程序优化
Program Optimization

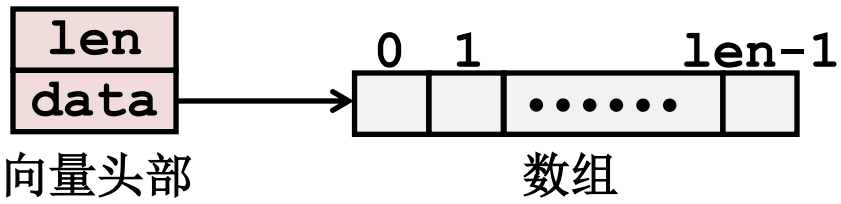
课 程 名 ：计算机系统

第 8 讲 （2025 年 5 月 14 日）

主 讲 人 ：华栋

基准测试benchmark举例：向量数据类型

```
/* 向量头部的数据结构 */
typedef struct {
    size_t len;
    data_t *data;
} vec;
```



- **数据类型**
  - 对 data_t 作不同的声明（如：typedef long data_t;）
  - int
  - long
  - float
  - double

```
/* 获取向量的元素并存至 *val */
int get_vec_element (*vec v,
    size_t idx, data_t *val)
{
    if (idx < 0 || idx >= v->len)
        return 1;
    *val = v->data[idx];
    return 0;
}
```

Benchmark 计算

```
typedef struct { … }
    *vec_ptr;  // 结构体指针类型
```

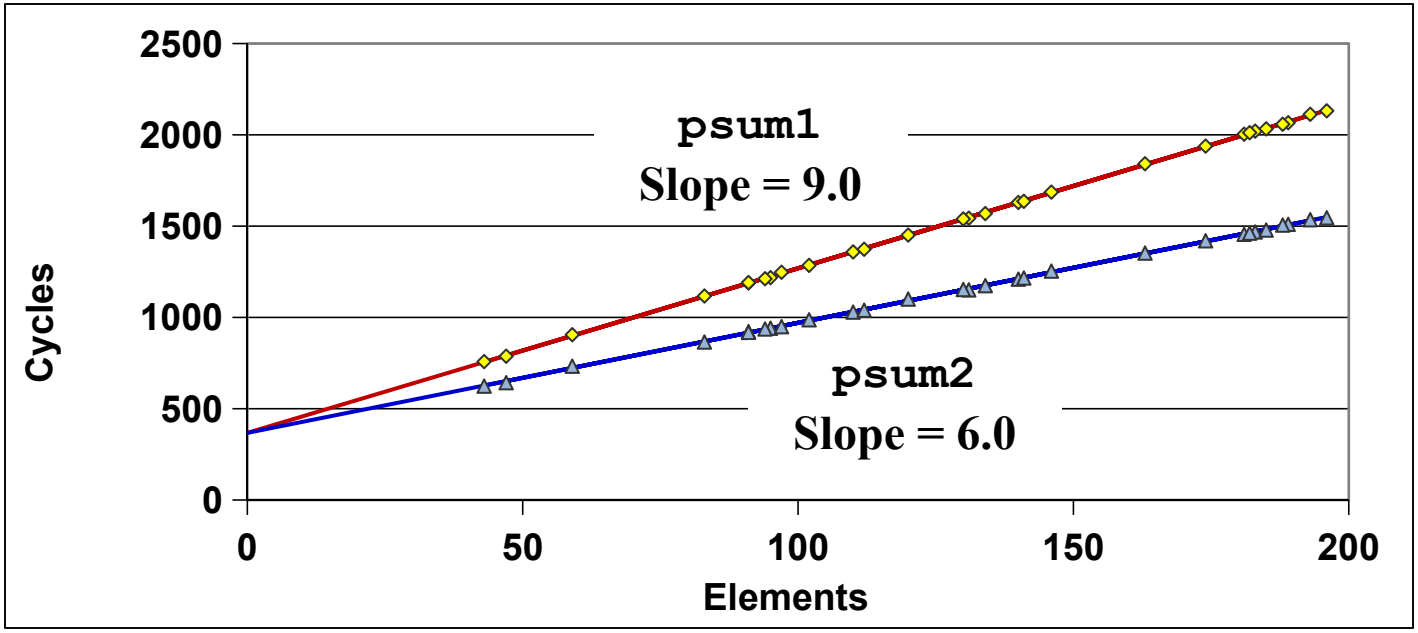
```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;          // 结果变量初始化
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
                                    // 原始代码
```

计算向量元素的 Σ 或 Π

- 数据类型
  - data_t 可以是：
    - int
    - long
    - float
    - double
- 操作
  - 宏 OP 和宏 IDENT 可以是：
    - + / 0
    - * / 1

每元素的周期数（Cycles Per Element, CPE）

- 衡量向量（列表）处理程序性能的指标
- length = *n*
- 对于本例而言，CPE = 每次操作所花费的周期数
- T = CPE × *n* + 固定开销overhead
  - CPE 为斜率slope



Benchmark 性能

```
void combine1(vec_ptr v, data_t *dest)
{
    long i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

计算向量元素的 Σ 或 Π

方法	整数类型		浮点类	
操作	+	*	+	*
combine1 未优化	22.68	20.02	19.98	20.18
combine1 –O1	10.12	10.12	10.17	11.14

基础优化

```
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    for (i = 0; i < length; i++)
        acc = acc OP data[i];
    *dest = acc;
}
```

- 将函数 vec_length 移到循环外（combine2）
- 避免每个循环的边界检查（combine3）
- 用寄存器作局部变量累积中间结果（combine4）

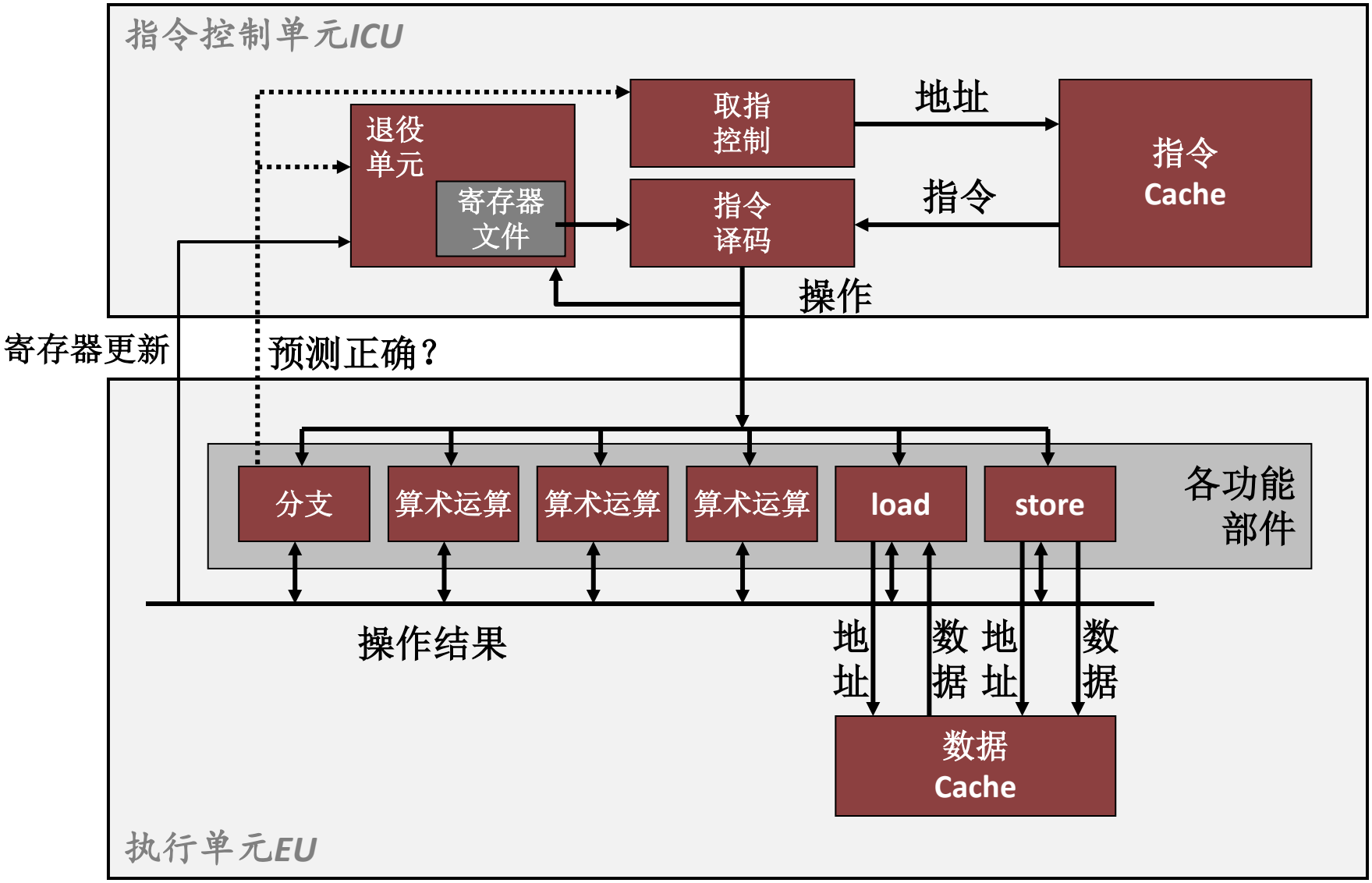
基础优化的效果

```
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;
    for (i = 0; i < length; i++)
        acc = acc OP data[i];
    *dest = acc;
}
```

方法	整型类		浮点类	
操作	+	*	+	*
combine1 –O1	10.12	10.12	10.17	11.14
combine4	1.27	3.01	3.01	5.01

- 消除循环中的经常性开销

现代 CPU 设计——超标量

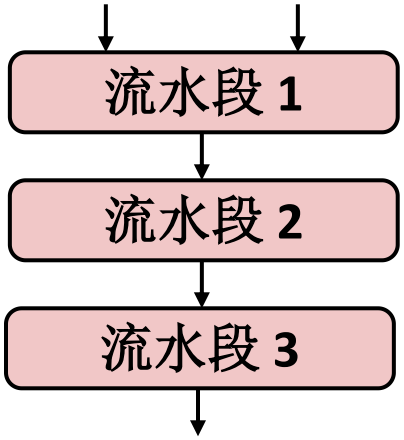

指令控制单元ICU
退役
单元
寄存器
文件
取指
控制
地址
指令
Cache
指令
译码
指令
操作
寄存器更新
预测正确？
分支
算术运算
算术运算
算术运算
load
store
各功能
部件
操作结果
地
址
数
据
地
址
数
据
数据
Cache
执行单元EU

超标量superscalar处理器

- 定义：超标量处理器一个周期可发射并执行多条指令；
  这些指令出自同一个顺序排列的指令流，接受动态调度
  - 多功能部件：每类功能部件 ≥ 1 个，实现同类运算的并行
  - 乱序*out-of-order*执行：实际执行顺序与程序中指令排列顺序不一致
- 分支预测*branch prediction*与投机执行*speculative execution*
  - 不确定条件是否满足就先开始执行
  - 但结果不写至寄存器或内存，直到确认分支预测正确
  - 错误预测有性能开销
- 好处：程序员什么也不用做，超标量处理器可自行利用程序普遍具有的指令级并行性*ILP* 提升性能
- 现代 CPU 多数都是超标量的
  - Intel 从 Pentium（1993）开始（乱序始自 Pentium Pro，1995）

流水线功能单元

```
long mult_eg(long a, long b, long c)
{
    long p1 = a * b;
    long p2 = a * c;
    long p3 = p1 * p2;
    return p3;
}
```



空间	时间						
	1	2	3	4	5	6	7
流水段 1	a*b	a*c			p1*p2		
流水段 2		a*b	a*c			p1*p2	
流水段 3			a*b	a*c			p1*p2

- 把计算分解为若干阶段
- 每一阶段完成计算的特定部分，经过所有阶段即完成运算
- 一旦将中间结果值交给阶段 $i+1$，阶段 i 即可开始新的计算
- 例：虽然每个乘法需 3 个周期，在 7 个周期里可完成 3 次乘法

示例机型： Intel Core i7 Haswell

- 8 个功能部件（#0 - #7），可并行执行多条指令：
  - 2 个 load（#2、#3），各自独立包括地址计算
  - 2 个 store（#4、#7），#7 专门计算写操作地址，#4 专门完成写操作）
  - 4 个整数运算（#0、#1、#5、#6），算术、位操作等
  - 2 个浮点乘（#0、#1）
  - 1 个整数乘（#1）
  - 1 个浮点加（#1）
  - 1 个浮点除（#0）

- 延迟：完成某运算所需要的总周期数
- 容量：能够执行某运算的功能部件数
- 发射时间：连续 2 次同类型运算间隔最小周期数

- 某些指令执行需要多于 1 个周期，但能够被流水化

指令	延迟 *latency*	发射时间
load / store	4	1
整数加	1	1
整数乘	3	1
integer / long 除	3-30	3-30
单、双精度浮点加	3	1
单、双精度浮点乘	5	1
单、双精度浮点除	3-15	3-15

观察结论：
- 浮点加、乘比定点只慢一点点
- 除法比较耗时
- 除法延迟与具体数有关，且定点除不一定快于浮点除

combine4 的 x86-64 编译

- 内层循环（以整数乘为例：acc = acc * data[i];）

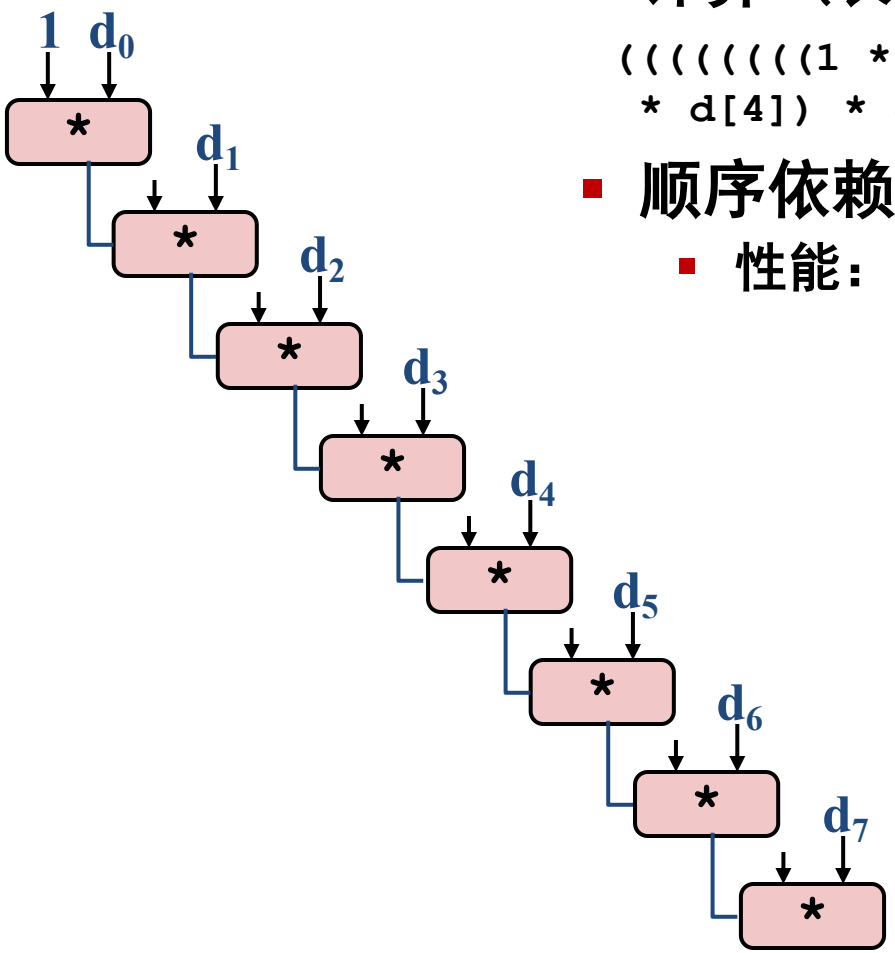
```
.L519:                                  # %rbp 的内容为数组长度
  imull (%rax,%rdx,4), %ecx             # t = t * d[i]
  addq  $1, %rdx                        # i++
  cmpq  %rdx, %rbp                      # 比较长度 : i
  jg    .L519                           # 大于则循环
```

方法	整型类		浮点类	
操作	+	*	+	*
combine4	1.27	3.01	3.01	5.01
延迟界限	1.00	3.00	3.00	5.00

mul 部件的流水和超标量并不能减少关键路径

延迟界限：必须严格按先后顺序完成 combine 的函数所需要的最小 CPE 值（本例中等于相应功能单元的延迟，见 § 5.7.3）

combine4 = 串行计算（OP = *）



- 计算（长度 = 8）

```
((((((((1 * d[0]) * d[1]) * d[2]) * d[3])
 * d[4]) * d[5]) * d[6]) * d[7])
```

- 顺序依赖性
  - 性能：由 OP 的延迟决定

循环展开（2 x 1）

```
void combine5(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    long limit = length-1;
    data_t *data = get_vec_start(v);        // p.354
    data_t acc = IDENT;

    /* 每次合并 2 个元素 */
    for (i = 0; i < limit; i += 2) {
        acc = (acc OP data[i]) OP data[i+1];
    }
    /* 计算剩余的元素 */
    for (; i < length; i++) {
        acc = acc OP d[i];
    }
    *dest = acc;
}
```

意义：①减少了循环控制变量 i 的修改次数、i 与 limit 的比较次数、条件转移的执行次数（其效果对于整数加尤为明显）
②凸显关键路径，指明优化方向

- 每次迭代的有效工作量 2 倍于此前

循环展开的效果

方法	整型类		浮点类	
操作	+	*	+	*
combine4	1.27	3.01	3.01	5.01
combine5 (2x1)	1.01	3.01	3.01	5.01
延迟界限	1.00	3.00	3.00	5.00

- 对整数加法有帮助
  - 降低到延迟界限
  - 原因：减少了循环开销（循环变量修改、判断次数）
  - 相对延迟为 1 的整数加，循环开销不能忽略
- 对其它运算没有改进
  - 原因：顺序依赖关系没有变，且其延迟为 3-5

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

循环展开（变换结合方式，记作 2 x 1a）

```
void combine7(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    long limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;
    /* 每次合并 2 个元素 */
    for (i = 0; i < limit; i += 2) {
       acc = acc OP (data[i] OP data[i+1]);
    }
    /* 计算剩余的元素 */
    for (; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = x;
}
```

与之前（2 x 1）比较：

```
acc = (acc OP data[i]) OP data[i+1];
```

- 会改变运算结果吗？整型不会，浮点会（结合律）
- 因此，编译器不会做优化，由程序员负责优化

变换结合方式的影响

方法	整型类		浮点类	
操作	+	*	+	*
combine4	1.27	3.01	3.01	5.01
combine5 (2x1)	1.01	3.01	3.01	5.01
combine7 (2x1a)	1.01	1.51	1.51	2.51
延迟界限	1.00	3.00	3.00	5.00
吞吐量界限	0.50	1.00	1.00	0.50

吞吐量界限：CPE 的最小界限

- 较 2 x 1 实现 2 倍速度提升：int *, FP +, FP *
  - 原因：打破了顺序依赖

```
acc = acc OP (data[i] OP data[i+1]);
```

  - 具体情况见下页

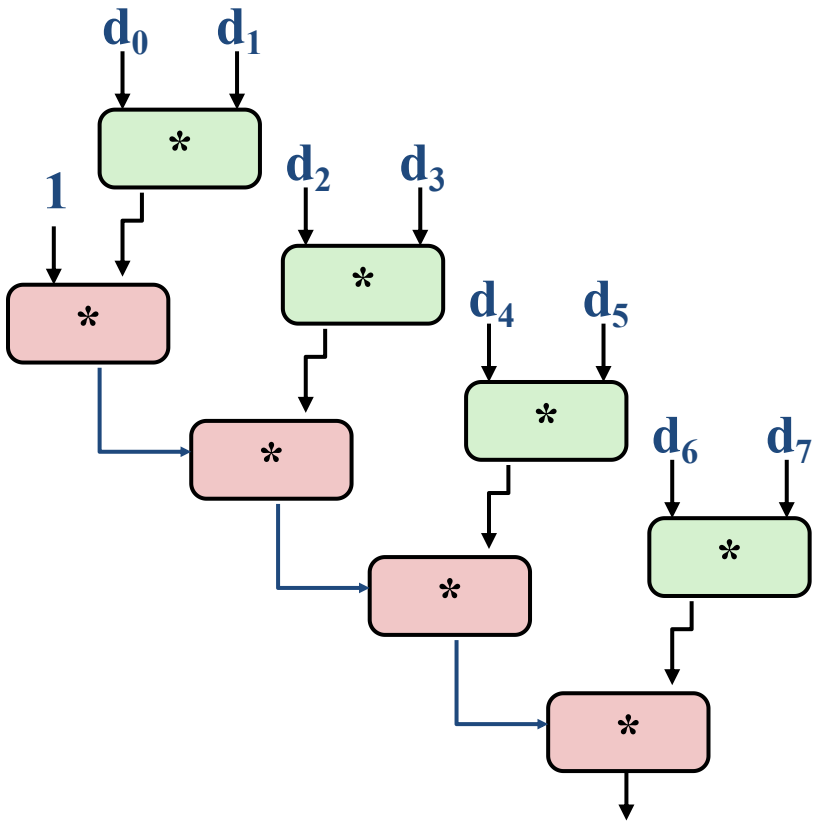
2 个流水浮点乘单元
2 个 load 单元

4 个整数加单元
但仅有 2 个 load 单元

改变结合方式后的计算

2个 load 单元操作时间与乘法完全并行

```
acc = acc OP (data[i] OP data[i+1]);
```



- 发生了哪些改变：
  - 结合图 5-29 分析
  - 对于浮点乘，本轮循环的第 2 乘与下轮循环的第 1 乘完全并行（超标量）
  - 对于定点乘，本轮循环的第 2 乘与下轮循环的第 1 乘基本并行（流水）
- 整体性能
  - 设有 *N* 个元素，每个操作 *D* 个周期延迟
  - (*N*/2+1)***D* 个周期：

    CPE = D/2

循环展开：使用多累加器（2 x 2）

- 重组的不同形式

```
void combine6(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    long limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc0 = IDENT;
    data_t acc1 = IDENT;
    /* 每次合并 2 个元素 */
    for (i = 0; i < limit; i += 2) {
        acc0 = acc0 OP data[i];
        acc1 = acc1 OP data[i+1];
    }
    /* 计算剩余的元素 */
    for (; i < length; i++) {
        acc0 = acc0 OP data[i];
    }
    *dest = acc0 OP acc1;
}
```

采用多累加器的效果

方法	整型类		浮点型	
操作	+	*	+	*
combine4	1.27	3.01	3.01	5.01
combine5 (2x1)	1.01	3.01	3.01	5.01
combine7 (2x1a)	1.01	1.51	1.51	2.51
combine6 (2x2)	0.81	1.51	1.51	2.51
延迟界限	1.00	3.00	3.00	5.00
吞吐量界限	0.50	1.00	1.00	0.50

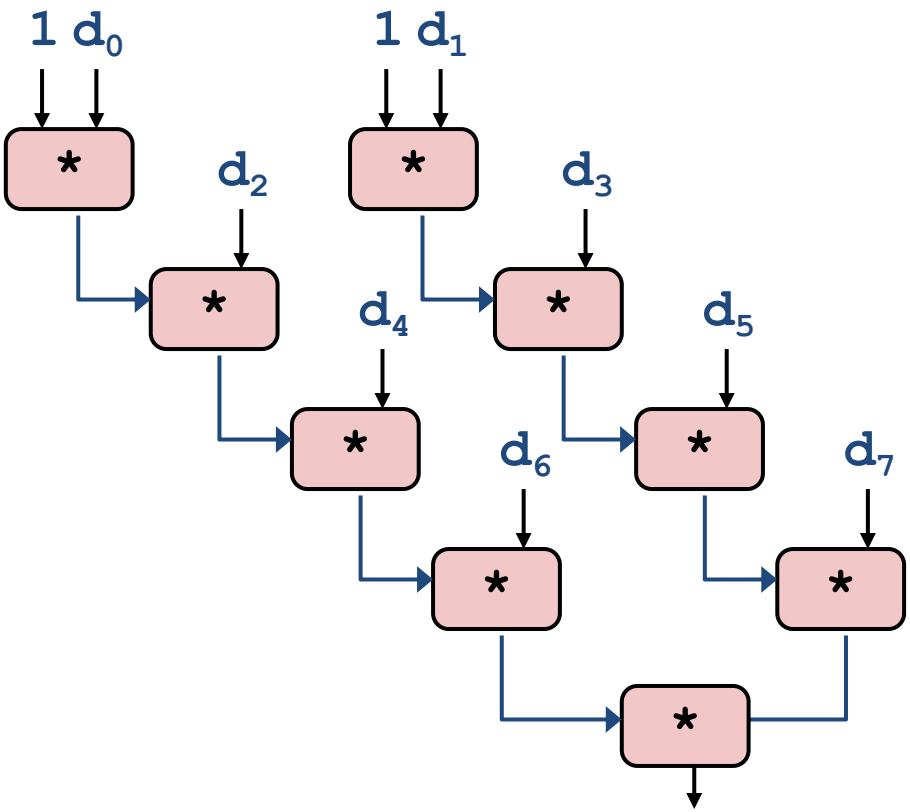
- 整数加法用到 2 个 load 单元

```
acc0 = acc0 OP data[i];
acc1 = acc1 OP data[i+1];
```

- 较 2 x 1 实现 2 倍速度提升：int *、FP +、FP *

多累加器

```
acc0 = acc0 OP data[i];
acc1 = acc1 OP date[i+1];
```



- 发生了哪些改变：
  - 两个独立的“操作流”
- 整体性能
  - 设有 *N* 个元素，每个操作 *D* 个周期延迟：
  - 应为 (*N*/2 + 1) * *D* 个周期
  - CPE = *D*/2
  - CPE 与预测匹配！

接下来呢？

循环展开 & 累加

- 思想
  - 可以任意因子 *L* 进行循环展开
  - 可并行累加 *K* 个结果
  - 可采用多种 *L* – *K* 组合，但 *L* 应是 *K* 的整数倍
- 只有保持能够执行该操作的所有功能单元的流水线都是满的，程序才能达到这个操作的吞吐量界限
- 局限性
  - 效果/收益递减 Diminishing returns
    - 不能超出执行单元的吞吐量限制
  - 当向量长度较短时开销较大
    - 顺序地完成循环

循环展开 & 累加：double *

- 案例
  - Intel Haswell
  - double 浮点乘法
  - 延迟界限：5.00；吞吐量界限：0.50

FP *	循环展开因子 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	
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

累加器数量

循环展开 & 累加：int +

- 案例
  - Intel Haswell
  - int 加法
  - 延迟界限：1.00，吞吐量界限：1.00

int +	循环展开因子 *L*							
累加器数量 *K*	1	2	3	4	6	8	10	12
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69		0.54		
3			0.74					
4				0.69		1.24		
6					0.56			0.56
8						0.54		
10							0.54	
12								0.56

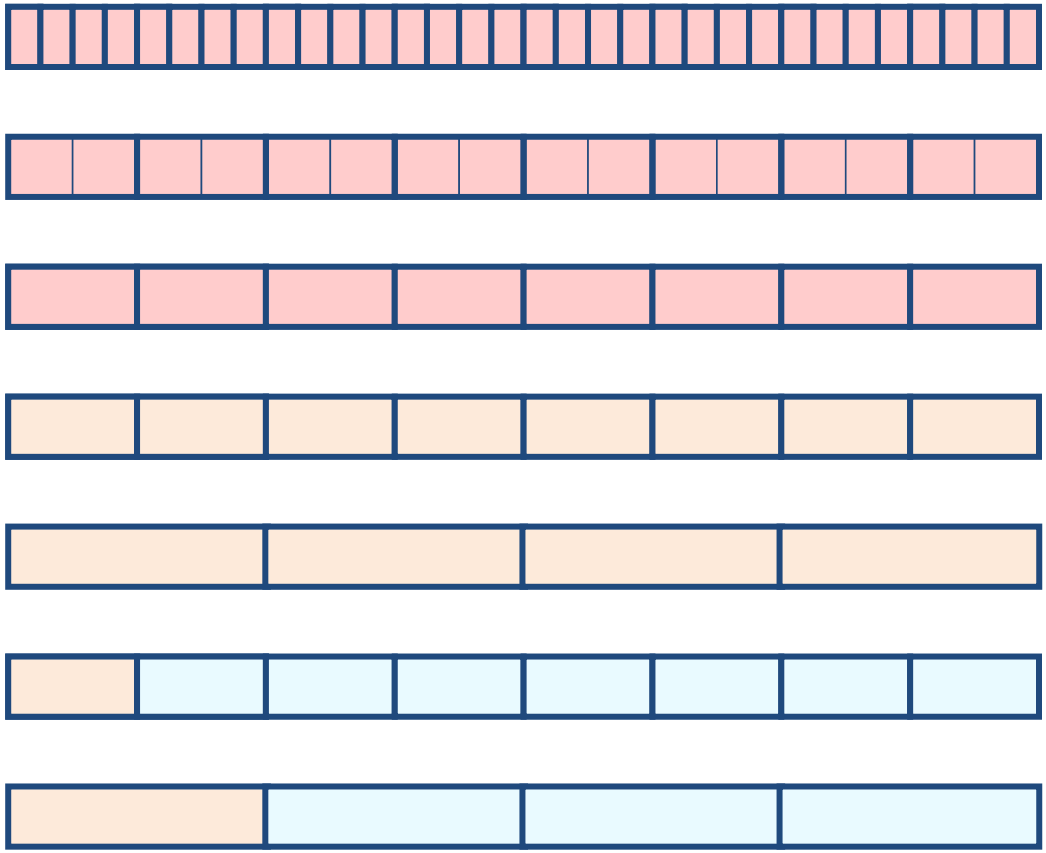
可达到的性能

- 只受功能单元的吞吐量限制
- 比初始的代码性能提高了 42 倍（图 5-25）

方法	整型类		浮点类	
操作	+	*	+	*
最优	0.54	1.01	1.01	0.52
延迟界限	1.00	3.00	3.00	5.00
吞吐量界限	0.50	1.00	1.00	0.50

用 AVX2 编程

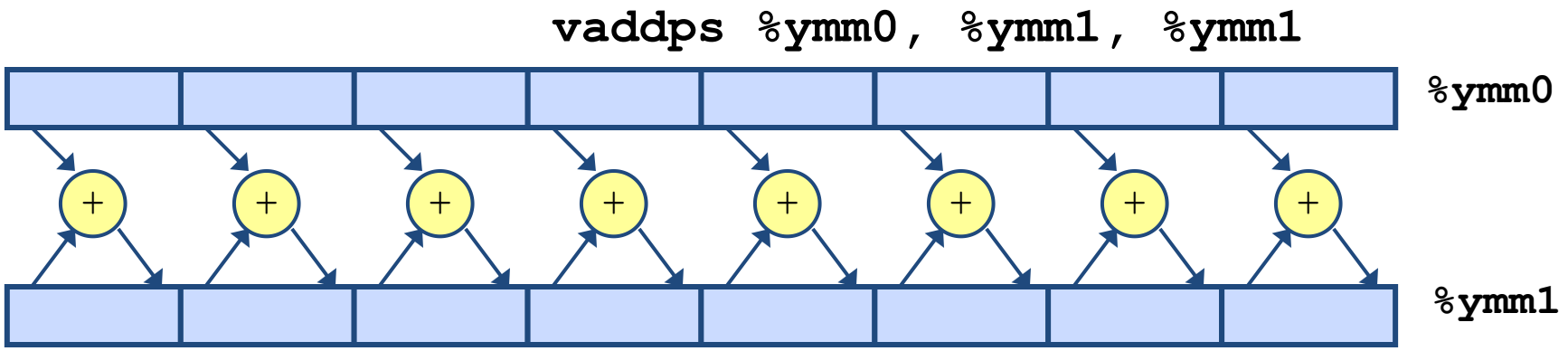
- YMM 寄存器：16 个（%ymm0 - %ymm15）每个 32 B
  - 32 个单字节整数
  - 16 个 16 位整数
  - 8 个 32 位整数
  - 8 个单精度浮点数
  - 4 个双精度浮点数
  - 1 个单精度浮点数
  - 1 个双精度浮点数



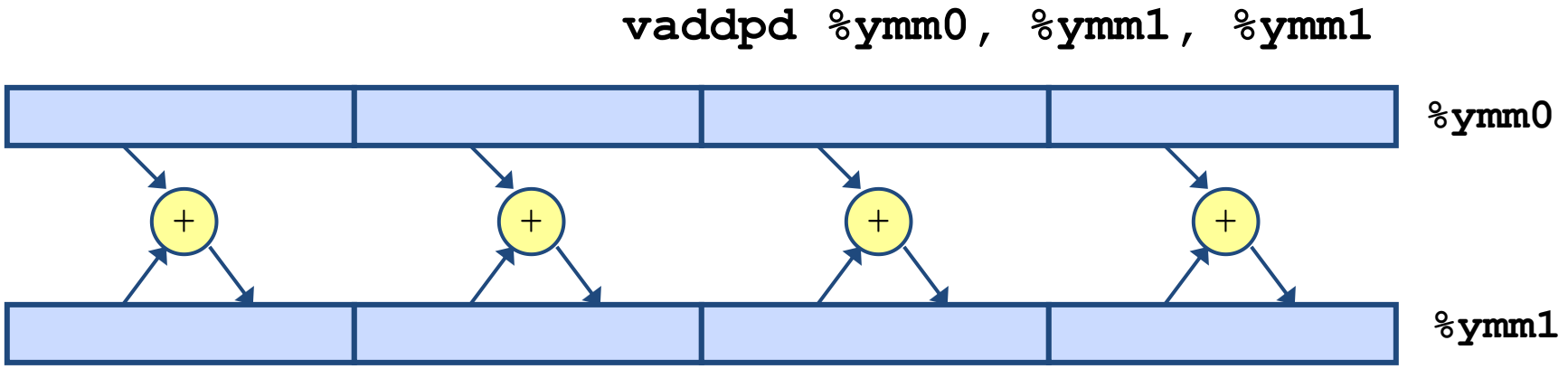
SIMD 操作

vaddpd：AVX add packed double-precision FP values

- SIMD 操作: 单精度



- SIMD 操作: 双精度



使用向量指令

方法	整型		双精度浮点型	
操作	+	*	+	*
标量最优	0.54	1.01	1.01	0.52
向量最优	0.06	0.24	0.25	0.16
延迟界限	0.50	3.00	3.00	5.00
吞吐量界限	0.50	1.00	1.00	0.50
向量吞吐量界限	0.0625	0.125	0.25	0.125

- 使用 AVX 指令
  - 多数据元素并行操作
  - 阅读网络旁注 OPT:SIMD（CS:APP 配套网站）

分支怎么处理？

- 挑战
  - 指令控制单元ICU 的工作进度远远超过执行单元EU，以产生足够的操作来使后者保持繁忙

```
404663:   mov     $0x0, %eax
404668:   cmp     (%rdi), %rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi), %rax

    . . .

404685:   repz retq ;repz可忽略
```

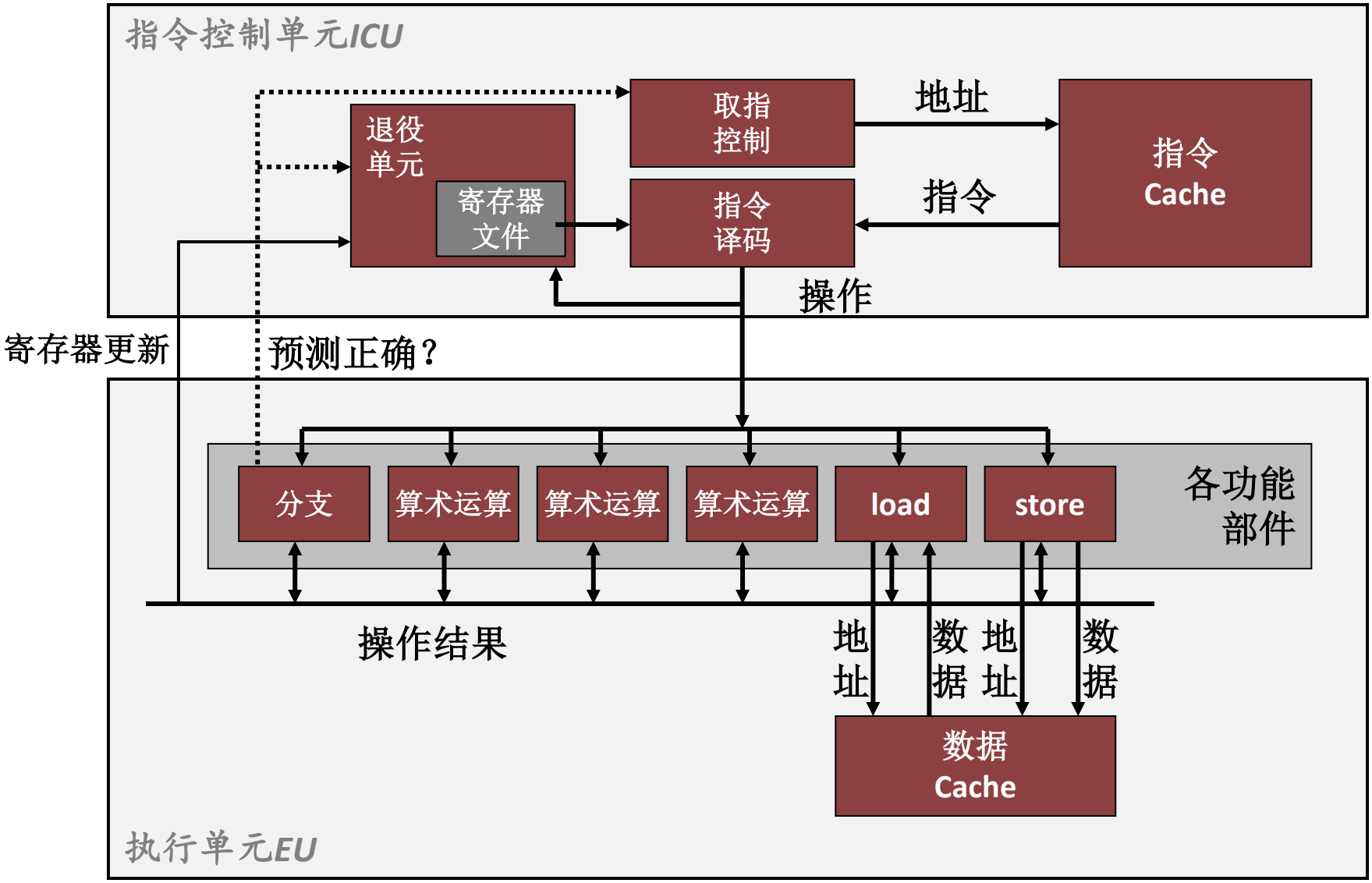
执行中

该如何继续？

见 §3.6.4 旁注

  - 遇到条件分支时，无法可靠地确定继续取指的位置

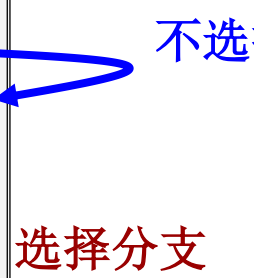
现代 CPU 设计


指令控制单元ICU
退役
单元
寄存器
文件
取指
控制
地址
指令
Cache
指令
指令
译码
操作
寄存器更新
预测正确？
分支
算术运算
算术运算
算术运算
load
store
各功能
部件
操作结果
地
址
数
据
地
址
数
据
数据
Cache
执行单元EU

分支的结果

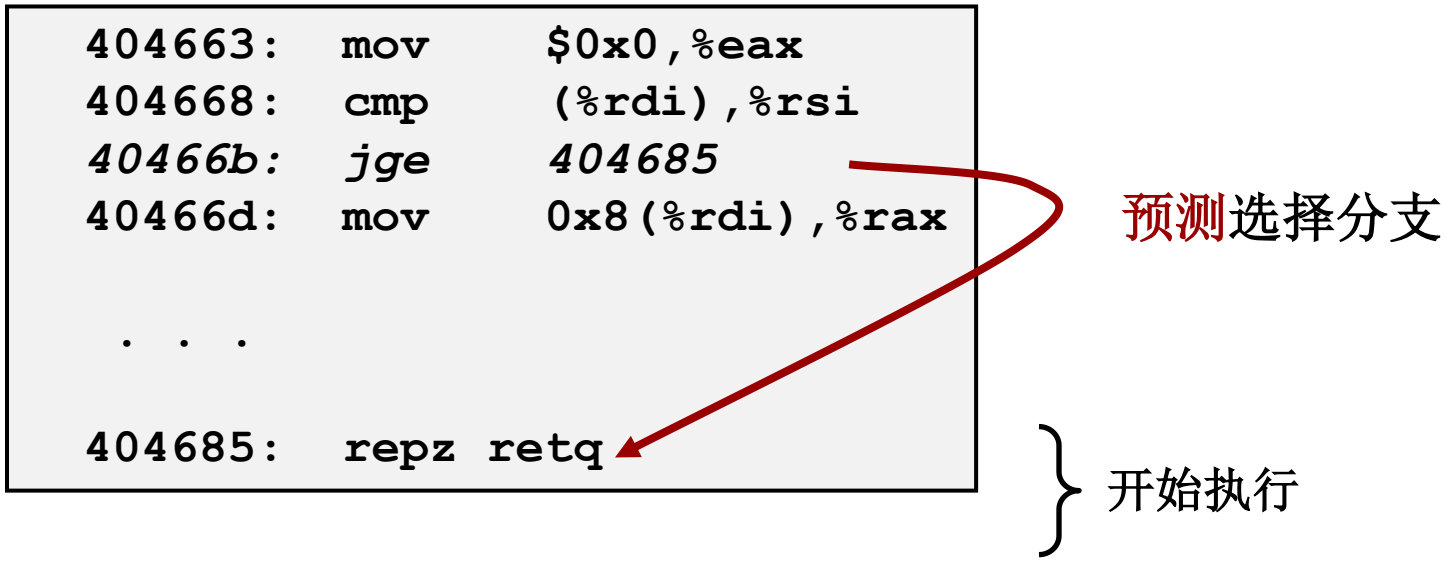
- 当遇到条件分支指令时，无法确定继续取指的位置
  - 选择分支：将控制转移到分支目标处执行
  - 不选择分支：继续执行下一条指令（fall through）
- 直到分支/整数单元的结果确定下来之后才能解决

```
404663:   mov    $0x0, %eax
404668:   cmp    (%rdi), %rsi
40466b:   jge    404685
40466d:   mov    0x8(%rdi), %rax

  . . .

404685:   repz retq
```



分支预测

- 思想
  - 猜测会走哪一分支
  - 在预测的位置开始执行指令
    - 但是，不要实际修改寄存器或内存中的数据



循环过程中的分支预测

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```

i = 98

假定
向量长度 = *100*

预测选择分支（正确）

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```

i = 99

预测选择分支（糟糕！）

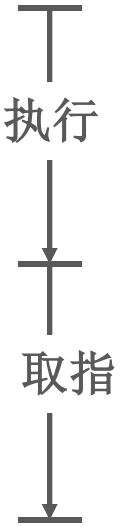
```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```

i = 100

取指位置
出错

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```

i = 101



错误分支预测的失效

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```

i = 98

假定
向量长度 = 100

预测选择分支(OK)

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```

i = 99

预测选择分支(糟糕!)

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```

i = 100

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```

i = 101

令所有错误操作失效

错误分支预测的恢复

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
401036:   jmp     401040
  . . .
401040:   vmovsd  %xmm0,(%r12)
```

i = 99

确定不会采纳

重新加载流水线

- 性能开销
  - 现代处理器上的多个时钟周期
  - 可能成为一个主要的性能限制因素

获得高性能

- 采用好的编译器和优化选项（如 GCC 的 -O3 选项）
- 谨慎行事
  - 留意导致算法效率低下的潜在问题
  - 编写“编译器友好型”代码
    - 留意优化障碍
    - 如：函数调用、存储器引用
  - 密切关注最内层的循环（多数工作在此完成）
- 根据机器优化代码
  - 利用指令级并行
  - 避免不可预测的分支
  - 编写 Cache 友好型代码 （在后续章节介绍）