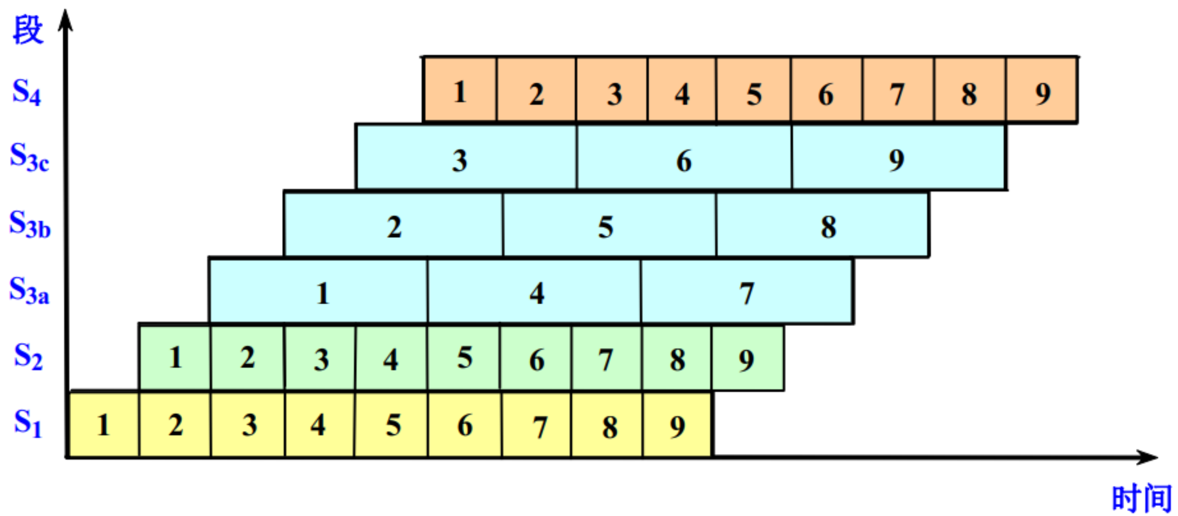
$$TP_{max} = \frac{1}{\Delta t}$$



## – 重复设置瓶颈段

- 缺点：控制逻辑比较复杂，所需的硬件增加了。
- 例如：对前面的4段流水线
- 重复设置瓶颈段S3：S3a，S3b，S3c






重复设置瓶颈段后的时空图

## 加速比

- **加速比：完成同样一批任务，不使用流水线所用的时间与使用流水线所用的时间之比。**
  - 假设：不使用流水线（即顺序执行）所用的间为Ts，使用流水线后所用的时间为Tk，则该流水线的加速比为

$$S = \frac{T_s}{T_k}$$

## 1. 流水线各段时间相等（都是Δt）

– k段流水线完成n个连续任务所需要的时间为

$$T_k = (k + n-1)\Delta t$$

– 顺序执行n个任务所需要的时间

$$T_s = nk\Delta t$$

– 流水线的实际加速比为

$$S = \frac{nk}{k+n-1}$$

– 最大加速比 $$S_{\max} = \lim_{n\to\infty} \frac{nk}{k+n-1} = k$$

– **当*n*>>*k*时，*S* ≈ *k*** → 理想

**思考：流水线的段数愈多愈好?**



## 2. 流水线的各段时间不完全相等时

– 一条*k*段流水线完成*n*个连续任务的实际加速比为

$$S = \frac{n\sum_{i=1}^{k}\Delta t_i}{\sum_{i=1}^{k}\Delta t_i + (n-1)\max(\Delta t_1, \Delta t_2, \cdots, \Delta t_k)}$$

## 加深流水线有什么影响?

### 正面影响

流水线的级数越多，意味着流水线被切得越细，每一级流水线内容纳硬件逻辑越少，意味着**运行更高的主频。主频越高，流水线的吞吐率越高**。

### 负面影响:

（1）每一级流水线都由寄存器组成，更多的流水线级数意味着**消耗更多的寄存器，产生更大的面积开销，功耗也会增大**。

（2）每一级流水线需要握手，流水线的最后一级反压信号可能会一直串扰到最前一级造成严重的**时序问题**。

（3）流水线的取指阶段得知条件跳转的结果是到底跳还是不跳，因此只能进行预测，而到了流水线的末端才能通过实际运算得知该分支是真的该跳还是不该跳。如果发现真实的结果与预期的结果不一致，意味着**预测失败**，需要将所有的预取指令全部丢失。流水线越深，则**意味着浪费和损失越严重**；流水线越浅，则浪费和损失越少。

（4）【面试官告知我的】流水级数越深，指令预测也越复杂！

这些负面影响有可能会抵消增加流水线级数带来的正面影响。

## 效率

- **效率：流水线中的设备实际使用时间与整个运行时间的比值，即流水线设备的利用率。**
  - 由于流水线有通过时间和排空时间，所以在连续完成n个任务的时间内，各段并不是满负荷地工作。
- **各段时间相等**
  - 各段的效率ei相同

$$e_1 = e_2 = \cdots = e_k = \frac{n\Delta t}{T_k} = \frac{n}{k+n-1}$$

– 整条流水线的效率为

$$E=\frac{e_1+e_2+\cdots+e_k}{k}=\frac{ke_1}{k}=\frac{kn\Delta t}{kT_k}$$

可以写成

$$E=\frac{n}{k+n-1}$$

最高效率为

$$E_{\max}=\lim_{n\to\infty}\frac{n}{k+n-1}=1$$

**当$n>>k$时，$E\approx1$。**

– 当流水线各段时间相等时，流水线的效率与吞吐率成正比。

E=TP△t

$$E=\frac{n}{k+n-1}\qquad TP=\frac{n}{(k+n-1)\Delta t}$$

- **流水线的效率是流水线的实际加速比S与它的最大加速比k的比值。**

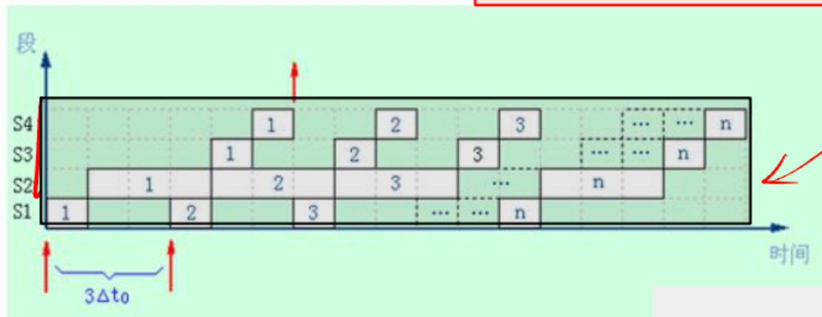
$$S=\frac{nk}{k+n-1}\qquad E=\frac{S}{k}$$

**当$E=1$时，$S=k$，实际加速比达到最大。**

- **从时空图上看，效率就是n个任务占用的时空面积和k个段总的时空面积之比。**

$$E = \frac{n\text{个任务实际占用的时空区}}{k\text{个段总的时空区}}$$ ←白块 ←矩形

当各段时间不相等时




$$E = \frac{n \cdot \sum_{i=1}^{k} \Delta t_i}{k\left[\sum_{i=1}^{k} \Delta t_i + (n-1)\cdot \max(\Delta t_1, \Delta t_2, \cdots, \Delta t_k)\right]}$$



$$TP_{\max} = \frac{1}{\max\{\Delta t_i\}}$$

$$TP = \frac{n}{\sum_{i=1}^{k} \Delta t_i + (n-1)\Delta t_j}$$

$$S = \frac{n \cdot \sum_{i=1}^{k} \Delta t_i}{\sum_{i=1}^{k} \Delta t_i + (n-1)\Delta t_j}$$

$$E = \frac{n \cdot \sum_{i=1}^{k} \Delta t_j}{k \cdot [\sum_{i=1}^{k} \Delta t_i + (n-1)\Delta t_j]}$$

$$E = TP \cdot \frac{\sum_{i=1}^{k} \Delta t_i}{k}$$

## 例如： Dual-port vs. Single-port

- 机器A: Dual ported memory (“Harvard Architecture”)
- 机器B: Single ported memory
- 存在结构相关的机器B的时钟频率是机器A的时钟频率的1.05倍
- Ideal CPI = 1
- 在机器B中load指令会引起结构相关，所执行的指令中Loads指令占 40%

**Average instruction time = CPI * Clock cycle time**

**无结构相关的机器A:**

Average Instruction time = Clock cycle time

**存在结构相关的机器B:**

Average Instruction time = (1+0.4*1) * clock cycle time /1.05
= 1.3 * clock cycle time



$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

| *Scheduling scheme* | *Branch penalty* | *CPI* | *speedup v. unpipelined* | *speedup v. stall* |
|---|---|---|---|---|
| Stall pipeline | 3 | 1.42 | 3.5 | 1.0 |
| Predict taken | 1 | 1.14 | 4.4 | 1.26 |
| Predict not taken | 1 | 1.09 | 4.5 | 1.29 |
| Delayed branch | 0.5 | 1.07 | 4.6 | 1.31 |

1.14 = 1 + 1*14%*100% ← predict taken
1.09 = 1+1*14%*65% ← pedict untaken
1.07 = 1+ 0.5*14%

Conditional & Unconditional = 14%, 65% change PC

猜猜

## 2 基本流水线扩展

### 2.1 异常和中断

这个在微机原理讲的够多了，入栈出栈，保留断点现场，关中断……

### 2.2 多周期操作

**Latency & Repeat Interval**

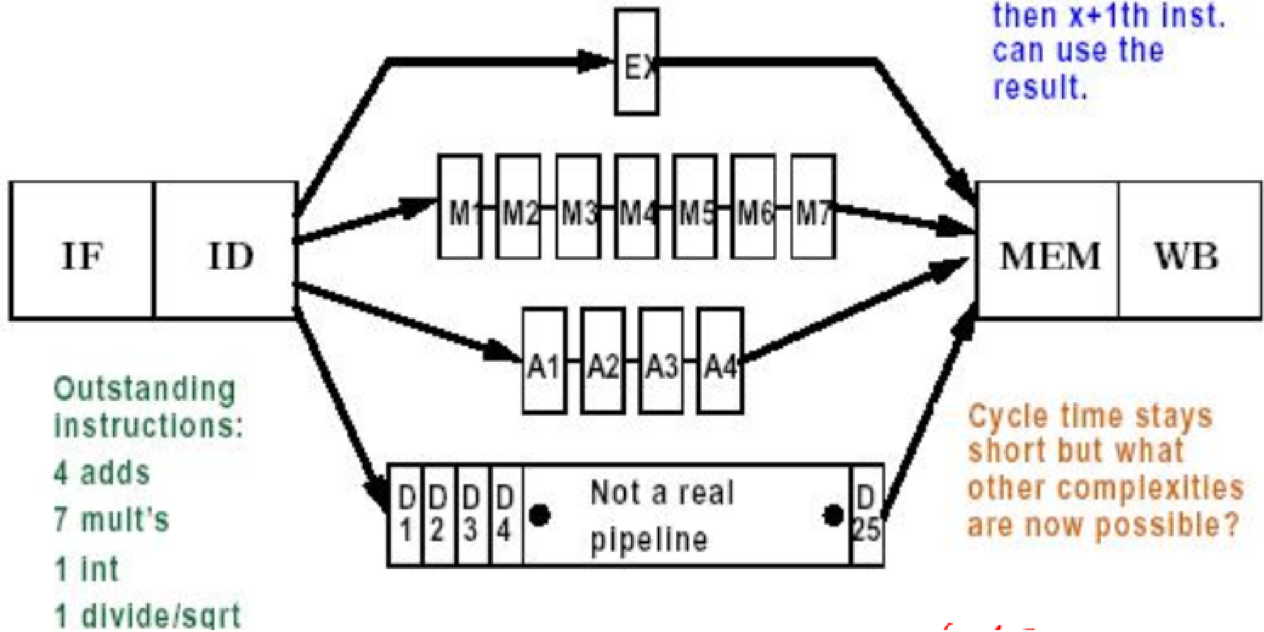
- **延时(Latency)**
  - 定义1：完成某一操作所需的cycle数
  - **定义2: 使用当前指令所产生结果的指令与当前指令间的最小间隔周期数**
- **循环间隔（Repeat/Initiation interval)**
  - 发射相同类型的操作所需的间隔周期数
- **EX部件流水化的新的MIPS**

| Function Unit | Latency | Repeat Interval |
| --- | --- | --- |
| Integer ALU | 0 | 1 |
| Data Memory (Integer and FP loads(1 less for store latency)) | 1 | 1 |
| FP Add | 3 | 1 |
| FP multiply | 6 | 1 |
| FP Divide (also integer divide and FP sqrt) | 24 | 25 |

3/22/2022 xhzhou@USTC 25

**Add需要4个周期EX，Multi需要7个周期EX**

**故Latency=3，6**



# 新的结构相关

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MULTD F0, F4, F6 | IF | ID | M1 | M2 | M3 | M4 | M5 | M6 | M7 | MEM | WB |
| ... | | IF | ID | EX | MEM | WB | | | | | |
| ... | | | IF | ID | EX | MEM | WB | | | | |
| ADDD F2, F4, F6 | | | | IF | ID | A1 | A2 | A3 | A4 | MEM | WB |
| ... | | | | | IF | ID | EX | MEM | WB | | |
| ... | | | | | | IF | ID | EX | MEM | WB | |
| LD F8, 0(R2) | | | | | | | IF | ID | EX | MEM | WB |

- **纵向检查指令所使用的资源**
  - 第10个cycle，三条指令同时进入MEM，但由于MULTD和ADDD在MEM段没有实际动作，这种情况没有关系
  - 第11个cycle，三条指令同时进入WB段，存在结构相关



EX段不同周期数

– 在一个周期内可能有多个寄存器写操作

– 可能指令 （乱序到达WB段） 有可能存在WAW

- 由于在ID段读， 还不会有 WAR 相关
- 乱序完成导致异常处理复杂

- 由于指令的延迟加大导致RAW相关的stall数增多
- 需要付出更多的代价来增加定向路径

# 解决方法

- **Option 1**
  - 在ID段跟踪写端口的使用情况，以便能暂停该指令的发射
  - 一旦发现冲突，暂停当前指令的发射
- **Option 2**
  - 在进入MEM或WB段时，暂停冲突的指令，让有较长延时的指令先做。这里假设较长延时的指令，可能会更容易引起其他RAW相关，从而导致更多的stalls



# 关于数据相关

| Instruction | Clock cycle number | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| L.D F4,0(R2) | IF | ID | EX | MEM | WB | | | | | | | | | | | | |
| MUL.D F0,F4,F6 | | IF | ID | stall | M1 | M2 | M3 | M4 | M5 | M6 | M7 | MEM | WB | | | | |
| ADD.D F2,F0,F8 | | | IF | stall | ID | stall | stall | stall | stall | stall | stall | A1 | A2 | A3 | A4 | MEM | WB |
| S.D F2,0(R2) | | | | | IF | stall | stall | stall | stall | stall | stall | ID | EX | stall | stall | stall | MEM |

S.D 多延迟一个cycle，以消解与ADD.D的冲突

**这里注意stall的位置：**

**第一个stall在EX前，是因为Mul需用到LD的结果，等MEM取地址之后bypass过来**

**但第二个stall在ID前，是因为前一个取指后还未执行，所以你不能再取一次指令，等上一个取完指令进入EX后再ID**

# 新的冲突源

- **GPR与FPR间的数据传送造成的数据相关**
  - MOVI2FP and MOVFP2I instructions
- **如果在ID段进行相关检测，指令发射前须做如下检测：**
  - 结构相关 repeat
    - 循环间隔检测 Interval
    - 确定寄存器写端口是否可用
  - RAW相关
    - 列表所有待写的目的寄存器
    - 不发射以待写寄存器做为源寄存器的指令，插入latency个stall
  - WAW相关
    - 仍然使用上述待写寄存器列表
    - 不发射那些目的寄存器与待写寄存器列表中的指令有WAW冲突的指令，延迟1个cycle发射。



## 2.3 MIPS R4000

# review：标量流水线

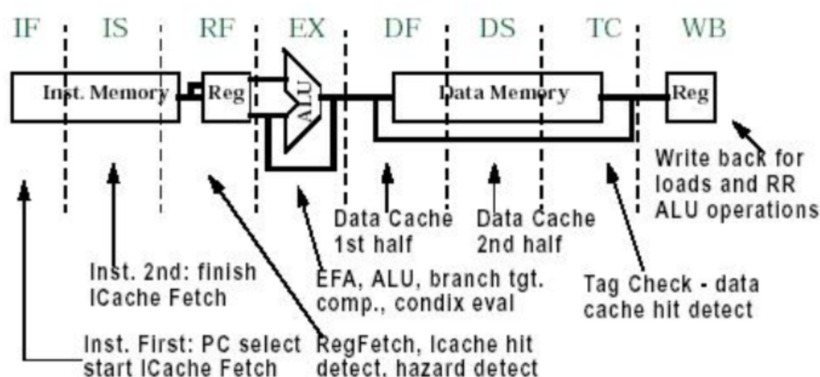
- **流水线提高的是指令带宽（吞吐率），而不是单条指令的执行速度**
- **限制流水线性能发挥的因素：相关（hazard）**
  - 结构相关：需要更多的硬件资源
  - 数据相关：bypassing or forwarding（硬件），编译器调度（软件）
  - 控制相关：尽早检测条件，计算目标地址，延迟转移，预测
- **编译器可降低数据相关和控制相关的开销**
  - Load 延迟槽、Branch 延迟槽、指令流静态调度
- **限制流水线性能发挥的因素：对异常事件的响应**
  - 引起控制流的变化，会冲刷（flush) 流水线



- **广义的异常：异常和中断**
  - 异常：程序运行过程中产生的**内部事件**，称为异常事件，因异常事件将控制权转移到监控程序，称为陷阱（Trap）
  - 中断：响应（处理）程序之外的**外部事件**，导致控制权转移到监控程序
- **同步异常 vs.异步异常**
  - 同步：在每次执行时异常事件发生在相同位置
  - 异步：在每次执行时异常事件可能发生在不同位置
- **精确异常 vs. 非精确异常**
  - 精确：响应异常后，可精确返回引起异常的指令位置
  - 非精确：响应异常后，无法精确返回引起异常的指令位置
- **异常处理:**
  - 异常处理的时机：指令commit阶段 即最后完成阶段
  - 优先级：外部事件引起的异常，内部事件引起的异常（同一条指令按时间顺序）
  - 过程：保存现场、处理、恢复现场

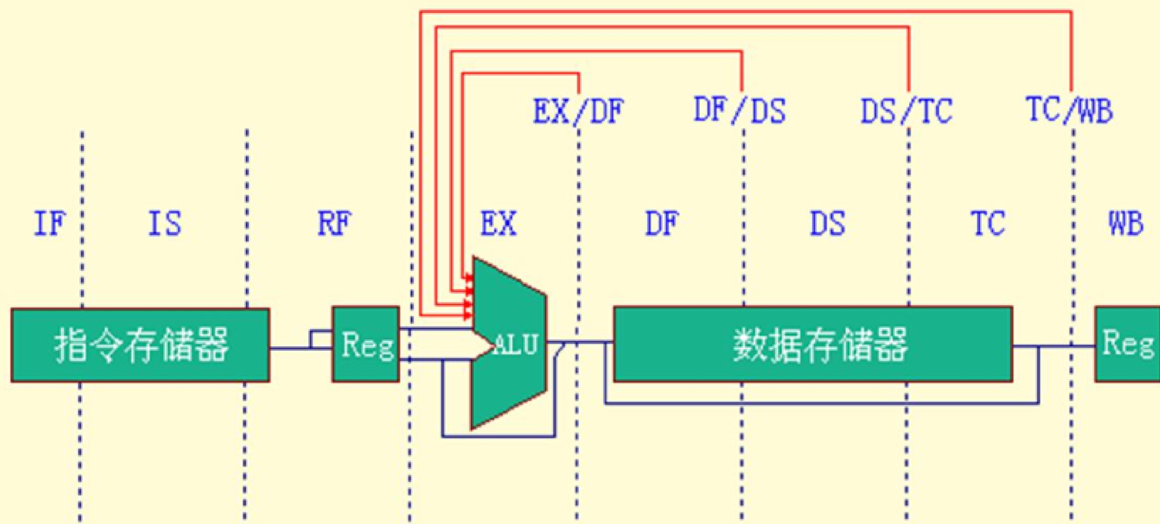
# MIPS R4000

- **实际的 64-bit 机器**
  - 主频100MHz ~200MHz
  - 较深的流水线（级数较多）(有时也称为 superpipelining)




- **IF–取指阶段前半部分；选择PC值，初始化指令cache的访问**
- **IS–取指阶段后半部分，主要完成访问指令cache的操作**
- **RF–指令译码，寄存器读取，相关检测及指令cache命中检测**
- **EX–执行，包括：计算有效地址，进行ALU操作，计算分支目标地址和检测分支条件**
- **DF–取数据，访问数据cache的前半部分**
- **DS–访问数据cache的后半部分**
- **TC–tag 检测，确定数据cache是否命中**
- **WB–Load操作和R-R操作的结果写回**

- **在使用定向技术的情况下，Load 延迟为2个cycles**
  - Load和与其相关的指令间必须有2条指令或两个bubbles
  - 原因：load的结果在DS结束时可用
- **分支延迟3个cycles**
  - 分支与目标指令间需要3条指令或3个bubbles
  - 原因：目标地址在EX段后才能知道




要看再看吧……