

$$\text{CPU执行时间} = \text{指令数目} \times \text{CPI} \times \text{一个时钟周期长度}$$

$$\frac{\text{系统总加速比}}{\text{加速比}} = \frac{\text{总执行时间}_{\text{改进前}}}{\text{总执行时间}_{\text{改进后}}} = \frac{1}{(1 - P) + \frac{P}{S}}$$

What if I am adding 25% of a

溢出检测方法三

(真实做法)

(因为进位会产生)

$$V = C_n \oplus C_{n-1}$$

C_n	C_{n-1}	V
0	0	0
1	1	0
1	0	1
0	1	1

溢出(Overflow) : 运算结果超出了数据表示范围

负数除以 Power of 2

> Want $\lceil x / 2^k \rceil$ (向零方向舍入)

> Compute as $\lfloor (x+2^{k-1}) / 2^k \rfloor$

• In C: $(x + (1 << k) - 1) \gg k$

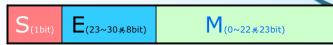
e 1: 没有舍入

① 被除的都是 0, 则加上修正并不会影响

② 商位的舍入, 则会产生进位.



单精度浮点数标准 IEEE754...



为什么采用偏移而不是补码：补码适合加法而不适合直接比较

◆ 规格化数(Normal)：

$$\text{代表数值: } (-1)^s \times 1.m \times 2^{e-\text{bias}}$$

➢ E的Bias:

表达范围：阶码全0或全1 表示特殊情况

- Single precision (8bits) : 127 (Exp: 1...254, E: -126...127)
- Double precision (11-bits) : 1023 (Exp: 1...2046, E: -1022...1023)

◆ 规格化数的最高数字位总是1， IEEE754将这个1缺省存储(隐藏位)，使得尾数表示范围比实际存储多一位

单精度非规格化数

◆ 非规格化数(Subnormal) (e=0, f非零)

➢ exp = 000...0, frac ≠ 000...0

$$\text{代表: } (-1)^s \times 0.m \times 2^{-126}$$

阶码全0时，表示的是非规格化，小数点前为0
为了使得数据连续

➢ 最小的规格化数: $1.0 \times 2^{1-127} = 1.0 \times 2^{-126}$



∞ (infinity)



$$\text{exp} = 111\dots1, \text{frac} = 000\dots0$$

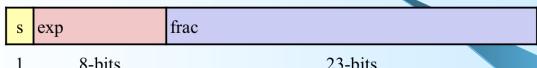
➢ 表示 ∞ (infinity)

➢ 注意不同的值： $+\infty$ and $-\infty$

➢ 一般是溢出 (overflows) 后得到的结果

➢ 例如： $1.0/0.0 = -1.0/-0.0 = +\infty, 1.0/-0.0 = -\infty$

NaN



$$\text{exp} = 111\dots1, \text{frac} \neq 000\dots0$$

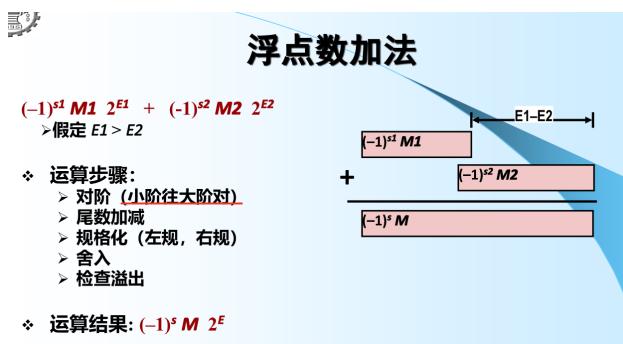
➢ 不是一个数 Not-a-Number (NaN)

➢ 表达当数值无法确定时，

➢ 例如， $\sqrt{-1}, \infty - \infty, \infty \times 0$

- 浮点0的形式和整数0的表达形式相同
- All bits = 0
- 几乎可以用 Unsigned Integer 比较器直接比较大小,
除了:
先比较符号位
必须考虑 $-0 = 0$
NaNs 比任何其他数值都大
其余部分 OK
 - 非规格化 vs. 规格化 ; 规格化 vs. 无穷大

注意符号



例题: 计算

$$2.6125 \times 10^1 + 4.150390625 \times 10^{-1}$$

$$2.6125 \times 10^1 = 26.125 = 11010.001 = 1.1010001000 \times 2^4$$

$$4.150390625 \times 10^{-1} = .4150390625 = .011010100111$$

$$= 1.1010100111 \times 2^{-2}$$

$$= 0.0000011010100111 \quad (\text{对阶, 小阶往大阶对, 小数点左移六位})$$

$$\begin{array}{r} 1.1010001000 \\ + 0.0000011010100111 \\ \hline \end{array}$$

1.1010100010 10 (尾数相加、舍入) and (尾数规格化检查)

1.1010100011 $\times 2^4$ (检查, 无溢出)

$$= 11010.100011 \times 2^0 = 26.546875 = 2.6546875 \times 10^1$$

浮点数乘法

相似的步骤:

- 阶相加(careful!)
- 尾数相乘 (set the binary point correctly)
- 尾数规格化
- 舍入 (potentially re-normalize)
- 设置符号位

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00 <u>011</u> ₂	10.00 ₂	(<1/2—down)	2
2 3/16	10.00 <u>110</u> ₂	10.01 ₂	(>1/2—up)	2 1/4
2 7/8	10.1 <u>100</u> ₂	11.00 ₂	(1/2—up)	3
2 5/8	10.10 <u>100</u> ₂	10.10 ₂	(1/2—down)	2 1/2

- $2.6125 \times 10^1 + 4.150390625 \times 10^{-1}$

$$2.6125 \times 10^1 = 26.125 = 11010.001 = 1.1010001000 \times 2^4$$

$$4.150390625 \times 10^{-1} = .4150390625 = .011010100111$$

$$= 1.1010100111 \times 2^{-2} \text{ (对阶, 小阶往大阶对)}$$

Shift binary point 6 to align exponents,

GR
 1.1010001000 00 |
 +.0000011010 10 0111 (Guard = 1, Round = 0, Sticky = 1)

 1.1010100010 10

(尾数相加) and (尾数规格化检查)

the extra bits (G,R,S) are more than half of the least significant bit (0). Thus, the value is rounded up. (舍入)

$$1.1010100011 \times 2^4 \text{ (检查, 无溢出)}$$

$$= 11010.100011 \times 2^0 = 26.546875 = 2.6546875 \times 10^1$$

对于IEEE754单精度浮点数加减运算，只要对阶时得到的两个阶码之差的绝对值 $|\Delta E|$ 大于等于 1 ，就无须继续进行后续处理，此时运算结果直接取阶大的那个数

- A. 24
- B. 25
- C. 126
- D. 128

正确答案: B 你选对了

哪些表达式恒为true ?

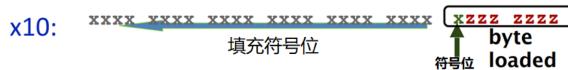
```
int x = ...;
float f = ...;
double d = ...;
```

Assume neither
d nor f is NaN

- $x == (int)(float) x$ ✗
- $x == (int)(double) x$ ✓
- $f == (float)(double) f$ ✓
- $d == (double)(float) d$ ✗
- $f == -(-f)$ ✓
- $2/3 == 2/3.0$ ✗
- $d < 0.0 \Rightarrow ((d*2) < 0.0)$ ✓
- $d > f \Rightarrow -f > -d$ ✓
- $d * d >= 0.0$ ✓
- $(d+f)-d == f$ ✗

$\infty > 2^4$

例如：lb x10, 4(x11)



算术右移n位不等价于除以 2^n

为了减轻保存和恢复寄存器的负担，将寄存器分为了两类：

1. 函数调用时，被调用函数帮助保存、恢复

- \$sp, \$gp, \$tp,
- \$s0-\$s11

2. 函数调用时，被调用函数不帮助保存、直接覆盖使用

- 返回地址 \$ra
- 参数寄存器 \$a0-\$a7,
- 临时寄存器 \$t0-\$t6

函数调用时寄存器使用规范

- a0-a7 for function arguments,
- a0-a1 for return values
- sp, stack pointer,
- ra return address
- s0-s11 saved registers
- t0-t6 temporarie

JAL x1, main

伪指令：JAL main, 对应的真实指令：JAL x1, main

伪指令：J main, 对应的真实指令：JAL x0, main

跳转到main函数，并将下一条指令存在x1寄存器中

被调用的函数 栈指针

Leaf: addi sp, sp, -8
sw \$1, 4(sp)
sw \$0, 0(sp)

- 举例：
- 数据传送伪指令 mv
 - mv dst, reg1 转换为 addi dst, reg1, 0
- 装入立即数伪指令：Load Immediate (li)
 - li dst, imm 转换为 addi 和 lui add的立即数，可表示范围很小 (12位)
 - LUI: Load Upper Immediate
- 装入地址伪指令：Load Address (la)
 - la dst, label 转换为 auipc dst, <offset to label>
 - AUIPC - Add Upper Immediate to PC
- 空操作伪指令(nop)
 - nop 转换为 addi x0, x0,

Pseudo	Real
nop	addi x0, x0, 0
not rd, rs	xori rd, rs, -1
beqz rs, offset	beq rs, x0, offset
bgt rs, rt, offset	blt rt, rs, offset
j offset	jal x0, offset
ret	jalr x0, x1, offset
call offset (if too big for just a jal)	auipc x6, offset[31:12] jalr x1, x6, offset[11:0]
tail offset (if too far for a j)	auipc x6, offset[31:12] jalr x0, x6, offset[11:0]

- LUI 将立即数设置在目标寄存器的高20位，低20位填0
- 作用：与ADDI指令一起，设置一个32位立即数

```
LUI x10, 0x87654      # x10 = 0x87654000
ADDI x10, x10, 0x321   # x10 = 0x87654321
```

- JALR rd, rs, immediate
- 设置rd, ra=PC+4
- 设置PC= rs + immediate (不乘以2)

即，jalr包含三部分功能：①加低12位到寄存器；②跳转到寄存器的地址；③将PC+4存到ra中

如果跳转目标离当前指令的距离 > 2^{10} 条指令



多段跳跃

```
beq x10, x0, far      →      bne x10, x0, next
# next instr           j far
# next instr
```

给出的RISC-V 指令以及它们的内存地址。

```
0x002cff00: loop: add x6, x7, x5
0x002cff04: jal ra, foo      // ra是x1的别名
0x002cff08: bne x6, zero, loop // zero是x0的别名
...
0x002cff2c: foo: jr ra      [??????] 00000 00110 001 [?????] 1100011
# next instr
```

接下来再给出Branch指令的格式：

imm	rs2=0	rs1=6	BNE	imm	BRANCH
-----	-------	-------	-----	-----	--------

部分指令编码表示一个立即数，这个立即数的（十进制）数值为多少？

- A. -2 B. -4 C. -8 D. 4

存的都是相对位置/2

举例: C \Rightarrow Asm \Rightarrow Obj (step1) \Rightarrow Exe \Rightarrow Run

替换伪指令、为jump指令添加相对偏移量

```
.text
.align 2
.globl main
main:
    addi sp, sp, -4      Pseudo-
    sw ra, 0(sp)        Instructions?
    addi t0, x0, 0        Underlined
    addi a1, x0, 0        前边的伪指令已
    addi t1, x0, 100     被替换
    jal x0, 16           相对偏移量已加
loop:
    mul t2, t0, t0
    add a1, a1, t2
    addi t0, t0, 1
```

check:
blt t0, t1 -16
lui a0, l.str
addi a0, a0, r.str
jal printf
mv a0, x0
lw ra, 0(sp)
addi sp, sp 4
jalr x0, ra
.data
.align 0
str:
.asciiz "The sum of sq from 0
.. 100 is %d\n"

举例: C \Rightarrow Asm \Rightarrow Obj (step2) \Rightarrow Exe \Rightarrow Run

- 创建符号表、重定位表 symbol table and relocation table

Symbol Table

Label	address (in module)	type
main:	0x00000000	global text
loop:	0x00000014	local text
str:	0x00000000	local data

Relocation Information

Address	Instr.	type	Dependency	创建目标文件(.o), 包含
0x00000002c	lui	l.str		• 代码段 text segment
0x00000030	addi	r.str		• 数据段 data segment
0x00000034	jal	printf		• 符号表、重定位表

```
1 .import print.s          # print.s is a different file
2 .data
3 array: .word 1 2 3 4 5
4 .text
5 sum:   la t0, array
6 li t1, 4
7 mv t2, x0
8 loop:  blt t1, x0, end
9 slli t3, t1, 2
10 addi t3, t0, t3
11 lw t3, 0(t3)
12 add t2, t2, t3
13 addi t1, t1, -1
14 j loop
15 end:   mv a0, t2
16 jal ra, print_int # Defined in print.s
```

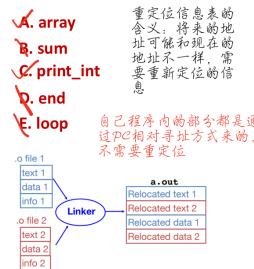
问题: 哪几行的指令是伪指令?

```
1 .import print.s          # print.s is a different file
2 .data
3 array: .word 1 2 3 4 5
4 .text
5 sum:   la t0, array
6 li t1, 4
7 mv t2, x0
8 loop:  blt t1, x0, end
9 slli t3, t1, 2
10 addi t3, t0, t3
11 lw t3, 0(t3)
12 add t2, t2, t3
13 addi t1, t1, -1
14 j loop
15 end:   mv a0, t2
16 jal ra, print_int # Defined in print.s
```

问题: 对于 branch/jump 指令,
which labels will be resolved in the
first pass of the assembler? The second?
第一次时, loop里面看到end是个未定义的label, 因此
无法处理

问题: What's contained in the relocation table (重定位信息表)?

```
1 0x00061C00: sum:   la t0, array
2 0x00061C08:   li t1, 4
3 0x00061C0C:   mv t2, x0
4 0x00061C10: loop:  blt t1, x0, end
5 0x00061C14:   slli t3, t1, 2
6 0x00061C18:   addi t3, t0, t3
7 0x00061C1C:   lw t3, 0(t3)
8 0x00061C20:   add t2, t2, t3
9 0x00061C24:   addi t1, t1, -1
10 0x00061C28:  j loop
11 0x00061C2C: end:   mv a0, t2
12 0x00061C30: jal ra, print_int
```



哪些需要重定位?

相对位置

- PC-Relative Addressing (beq, bne, jal)
 - never relocate 一般自己程序内部都是无需重定位的
- External Function Reference (usually jal)
 - always relocate
- Static Data Reference (often auipc and addi)
 - always relocate
 - RISC-V often uses auipc rather than lui so that a big block of stuff can be further relocated as long as it is fixed relative to the pc

什么时候以下指令对应的机器码可以完全确认？

1) addu x6, x7, x8

2) jal printf

A: 1) & 2) After compilation

B: 1) After compilation, 2) After assembly

C: 1) After assembly, 2) After linking

D: 1) After compilation, 2) After linking

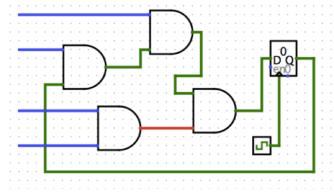
E: 1) After compilation, 2) After loading

Period = Max Delay = CLK-to-Q Delay + CL Delay + Setup Time
组合逻辑电路延迟

眼hold没有关系

What is maximum clock frequency? (assume all unconnected inputs come from some register)

- A: 5 GHz
- B: 200 MHz
- C: 500 MHz
- D: 1/7 GHz
- E: 1/6 GHz



Clock->Q 1ns
Setup 1ns
Hold 1ns
AND delay
1ns

CL Delay = 3 ns ~ 1 ns ~ 1 ns

total 5 ns => 200MHz

1
2

1
2



中断处理

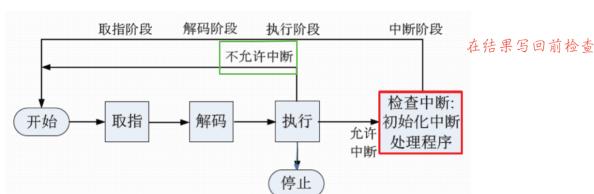
三个步骤！！

- 设置MEPC为发生异常的指令的地址 通常也即当前指令的地址
- 设置MSTATUS寄存器，强迫CPU进入kernel态，禁止中断响应，即“关中断”。
即不会在处理中断的过程中再次被中断去处理另一个中断
- 设置MCause寄存器，使软件可以得到异常的类型信息；
- CPU开始从一个统一入口取指令(mtvec中的base)，以后的所有事情都交由软件处理了。

精确中断响应

28

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



性能衡量单位: CPE

Method	Integer		Double FP	
	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 -O1	10.12	10.12	10.17	11.14 ✓

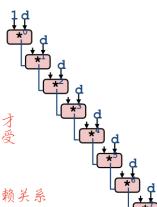
*dest) cycles per element (每个元素完成的周期数)

- 内层循环 (Integer Multiply)

```
for (i = 0; i < length; i++)
    t = t OP d[i];
```

由于有数据依赖，循环中每个步骤都得等上一步执行完才能开始执行（假如流水线满载，应该可以做得更快，但受限于数据依赖，只能这样）

是否有处理方法：循环展开（其实就是消除掉相近的依赖关系）



1

2

```
x = x OP (d[i] OP d[i+1]);
```

```
for (i = 0; i < limit; i+=2) {
    x = (x OP d[i]) OP d[i+1];
}
```

<=

这种写法是没有意义的

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

循环展开、 独立计算: Double *

23

执行一个指令所需时间，注意不是cpe (per element)
应该是1)

Accumulators

→ 循环展开数

FP *	Unrolling Factor L								
	K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	5.01	5.01
2		2.51		2.51		2.51			
3			1.67						
4				1.25		1.26			
6					0.84			0.88	
8						0.63			
10							0.51		
12								0.52	

2个计算单元，每个指令5个周期，因此2*5就是恰好可以在两个计算单元内完成连续流水的循环展开数

Method	Integer		Double FP	
	Add	Mult	Add	Mult
Operation	0.54		1.01	
Best	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

但是有个问题，有4个int加法器，为什么add不是0.25?

- 受限于功能单元的吞吐率
- 相比未优化的基准程序快了42倍

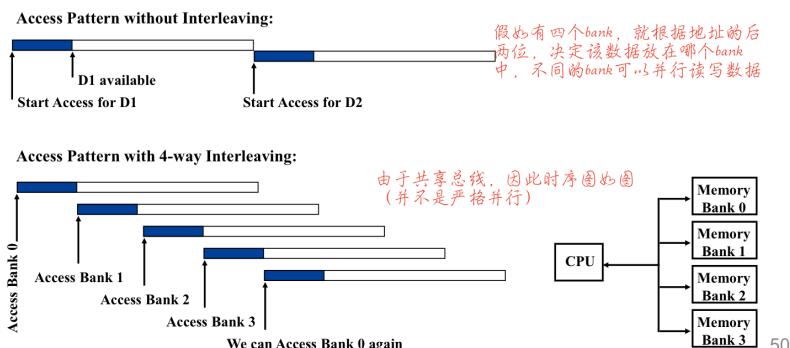
因为只有两个Load单元，则
加法的速度还受限于数据
加载的速度（总得取到了
数据才可以开始计算吧）

刷新其实也和正常读写过程是一样的，也是读到buffer中然后写回，可以理解为刷新穿插在读写过程中，并且每行必须要在刷新周期内完成一次

一个 256×256 结构的DRAM芯片，每隔2ms要刷新一次，采用异步刷新方式，且刷新是按顺序对所有256行存储元进行内部读操作和写操作实现的。设存取周期为 $0.5\mu s$ ，求刷新开销（即刷新操作的时间所占的百分比）

☆ 存取开销！

- A. 6.4%
- B. 3.2%
- C. 12.8% $\frac{2 \times 0.5\mu s}{2 ms} \times 100\% = 12.8\%$
- D. 1.6%



DDR3 SDRAM 芯片内部核心频率是133.25Mhz, 与之相连的

存储总线每次传输8B, 下面描述错误的是：

- A. 存储器总线的时钟频率是1066Mhz
- B. 芯片内部输入输出缓冲采用8位预取技术
- C. 存储器总线每秒传1066M次数据
- D. 存储器总线带宽约为8.5GB每秒

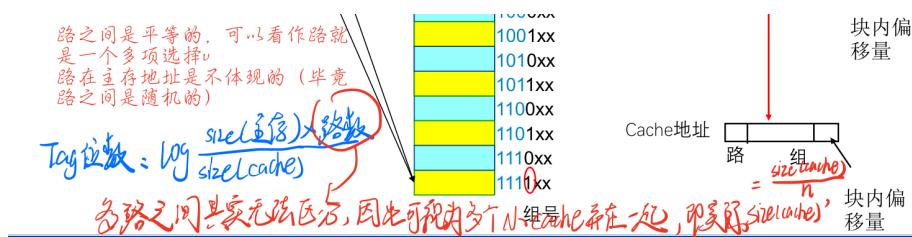
$$133.25 \times 8 = 1066 \text{ M 数据}$$

但由于 DDR，每 S 2 次数据，故 A，不是 1066Mhz

t_a 平均访问时间

$$t_a = ht_c + (1-h)t_m$$

也可以先计算 index 的值和 offset, cache 内的块数就可以计算出 index



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

直接映射快，但是命中率低
组相联稍慢，但命中率较高

AMAT 是内存访问的平均 (expected) 时间，可以用以下公式来估算：

$$AMAT = hit_time + miss_rate \times miss_penalty$$

- Hit_time: cache hit 时，访问 cache 所花的时间
- Miss_rate: 高速缓存的失效率
- miss penalty: 当发生 cache miss 时，需要花的额外的访存时间，所以一次 cache miss 需要花费 (hit_time + miss_penalty) 的时间

Global miss rate 和 Local miss rate 的定义请参考如下描述：

Global miss rate:

- the fraction of references that miss some level of a multilevel cache
- misses in this cache divided by the total number of memory accesses generated by the CPU

Local miss rate – the fraction of references to one level of a cache that miss

Local Miss rate L2\$ = L2\$ Misses / L1\$ Misses

Cache失效 (miss) 时的写策略

以下两个方法，差别就在于是否将新的内容放到cache中

- 按写分配法 Write-allocate (将数据读入缓存，在缓存中更新内容)
 - 如果要连续访问同一位置，这种策略较优
 - 出现写失效时，首先会出现读失效
 - 在缓存中先为写失效分配一行 (line)，然后在这一行发生一次写命中
 - 常与写回法 (write-back) 搭配使用 本质上就是先把内容xx位置的数据读到内存中，接下来就相当于写命中了，只修改cache中的值，内存中的值等cache替换再写入
- 不按写分配法 No-write-allocate (直接修改内存中的内容，无需读入缓存)
 - 写失效不影响缓存中的数据
 - 数据仅在低级的存储器中被修改
 - 常与写直达 (write-through) 搭配使用

无论如何
来源于 cache

1

2

课堂讨论：In the Table 1, we have given you four different sequences of addresses generated by a program running on a processor with a data cache. Cache hit ratio for each sequence is also shown below. Assuming that the cache is initially empty at the beginning of each sequence, find out the following parameters of the processor's data cache:^④

- Associativity (1, 2 or 4 ways)^④ 由①得，block size肯定不会是32或16bytes，因为这样hit ratio应该是4/6和1/2，也不能是小于8，否则hit应该会小于2/6，故block size只能是8
- Block size (1, 2, 4, 8, 16, or 32 bytes)^④ 由②得，该序列所有的地址写为二进制表示，其组号部分都是全0（最后三位得拿出来作块偏移），故只有associativity因为4时才会是1/6
- Total cache size (256 B, or 512 B)^④
- Replacement policy (LRU or FIFO)^④

Assumptions: all memory accesses are one byte accesses. All addresses are byte addresses.^④

Sequence No.	Address Sequence	Hit Ratio
1	0, 2, 4, ..., 8, 16, 32	0.33
2	0, 512, 1025, 1536, 2048, 1536, 1025, 512, 0	0.33
3	0, 64, 128, 256, 512, 256, 128, 64, 0	0.33
4	0, 512, 1024, 0, 1536, 0, 2048, 512	0.25

由③可以得到total cache size，由④可以得到replacement policy，具体可由地址的二进制编码确定

1

2

3

4

5

6

TLB 事件组合

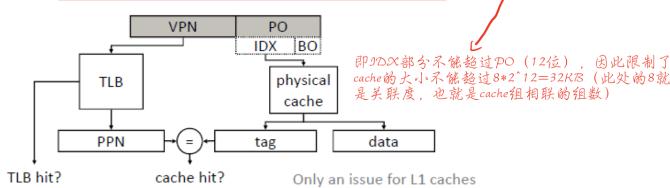
TLB	Page Table	Cache	可能发生吗? 在何种情况下?
Hit	Hit	Hit	是的 – 我们希望这种情况发生!
Hit	Hit	Miss	是的 – TLB命中, 不需要访问页表, 但数据没有调入缓存, 只能从内存读取
Miss	Hit	Hit	是的 – TLB失效, 地址转换通过查页表获得
Miss	Hit	Miss	是的 – TLB失效, 地址转换通过查页表获得, 但数据没有调入缓存, 只能从内存读取
Miss	Miss	Miss	是的 – 页面失效
Hit	Miss	Miss/ Hit	不可能 – 如果这页没有调入内存, TLB不可能命中
Miss	Miss	Hit	不可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

22

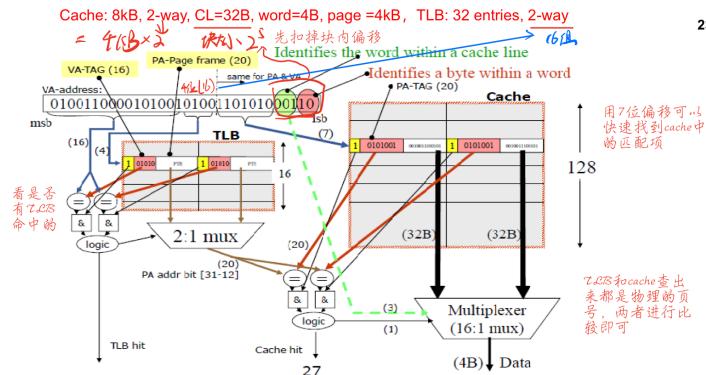
Virtually-Indexed Physically-Tagged: VIPT

- If $C \leq (\text{page_size} \times \text{associativity})$, the cache index bits come only from page offset (same in VA and PA)
- If both cache and TLB are on chip
 - index both arrays concurrently using VA bits
 - check cache tag (physical) against TLB output at the end

优点: 既速度快 (TLB前无需地址转换), 又没有别名问题 (cache中是物理地址)



Putting it all together: VIPT



Bus Based Snooping Protocol

- Write Update (Broadcast)
 - 早期方法：所有处理器共用一个总线和memory交流，处理器间没有直连
- Writes are broadcast and update all other cache copies

- Write Invalidate
 - Writes invalidate all other cache copies

基于总线的更新方法：

- 写更新：每次更新就让其他所有cache也进行更新
- 写失效：每次更新就让其他所有cache的相关条目失效

写失效

MSI Protocol

增加了一个状态

35

- Three states to differentiate between clean or dirty
 - Modified, Shared and Invalid
- Modified**
 - The cached copy is the only valid copy in the system.
 - Memory is stale.
- Shared**
 - The cached copy is valid and it may or may not be shared by other caches.
 - Initial state after first loaded.
 - Memory is up to date.
- Invalid**
 - The cached copy is not existence.

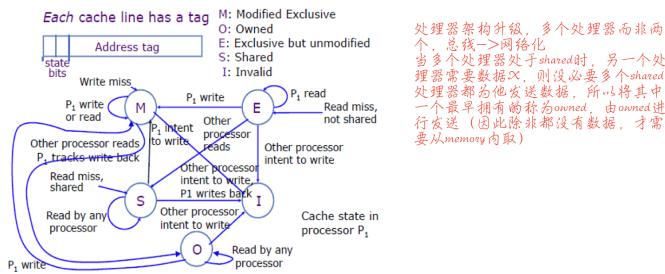


MESI states

- Modified
 - Main Memory's value is stale
 - No other cache possesses a copy
- Exclusive
 - Main Memory's value is up to date
 - No other cache possesses a copy
- Shared
 - Main Memory's value is up to date
 - Other caches have a copy of the variable
- Invalid
 - This cache have a stale copy of the variable

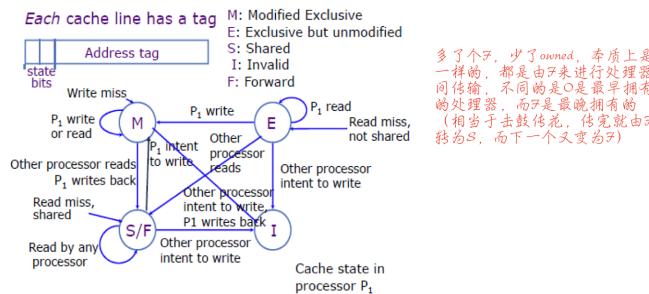
唯一的区别就是memory的数据备份是否是有效的

MOESI (Used in AMD Opteron)



处理器架构升级，多个处理器而非两个，总线->网络化
当多个处理器处于shared时，另一个处理器需要数据时，则没必要多个shared处理器都为他发送数据，所以将其中一个最早拥有的称为owned，由owned进行宏送（因此除非都没有数据，才需要从memory内取）

MESIF (Used by Intel Core i7)



多了个F，少了owned，本质上是一样的，都是由S来进行处理器间传输，不同的是O是最早拥有的处理器，而F是最晚拥有的（相当于击鼓传花，传来就由S转为F，而下一个又变为S）

False sharing

假共享问题：一个cache里一个块包含多个数据，改其中的一个数据会导致其他数据也变为无效（尽管他们事实上确实是最新的）

state	blk	addr	data0	data1	...	dataN
-------	-----	------	-------	-------	-----	-------

- A cache block contains more than one word
- Cache-coherence is done at the block-level and not word-level
- Suppose P₁ writes word_i and P₂ writes word_k and both words have the same block address.
- What can happen?
后一个处理器必须等前一个数据写完并更新后才可以读

这个五阶段流水线处理器的最小时钟周期长度和最大时钟频率分别是多少？

- 在T3的基础上，五个阶段前后都加上流水段寄存器的时间即可
 - IF : tPC clk-to-q + tIMEM read + tReg setup = 30 + 250 + 20 = 300 ps
 - ID : tReg clk-to-q + tRF read + tReg setup = 30 + 150 + 20 = 200 ps
 - EX : tReg clk-to-q + tmux + tALU + tReg setup + tmux = 30 + 25 + 200 + 20 + 25 = 300 ps
 - MEM : tReg clk-to-q + tDMEM read + tReg setup = 30 + 250 + 20 = 300 ps
 - WB : tReg clk-to-q + tmux + tRF setup = 30 + 25 + 20 = 75 ps
- 注意最小时钟周期取每个阶段的最大时间

S中

5 单选 (1分) 寄存器中的值有时是地址，有时是数据，在指令中，它们在形式上没有差别，只有通过（ ）才能识别它是数据还是地址。

得分/总分

A. 指令操作码或寻址方式位

✓1.00/1.00

B. 时序信号

C. 判断程序

D. 寄存器编号

3 单选 (1分) 假设不考虑中断和异常处理（这个内容在以后的章节介绍），关于程序计数器PC的叙述中，哪个是错误的？

得分/总分

A. 条件转移指令（例如:beq）指令执行后，PC的值一定是跳转到的目标地址

✓1.00/1.00

B. 指令顺序执行时，PC的值会改变为下一条指令的地址，在MIPS、RISCV32I中，PC的值加4

C. 每条指令执行后，PC的值都会改变



D. 无条件转移指令（jump）指令执行后，PC的值一定是跳转到的目标地址

正确答案：A 你选对了

解析： A. 只有在条件满足的情况下，PC的值才是跳转到的目标地址

1 单选 (1分) 哪些相关会引起流水线冲突？

得分/总分

A. 控制相关、结果相关、数据相关

其实就是结构冒险，换个名换个寄存器就好了

✓1.00/1.00

B. 数据相关、名字相关、控制相关

C. 指令相关、数据相关、控制相关

D. 名字相关、控制相关、指令相关

10 多选 (1分) 下面列出了开发指令级并行性所使用的技术，哪些技术是只基于“硬件”的？ 得分/总分

- A. Very long instruction word (超长指令字) ✓0.33/1.00
- B. Superscalar (超标量) ✓0.33/1.00
- C. Dynamic scheduling (动态调度) ✓0.33/1.00
- D. Reorder buffer (重排序缓冲器) ✓0.33/1.00

正确答案：B、C、D 你选对了

解析：A、这是用软件，编译的时候，实现多发射 B、这是用硬件，运行时多发射（的专用术语） C、“动态”调度，指执行时再调度，而不是执行前调度，所以是硬件完成的。 D、这是一个硬件单元

10 多选 (1分) 下面列出了开发指令级并行性所使用的技术，哪些技术是只基于“硬件”的？ 得分/总分

- A. Multiple-issue(多发射) ✗该题无法得分/1.00
- B. Superscalar (超标量) ✓该题无法得分/1.00
- C. Register renaming (寄存器换名) ✗该题无法得分/1.00
- D. Dynamic scheduling (动态调度) ✓该题无法得分/1.00

正确答案：B、D 你错选为A、B、C、D

解析：B、这是用硬件，运行时多发射（的专用术语） C、软硬件均可实现 D、“动态”调度，指执行时再调度，而不是执行前调度，所以是硬件完成的。

4 单选 (1分) 缓存到地址映射中_____比较多的采用“按内容寻址”的相联存储器来实现 得分/总分

- 地址
- A. 全相联映射 ✓1.00/1.00
 - B. 自动映射
 - C. 组相联映射
 - D. 直接映射

正确答案：A 你选对了

1 单选 (1分) 采用虚拟存储器的目的是 得分/总分

- A. 增加存储系统结构的层次性
- B. 提高主存的访问速度；
- C. 扩大存储器的寻址空间； ✓1.00/1.00
- D. 扩大辅存的存取空间；

正确答案：C 你选对了

2. 如 (单选) (1分)

得分/总分

下C
语言
程序

```
for(k=0; k<1000; k++)  
    a[k] = a[k]+32;  
若数组a及变量k均为int型, int型数据占4B, 数据cache采用直接映射方式, 数据区大小为1KB, 块大小为16B, 该程序段执行前cache为空, 则该程序段执行过程中访问数组a的cache缺失率为
```

狗

- A. 25%
- B. 1.25%
- C. 2.5%
- D. 12.5%

 每个循环周期, 每个元素都访问了两次

✓1.00/1.00

正确答案: D 你选对了

解析: D、 $a[k] = a[k]+32$ 中, 先读取 $a[k]$, 和写 $a[k]$ 是两次访问存储器。所以, 每个主存块中的4个数据, 只有读第一个时会miss, 其余的7次访问全部命中。

6 (单选) (1分) 选出正确的说法:

得分/总分

- 1) TLB是页表的cache;
 - 2) 主存是磁盘的cache;
 - 3) TLB采用一般使用全相联映射; 
 - 4) 如果发生TLB miss, 就一定会发生cache miss
-
- A. 1) 2) 4)
 - B. 1)
 - C. 全对
 - D. 1) 和2)

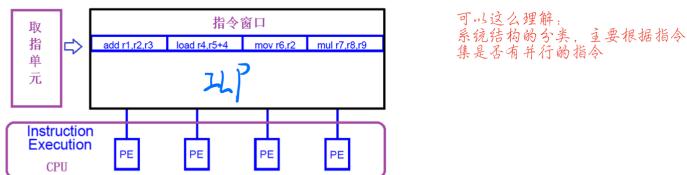
✓1.00/1.00

正确答案: D 你选对了

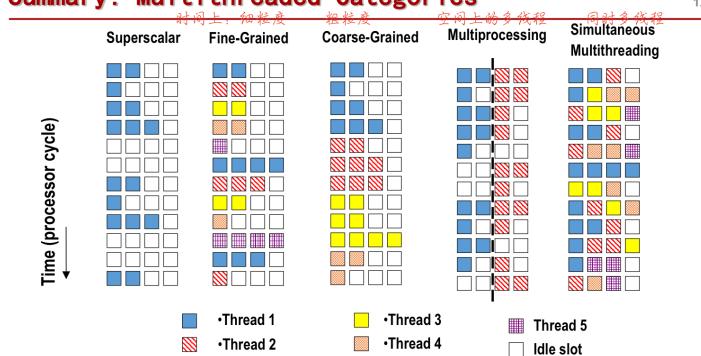
分类

- 单指令流单数据流 (SISD)：传统计算机：单个CPU，从内存中取一条指令，该作用于单一的数据流；
- 单指令流多数据流 (SIMD)：单个指令流作用于多个数据流上。
- 多指令流单数据流 (MISD)：很少见。冗余，多用于容错系统。
- 多指令流多数据流 (MIMD)：类似于多个SISD系统。例如：多处理器计算机、企业级服务器。

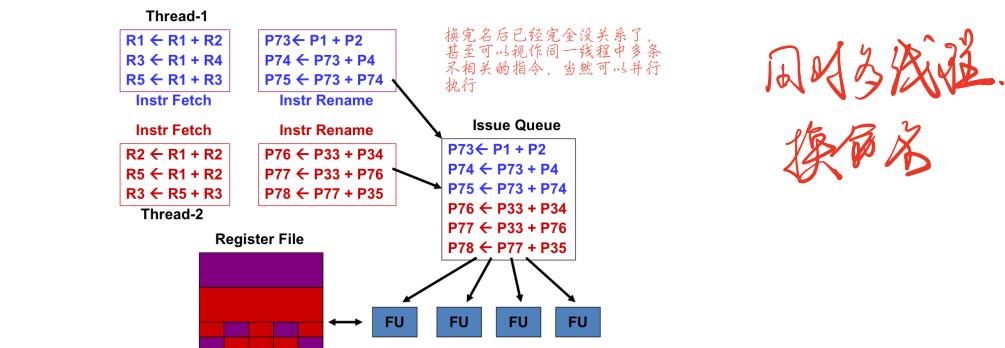
- 超标量依旧是标量处理器，即可以理解为高效的SISD
- 问题：超标量处理器是SISD？ SIMD？ MIMD？
 - SuperScalar：多个独立的操作由硬件调度单元动态发射



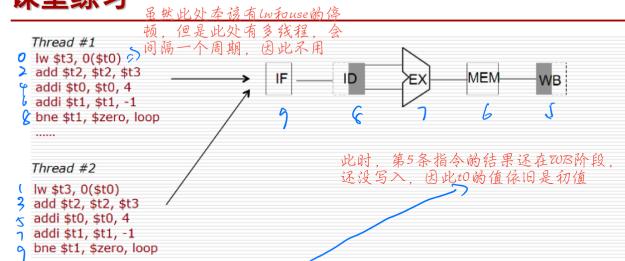
Summary: Multithreaded Categories



寄存器换名对同时多线程的支持



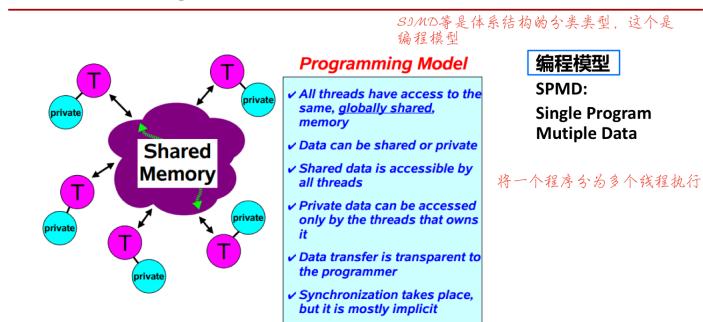
课堂练习



Q. 流水线具有前向通路 (bypassing) 以消除部分数据相关性。这两个线程采用交叠执行 (interleaved) 的方式共享流水线 -- 线程1的第一条指令在cycle 0 被取指，然后每一个线程的指令采用交叉执行的方式轮流进入流水线执行。为了支持这两个线程共享流水线，每个线程都有一套独立的寄存器。除了线程1的\$t0寄存器被初始化为0X10010000、线程2的\$t0寄存器被初始化为0X10010024，其它寄存器都初始化为0；第一条指令 (load指令) 存储在0X40040048。在第9周期时 (注意：初始周期是0) 线程#2 的寄存器 \$t0 的内容？

Shared Memory Model

29



D.

```
// 将arr数组的每一个元素减去其数组下标值。  
// Decrement element i of arr, n is a multiple of omp_get_num_threads()  
#pragma omp parallel {  
    int threadCount = omp_get_num_threads();  
    int myThread = omp_get_thread_num();  
    for (int i = 0; i < n; i++) {  
        if (i % threadCount == myThread) arr[i] -= i;  
    }  
}
```

功能一致，但是反而更慢，更慢的原因无关openMP，而是由于多CPU，访问数组的邻近元素，为了维持缓存一致性，而经常性出现假共享问题，需要进行cache的更新等，反而速度慢了很多
多CPU的多线程，一定要注意数组元素不要相邻

多条指令流执行相同的程序

- 每个程序
 - ▣ 操作不同的数据
 - ▣ 运行时可以执行不同的控制流路径
- 许多应用以这种方式编程，运行在MIMD硬件结构上(multiprocessors)
- 现代 GPUs 以这种类似的方式编程，运行在SIMD硬件上

多线程->多数据

**A Thread of SIMD Instructions
(SIMD thread)
= A warp**

把一个warp看做一个并行的线程

所有线程的寄存器值保存在
寄存器文件中

寄存器数目特别多，
所以cache很小。

Warp-based SIMD vs. Traditional SIMD

- 传统的SIMD仅包含一个线程
 - Lock step: 一条向量指令执行完，然后启动下一条向量指令
 - 编程模型为SIMD (no extra threads) → SW 需要知道向量长度
 - ISA 包含vector/SIMD指令
- Warp-based SIMD 由多个标量线程构成，以SIMD方式执行 (即所有的线程执行相同的指令)
 - 每个线程可以映射到不同的warp中，编程模型不是SIMD
 - ▣ SW 不必知道向量长度
 - ▣ 多个线程可以动态构成warp
 - ISA是标量ISA → 可以动态形成逻辑上的向量指令
 - 是一种在SIMD硬件上实现的SPMD编程模型

7 [单选] (1分) 关于计算机系统性能和程序执行时间，以下说法哪些是正确的? 得分/总分

1. 机器的时钟频率越高，机器的速度就越快
2. 计算机的MIPS数越大，性能就越好
3. 计算机系统性能的主要衡量指标包括：响应时间和吞吐率
4. 基准测试程序执行得越快，说明机器性能越好
5. 一个程序的执行时间，不仅仅是执行这个程序所有指令所用的时间，因为在程序执行过程中，还会执行操作系统代码或者其他用户程序，也可能等待I/O操作。

A. 3、5 1.00/1.00

B. 1、3、5

C. 3、4、5

D. 全部正确

正确答案: A 你选对了

21 [单选] (1分) 在IEEE 754浮点数运算中，判断浮点运算的结果是否溢出的描述，哪些是正确的? 得分/总分

1. 浮点运算结果是否溢出，并不以尾数溢出来判断，而主要看阶码是否溢出。
2. 尾数溢出时，可通过规格化操作进行纠正。
3. 阶码上溢时，说明结果的数值太大，无法表示；阶码下溢时，说明结果数值太小，可以把结果近似为0。
4. 在进行对阶、规格化、舍入和浮点数的乘/除运算等过程中，都需要对阶码进行加、减运算，可能会发生阶码上溢或阶码下溢。

A. 2, 3, 4

0.00/1.00

B. 1, 2, 3

C. 1, 3, 4

D. 全对

正确答案: D 你错选为B

27 [单选] (1分) 某计算机主存按字节编址,由4个64M*8位的DRAM芯片采用交叉编址方式构成,并与宽度为32位的存储器总线相连。主存每次最多读写32位数据。若double型变量x的主存地址位804 001AH,则读取x需要的存储周期数为:

得分/总分

- A. 1
- B. 2 0.00/1.00
- C. 4
- D. 3

正确答案: D 你错选为B

49 [单选] (1分) 下列关于外部输入输出中断的叙述中,正确的是:

得分/总分

- A. 有中断请求时, CPU立即暂停当前指令执行,转去执行中断服务程序
- B. CPU只有在处于中断允许状态时,才能响应外部设备的中断请求
- C. CPU响应中断时,通过执行中断隐指令完成对通用寄存器的保护
- D. 中断控制器按所接收中断请求的先后次序进行中断优先级排队

正确答案: B 你没选择任何选项