计算机系统结构

不涵盖的内容：多周期处理器，乱序执行，MSI协议，x86指令集相关

目录按照内容层次划分，并非考试占比

Chap1 计算机系统性能指标

执行时间

完成某任务的总时间

性能: 1/执行时间；加速比: 性能比

Example

A=200 cycles, B=350 cycles

speed up: 1.75

% increase: ((350-200)/200) * 100 = 75%

% decrease: ((350-200)/350) * 100 = 42.3%

吞吐率

单位 时间内完成的任务数量

执行时间与吞吐率

- 吞吐率高：在单位时间完成的任务越多越好（管理员角度）
- 执行时间短：提交的任务越快完成越好（使用者角度）

优化方法

- 使用更快的处理器

  既能缩短一个程序的执行时间，又能提高系统整体的吞吐率
- 将一个处理器增加为多个处理器

  只增加吞吐率，无法缩短单个任务的执行时间

CPU时间

给定程序任务占用处理器的时间（处理器有进程切换）

公式：CPU time = IC * CPI * Clock time

IC: # instructions (instruction count)

CPI: average clock per instruction

Clock time: duration of clock

优化CPU性能

三个要素之间折衷

- IC
  - Compiler optimizations (constant folding, constant propagation)
  - ISA (More complex instructions)
- CPI
  - Microarchitecture (Pipelining, Out of order execution, branch prediction)
  - Compiler (Instruction scheduling)
  - ISA (Simpler instructions)
- Clock period

- Technology (Smaller transistors)
- ISA (Simple instructions that can be easily decoded)
- Microarchitecture (Simple architecture)

CPU吞吐率

MIPS：每秒执行百万条指令数

FLOPS：每秒浮点运算次数

Amdahl定律

加快某部件执行速度所获得的系统性能加速比，受限于该部件在系统中所占的重要性比例

$$\eta = \frac{1}{1 - P + P/S} \rightarrow \frac{1}{1 - P}$$

失效情况

不适用于规模可扩展的并行应用程序的性能分析

- Strong scaling

  shrink the serial component
- Weak scaling

  increase problem size (e.g. video game 2K/4K)

Chap2 数据的表示和运算

Chap2.1 整数的表示和运算

按位取反后+1

MSB表示负权重

补码 (Two's complement)

- 0是唯一的
- 符号位参与运算

整数溢出

正正负，负负正

Tip：将运算数的符号位设置为00(正数)或11(负数)，如果结果的符号位是01或10则溢出(e.g. 11100+11011=10111, -4+-5=7)

类型转换

零扩展 VS 符号扩展

note: 先变大小，再变类型

降阶规则: signed与unsigned一起，则转化成unsigned

乘法运算

无符号乘法: Out = u * v mod 2^w

无符号乘法和补码乘法位级一致

编译器优化

- 用位移计算乘法

  u<<3 = u * 8

  (u<<5) - (u<<3) = u * 24

- 用位移计算除法

  右移向下舍入，而非向0舍入

  ```
  (x < 0 ? x + (1 << k) - 1 : x) >> k
  ```

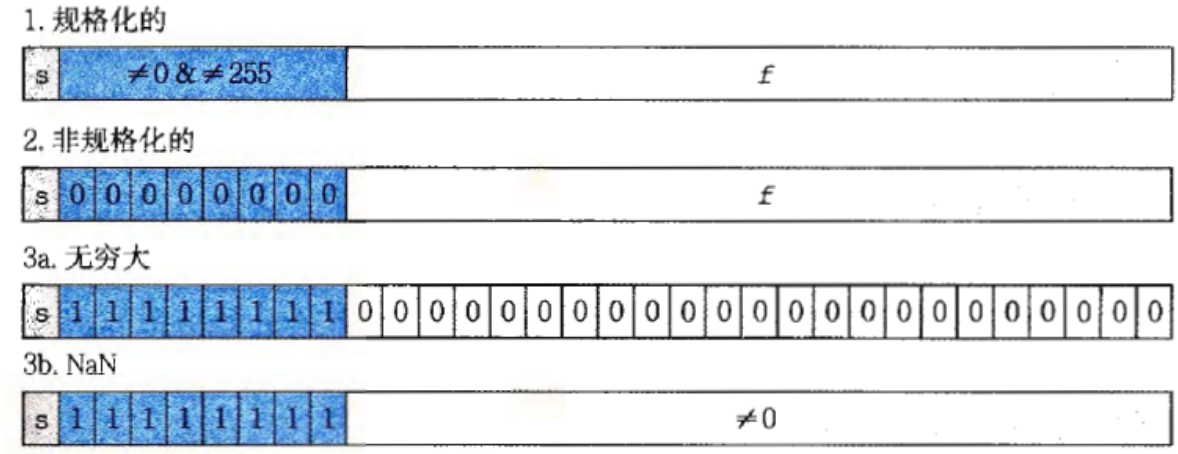
Chap2.2 浮点数表示和运算

IEEE754标准

表达形式

$N = (-1)^s \times M \times 2^E$，符号+尾数+阶码

- 单精度: 1+8+23
- 双精度: 1+11+52



规格化

$E = e - Bias, Bias = 2^{k-1} - 1$ (e.g. double: -1022~1023)

尾数隐含1开头，$M = 1 + f$

非规格化

$E = 1 - Bias, M = f$，不隐含1开头

用于表示接近0的数

1-Bias用于平滑过渡

描述	位表示	指数			小数		值		
		e	E	2^E	f	M	$2^E \times M$	V	十进制
0	0 0000 000	0	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{0}{8}$	$\frac{0}{512}$	0	0.0
最小的非规格化数	0 0000 001	0	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$	$\frac{1}{512}$	0.001953
	0 0000 010	0	−6	$\frac{1}{64}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$	$\frac{1}{256}$	0.003906
	0 0000 011	0	−6	$\frac{1}{64}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$	$\frac{3}{512}$	0.005859
	⋮								
最大的非规格化数	0 0000 111	0	−6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0.013672
最小的规格化数	0 0001 000	1	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0.015625
	0 0001 001	1	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$	$\frac{9}{512}$	0.017578
	⋮								
	0 0110 110	6	−1	$\frac{1}{2}$	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$	$\frac{7}{8}$	0.875
	0 0110 111	6	−1	$\frac{1}{2}$	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$	$\frac{15}{16}$	0.9375
1	0 0111 000	7	0	1	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	1	1.0
	0 0111 001	7	0	1	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	1.125
	0 0111 010	7	0	1	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$	$\frac{5}{4}$	1.25
	⋮								
	0 1110 110	14	7	128	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{1792}{8}$	224	224.0
最大的规格化数	0 1110 111	14	7	128	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240.0
无穷大	0 1111 000	—	—	—	—	—	—	∞	—

图 2-35　8 位浮点格式的非负值示例($k=4$ 的阶码位的和 $n=3$ 的小数位。偏置量是 7)

note: 按照字典序排列 (阶码不用补码的原因)。比较大小注意符号位，-0=0以及NaN

描　述	exp	frac	单精度		双精度	
			值	十进制	值	十进制
0	00 ··· 00	0 ··· 00	0	0.0	0	0.0
最小非规格化数	00 ··· 00	0 ··· 01	$2^{-23} \times 2^{-126}$	1.4×10^{-45}	$2^{-52} \times 2^{-1022}$	4.9×10^{-324}
最大非规格化数	00 ··· 00	1 ··· 11	$(1-\varepsilon) \times 2^{-126}$	1.2×10^{-38}	$(1-\varepsilon) \times 2^{-1022}$	2.2×10^{-308}
最小规格化数	00 ··· 01	0 ··· 00	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
1	01 ··· 11	0 ··· 00	1×2^{0}	1.0	1×2^{0}	1.0
最大规格化数	11 ··· 10	1 ··· 11	$(2-\varepsilon) \times 2^{127}$	3.4×10^{38}	$(2-\varepsilon) \times 2^{1023}$	1.8×10^{308}

Example

练习把一些整数值转换成浮点形式对理解浮点表示很有用。例如，在图 2-15 中我们看到 12 345 具有二进制表示[11000000111001]。通过将二进制小数点左移 13 位，我们创建这个数的一个规格化表示，得到 $12345 = 1.1000000111001_2 \times 2^{13}$。为了用 IEEE 单精度形式来编码，我们丢弃开头的 1，并且在末尾增加 10 个 0，来构造小数字段，得到二进制表示[10000001110010000000000]。为了构造阶码字段，我们用 13 加上偏置量 127，得到 140，其二进制表示为[10001100]。加上符号位 0，我们就得到二进制的浮点表示

加法

性质：不可结合，加法乘法满足单调性 (a>b, 则a+x>b+x)，a*a>=0 (a!=NaN)

1. 对阶

   小阶对大阶，小数点左移，此时阶码不会溢出

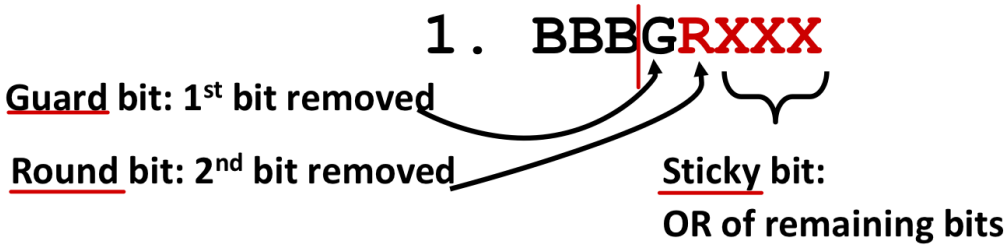
2. 尾数加减
3. 规格化
4. 舍入
5. 检查溢出

e.g. single 加法，阶数>=25直接取大的（24可能需要舍入）

舍入

相似地，向偶数舍入法能够运用在二进制小数上。我们将最低有效位的值 0 认为是偶数，值 1 认为是奇数。一般来说，只有对形如 $XX\cdots X.YY\cdots Y100\cdots$ 的二进制位模式的数，这种舍入方式才有效，其中 X 和 Y 表示任意位值，最右边的 Y 是要被舍入的位置。只有这种位模式表示在两个可能的结果正中间的值。例如，考虑舍入值到最近的四分之一的问题(也就是二进制小数点右边 2 位)。我们将 $10.00011_2\left(2\frac{3}{32}\right)$ 向下舍入到 $10.00_2(2)$，$10.00110_2\left(2\frac{3}{16}\right)$ 向上舍入到 $10.01_2\left(2\frac{1}{4}\right)$，因为这些值不是两个可能值的正中间值。我们将 $10.11100_2\left(2\frac{7}{8}\right)$ 向上舍入成 $11.00_2(3)$，而 $10.10100_2\left(2\frac{5}{8}\right)$ 向下舍入成 $10.10_2\left(2\frac{1}{2}\right)$，因为这些值是两个可能值的中间值，并且我们倾向于使最低有效位为零。

GRS: 以100为界



- Round up conditions

Value	*Fraction*	*GRS*	*Incr?*	*Rounded*
128	1.000**0000**	000	N	1.000
15	1.101**0000**	000	N	1.101
17	1.000**1000**	100	N	1.000
19	1.001**1000**	100	Y	1.010
138	1.000**1010**	101	Y	1.001
63	1.111**1100**	110	Y	10.000

类型转换

- int ⇒ single: 可能舍入，不会溢出
- int ⇒ double: 保留精度
- double ⇒ int: 向零舍入，可能溢出

Chap3 指令集架构(ISA)

Chap3.1 ISA简介

word: 32位 (in MIPS)

用户级ISA VS 系统级ISA (e.g. HLT停机)

小端 VS 大端

数据对齐: 任何K字节的基本对象的地址必须是K的倍数

CISC

- 编译简单，程序大小较小
- Register-Memory架构，运算指令可以访存
- 指令繁多，不定长

RISC

- 简单，指令总类少

  可能需要更多的指令完成相同操作
- Load-Store架构，用于数据传送
- 运算指令使用立即数或寄存器
- 便于流水化

RISC V

- 通用ISA，面向并行
- 架构简单
- 方便微体系结构实现
- 开源

Chap3.2 MIPS指令集

寄存器

编号	名称	用途	编号	名称	用途
0	$zero	常量0 (The Constant Value 0)	24-25	$t8-$t9	临时变量 Temporaries
1	$at	汇编器临时变量	26-27	$k0-$k1	为操作系统内核保留
2-3	$v0-$v1	函数返回结果、表达式值	28*	$gp	全局指针 Global Pointer
4-7	$a0-$a3	函数调用时传递的参数	29*	$sp	栈指针 Stack Pointer
8-15	$t0-$t7	临时变量 Temporaries	30*	$fp	结构指针/桢指针 Frame Pointer
16-23*	$s0-$s7	需要保存的临时变量	31*	$ra	返回地址 Return Address

•* 在过程调用时需要保存

常用指令

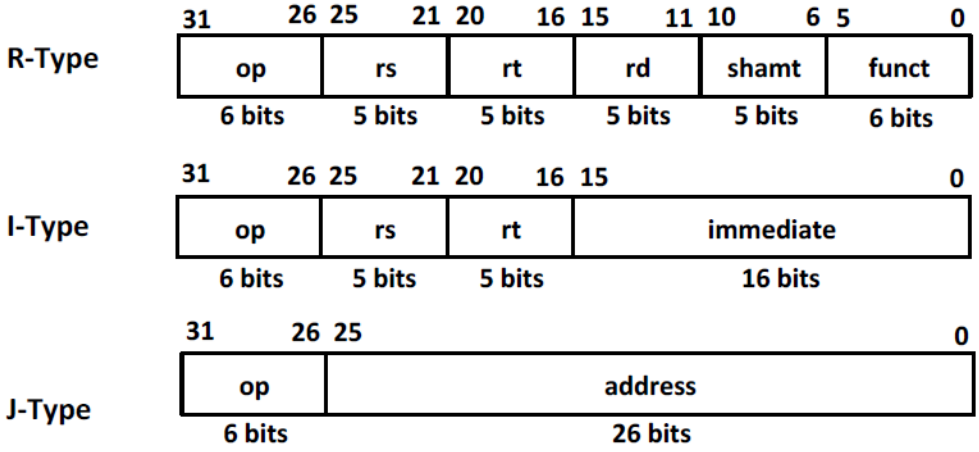
见`mips-ref-sheet.pdf`

```
add     $s0,$s1,$s2     # s0=s1+s2
add     $s0,$s1,$s2     # s0=s1-s2
addi    $s0,$s1,10      # s0=s1+10
lw      $s0,12($s1)     # s0=[$s1+12]    需要对齐
sw      $s0,12($s1)     #
lb/sb   # sign-extend

and or not sll srl/sra

slt/slti
beq     # jump if equal
```

```
bne     # jump if not equal
j
# example
slt     $t0,$s0,$s1     # if (a<b)
bne     $t0,$zero,xxx
# example
slt     $t0,$s1,$s0     # s0<=s1 -> !(s1<s0)
slti    $t0,$s0,2       # s0>1   -> !(s0<2)
```

```
# while (a[i]==k) i++;
Loop:
        sll     $t1, $s3, 2     # *4
        add     $t1, $t1, $s6   # calc addr
        lw      $t0, 0($t1)     # load a[i]
        bne     $t0, $s5, Exit
        addi    $s3, $s3, 1     # i++
        j       Loop
```

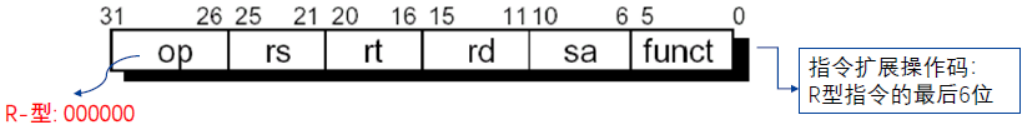
指令格式



I28~I26 / I31~I29	000	001	010	011	100	101	110	111
000	R 格式	Bltz/gez	j	jal	beq	bne	blez	bgtz
001	addi	addiu	slti	sltiu	andi	ori	xori	lui
010	TLB 指令	浮点指令						
011								
100	lb	lh	lwl	lw	lbu	lhu	lwr	
101	sb	sh	swl	sw			swr	
110	lwc0	lwc1						
111	swc0	swc1						

R指令需要结合`funct`继续解码

I5~I3 \ I2~I0	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srlv	srav
001	j $r	jalr			syscall	break		
010	mfhi	mthi	mflo	mtlo				
011	mult	multu	div	divu				
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								



R型

```
and     $t0,$t1,$t2
#   op  rs  rt  rd  shamt   funct
#    0  t1  t2  t0      0   and
sll     $t0,$t2,8
#   op  rs  rt  rd  shamt   funct
#    0  0   t2  t0      8   sll
#   shamt 有5位，最多可以移动31位
```

I型

```
sw      $t0,4($s3)
#   地址: 32位寄存器 + 16位偏移量 (补码,可以负数)
slti    $t1,$t2,15
#   imm 保存立即数 (零扩展/符号扩展)
bne     $t0, $s5, Exit
#   保存offset
#   branch addr = PC + 4 + sext(imm) * 4
#   跳转范围 -2^15 ~ 2^15-1 (word)
#   jump further
bne     x,x,NotJ
j       xxx
NotJ:   xxx
```

J型

```
j       label
#   26位addr
#   jump addr = {(PC+4)[31:28],addr,00}
#   256 MB = 2^28 B
lui     $at,xxx         # {upper 16 bits of Foo}
ori     $at,$at,xxx     # {lower 16 bits of Foo}
jr      $at             # jump further
```

函数调用

```
jal     func addr       # call, write PC+4 to $ra
jr      $ra             # ret
```

过程执行的六个步骤

1. 主程序（调用者）将参数放置在过程（被调用者）可以访问到的位置
   - $a0 - $a3: 四个参数寄存器
2. 调用程序将系统控制权转移给被调用过程（jal）
3. 被调用过程申请所需的存储资源
4. 被调用过程执行相应的任务
5. 被调用过程将执行结果存放在主程序可以访问的位置
   - $v0 - $v1: 两个结果值寄存器
6. 被调用过程将系统控制权移交给调用程序(jr)
   - $ra: 返回地址寄存器

多余的参数用栈(stack)传递

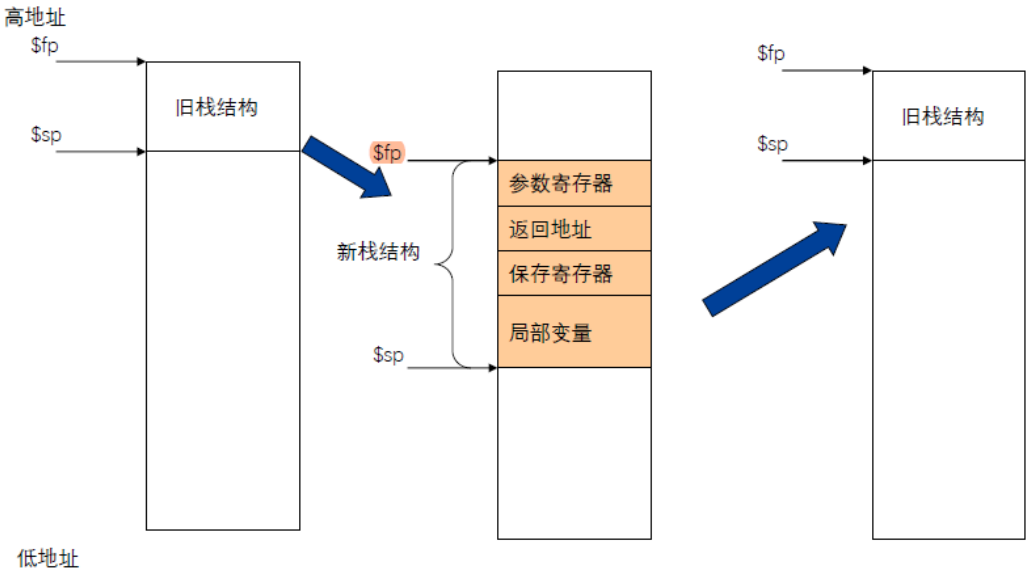
```
addi    $sp,$sp,-4
sw      $s0,0($sp)
#   ...
lw      $s0,0($sp)
addi    $sp,$sp,4
```

- 过程调用里保存

  `$ra` , `$sp` , `$gp` , `$fp` , `$s0 ~ $s7` (store), `$a0 ~ $a3`

- 过程调用里不保存

  `$v0` , `$v1` , `$t0 ~ $t9`



伪指令

```
move    dst,src         # add dst,src,$zero
# get translated into actual instructions later
li      $t0,0xABABCDCD  # lui + ori
la      dst,label       # load address
```

Chap4 处理器设计

Chap4.1 单周期处理器设计

~~实践出真知，后六周实验的好处~~

设计处理器的五个步骤：

1. 分析指令系统得出对数据通路的需求
2. 选择数据通路上合适的组件
3. 连接组件构成数据通路
4. 分析每一条指令的实现以确定控制信号
5. 集成控制信号完成控制逻辑

单周期处理器

时钟周期受限于最慢的一条指令

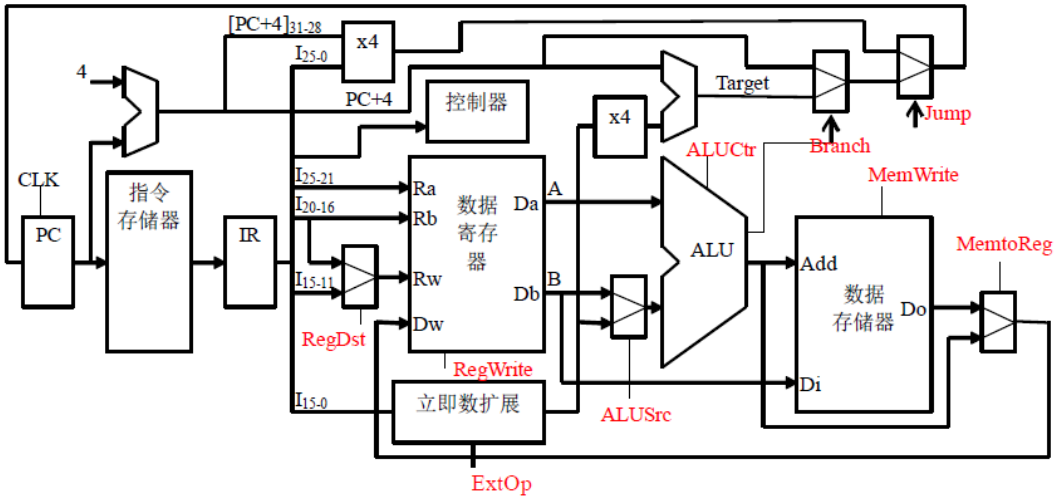
功能部件(e.g. 加法器)每个周期只能使用一次

寄存器/存储器：在时钟边沿来到，写允许信号有效时才更新状态

PC+4由clock控制

note: 一般`load`最慢，`jump`最快

支持`add`,`ori`,`lw`,`sw`,`beq`,`j`，ALUCtr结合`funct`继续解码R指令

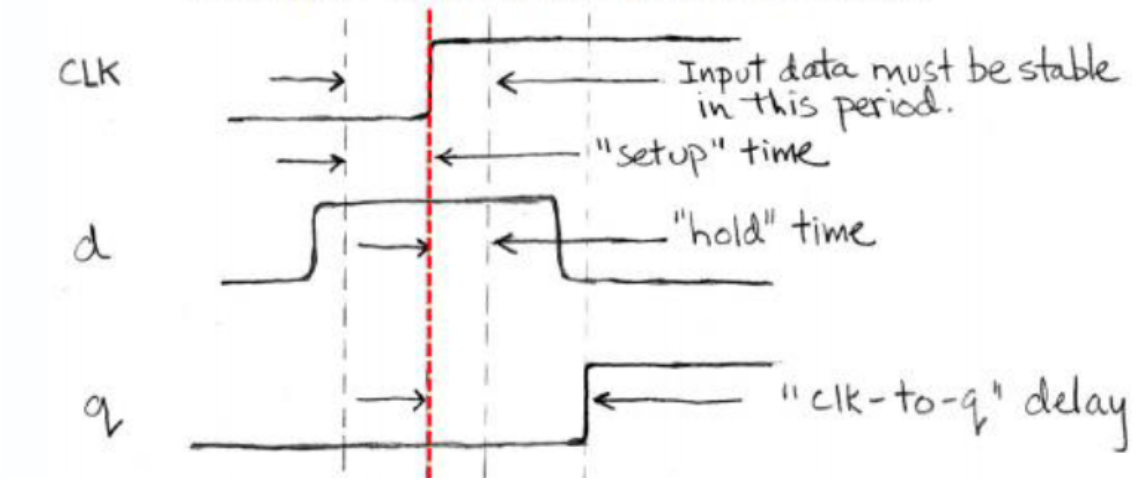


Chap4.2 流水线处理器

数字逻辑基础

D触发器

上升沿触发



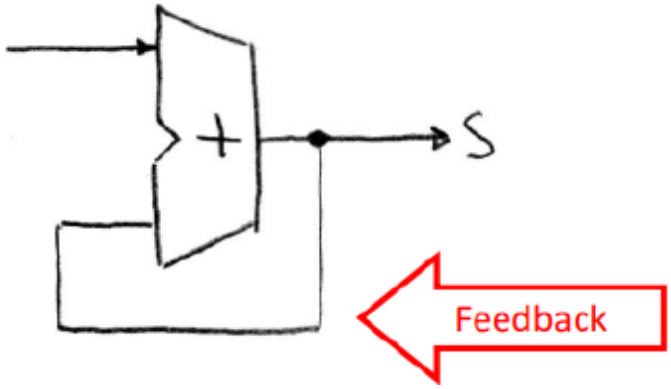
setup time: 输入在上升沿前必须稳定的时间

hold time: 输入在上升沿后必须稳定的时间

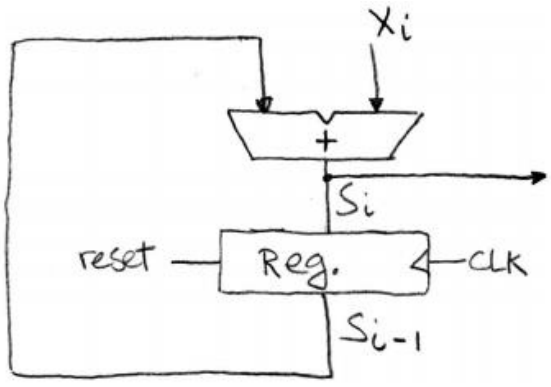
clk-to-q delay: 输出变化需要的时间

加法器

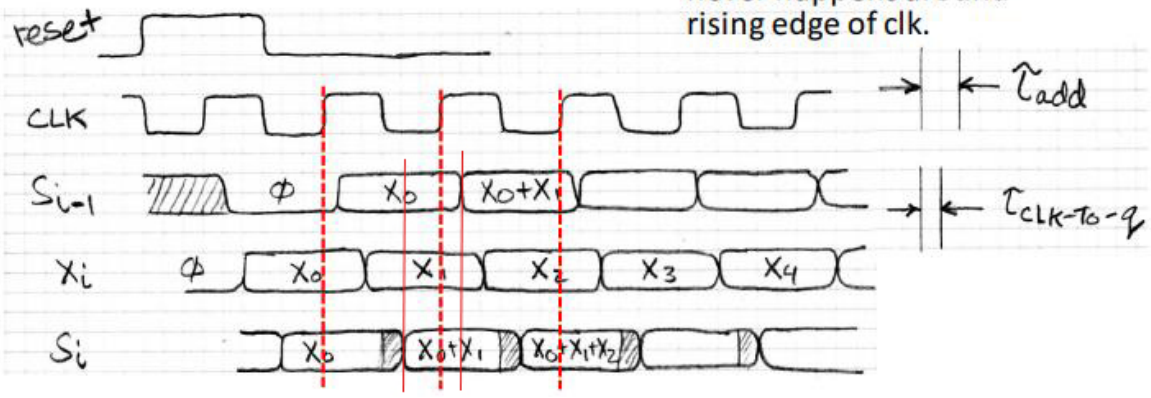
错误的累加器：无法控制下一次迭代，无法置0



正确的累加器



- reset signal shown.
- Also, in practice X might not arrive to the adder at the same time as S_{i-1}
- S_i temporarily is wrong, but register always captures correct value.
- In good circuits, instability never happens around rising edge of clk.



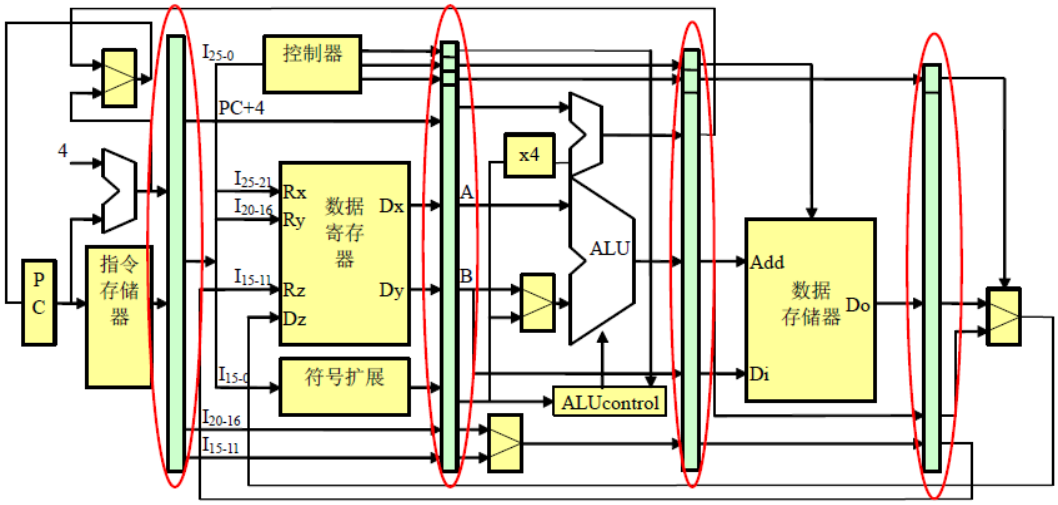
Max Delay = clk-to-q + combinational logic + setup time

五阶段流水

`IF`,`ID`,`EXE`,`MEM`,`WB`

note: IF阶段更新PC; 寄存器写入地址需要传递到WB阶段

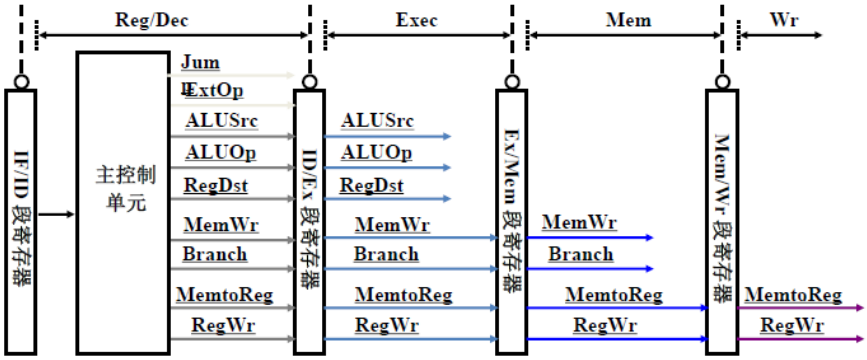
MIPS：指令等长，指令格式少，只有sw/lw访存，最多一次写回



$$t_{clkpipe} >= \max \begin{cases} t_{clk\text{-}to\text{-}q} + t_{MEMread} + t_{setup} \text{ (Fetch)} \\ t_{clk\text{-}to\text{-}q} + t_{RFread} + t_{setup} \text{ (Decode)} \\ t_{clk\text{-}to\text{-}q} + t_{ALU} + t_{mux} + t_{setup} \text{ (Execute)} \\ t_{clk\text{-}to\text{-}q} + t_{MEMread} + t_{setup} \text{ (Memory)} \\ t_{clk\text{-}to\text{-}q} + t_{mux} + t_{RFsetup} \text{ (Writeback)} \end{cases}$$

控制信号传递

主控制单元在译码段（Reg/Dec）产生所有控制信号
- Exec 段需要的控制信号，在一周期后使用
- Mem 段需要的控制信号，在两周期后使用
- Wr 段需要的控制信号，在三周期后使用

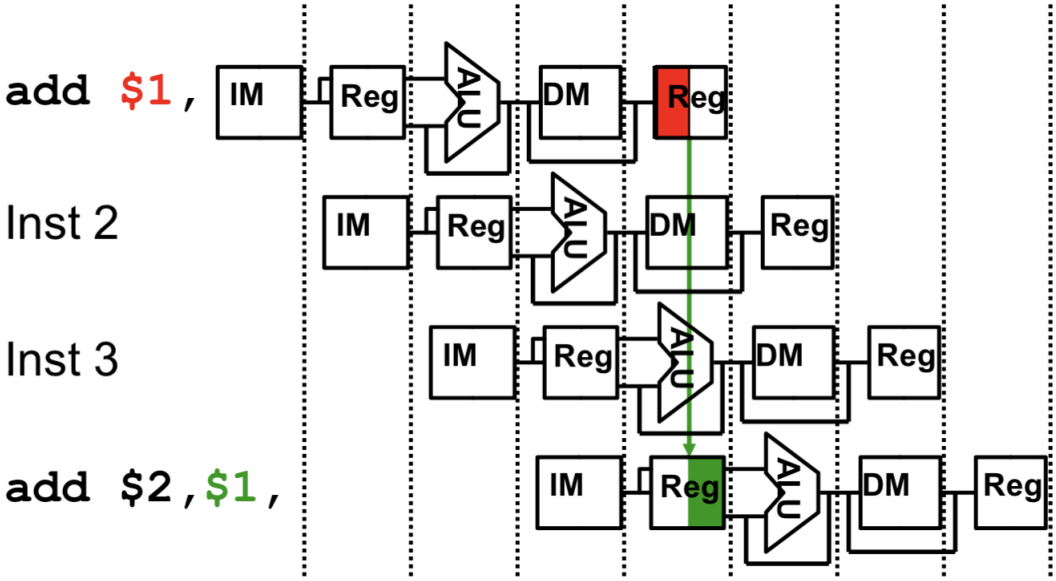


相关性和冒险

结构冒险

所需的硬件部件正在为之前的指令工作

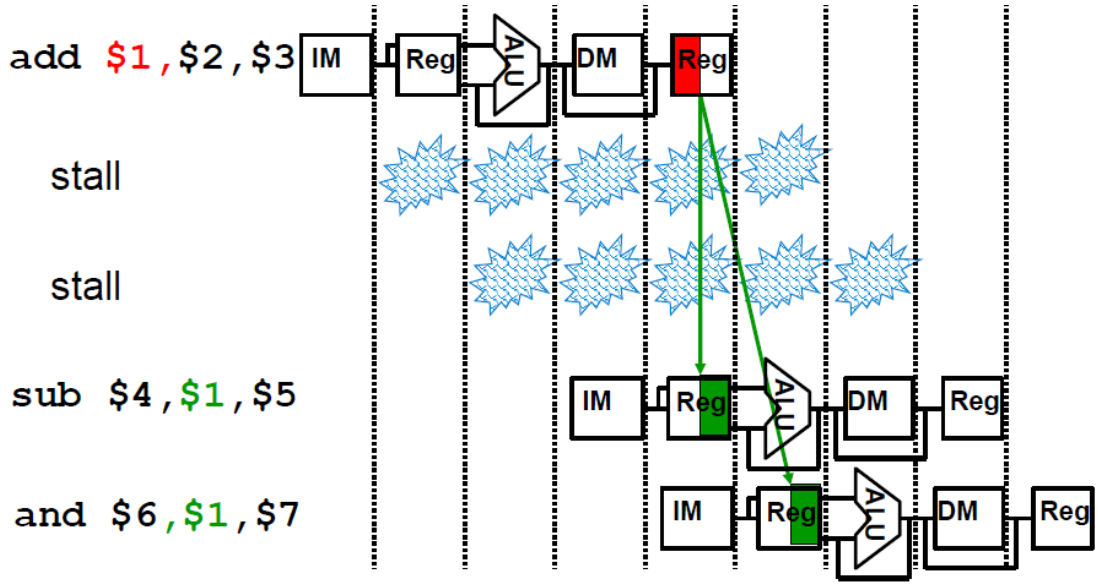
1. 指令内存和数据内存分离
2. 前半周期写，后半周期读



数据冒险

RAW，Load - Use（均需要停顿2个周期）

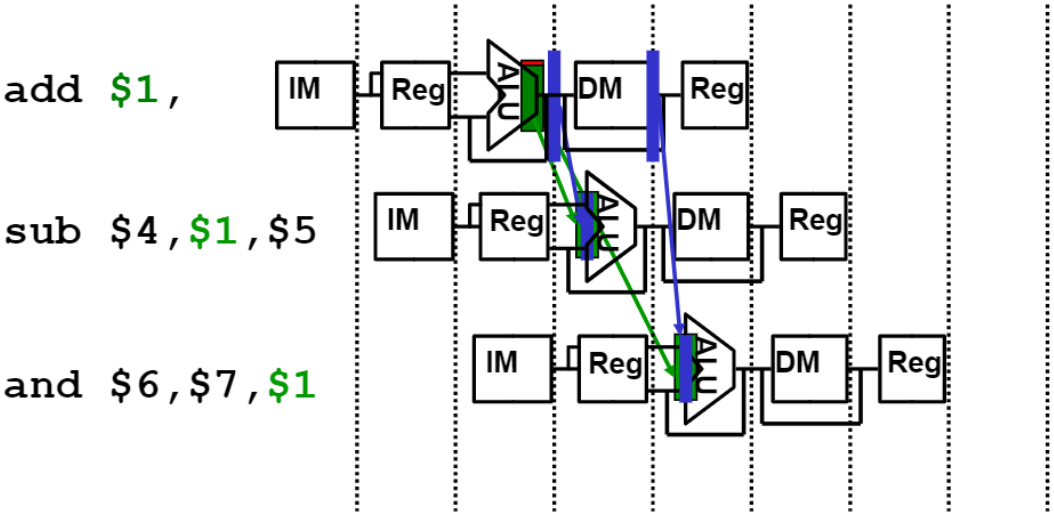
1. 插入空指令(nop)
2. 停顿流水线



3. 前向传递

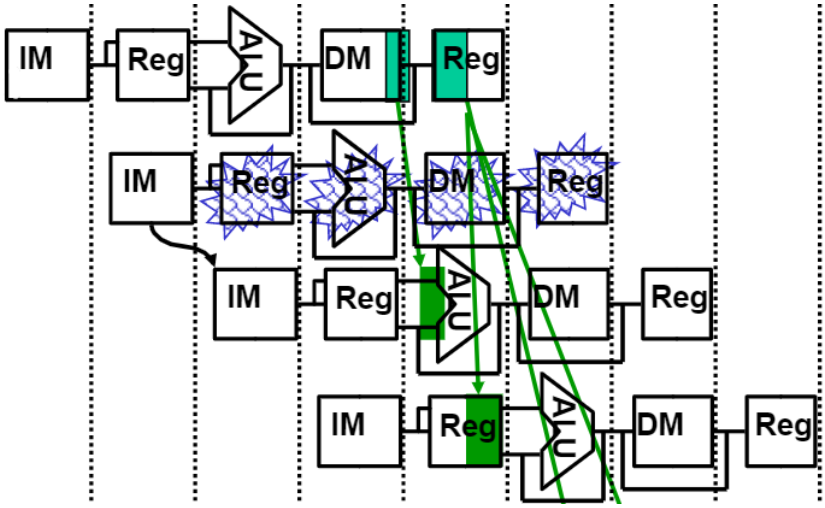
   将产生的结果直接传送到当前周期需要结果的功能单元

   - EXE到下一个EXE



- MEM到下一个EXE

  仍需要一个周期的停顿

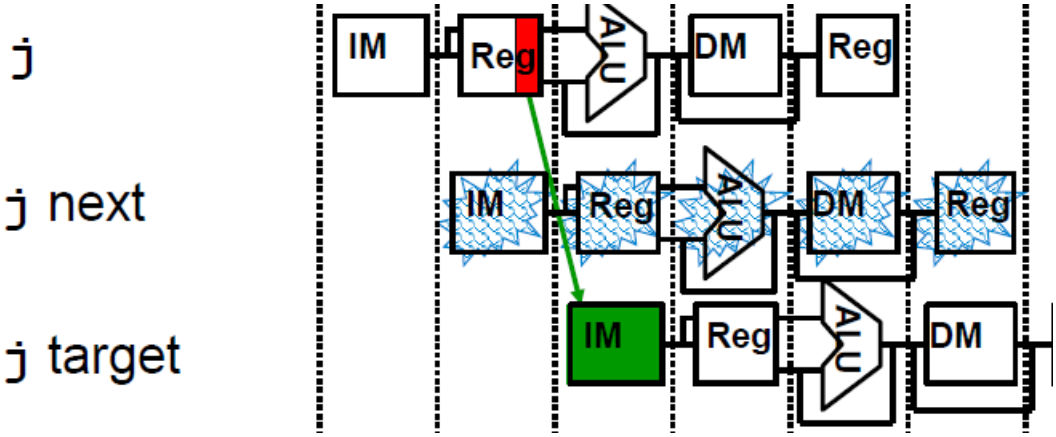


4. 指令重排

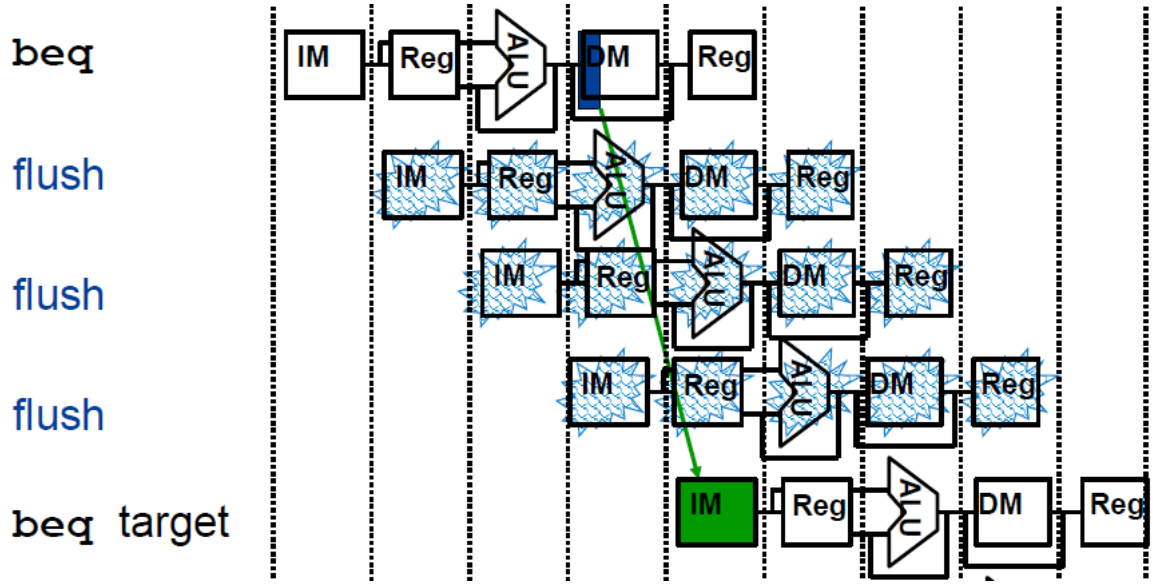
控制冒险

由转移指令引起

jump需要flush 1条，新的PC在ID后可以计算出



branch需要flush 3条，新的PC在EXE后才能计算出



流水线越长，分支开销越大

1. 分支提前决策
    - 增加硬件

      在ID阶段增加硬件，提前计算出branch指令的结果

      branch指令只需要flush 1条
    - 分支预测
2. 转移延迟槽

   编译器支持，确保本应该flush的指令是必须执行的 (nop)

   ```
   # example
   add     xx,xx,xx        # must execute
   j       label
   # reorder
   j       label
   add     xx,xx,xx        # do not flush this
   ```

   转移延迟槽的大小取决于体系结构

   note: 五阶段无提前决策，jump指令为1，branch指令为3

精准中断

MIPS中异常寄存器

EPC：存放发生异常的指令地址

Cause：保存异常的原因

STATUS：interrupt mask以及控制CPU状态

处理异常

中断处理

1. 设置 EPC
2. 设置 STATUS 寄存器，进入kernel态，禁止中断响应
3. 设置 Cause 寄存器
4. CPU 开始从一个统一入口取指令

中断返回

1. 执行指令`eret`会将 EPC 中的内容移入 PC 并设置 STATUS 寄存器，开中断，CPU 进入用户态
2. 重新执行被异常事件中断的那条指令。

精准中断

可能同时在不同阶段发生多个中断

精准中断：传递中断信息，在指令执行周期的最后一个阶段增加一个“检查中断”的操作，以响应中断

同一条指令仅保存最先发生的异常

动态转移预测

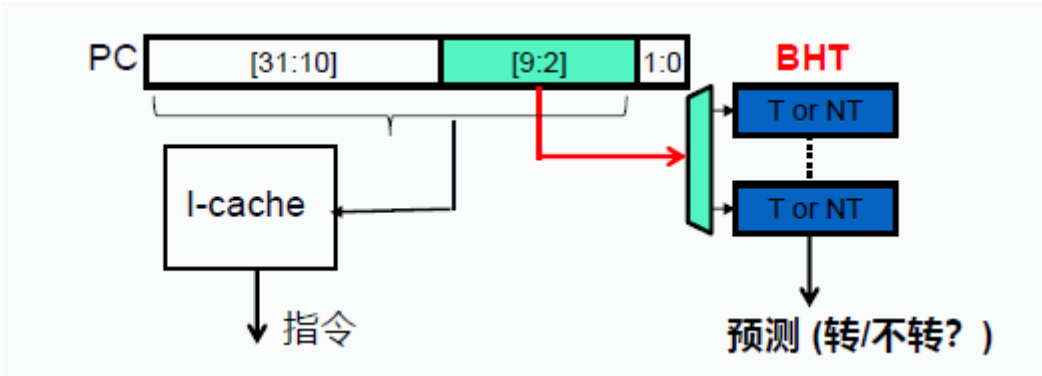
仅要求掌握概念

局部相关：和自己的历史有关

全局相关：和其它语句有关 (e.g. i>0 ⇒ !i<=0)

预测转移目标

- 转移历史表 (BHT)



  一边取指一边查表，索引位重合只是降低准确率

- 两位预测器
- Gshare预测器

  根据指令和历史预测

  f: inst x {N,T}^k ⟶ {N,T}
- 锦标赛预测器

  综合多个预测器

预测转移目标

- Branch target buffer (BTB)

  记住每次PC和jump addr

  可在BTB中加入return标记，记录栈中的程序返回地址

Chap4.3 现代处理器

仅要求掌握概念

多发射、超标量处理器

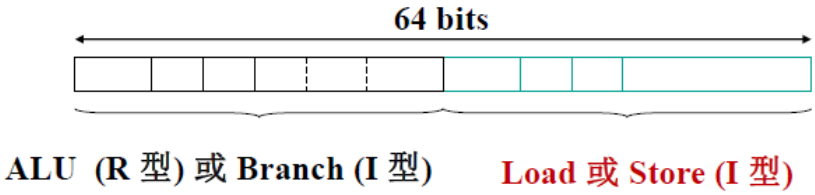
指令级并行 (ILP)：深度流水，超标量，乱序执行等

超标量：每个周期发射和执行多条指令

静态多发射

超长指令字 (VLIW)，由编译器生成

双发射MIPS处理器，两条指令(nop)组成指令束，硬件改动小



缺点：编译复杂，代码膨胀，目标代码不兼容

动态多发射

CPU每次发射若干条指令

运行时检测并处理冒险

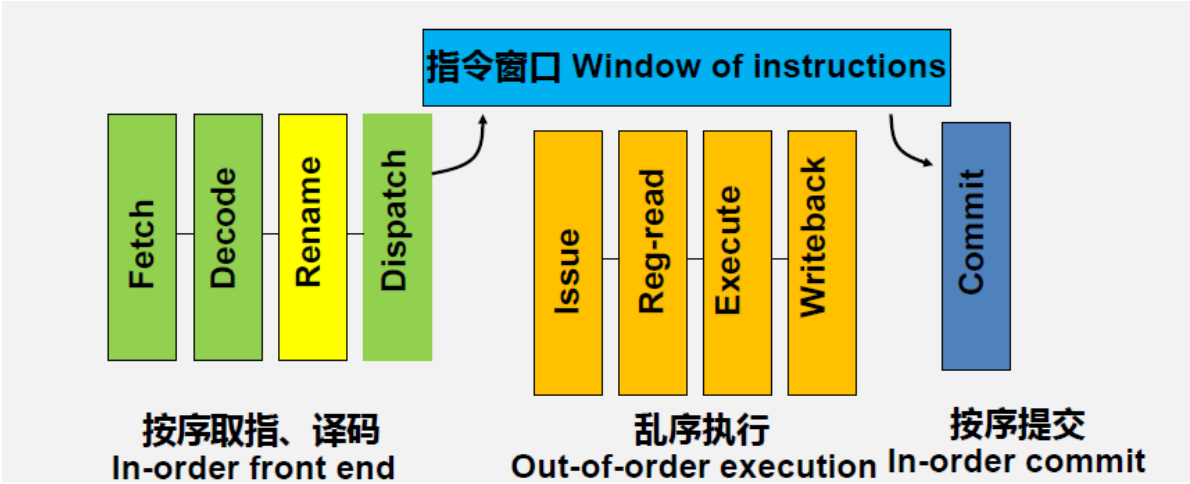
乱序执行

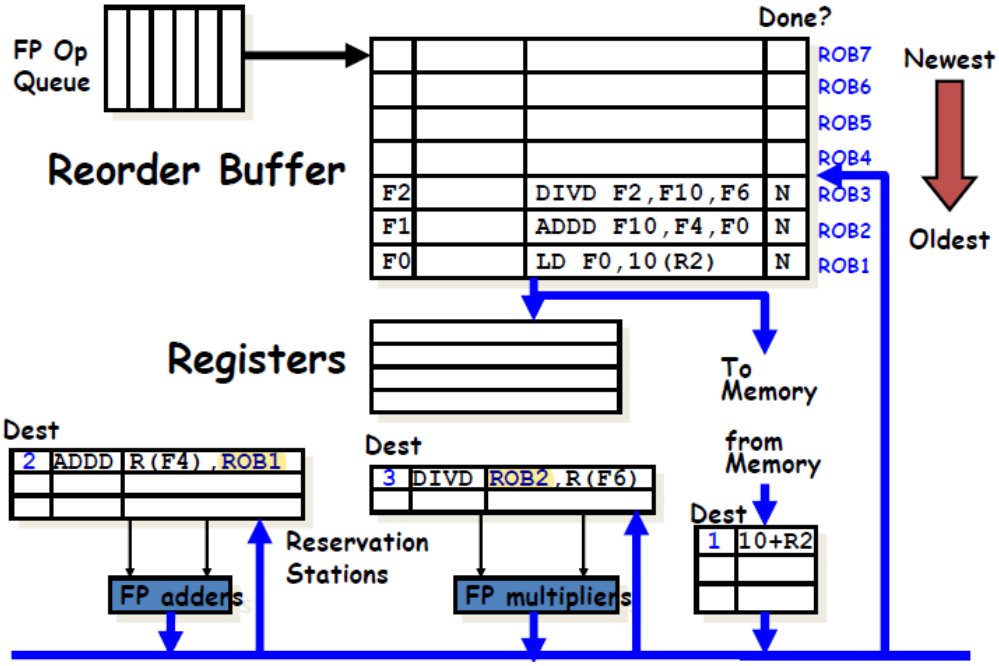
名字引起假相关

寄存器重命名以解决WAW(输出相关)和WAR(反相关)

```
#   WAW
lw      $t0(a),xxx
add     $t0(b),xx,xx
sw      $t0(b),xx           # need result of add, not lw
#   WAR
add     xx,xx,$t0(a)
sub     $t0(b),xx,xx        # must read first
```

真相关：RAW





Tomasulo算法无法解决memory hazard (lw after sw)

多线程处理器

ILP局限性：程序的相关性，大量复杂的硬件资源

线程级并行 (TLP)

交叉执行不同线程的指令以消除数据依赖

多线程开销

- 每个线程需要保存：PC, system state等
- OS调度，TLB miss等

粗粒度多线程

当线程运行时存在较长时间延时时切换到另一线程 (e.g. cache失效)

同一线程指令间的较短延时不切换

`fig3` 所示，减少垂直浪费

细粒度多线程

多个线程的指令交叉执行

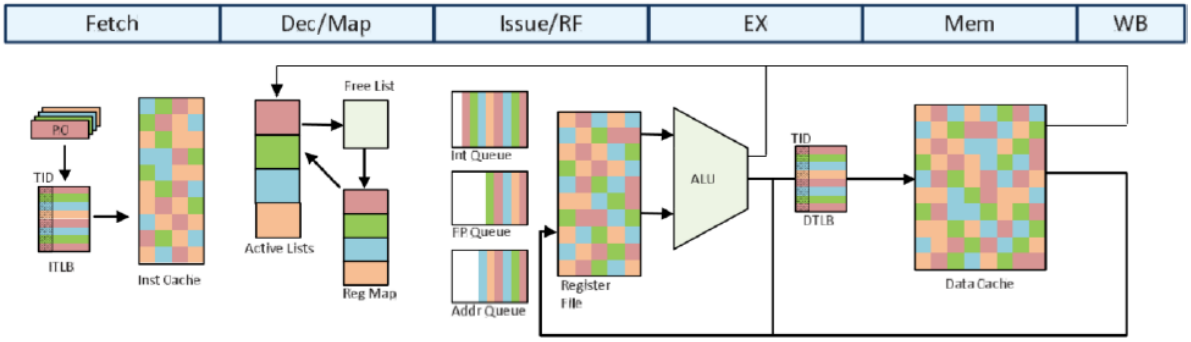
`fig2` 所示，减少甚至消除垂直浪费

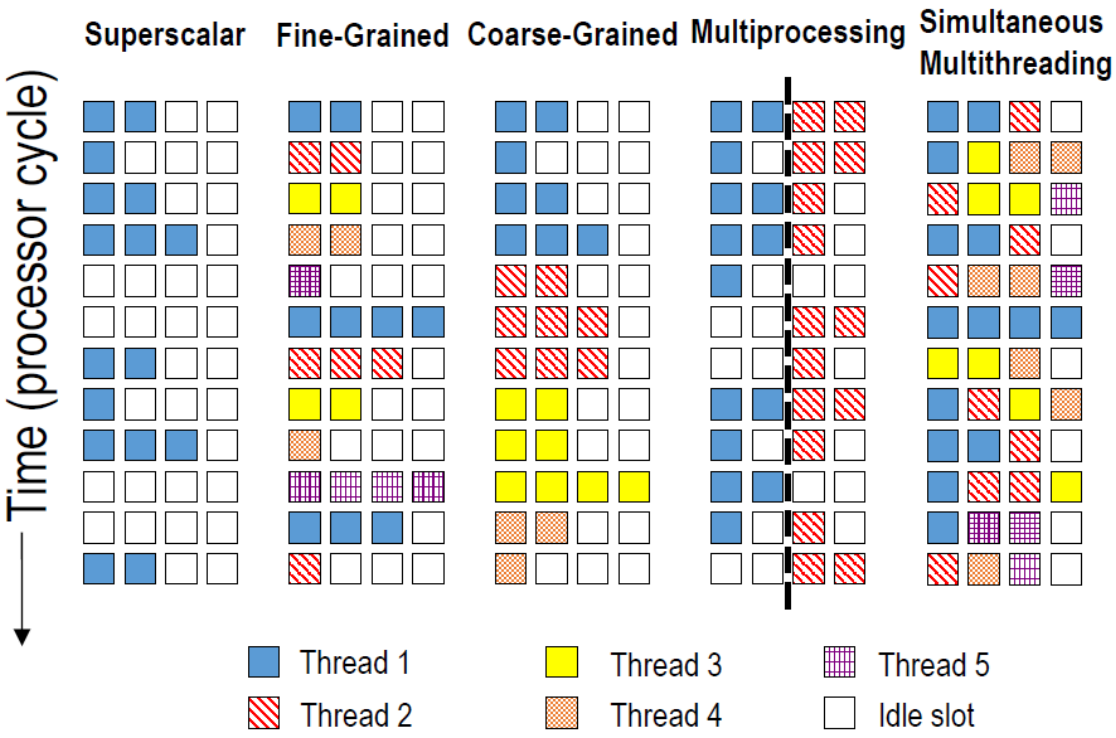
同时多线程

在相同时钟周期运行多个线程的指令

乱序超标量处理器 + 可以从多个线程取指令

`fig5` 所示，消除水平浪费





开发程序中的ILP

1. 基本优化
    - 把同一个值放入变量，避免多次计算
    - 结果保存在临时变量，避免每次写入
2. 循环展开，改变计算顺序/独立计算(分s0,s1)
    - 指令增多，增加指令级并行性以填充流水线
    - 转移指令减少
    - 寄存器换名以消除假相关性

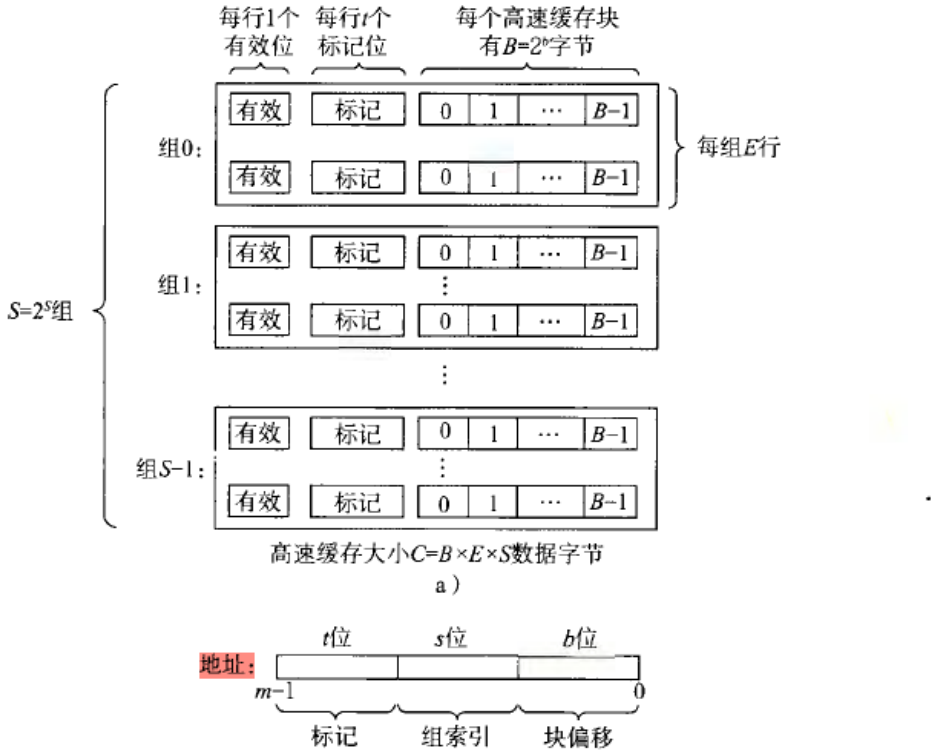
缺点：代码膨胀，易读性下降

Chap5 高速缓存

局部性：时间局部性，空间局部性

局部性体现：数据访问，指令访问

Chap5.1 基本概念



块/行(block)：cache以块为内存最小单位

命中率：$h = hit/total$

平均访问时间：$t = ht_c + (1-h)t_m$

关键问题

- 数据查找
- 地址映射：决定把block放在哪里
- 替换策略：替换掉哪个block
- 写入策略：保证cache和memory一致

地址映射方式

valid bit: 数据是否有效（上下文切换后清空valid bit)

直接映射

每组只有一行

优点：地址变换速度快，一对一映射；替换算法简单、容易实现

缺点：容易冲突，cache利用率低；命中率低

全相联

一个包含所有高速缓存行的组

优点：一对多映射，cache全部装满后才会出现块冲突；块冲突的概率低，cache利用率高

缺点：硬件开销增加，替换算法复杂

组相联

每个组都保存有多千一个的高速缓存行（4路组相联：每组4个路/行）

关联度：组内行数

降低关联度：减少一组中的路数、增加组数

增加关联度：增加一组中的路数、减少组数

总结

- 不命中类型

  强制不命中，冲突不命中，容量不命中
- cache的容量
  - 小：采用组相联映射或全相联映射
  - 大：采用直接映射方式，查找速度快，命中率相对前者稍低
- cache 的访问速度
  - 要求高的场合采用直接映射
  - 要求低的场合采用组相联或全相联映射

Chap5.2 缓存相关策略

更新策略

dirty bit: 表示是否修改

写直达法 (write-through)

修改内容直接写入内存

特点：控制逻辑简单，时间更容易预测，高可靠性

写回法 (write-back)

修改内容先写入cache，等被替换时再更新

特点：控制逻辑复杂，每次不命中时间不一致，减少写次数，缺乏可靠性

cache失效时的写策略

写分配

miss后数据载入缓存，在缓存中更新

特点：连续访问较优，写miss会先导致读miss（读miss后在缓存中分配一行后写hit）

不按写分配

miss后直接修改内存的内容

特点：不影响缓存中的数据，仅在第一级设备被修改

总结：写直达+不按写分配；写回+按写分配

替换策略

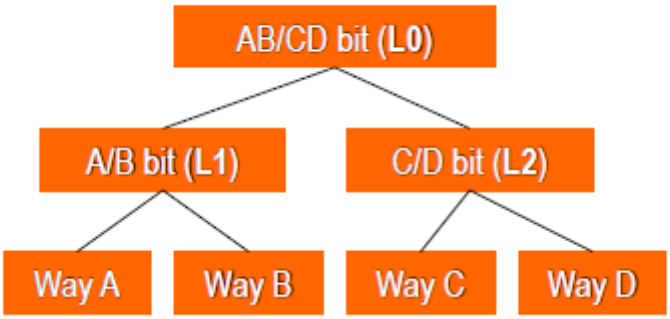
LRU算法

$O(n \log n)$bits来表示n!种情况

颠簸(trashing)：循环访问（1234512345...容量为4）

伪LRU

$O(n)$bits，减少硬件代价



Chap5.3 其它

虚拟地址与TLB

VPN1 | VPN2 | VPO $\longrightarrow$ PPN | PPO

offset: VPO=PPO

page number: 通过页表查询

TLB：加快VPN到PPN的翻译

VPO长度大于组行+块偏移，翻译虚拟地址与cache查找同时进行

TLB 事件组合

TLB	Page Table	Cache	可能发生吗？在何种情况下？
Hit	Hit	Hit	是的 – 我们希望这种情况发生!
Hit	Hit	Miss	是的–TLB 命中，不需要访问页表，但数据没有调入缓存，只能从内存读取
Miss	Hit	Hit	是的 – TLB失效, 地址转换通过查页表获得
Miss	Hit	Miss	是的 – TLB失效, 地址转换通过查页表获得，但数据没有调入缓存，只能从内存读取
Miss	Miss	Miss	是的 – 页面失效
Hit	Miss	Miss/ Hit	不可能 – 如果这页没有调入内存，TLB不可能命中
Miss	Miss	Hit	不可能 –如果这页没有调入内存，就不可能存放在cache中

编写缓存友好的代码

- 关注内存循环
- 时间局部性

  重复使用变量
- 空间局部性

  优先访问相邻变量

```
// optimized matrix multiplication
for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
    {
        r = b[k][j];
        for (i = 0; i < n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

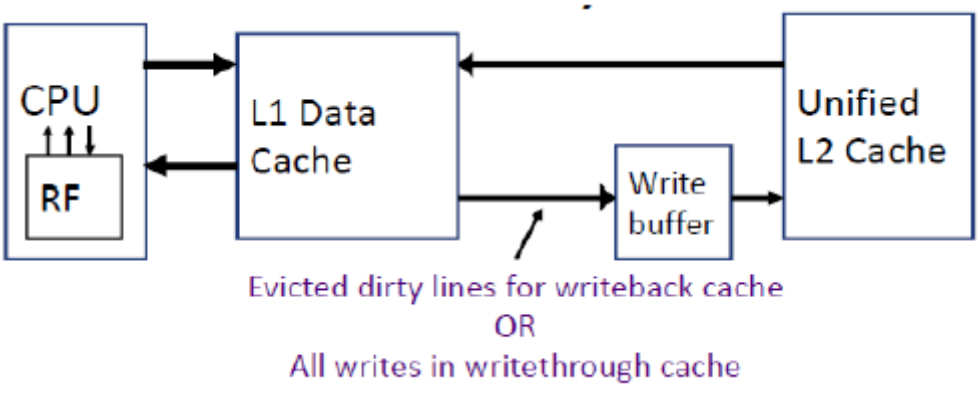
缓存设计

technique	Miss rate	Miss Penalty	Hit time	Complexity
large block size	☺	☹		☺
high associativity	☺		☹	☹
victim cache	☺	☺		☹
hardware prefetch	☺			☹
compiler prefetch	☺			☹
compiler optimizations	☺			☹
prioritisation of reads		☺		☹
critical word first		☺		☹
nonblocking caches		☺		☹
L2 caches		☺		☹
small and simple caches	☹		☺	☺
virtual-addressed caches			☺	☹

Impact of Cache Performance and Complexity

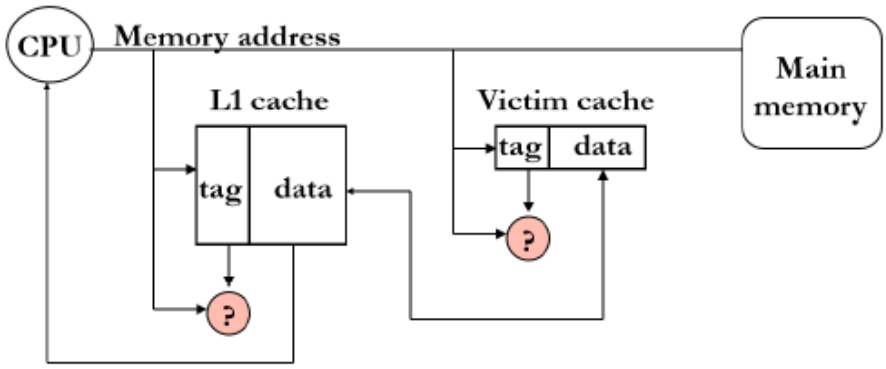
Technique	Hit time	Miss penalty	Miss rate	Hardware complexity
Larger block size		–	+	0
Larger cache size	–		+	1
Higher associativity	–		+	1
Multilevel caches		+		2
Read priority over writes		+		1
Avoiding address translation during cache indexing	+			1

- 减少hit时间
  - 小cache
  - 直接映射
  - 小block
  - write buffer



- 减少miss rate

    - 大cache
    - 增加关联度
    - 大block
    - victim cache：保存最近被替换出的cache
    - 预取：硬件预取；软件预取，编译器插入指令



- 减少miss惩罚
    - 小block
    - write buffer
    - victim cache
    - early restart：读到块中被用到部分时提前送入cache
    - critical word first：优先读入块中被用到的部分
    - 写优先于读
    - 非阻塞：在miss后也允许先hit而不是等待miss处理完成
    - 多级缓存

Chap6 并行与异构计算

Chap6.1 数据级并行 (DLP)

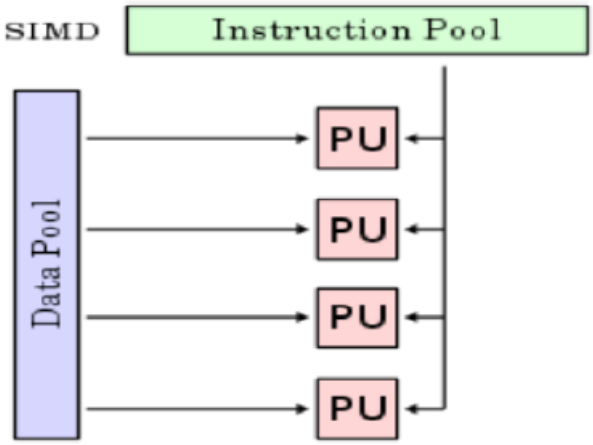
分类

SISD

单指令单数据（包括超标量处理器，因为模型是顺序的）

SIMD

单指令多数据



MIMD

多个处理器同时处理不同数据的不同指令（e.g. 多核处理器）

向量处理模型

两个向量对应分量做运算，产生结果向量

向量内的操作无关，从而可以使用简化控制的深度流水

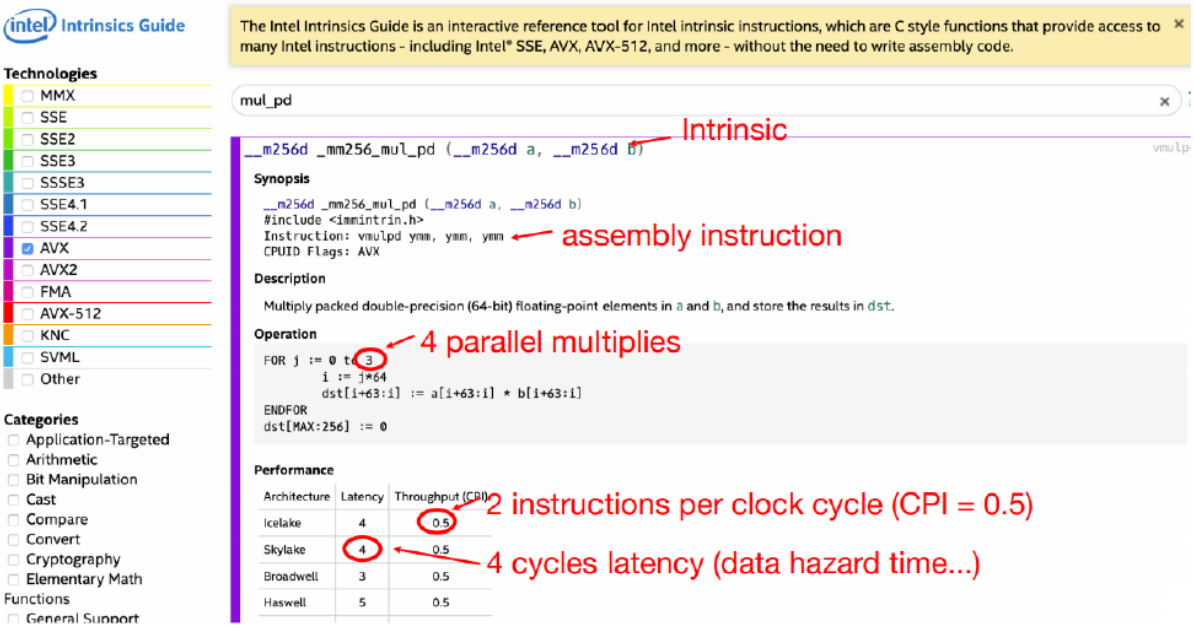
存储控制：流水读入数据

条件控制：使用mask向量寄存器

SIMD扩展

AVX (Advanced Vector Extension)

注：p(packed)表示封装，pd表示packed-double




```
// AVX example
// origin version
for(int i=0;i<n;++i)
    aa[i]=bb[i]>0?cc[i]+2:dd[i]+g;
// AVX version

__m128i a, b, c, d, mask, zero, two, g_broadcast;
// broadcase 16-bit integer to all elements
zero = _mm_set1_epi16(0);
two = _mm_set1_epi16(2);
g_broadcast = _mm_set1_epi16(g);
for (i = 0; i < SIZE; i += 8)
{
    b = _mm_load_si128(ToVectorAddress(bb[i]));
    c = _mm_load_si128(ToVectorAddress(cc[i]));
    d = _mm_load_si128(ToVectorAddress(dd[i]));
    c = _mm_add_epi16(c, two);
```

```
    d = _mm_add_epi16(d, g_broadcast);
    mask = _mm_cmpgt_epi16(b, zero);
    // use mask to eliminate ?:
    // each mask is 0xFFFF
    a = _mm_and_si128(c, mask);
    mask = _mm_andnot_epi16(mask, d); // (not a) and b
    a = _mm_or_si128(a, mask);
    _mm_store_si128(ToVectorAddress(aa[i]), a);
}
```

Chap6.2 线程级别并行 (TLP)

共享内存多核处理器

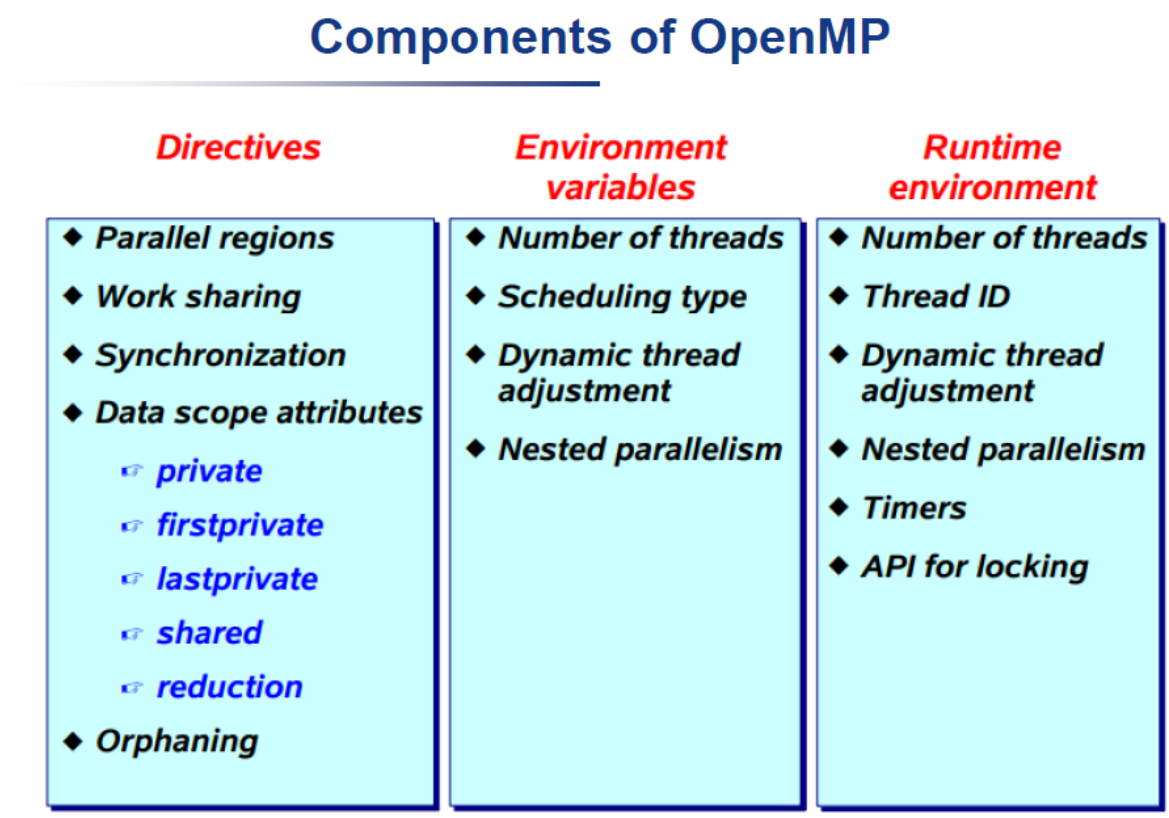
MIMD: 每个处理器有独立的指令流

通过共享内存通信

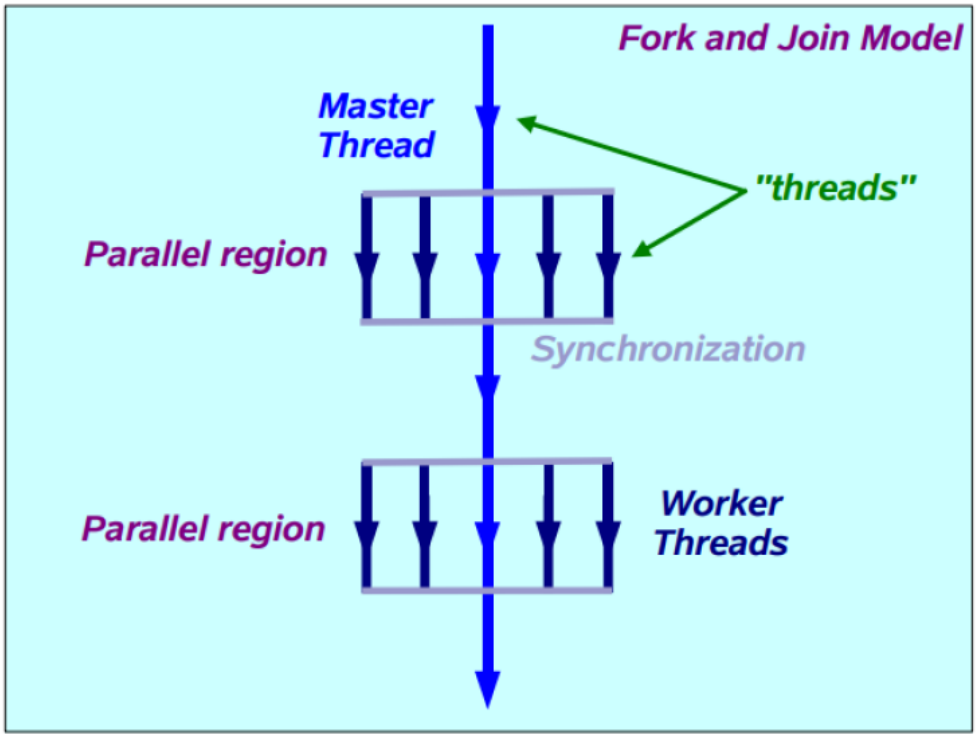
共享内存类型：对称多处理器 VS 分布式内存

共享内存多线程编程

OpenMP



特点：可移植，标准化，编译简单



更多见`OpenMP.md`

```
#pragma omp parrallel [...]
{
    ...
} // implicit barrier
// env
omp_set_num_threads(x);
num_th = omp_get_num_threads();
th_ID = omp_get_thread_num
// example
#pragma omp parallel for \
            shared(n,x,y) private(i)
for(i=0;i<n;++i)
    x[i]+=y[i];
// critical section
#pragma omp critical
global+=my;
```

共享内存的问题

内存一致性

不同线程缓存了同一个内存block

当其中一个线程修改后，所有线程缓存需要保持一致性

一致性策略

- write invalidate

  mark all copies as invalid except most recent one

- write update

  all other copies are updated

不同场景

- write miss

  update: broadcast on bus

  invalidate: the address is invalidated

- read miss

  update: update memory

  invalidate: check if there's a dirty copy in some cache and writeback if so

cache假共享问题

cache的block较大，又需要保持一致性

不同线程处理同一个block的不同位置时，每一次修改都会使得其它线程的cache失效

解决方法：让每个线程处理连续部分；用空位置补齐block的大小

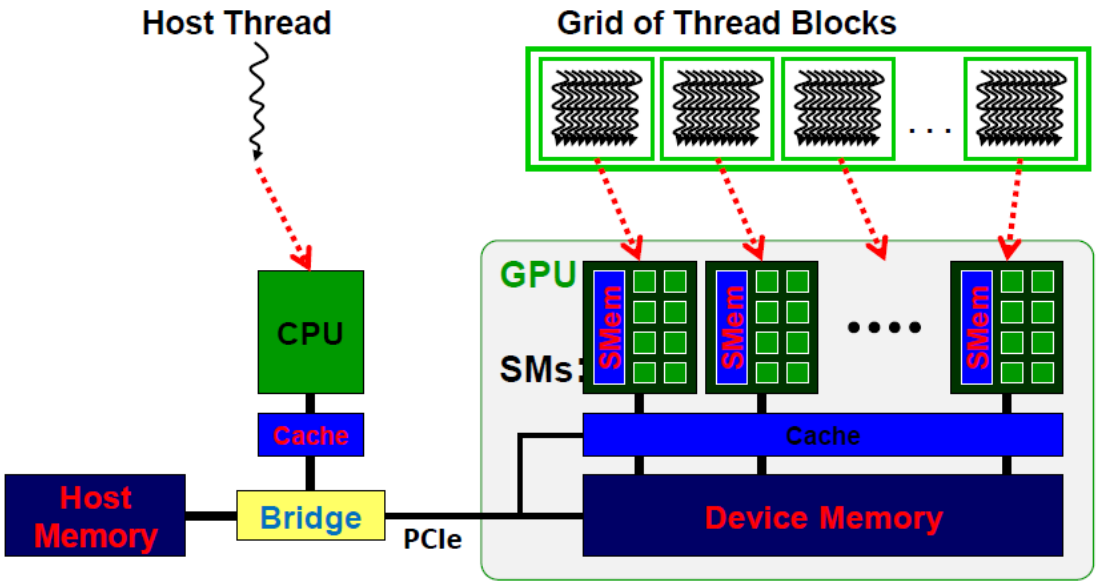
Chap6.3 GPU线程模型

GPU编程模型

SPMD

single procedure multiple data

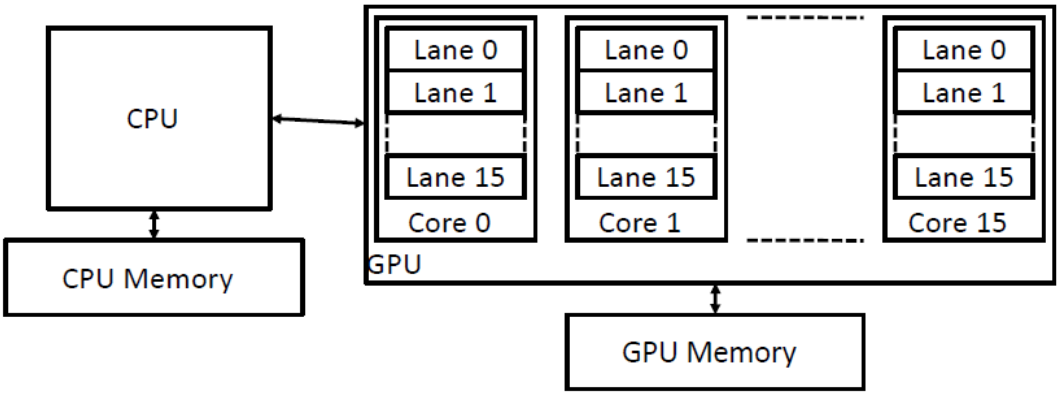
- 每个线程执行同样的过程，处理不同的数据
- 每个线程有自己的上下文



GPU硬件执行模型

SIMD

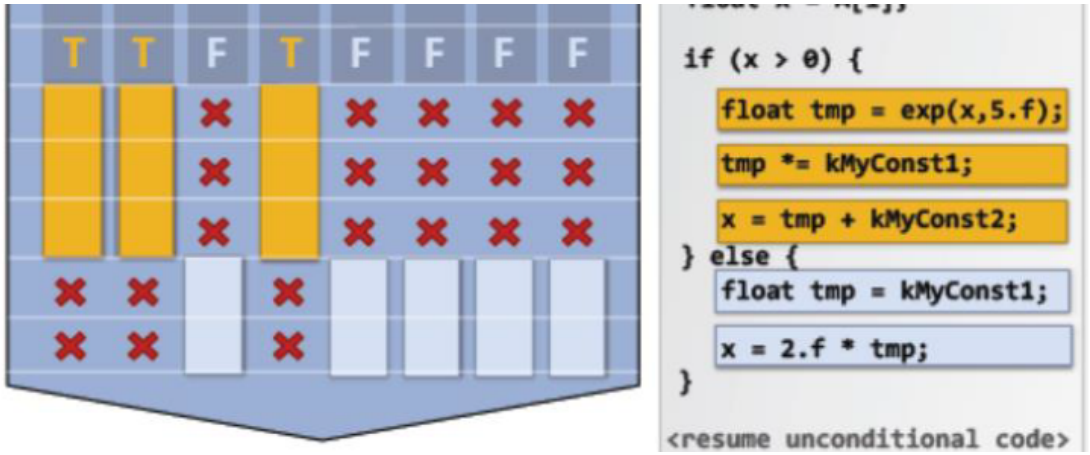
一组执行相同指令的线程由硬件动态组织成 warp，一个 warp 是由硬件形成的 SIMD 操作



每个core(SM)包含多个lane，运行一个线程块

SIMT模型: 每个线程的标量指令流汇聚在一起在硬件上以 SIMD 方式执行 (32 threads 组成一个 warp)

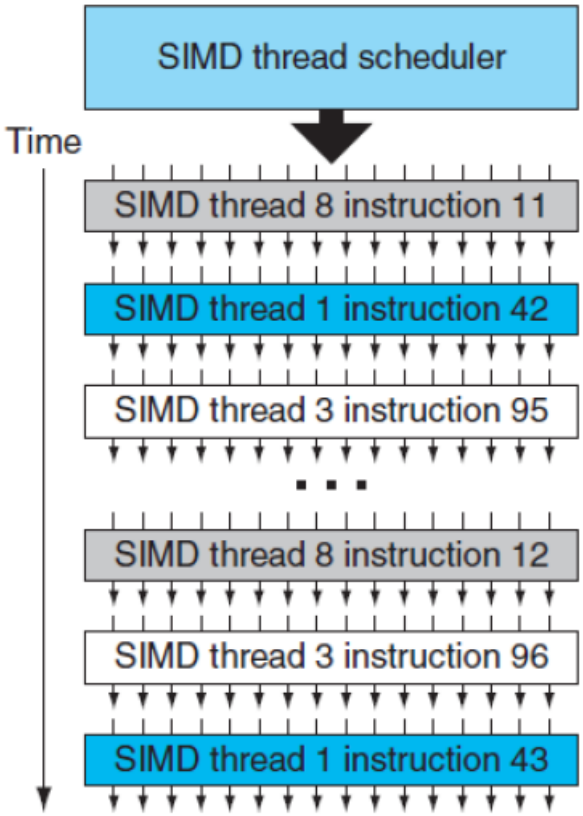
每一条指令都会在所有的lane执行，分支会导致性能下降



threads $\rightarrow$ warp $\rightarrow$ lane

调度器每次选择不同的warp，一个warp可能要分多次在lane上执行

注：SIMD thread = warp



利用多线程交替执行隐藏访存延迟，因此warp不能太少。吞吐率增加，但单个线程时间变长

细粒度多线程，每次执行每个warp的一条指令，所有warp的寄存器都保存在寄存器文件中，实现零开销切换。

寄存器数量有上限(32768 in 720M)，而warp使用的寄存器数量有最小限制，故warp不能太多

GPU存储模型

建议方式：将内存先分给不同的shared memory，每个warp处理自己的shared memory