

组成原理笔记

上官凝

September 24, 2020

Contents

0.1	序幕	2
0.2	计算需要用的公式	2
0.3	这是一些概念	4
0.4	另外一个 perspective 来看待组成原理	7
0.4.1	计算机体系结构的 8 种属性	7
0.4.2	计算机系统概论	7
0.4.3	关于指令系统	7
0.4.4	关于几种 CPU 的设计	8
0.4.5	中断系统	9
0.4.6	存储系统	9
0.5	几个比较复杂的 points	9
0.5.1	寻址	9
0.5.2	中断 & 异常（理论）	11
0.5.3	流水线 CPU 中的异常处理	14
0.5.4	容错与校验	18
0.5.5	Cache	20

0.1 序幕

组成原理的实验很重要。它会让你对这门课学的有一个比较深入的理解。在写 CPU 的时候多翻 COD 书，可以参考我在 GitHub 上的代码 (Username:OrigamiAyc)。所有的策略都会有不同场景下的 tradeoff，最好思考一下优缺点和适用范围。图片的位置可能会有偏移（当这一页放不下就会挪到下一页正文当中，但是与正文的相对位置就改变了），请以图片标题为准

0.2 计算需要用到的公式

1 运算速度与处理器性能：(这里需要注意机器周期和时钟周期未必相等)

(a) CPI：指令周期，表示执行一条指令所需的平均时钟周期数，即

$$\frac{\text{执行程序所需 CPU 时钟周期数}}{\text{程序包含的指令条数}}$$

(b) MIPS：百万条指令每秒，即单位时间内执行的指令数，即

$$\frac{\text{程序的指令条数}}{\text{程序的执行时间} \times 10^6}$$

(c) MFLOPS：百万次浮点操作每秒，用来衡量机器浮点运算的性能，即

$$\frac{\text{程序的浮点操作次数}}{\text{程序的执行时间} \times 10^6}$$

注意这里计算的指包括浮点运算次数，却除以的是总时间

(d) 时间计算公式 (IC 是程序执行过程中所处理的指令数)：

$$\begin{aligned} T_{\text{CPU}} &= \text{CPI} \times \text{IC} \times T_{\text{CLK}} \\ &= \sum (\text{CPI}_i \times \text{IC}_i) \times T_{\text{CLK}} \end{aligned}$$

(e) Gibson 法求平均 CPI：

$$\overline{\text{CPI}} = \sum (\text{IC}_i / \text{IC}) \times \text{CPI}_i$$

注意在指令集改变之后，有可能会出现所有指令所占比例加起来不等于 100% 的情况，例如 ch1 page 68-70 例 1¹

(f) 性能与响应时间（从事件开始到结束之间的时间）成反比，与吞吐率（在单位时间内所能完成的工作量(任务)(多用户系统)）不同

(g) 可靠性与可用率：可靠性用平均无故障运行时间 (MTBF) 衡量，可用率为 $\text{MTBF}/(\text{MTBF} + \text{MTTR})$ (平均故障修复时间)

2 Amdahl 定律：适用于改进一段系统的一部分，用于计算改进之后的执行总时间。

$$T_e = T_0 [(1 - f_e) + \frac{f_e}{S_e}]$$

$$S = \frac{1}{(1 - f_e) + \frac{f_e}{S_e}}$$

$$f_e = T_1/T_0 \quad S_e = T_1/T_2$$

¹这个例子的答案有坑，×恰好标在了正确答案的位置，建议听课

其中 T_0, T_1, T_2, T_e 分别原总时长、原改进段时长、新改进段时长、新总时长, f_e, S_e 分别为可改进比例与部件加速比, S 为总加速比。理解阿姆达尔定律的计算实质, 即加速比 = 增强前时间/增强后时间, 可以计算同时多个优化

3 计算总共可能有多少种 X 地址指令: 指令字长 n , 可选有 2^n 种指令, 若不定长指令, 比如 4 个 bit 的 op code 是二地址, 10 个 bit 的 op code 是一地址, 那么用 4 个 bit 全是 1 则为一地址的方式来区分

4 单周期 CPU 确定主频: 关键路径法; 多周期和流水线都是用时最长的那一段

5 流水线最大加速比: 流水线站数 (ch4 page 13)

6 流水线的性能指标:

(a) 吞吐率: 单位时间内流水线所完成的任务数或输出结果的数量。分各段时间相等和不等来讨论 (设 m 为流水深度, n 为任务数)。

	各段时间相等	各段时间不等
最大吞吐率	$TP_{\max} = 1/\Delta t_0$	$TP_{\max} = 1/\max_i \Delta t_i$
实际吞吐率	$TP = \frac{TP_{\max}}{1 + (m - 1)/n}$	$TP = \frac{n}{\sum_{i=1}^m (\Delta t_i) + (n - 1) \max_i \Delta t_i}$

(b) 加速比: 流水线速度与等功能的非流水线速度之比。若为各段时间相等的情况, 加速比

$$S = \frac{T_{\text{非流水}}}{T_{\text{流水}}} = \frac{m}{1 + (m - 1)/n}$$

(c) 效率: 流水线的设备利用率(部件运行时间与总时间的比率, 即时空图上阴影部分与大矩形面积之比), $E = \frac{1}{1 + (m - 1)/n}$

(d) 三者关系: 效率是实际加速比 S 与最大加速比 m 之比; $E = TP\Delta t_0$

7 计算第几个周期在做什么。比如说, 在一个充分转发并有着异常处理的流水线 CPU 里面, 给一个汇编段, 第四条指令是 **add**, 问这个指令算数溢出的时候是第几个周期。算数溢出是 EX 段, 那画一下时空图, 就是 CC6。在 **add** 之前不巧还有一个 **sw** 于是 **interlock**, 再加一个周期, 就是 CC7。这种题要考虑到各种特殊情况 (只存在 ALU 之间转发、延迟槽以及分支预测、奇怪的结构比如 **beq** 需要三个周期)

8 类似的还有判断数据相关。如果有数据通路 (可以手画一个 draft) 可以把寄存器编号标在 wire 上面, 表示在这个时钟周期的时候这上面 (应该) 跑的是哪一个寄存器的值。简便起见可以把转发之后的结果放上去, 就不需要再画一大堆 MUX 了

9 控制信号的值也可以写上去的唉...以上三点同样适用于人工调试代码, 无论是实验还是考试的调试题目, 具体写法可以参考下文流水线非精确异常处理的例题 (COD 4th page 388)

10 评价指标

11 cache 性能效率指标: 设程序执行过程中, N_c 表示访问 Cache 完成存取的总次数, N_m 表示访问主存完成存取的总次数, 则命中率 $h = N_c/(N_c + N_m)$; 设 t_c 为命中时的 Cache 访问时间, t_m 为未命中时的主存访问时间, 则 Cache/主存平均访问时间 $t_a = h \times t_c + (1 - h) \times t_m$; 用 e 表示访问效率, 有 $e = \frac{t_c}{t_a} \times 100\%$

□CPU时间	指令字长
✓CPI	机器字长
✓IC	存储字长
✓主频	
□加速比 (Amdahl定律)	存储器带宽
✓部件加速比	总线带宽
✓可改进比例	
□流水线	机器周期
✓吞吐率	时钟周期
✓加速比	指令周期
✓效率	

Figure 1: 评价指标

0.3 这是一些概念

1. 机器字长：CPU 一次能处理的二进制数据的位数，通常与 CPU 的寄存器位数有关，多数计算机采用变字长运算
2. 字：
3. 机器字长、存储字长、数据字长都是字节的整数倍，不是字
4. 三二一零地址指令：计算访存次数（取指 + 取操作数/从 ACC + 存操作数/存 ACC）(4, 4/3, 2, 0)
5. 指令字长，单位为 bit
6. 地址是无符号整数，但是 MIPS 指令集里面的 I-type 指令，其 imme/addr 段是有符号数，所以能访问的地址量是正负偏移各自 2^{15} ，加起来 2^{16}
7. 小尾端和大尾端 (endians) 低/高位字节在低地址



Figure 2: 字地址与字节地址

8. 陷阱：一种意外事故中断，如除 0、运算溢出等 (ch2.1 page 36)
9. x86 寄存器（见下面的图）
10. 寄存器堆：在一个周期内，某个 REG 可以同时完成读写操作，但读出的是上一个周期写入的值

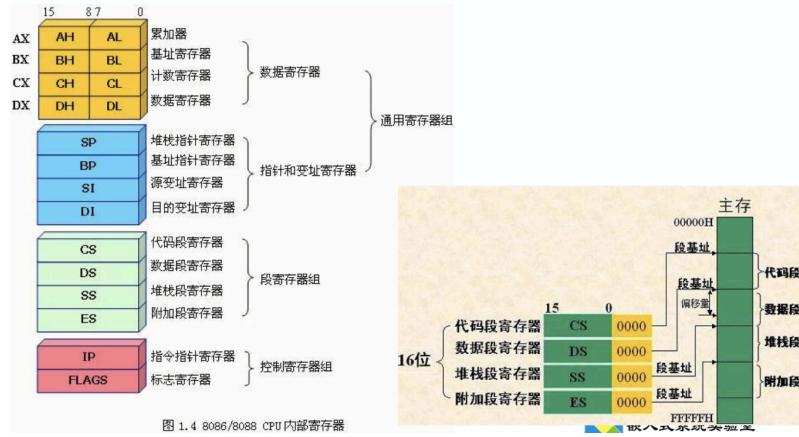


Figure 3: x86 registers

11. 多周期的 $\text{beq} = \text{PC} + \text{offset}$, 而单周期是 $\text{beq} = \text{NPC} + \text{offset}$, 这是因为多周期的 PC 是在从上一条指令的第二个周期保持到自己的第一个周期
12. MIPS 操作数在内存中要字对齐
13. 程序状态字 PSW, 记录各种异常的一个数组
14. 存储器件的 Hierarchy 以及延迟

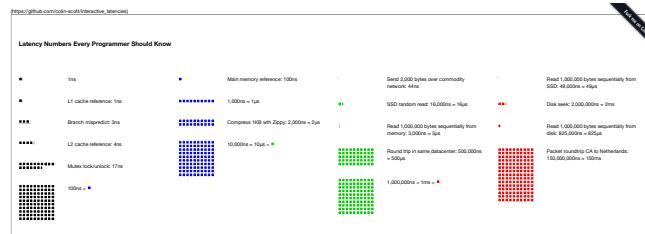


Figure 4: 存储系统的层次结构

15. 主存储器的技术指标
 - (a) 存储容量: 一个存储器中存储单元的总数, 一般用字数 (W) 或字节数 (B) 表示
 - (b) 存取时间: 存储器访问时间, 指一次读操作命令发出到该操作完成, 将数据送到数据总线上所经历的时间。写操作时间通常等于读操作时间
 - (c) 存储周期: 连续两次读写操作所需的最小间隔时间。存储周期一般大于存取时间
 - (d) 存储器带宽: 单位时间内存储器存取的信息量, 用字/秒、字节/秒或位/秒表示。是衡量数据传输速率的重要指标, 决定了以存储器为中心的计算机系统获得信息的速度, 是改善机器性能瓶颈的一个关键因素。提高存储器带宽的措施有:
 - i. 缩短存取时间、存储周期
 - ii. 增加存储字长, 使每个存取周期可读写更多二进制位数

- iii. 增加存储体：存储容量的扩展。位扩展：增加存储器的横向容量即存储字长；字扩展：增加存储器的纵向容量即存储单元的数量
16. 存储器与 CPU 的链接：① CPU 地址线的低位线与存储芯片的地址线相连，CPU 地址线的高位线或用于存储扩展，或用于片选等其他用途；② 数据线应当位数一样，否则位扩展存储芯片；③ CPU 的读/写命令线一般可以直接连到存储芯片的读/写控制端，通常高电平为读，低电平为写；④ 片选控制信号还与 CPU 的访存控制信号 $\overline{\text{MREQ}}$ (低电平有效) 有关
17. cache 是全硬件实现，注意和虚存的软件实现相区分。cache 只有很少的 OS 参与量
18. Cache 的写回策略（写命中，不命中还要访存的，分为写时取（调入 cache 再写）和绕写法（直接写入下一级存储器）。需要和内存进行同步，所以出现了两种方案：① 写直达：同时写 cache 和内存，用时为写内存的时间。② 写回：只有被替换的时候才写回，节约了时间，但是会增加 cache 的复杂性
19. 页式虚存
-
- ```

graph TD
 MMU[Memory Management Unit] -- "1. 使用虚地址查询" --> TLB[TLB]
 TLB -- "2. 未命中" --> MainMemory[主存]
 TLB -- "2. 命中" --> RealAddress[实地址]
 MainMemory -- "3. 使用实地址同时访问" --> Cache[Cache]
 Cache -- "4. 有一处命中" --> End[访存结束]
 Cache -- "4. 未命中" --> MainMemory
 MainMemory -- "5. 将所需页调入主存，并装入cache" --> Cache
 Cache -- "6. 修改相应页表项和TLB表项" --> TLB

```
- Figure 5: 页式虚存的访问流程
20. 总线送入 CPU 的信号称为输入信号，CPU 发出的信号称为输出信号
21. 总线的性能指标
- (a) 总线宽度：通常指数据总线的位数
  - (b) 总线频率： $1/\text{传输一次数据时间}$
  - (c) 总线带宽：总线的数据传输速率，即单位时间内总线传输数据的位数，通常用每秒传输信息的字节数来衡量（比特率和波特率）
  - (d) 总线复用：一条信号线上分时传送多种信号
  - (e) 其他指标：如负载能力、电源电压、总线宽度扩展等
22. 总线结构：由 4 部分构成
- (a) 数据传送总线：数据、地址、控制
  - (b) 仲裁总线：包括总线请求线和总线授权线
  - (c) 中断和同步总线：用于处理带优先级的中断操作，包括中断请求线和中断响应线
  - (d) 公用线：包括时钟线、电源线、地线、复位线及加电/断电的时序信号线等
23. 比特率：单位时间内传送的二进制有效数据的位数，单位为 bps；波特率：单位时间内传送的二进制数据的位数，单位为 bps

## 0.4 另外一个 perspective 来看待组成原理

### 0.4.1 计算机体系结构的 8 种属性

1. 数据表示：硬件能直接辨识和操作的数据类型和格式
2. 寻址方式：最小可寻址单位、寻址方式的种类、地址运算（ch2）
3. 寄存器组织：操作寄存器、变址寄存器、控制寄存器及专用寄存器的定义、数量和使用规则（ch3、ch4）
4. 指令系统：机器指令的操作类型、格式、指令间排序和控制机构（ch2、ch3、ch4）
5. 存储系统：最小编址单位、编址方式、主存容量、最大可编址空间（ch6）
6. 输入输出结构：输入输出的连接方式、处理机/存储器与输入输出设备间的数据交换方式、数据交换过程的控制（ch7、ch8）
7. 中断机构：中断类型、中断级别，以及中断响应方式等（ch5）
8. 信息保护：信息保护方式、硬件信息保护机制

### 0.4.2 计算机系统概论

1. 指令和数据都存储于存储器中，计算机如何区分它们？
2. 综述计算机技术的发展历程及热点问题
3. 最新 Intel 处理器的性能指标？
4. 计算机的开机过程？
5. C 语言计算机模型？
6. `getchar()` 的实现过程？
7. PC 系统活动与性能分析
  - (a) 跑一个 Benchmark，给出结果？每次执行同一个程序，结果都一样？
  - (b) 主频与计算性能的关系？

### 0.4.3 关于指令系统

1. CPU 的 ISA 要定义哪些内容？
2. 8086 为什么要采用段式内存管理模式？
3. Windows 系统中可执行程序的格式？
4. 调研现在常用的处理器是大尾端还是小尾端，如 x86, ARM, MIPS, PowerPC 等？

#### 0.4.4 关于几种 CPU 的设计

众所周知，组成原理实验是让设计 CPU 的，所以也可以说这门课是围绕着 CPU 来看的（中断和 memory 都在 CPU 里面），总线和外设也是由 CPU 来处理的

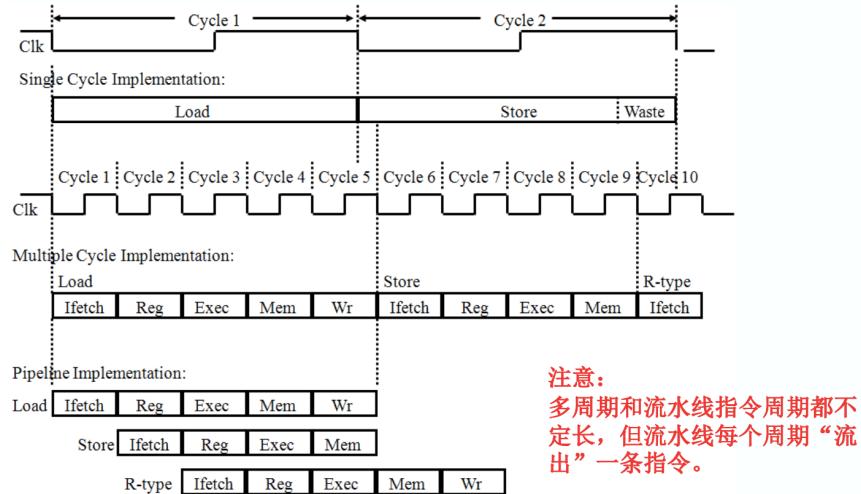


Figure 6: SingleCycle, MultiCycle and Pipeline CPU Design

1. 单周期和流水线的关系？

流水线的控制信号主要继承了单周期的

2. 流水线中实现一个周期访存(取指, 取数)的前提条件是什么？

3. 流水线执行第一条指令时, IF 在取指, 其他段在做什么?

如果控制信号清 0，则都在执行 **nop** 指令，如果没有清 0，由于控制信号未知则……

4. 各流水段的控制信号何时生成与释放?

都在本指令的 ID 段生成，在用到的时候从段间寄存器取出

5. 寻址方式如何实现的? 哪些寻址方式适合流水线?

对寄存器堆的访问是在 ID 段由段间寄存器 IR 取出 **rs = [21:25], rt = [16:20]** 得到的寄存器编号，对寄存器进行读操作。隐含寻址 (PC)、相对寻址 (跳转)、基址寻址 (访存)

6. 如果 R-type 指令也可访存，则应如何设计流水线?

7. MIPS 能否采用“取指、译码、执行”三段流水线?

8. Stall 原因与判断规则? 如何实现 stall?

### 0.4.5 中断系统

1. 中断周期要完成哪些微操作?

响应中断、存断点、保存现场、设置屏蔽字、关中断以及跳转 ISR

2. 多周期状态机中，出现溢出的指令是否将错误结果写回?

有可能会，也有可能不会

3. 多周期中状态机中，如何响应中断?

当前指令周期结束之后响应（下一次 IF 之前）

4. 指令顺序执行，中断“精确”；指令流水执行，中断“精确”或“非精确”可选

5. 精确中断，为何提交点是 M 段?

6. EPC 和 cause 应该在哪个段？异常检测电路?

7. mips 异常返回指令 eret 如何实现?

8. 异常与中断同时发生，优先级?

9. (分支) 延迟槽中的指令发生异常，EPC = ?

10. 比较中断、异常、过程调用(请求时间、响应时间，断点与现场，返回点，同步异步，中断周期、系统状态)

### 0.4.6 存储系统

1. 理解“码距”

(a) 具有 1 位纠错能力的编码系统最小码距是多少?

(b) 单纠错海明码码距是多少?

(c)  $(n+k, n)$ CRC 码码距是多少?

2. 比较海明码与 CRC 码的容错能力

## 0.5 几个比较复杂的 points

### 0.5.1 寻址

指令寻址比较简单，就 PC+4 或者计算跳跃地址。数据寻址，就是把操作数的形式地址 A，通过寻址特征，变换为操作数的有效地址 EA 的过程

下面以 MIPS 指令集为例解释一下这些寻址方式的一部分，没写的就是没有对应的例子：

1. 隐含寻址：指令中不明显给出操作数的地址，而是隐含在特定的寄存器中，有利于缩短指令字长

|         | Algorithms | Main Merits | Main Drawbacks |
|---------|------------|-------------|----------------|
| 隐含寻址    | 操作数在专用寄存器  | 无存储器访问      | 数据范围有限         |
| 立即寻址    | 地址段直接存立即数  | 无存储器访问      | 操作数幅值有限        |
| 直接寻址    | $EA=A$     | 简单          | 地址范围有限         |
| 间接寻址    | $EA=(A)^2$ | 大的地址范围      | 多重存储器访问        |
| 寄存器寻址   | $EA=R$     | 无存储器访问      | 地址范围有限         |
| 寄存器间接寻址 | $EA=(R)$   | 大的地址范围      | 额外存储器访问        |
| 偏移寻址    | $EA=A+(R)$ | 灵活          | 复杂             |
| 段寻址(偏移) | $EA=A+(R)$ | 灵活          | 复杂             |
| 堆栈寻址    | EA= 栈顶     | 无存储器访问      | 应用有限           |

2. 立即寻址：指令的地址字段指出的不是操作数地址，而是操作数本身。取出指令就可同时获得操作数，不必访问内存 **addi**
3. 直接寻址：形式地址即为真实地址 **jump**
4. 间接寻址：访存来解锁下一步线索/地址。另一个优点是便于编程，尤其是调用函数返回的时候，调用子程序前，将返回地址存入子程序最末条指令的形式地址的存储单元，便可准确返回原程序，如图所示 †<sup>3</sup>
5. 寄存器寻址：取出寄存器内部的数据 对**rs, rt**的访问
6. 寄存器间接寻址
7. 偏移寻址：看下面的 **mindnode**‡（注释一下段寻址方式：将主存空间分为若干段，每段的首地址存于基址寄存器，段内偏移量由指令字中的形式地址 A 指出）
8. 相对寻址：偏移寻址的一种，偏移量常用补码表示，基于局部性原理
9. 堆栈寻址

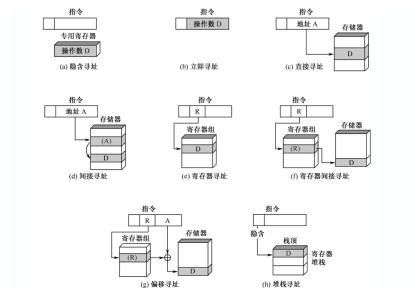


Figure 7: Big Picture

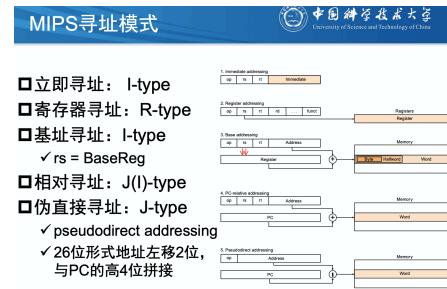


Figure 8: MIPS Address

对基址寻址和变址寻址的对比，其实我觉得记住lw是基址寻址就OK。

### 1. 基址寻址 (Base or Displacement addressing)

<sup>3</sup>附图均在本节最后

- (a) 以基址寄存器 (BR) 为基准进行寻址
  - (b) EA = 形式地址 + BR
  - (c) 基址寄存器：专用、通用 (显式 (通用寄存器)、隐式 (专用寄存器))
2. 变址寻址 (index)
- (a) 以变址寄存器 (IX) 的值为基准寻址
  - (b) EA = 形式地址 + IX
  - (c) 变址寄存器：专用、通用 (显式 (通用寄存器)、隐式 (专用寄存器))
3. 基址寻址 vs. 变址寻址
- (a) 基址：BR 由 OS 赋值，不变；形式地址可变
  - (b) 变址：IX 由程序员赋值，可变；形式地址不变
  - (c) 用途不同：基址—段寻址，变址—数组、字符串、循环
4. 基址寄存器是一个很长的寄存器，因为需要访问到所有的内存空间，才能给出每一个段的首地址
5. 变址寻址多数用在循环或者数组的自增操作上，所以 IX 寄存器不需要特别大
6. 从实现的角度来讲，也就是寄存器名字，叫成 BR 还是 IX，并非对应 WIDTH
7. 由于虚拟内存的地址映射，进程在调用地址空间时是一块连续的 memory，故基址寻址偏移不会 segmentation fault。BR 是在 OS 创建进程的时候，调进去的，所以说是由 OS 管理

### 0.5.2 中断 & 异常（理论）

三种中断：Exceptions 异常（随时发生，随时处理）、Interrupts 中断（随时发生，延迟响应）、Traps 陷阱（专用指令，特殊处理）

中断系统需要解决的问题：

- (a) 各中断源如何向 CPU 提出请求?(INTR, INTA)
- (b) 各中断源同时提出请求怎么办? (中断判优-程序查询，硬件排队-集中、分布)
- (c) CPU 什么条件、什么时间、以什么方式响应中断? (指令周期结束，中断隐指令)
- (d) 如何保护现场?(中断隐指令断点、ISR、堆栈)
- (e) 如何寻找入口地址?(硬件向量，软件查询)
- (f) 如何恢复现场，如何返回?(中断隐指令断点、ISR、堆栈)
- (g) 处理中断的过程中又出现新的中断怎么办? (多重中断，可屏蔽，设置屏蔽字)

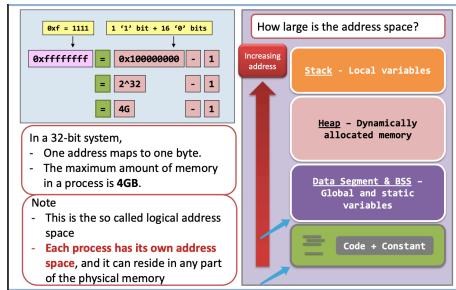


Figure 9: Data Segment (quoted from OS)



Figure 10: 偏移寻址

| 指令类型 |        | 助记符  |                | 指令格式    |           |                     |        | 寻址方式                        |       | 示例                          |  |
|------|--------|------|----------------|---------|-----------|---------------------|--------|-----------------------------|-------|-----------------------------|--|
| 3    | R-type | add  | op 00000       | rs      | rt        | rd                  | shamt  | function                    | 立即数寻址 | add \$1\$1, \$2\$2, \$3\$3  |  |
| 4    | R-type | sub  | op 00000       | rs      | rt        | rd                  | 00000  | 100000                      | 寄存器寻址 | sub \$1\$1, \$2\$2, \$3\$3  |  |
| 5    | R-type | and  | op 00000       | rs      | rt        | rd                  | 00000  | 100000                      | 寄存器寻址 | and \$1\$1, \$2\$2, \$3\$3  |  |
| 6    | R-type | or   | op 00000       | rs      | rt        | rd                  | 00000  | 100000                      | 寄存器寻址 | or \$1\$1, \$2\$2, \$3\$3   |  |
| 7    | R-type | nor  | op 00000       | rs      | rt        | rd                  | 00000  | 100000                      | 寄存器寻址 | nor \$1\$1, \$2\$2, \$3\$3  |  |
| 8    | R-type | slt  | op 00000       | rs      | rt        | rd                  | 00000  | 100000                      | 寄存器寻址 | slt \$1\$1, \$2\$2, \$3\$3  |  |
| 9    | R-type | sltu | op 00000       | rs      | rt        | rd                  | 00000  | 100110                      | 寄存器寻址 | sltu \$1\$1, \$2\$2, \$3\$3 |  |
| 10   | R-type | sll  | op 00000 00000 | rt      | rd        | shamt               | 00000  | 000000                      | 寄存器寻址 | sll \$1\$1, \$2\$2, \$3\$3  |  |
| 11   | R-type | srl  | op 00000 00000 | rt      | rd        | shamt               | 00000  | 000000                      | 寄存器寻址 | srl \$1\$1, \$2\$2, \$3\$3  |  |
| 12   | R-type | sra  | op 00000 00000 | rs      | rt        | rd                  | 00000  | 000000                      | 寄存器寻址 | sra \$1\$1, \$2\$2, \$3\$3  |  |
| 13   | R-type | jr   | op 00000       | rs      | 00000     | 00000               | 00000  | 001000                      | 寄存器寻址 | jr \$1\$1                   |  |
| 14   |        | 助记符  |                | 指令格式    |           |                     |        | 寻址方式                        |       | 示例                          |  |
| 15   | I-type | andi | op 00100       | rs      | rt        | constant or address |        | 立即数寻址                       |       | andi \$1\$1, \$2\$2, \$3\$3 |  |
| 16   | I-type | lw   | op 10011       | rs      | rt        | immediate           |        | 基址寻址                        |       | lw \$1\$1, \$2\$2(\$3\$3)   |  |
| 17   | I-type | sw   | op 10110       | rs      | rt        | immediate           |        | 基址寻址                        |       | sw \$1\$1, \$2\$2(\$3\$3)   |  |
| 18   | I-type | lh   | op 10010       | rs      | rt        | immediate           |        | 基址寻址                        |       | lh \$1\$1, \$2\$2(\$3\$3)   |  |
| 19   | I-type | lhud | op 10010       | rs      | rt        | immediate           |        | 基址寻址                        |       | lhud \$1\$1, \$2\$2(\$3\$3) |  |
| 20   | I-type | lhuh | op 10010       | rs      | rt        | immediate           |        | 基址寻址                        |       | lhuh \$1\$1, \$2\$2(\$3\$3) |  |
| 21   | I-type | sh   | op 10010       | rs      | rt        | immediate           |        | 基址寻址                        |       | sh \$1\$1, \$2\$2(\$3\$3)   |  |
| 22   | I-type | lb   | op 10010       | rs      | rt        | immediate           |        | 基址寻址                        |       | lb \$1\$1, \$2\$2(\$3\$3)   |  |
| 23   | I-type | lbu  | op 10010       | rs      | rt        | immediate           |        | 基址寻址                        |       | lbu \$1\$1, \$2\$2(\$3\$3)  |  |
| 24   | I-type | sb   | op 10010       | rs      | rt        | immediate           |        | 基址寻址                        |       | sb \$1\$1, \$2\$2(\$3\$3)   |  |
| 25   | I-type | shd  | op 10010       | rs      | rt        | immediate           |        | 基址寻址                        |       | shd \$1\$1, \$2\$2(\$3\$3)  |  |
| 26   | I-type | sc   | op 10010       | rs      | rt        | immediate           |        | 基址寻址                        |       | sc \$1\$1, \$2\$2(\$3\$3)   |  |
| 27   | I-type | lui  | op 00111 00000 | rt      | immediate | 立即数寻址               |        | 立即数寻址                       |       | lui \$1\$1                  |  |
| 28   | I-type | andi | op 00100       | rs      | rt        | immediate           |        | 立即数寻址                       |       | andi \$1\$1, \$2\$2, \$3\$3 |  |
| 29   | I-type | ori  | op 00100       | rs      | rt        | immediate           |        | 立即数寻址                       |       | ori \$1\$1, \$2\$2, \$3\$3  |  |
| 30   | I-type | beq  | op 00010       | rs      | rt        | immediate           | PC相对寻址 | beq \$1\$1, \$2\$2, \$3\$3  |       |                             |  |
| 31   | I-type | bne  | op 00010       | rs      | rt        | immediate           | PC相对寻址 | bne \$1\$1, \$2\$2, \$3\$3  |       |                             |  |
| 32   | I-type | slt  | op 00010       | rs      | rt        | immediate           | 立即数寻址  | slt \$1\$1, \$2\$2, \$3\$3  |       |                             |  |
| 33   | I-type | sltu | op 00010       | rs      | rt        | immediate           | 立即数寻址  | sltu \$1\$1, \$2\$2, \$3\$3 |       |                             |  |
| 34   |        |      |                |         |           |                     |        |                             |       |                             |  |
| 35   |        | 指令类型 |                | 指令格式    |           |                     |        | 寻址方式                        |       | 示例                          |  |
| 36   | J-type | jal  | op 00000       | address | address   | 直接地址寻址              |        | 直接地址寻址                      |       | jal \$1\$1                  |  |
| 37   | J-type | jalr | op 00001       | address | address   | 直接地址寻址              |        | 直接地址寻址                      |       | jalr \$1\$1                 |  |

Figure 11: MIPS Address Seek

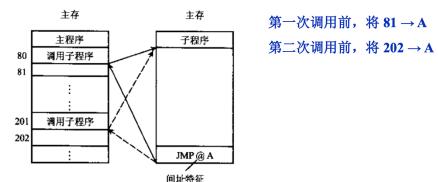


Figure 12: †间接寻址方式的优点——便于编程

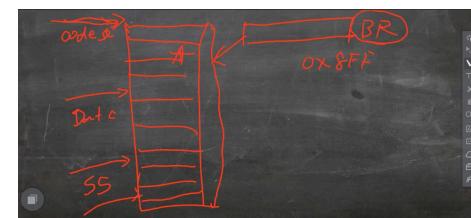
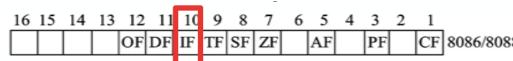


Figure 13: BR and Data Segment



中断机构组成

1. CPU 中断禁止/允许:IF@PSW。PSW 即程序状态字(程序状态寄存器), Program Status Word。
  2. CPU 中断请求/响应控制:INTR、INTA
  3. 中断响应/返回:中断隐指令
  4. 断点/现场保存:MEM(stack)
  5. 中断服务 ① 中断源识别/判优: 中断控制器 ② ISR (Interrupt Service Routine) 入口: 向量方式、非向量方式

**中断请求标记 INTR** (各中断源如何向 CPU 提出请求?) 一个请求源对应一个 INTR 中断请求标记触发器, 多个 INTR 组成中断请求标记寄存器。ITR 可以分散在各个中断源的接口电路里面或者集中在 CPU 的中断系统内部。INTR (interrupt request) 和 INTA (interrupt acknowledge) 都是控制信号。

**中断判优逻辑** (各中断源同时提出请求怎么办?) **if-else** 优先级

1. 软件实现 (程序查询)
2. 硬件实现 (排队器), 分为 ① 分散在各个中断源的接口电路中的链式排队器 ② 集中在 CPU 内的链式排队器

**中断响应** (*CPU* 在什么条件、什么时间、以什么方式响应中断?) (如何寻找入口地址?)

**响应中断的条件** *CPU* 允许中断触发器 **EINT == 1**

**响应中断的时间** 指令执行周期结束时刻由 *CPU* 发查询信号

1. 外部中断 (IO): 异步 (延迟响应), 指令周期结束
2. 内部中断 (陷阱和异常): 同步 (“马上”响应)

注意一个指令周期可能包含多个时钟周期

**中断隐指令** 中断周期完成的主要操作, 通过中断隐指令完成

1. 保护程序断点<sup>4</sup>: 断点存于特定地址 (如 0 号地址) 内, 或者在堆栈
2. 寻找服务程序入口地址:
  - (a) 向量地址 → PC (硬件向量法): 在 ISR 里面有很多 **JMP** 跳转到特定的内存空间, 排队器输出到向量地址形成部件里面, 然后间接访存跳转到 ISR 的入口地址 (把入口地址哈希起来是一个连续数组可以提高性能? (雾))
  - (b) 中断识别程序入口地址 M → PC (软件查询法)
3. 硬件关中断 (其实也可以不做, 实现嵌套中断): ① ENIT 允许中断触发器置为 0 ② INT 中断标记触发器置为 1

**中断周期** 由于中断的延迟响应特性, 在 *CPU* 写回之后才会着手查询和响应可能出现的 *Interrupts*。中断即 *CPU* 与中断控制器通信, 响应外部事件 (INTR,INTA), 包括 ① 是否有中断请求 ② 识别中断源, 获得中断向量。采取的动作: 存 PC 和 PSW, 关中断, 转 ISR

**保护和恢复现场** (如何保护现场?) (如何恢复现场、如何返回?) 保护现场分为断点和寄存器内容两部分。前者由中断隐指令, 通过压栈<sup>5</sup>或者存在特定地址完成; 后者由 ISR 压栈完成, 是一个被调用者保存策略, 只需保存部分寄存器 (ISR 中可能还会涉及关中断和开中断, (参考 OS 里面 critical section should be as tight as possible 的说法, ) 在保护现场两端需要关掉中断)。恢复就是反过来, 返回是从栈里面再读出来 PC 值 (为什么不用间接寻址?)

**中断屏蔽技术** (处理中断的过程中又出现新的中断怎么办?)

---

<sup>4</sup>断点仅指 PC, 或者 NPC

<sup>5</sup>栈的地址由高向低生长 (OS)

**多重中断的概念** 中断嵌套，即在执行 ISR 的过程中出现并响应了新的中断请求，亦即中断服务函数的嵌套调用

### 实现多重中断的条件

1. 提前设置开中断，即中断隐指令响应，存断点之后关中断；中断服务开始之后（保护现场结束之后，ISR 核心程序开始之前）开中断。以及在恢复现场之前关中断，之后开中断
2. 优先级别高的中断源有权中断优先级别低的中断源

### 屏蔽技术

- ① 屏蔽触发器的作用：动态优先级设定，若某高优先级的中断源其  $MASK_i$  为 1，则它的请求将被屏蔽，即手动调整它的优先级为无穷低
- ② 屏蔽字：每一个中断源都有自己的屏蔽字（姑且称为  $\bar{x}_i$ ），其位数均等于中断源总数。若  $\bar{x}_i$  的第  $m$  项为 0，则中断源  $i$  不能屏蔽中断源  $m$ 。
- ③ 新屏蔽字的设置：新的中断服务流程为：保护现场 - 置屏蔽字 - 开中断 - 中断服务 - 关中断 - 恢复现场 - 恢复屏蔽字 - 开中断
- ④ 屏蔽技术可以改变处理的优先等级（响应优先级不可改变；处理优先级可改变（通过重新设置屏蔽字））

新屏蔽字图中，红色圈里面的锯齿为 BCD 三个中断请求的响应时间（响应时间包括保存断点、

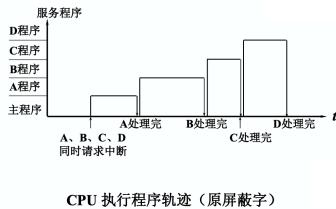


Figure 14: 响应优先级即处理优先级

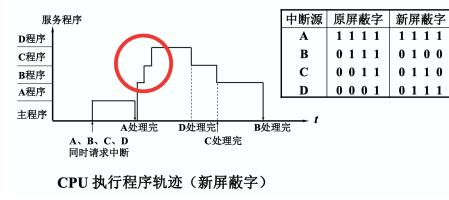


Figure 15: 响应后不一定处理

保存现场、设置新的屏蔽字）。出现这种情况的原因是响应优先级与处理优先级不一致。由原屏蔽字表每一行 0 的数量可知，响应优先级  $A \rightarrow B \rightarrow C \rightarrow D$  降序排列，当响应某一个中断请求（如 B）并进入 ISR 时，置屏蔽字一步会将屏蔽触发器组按照 B 的新屏蔽字置数，即只有 B 的请求被屏蔽。在执行 ISR 时，和以前一样会在 WB 之后检查中断请求，这时 A 已经处理完成，B 已经响应，C 和 D 正在请求且未被屏蔽，于是按照响应优先级响应 C 的请求。这样做的目的是可以提供一个人为改变实际处理顺序的机会，让在物理上优先级低的中断源设备也可以先处理更重要的请求，或者人为地屏蔽某个中断源的请求，便于程序控制

### 0.5.3 流水线 CPU 中的异常处理

多周期的比较简单，就只放几个图吧

流水线的处理较多周期要复杂很多，这是由流水线的并行导致的。在流水线里面，“顺序执行”只是一种逻辑关系：断点和状态难以确定。

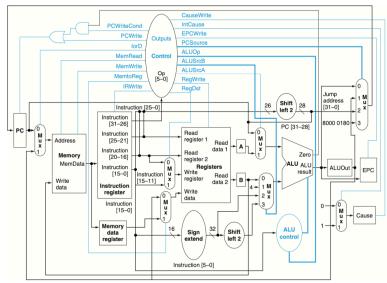


Figure 16: 多周期异常处理数据通路



Figure 17: 多周期异常处理主要内容

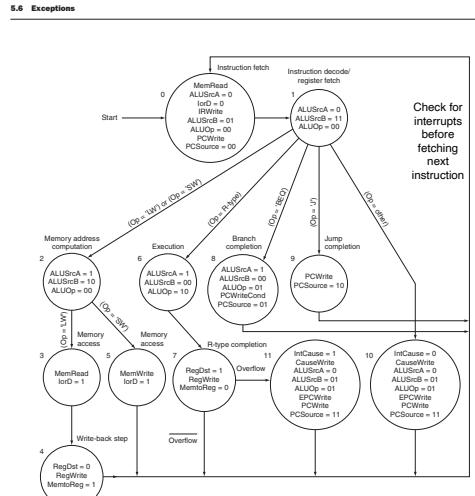


Figure 18: 多周期异常处理状态机

- 后续指令在产生异常指令完成之前改变了系统的部分状态，如 `lw` 指令在 MEM 段产生异常，而其后的 R-type 指令设置了 0 标志（PSW 内程序状态字）
- 异常发生顺序与指令执行顺序不一定相同，如 `addi` 指令在 ID 段 CC3 发生异常，但 `lw` 指令在 MEM 段 CC4 异常
- 转移指令和分支预测给异常处理带来了麻烦

流水线处理分为非精确处理和精确处理两种，非精确处理也有简单一些的和复杂一些的

**非精确方案 1：甩锅** 允许已进入流水线中的指令都做完再去做异常处理。断点：无论在第 i 条指令的哪一级流水段发生异常，都不再允许后继指令进入流水线，断点为最后进入流水线那条指令的地址（非精确：断点不是当前指令；可变：不同段发生异常，EPC 增量不同）。这种方案需要比较强的 OS 支持，因为没有给出来具体哪一条指令的哪一个段出了问题，需要 OS 自行判断

**非精确方案 2：挑简单的做** 将异常指令的后续指令排空。视为一种控制相关，按照 interlock 来加 `nop` 指令，将控制信号清 0，并不允许后续指令执行。处理步骤如下（前三步不就是分支的清空延迟槽嘛 ówò）：

- 暂停指令流中导致异常的指令
- 执行完异常指令之前的所有指令
- 清除异常指令之后的所有指令

4. 记录异常原因
5. 保存断点
6. 转异常处理程序

在记录 EPC 的时候，有两种处理方式。一个是把 PC + 4 一级一级传下来，另一个是把现在的 PC + 4 写到 EPC 里面，即进到流水线的最后一条指令

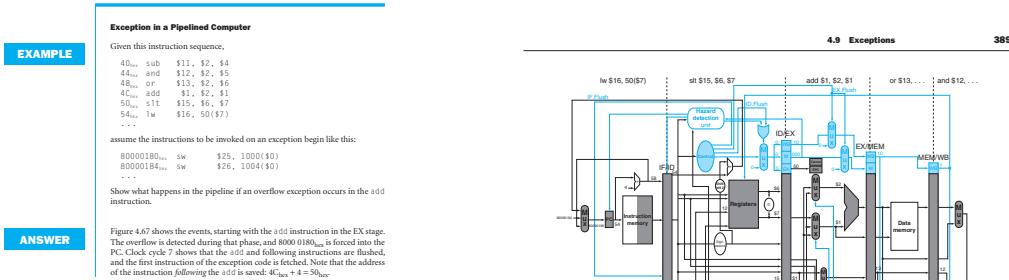


Figure 19: 非精确处理的一个例子

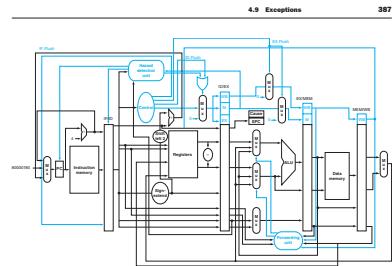


Figure 20: 流水线非精确处理异常的数据通路

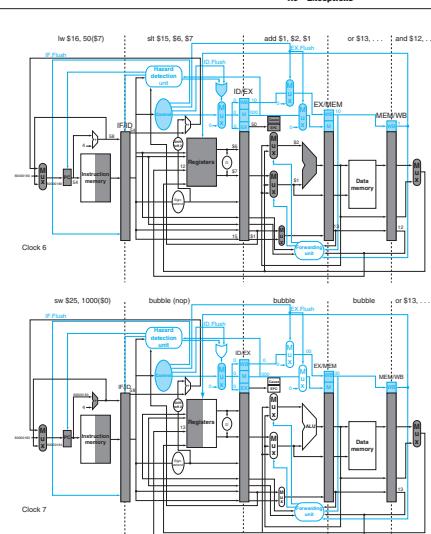


Figure 21: 非精确处理例题的数据通路分析

### 非精确异常的问题 Drawbacks

1. 异常响应时间较长
2. 如果等进入流水线的指令执行结束，可能会导致程序出错
  - (a) i: ADD R1, R2; (R1)+(R2) -> R1, 如果此时溢出?
  - (b) i+1: MUL R3, R1; (R3)×(R1) -> R3, 无效执行!
  - (c) (此处假设有 forwarding)
3. 程序调试不便：程序员在第 i 条指令设置断点，但程序不能准确中断在所设置的断点处。

**精确处理异常** 遇到的困难主要是开销会很大，因为需要保证异常指令可以重启，就得做很完全的保护现场：① 安全停止流水线，并完整保存当前状态 ② 需要大量后援寄存器保存流水线中各指令的现场，包括 RegFile、PSW、流水段寄存器（含各段的控制寄存器）。MIPS 采用提交点技术来实现：先发生的异常并不立即处理，只是被标记。每一条指令在执行过程中记录下来自己在哪一个阶段出了错，这些错在 WB 段结束之后再去处理。这样做就像前面要求 R-type 指令也要到第五个周期写回是类似的思想，统一的处理位置使得先进入流水线的指令可以先处理问题。EXCn 寄存器：保存异常类型。流水线中最深的指令引起的异常最优先。直到 MEM 段才 Flush

1. 保持流水线的异常标记直到提交点 (MEM 段)
2. 如果提交点有异常，则更新 cause 和 EPC，清除所有流水段，新 PC 值到 fetch 段
3. 早期流水段的异常抑制后来的异常 (flush IF/ID/EX)
4. 提交点处引入中断处理

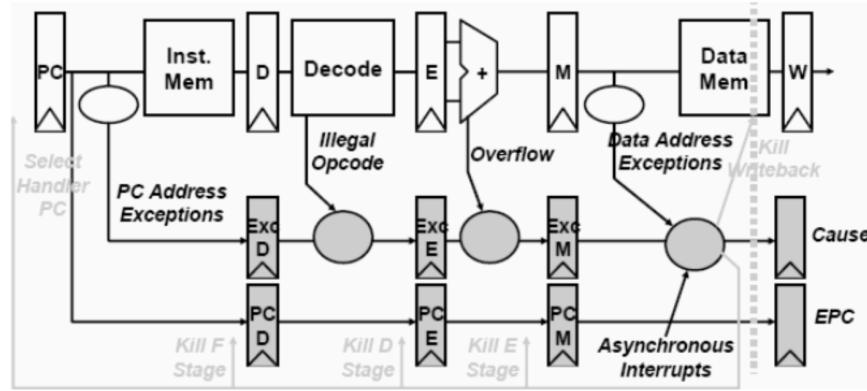


Figure 22: 提交点和两条流水线

**两条流水线** 提交点方案可以被看作是两条流水线，其中一个是之前普通的流水，另一个则是用于收集并在 MEM 段处理在各段发生的异常。这是因为不能保证前面的段出错了后面不再出错，所以每一级都需要有接收；另一个原因是在同一时钟周期，每一级流水段中是不同的指令。为了简单起见，每一条指令只保留最先发生的异常，即上文的第三条策略。

#### 为什么要在 MEM 段提交 前提：WB 段不会出错

1. (需要异步在统一的段处理) 都在 MEM 段才处理，则可以使处理顺序即为流水指令顺序，保证一个 FCFS 的顺序，即先进入流水线的指令先进入异常处理程序
2. (EX 段不可以) 在 MEM 段还有可能出错，且放到 MEM 段可以保证前一条指令已执行完成
3. (WB 段不可以) 后续指令被 flush (kill)，保证没有指令完成写回

#### 0.5.4 容错与校验

基本解决方案是冗余 Redundancy，用时间和/或空间的代价来换取信息的正确性，主要包括 Information redundancy 信息冗余（海明码、CRC 码、奇偶校验码）、Time redundancy 时间冗余（回滚）、Physical redundancy 空间冗余（复用、磁盘冗余阵列）

**码距纠错理论** 理论分析纠错方法的可行性。比如校验一组数据的传输是否正确， $L$  就是传送的数据和收到的数据之间的码距  $d$  的最小值（因为每一位都有可能翻转，所以说一般来说是只有一位错误的时候，再加上校验位的翻转数量是  $L$ ）

1. 一个编码系统中任意两个合法编码（码字）之间不同的二进数位（bit）数叫这两个码字的码距，也称为汉明距离，用  $d$  表示。例如码字 10010 和 01110，有 3 个位置的码元不同，所以  $d = 3$ 。
2. 整个编码系统中任意两个码字的最小距离就是该编码系统的码距。任何一种编码是否具有检测能力或纠错能力，都与编码的最小距离有关。
3. 在一个码组内为了检测  $D$  个误码，需要最小码距  $L \geq D + 1$
4. 在一个码组内为了纠正  $C$  个误码，需要最小码距  $L \geq 2C + 1$
5.  $L - 1 \geq D + C$  且  $D \geq C$
6. 即编码最小距离  $L$  越大，则其检测错误的位数  $D$  也越大，纠正错误位数  $C$  也越大，且纠错能力恒小于或等于检测能力。
  - (a) 例如， $L = 2$ ，则  $D = 1$ ,  $C = 0$ 。码距 = 2 才能检测 1 位错
  - (b) 例如， $L = 3$ ，则  $D = 2$ ,  $C = 0$ ; 或  $D = 1$ ,  $C = 1$ 。码距 = 3 才能检测 2 位错，或者检测 1 位错，纠错 1 位。

**奇偶编码校验** 在被传送的  $n$  位代码  $b_{n-1}b_{n-2}\cdots b_1b_0$  上增加一位校验位  $P$  (Parity)。奇校验是使 1 的个数为奇数，偶校验反之。可以检测一位错

**Hamming 码** 海明码是可以一位检验一位纠错的

**写出海明码  $H_n \cdots H_1$**  假如说有  $k$  位数据  $D_k \cdots D_1$ ，那么海明码的位数  $r$  需要满足  $2^r - 1 \geq k + r$ 。另一种考虑的 perspective 是，从最右端 ( $H_1$ ) 开始写，下标为  $2^i$  则填  $P_{i+1}$ ，其余位置顺序填  $D_j$ 。当数据都填完的时候就得到一个 Hamming 码数组（下表前两行

**计算校验位** 下面都假设是偶校验，如果是奇校验那么得到的校验位再取一次反首先填入每一位的校验位号。规则为该下标的二进制表示中 1 的位置，即为此海明位所用到的校验位。如  $H_7$ ,  $7_{Dec} = (111)_{Bin}$ ，则它用到的校验位为  $P_1, P_2, P_3$ ，即  $H_1, H_2, H_4$ 。这时候从另一个角度来看，就是每一个校验位在除了自己以外的一些位  $i_1, \dots, i_k$  被用到，那么校验位的值就是  $\sum H_{i_k}$ ，求和为  $F_2$  域上求和，即异或  $\oplus$ 。如  $P_1 = D_4 \oplus D_2 \oplus D_1$

**接收端校验** 检验数位数等于校验位位数。检验数计算规则为  $S_i = P_i \oplus \sum H_{i_k}$ ，即校验位异或所有用到这个校验位的数据位（都是接收到的值）。如  $S_1 = P_1 \oplus D_4 \oplus D_2 \oplus D_1$ 。然后检验数  $S_r \cdots S_1$  的十进制值  $x$  就表示  $H_x$  是错误的

| Hamming 码位号 | $H_7$ | $H_6$ | $H_5$ | $H_4$ | $H_3$ | $H_2$ | $H_1$ |
|-------------|-------|-------|-------|-------|-------|-------|-------|
| 数据位/校验位     | $D_4$ | $D_3$ | $D_2$ | $P_3$ | $D_1$ | $P_2$ | $P_1$ |
| 数值 (先填入数据位) |       |       |       |       |       |       |       |
| 参与检验的校验位号   | 1,2,4 | 2,4   | 1,4   | 4     | 1,2   | 2     | 1     |

## CRC 循环冗余码

**模 2 运算** 没有借位和进位。记住  $0 - 1 = 1$  然后乘除法都老实用竖式算就 OK。比如说除法，上 0 还是 1 就是看第一位

**CRC 的生成** 校验位与海明码不同，统一放在数据位的后面。

已知：n 位原始数据和 k + 1 位生成多项式。

1. 在数据后面添加 k 个 0，变成  $n + k$  位
2. 将得到的这个二进制数除以生成多项式，得到商和余数（模 2 运算）
3. 把余数补在 n 位数据后面，得到  $(n + k, n)$  码

**CRC 的错误与余数循环** 用接收到的  $n + k$  位数据串模 2 除生成多项式，得到一个余数。如果某一位出错，则余数不为 0。不同位出错，余数不同，余数和出错位之间的对应关系不变。与待测码无关，与生成多项式有关。这个余数具有一个循环特性，即对余数补 0，再除以生成多项式，便得到错误位左移一位的那个余数

**纠错办法** 一个是把整张错误-余数对照表列出来，然后对比余数得到错误位。另一个是用所谓的循环移位法。

1. 将 CRC 码进行左循环移位，至出错位被移至最高位，即
  - (a) 将接收数据模 2 除以生成多项式，得到一个余数
  - (b) 将这个余数右边补 0，除以生成多项式，得到一个新的余数，同时把接收数据左移一位，即 `CRC_next <= {CRC_curr[WIDTH-2:0], CRC_curr[WIDTH-1]}`，亦即将数据看成一个循环队列
  - (c) 直到得到的余数等于最高位出错时的余数（这个可以单独算一下，把原始数据最高位取反然后除以生成多项式）
2. 对最高位取反纠错
3. 继续左循环移位，直到余数和一开始的相等

原理：余数和 CRC 码在开始移位之后便相互独立，余数的补 0 除实际上是一个计数器的作用，判断错误位转到最高位以及循环一周。有一点像转魔方，转到某一个样子之后做一个处理，然后恢复不需要改变的块的位置

## 0.5.5 Cache

cache 是全硬件实现，注意和虚存的软件实现相区分。cache 只有很少的 OS 参与量

**思考这 4 个各自的优点** 根据不用的应用特征，执行的轨迹或者数据通路、需要的数据量及其相关程度，以及应用的场景来选择不同的有针对性的 cache

### 1. 单一 Cache 和多级 Cache

- (a) 单一 Cache：CPU 和主存之间只设一个 Cache。随着技术发展，与 CPU 制作在同一芯片中——片内 Cache。存取速度快，不占用片外系统总线，但容量受限
- (b) 多级 Cache：在片内 Cache 基础上，再增加一(多)级片外 Cache。与 CPU 之间使用独立数据通路

### 2. 统一 Cache 和分离 Cache

- (a) 统一 Cache，指令和数据存放在同一 Cache 内
- (b) 分离 Cache，指令和数据分别存放——指令 Cache 和数据 Cache

**Cache 与 Memory 的地址映射** 地址映射是将主存地址定位到 Cache 地址的方法，Cache 大小要小于主存大小，且数据传输的最小单元是块，但是 CPU 取数据是以字为单位的。联想一下 OS 里面 page 的概念（虽然那个是内存对于硬盘而言的）。CPU 给出来的是在内存里面的字/字节编号，由于分块，故低位拿出来判断在一块之内的偏移量，高位拿出来判断块号。直接映射由于是把 (Cache 块数) 个块绑成一组平行映射到 Cache，那就类似的再次高低位拆分检索，标记位数缩短

**全相连映射** 适用于小容量的 Cache。设主存共有  $2^s$  个块，每一个块里面又有  $2^w$  个字（或字节），那么 cache 就要有每一个 block 都能够偏移寻址到整个内存的能力，所以维持一个标记表：cache 里面每一块的标记处存着这一块对应在内存里面的块号（块里面就对应把整个块的内容都 copy 下来）。为了快速检索，内存地址的块号与 Cache 中所有行的标记同时进行比较。优点：冲突概率小，Cache 的利用高。缺点：比较器难实现，需要一个访问速度很快代价高的相联存储器

**直接映射** 适合大容量的 Cache。每个主存块只能映射到 Cache 的一个特定块位置，每个 Cache 块可以和若干个主存块对应。 $i$  表示 Cache 块号， $j$  表示主存块号， $m$  表示 Cache 的总块数（一般为 2 的幂数），则  $i = j \bmod m$ 。判断命中的时候，先由块号的低位优点：比较电路少，硬件实现简单，Cache 地址为主存地址的低几位，不需变换。缺点：冲突概率高

**组相连映射** 是前两种方案的折中考虑。将 Cache 块分为  $u$  组，每组有  $v$  块，主存块存放在哪个组是固定的，而存到组内哪个块则是灵活的：间直接映射，组内全相联映射。Cache 块数  $m = u \times v$ ；组号  $q = j \bmod u$ ：主存的第  $j$  块内容映射到 Cache 的第  $q$  组中的某块 ( $v = 1$ , 直接映射;  $u = 1$ , 全相联映射)  $v$  的取值一般是 2 的幂，称之为  $v$  路组相联 cache

**联想 OS** 想一下进程调度里面，允许踢出其他进程的 page 就像全相连映射，可以将内存中的一个 block 映射到随便哪一个位置。只可以踢出自己 frames 里面的页，如果有好多页就像是组相连映射，同组之内可以有轮转的算法