

# **Code Optimization II : Machine-dependent Optimizations**

COMP400727: Introduction to Computer Systems

**Hao Li**  
**Xi'an Jiaotong University**

# Today

- **Machine-Dependent Optimizations**
- Instruction-Level Parallelism
- Branch Predictions

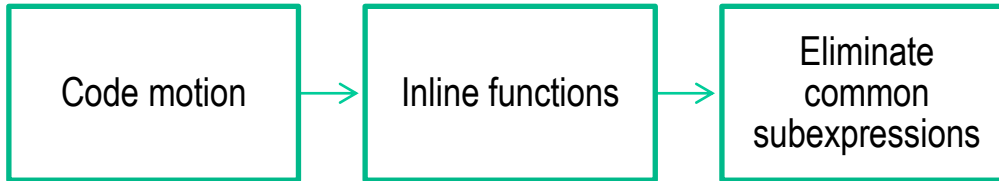
# Multiple Levels of Optimizations

## *Human-level optimization*



Preprocessing

## *Machine-independent optimization*



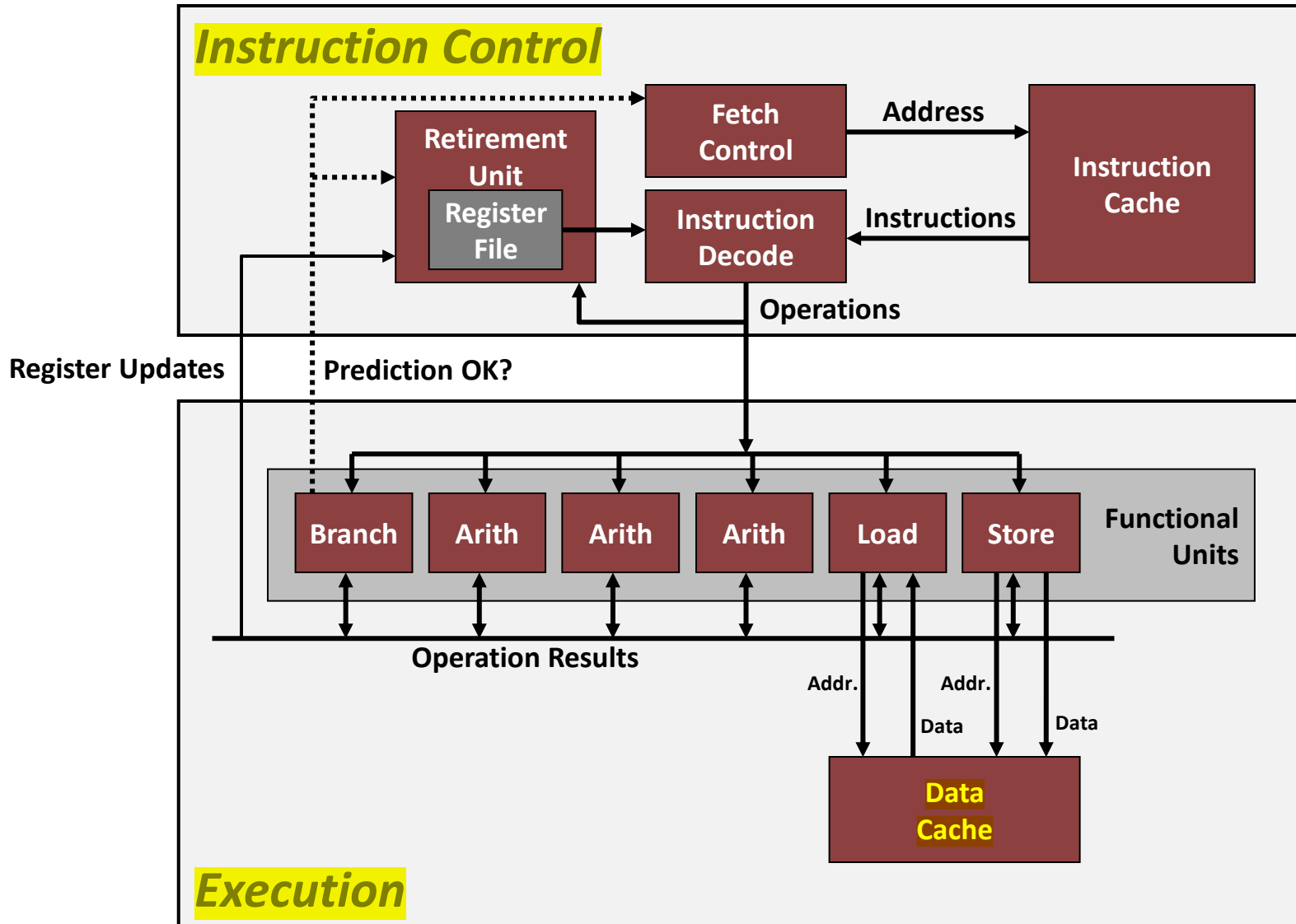
## *Machine-dependent optimization*



Compilation

Assembling

# Modern CPU Design



# Today

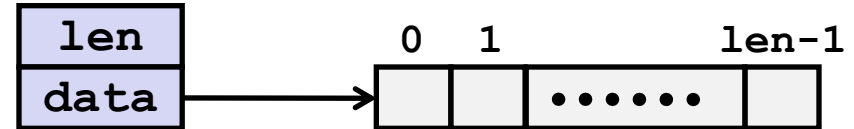
- Machine-Dependent Optimizations
- **Instruction-Level Parallelism**
- Branch Predictions

# Exploiting Instruction-Level Parallelism

- **Need general understanding of modern processor design**
  - Hardware can execute multiple instructions in parallel
- **Performance limited by data dependencies**
- **Simple transformations can cause big speedups**
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic

# Benchmark Example: Data Type for Vectors

```
/* data structure for vectors */  
typedef struct{  
    size_t len;  
    data_t *data;  
} vec;
```



## ■ Data Types

- Use different declarations for data\_t
- int
- long
- float
- double

```
/* retrieve vector element  
and store at val */  
int get_vec_element  
(*vec v, size_t idx, data_t *val)  
{  
    if (idx >= v->len)  
        return 0;  
    *val = v->data[idx];  
    return 1;  
}
```

# Benchmark Computation

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

Compute sum or  
product of vector  
elements

## ■ Data Types

- Use different declarations for `data_t`
- `int`
- `long`
- `float`
- `double`

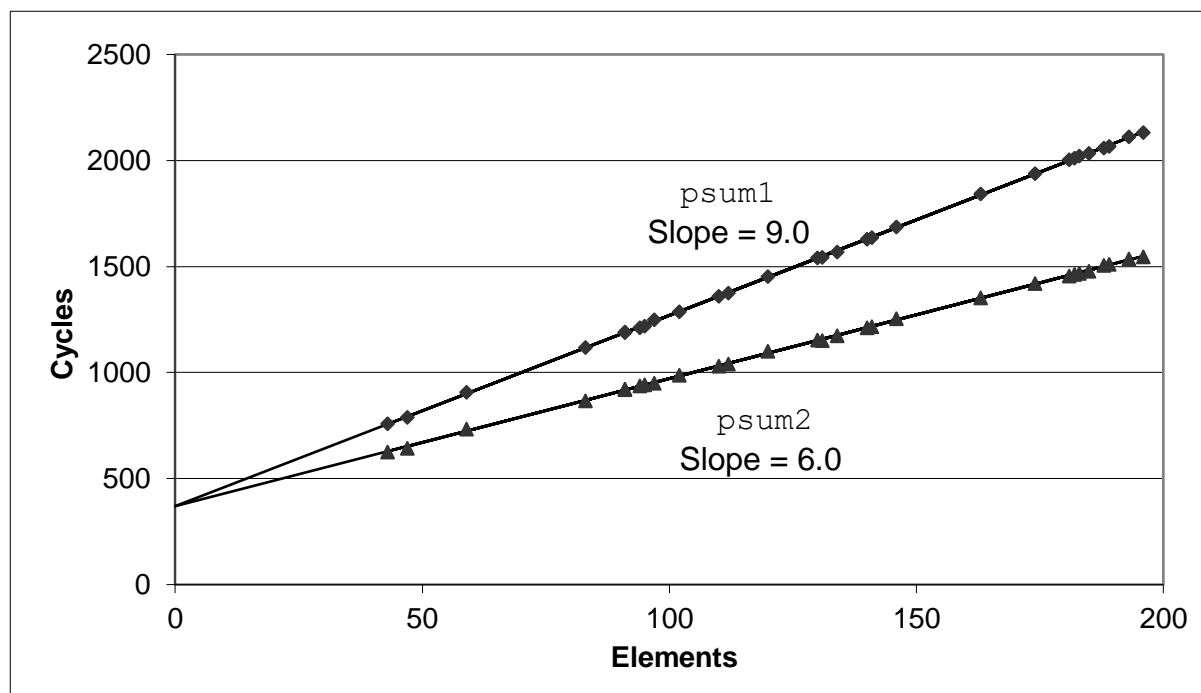
## ■ Operations

- Use different definitions of `OP` and `IDENT`
- `+` / `0`
- `*` / `1`



# Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length =  $n$
- In our case: **CPE = cycles per OP**
- $\text{Cycles} = \text{CPE} * n + \text{Overhead}$ 
  - CPE is slope of line 斜率



# Benchmark Performance

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

Compute sum or  
product of vector  
elements

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 -O1	10.12	10.12	10.17	11.14
Combine1 -O3	4.5	4.5	6	7.8

Results in CPE (cycles per element)

# Basic Optimizations

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

- Move vec\_length out of loop 不然每次循环都需要调用get\_length()函数
- Avoid bounds check on each cycle
- Accumulate in temporary 减少访存次数

# Effect of Basic Optimizations

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

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 -O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

- What are the bounds?

# Latency Bounds

## *Instruction*

Integer Add

Integer Multiply

Integer/Long Divide

Single/Double FP Multiply

Single/Double FP Add

Single/Double FP Divide

## *Latency*

1

3

3-30

5

3

3-15

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 -O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

- Why **1.27**, instead of **1.0**, for Int+?

# Overhead in Loop

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

# Loop Unrolling (2x1)

```

void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}

```

l=0: 0 1  
l=2: 2 3  
l=4: 4 5  
.....

- Perform 2x more useful work per iteration

# Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
<i>Latency Bound</i>	<i>1.00</i>	<i>3.00</i>	<i>3.00</i>	<i>5.00</i>

## ■ Helps integer add

- Achieves latency bound 循环展开：有利于达到物理下限

## ■ Others don't improve

## ■ Can we break the latency bound?

pipeline：突破物理下限



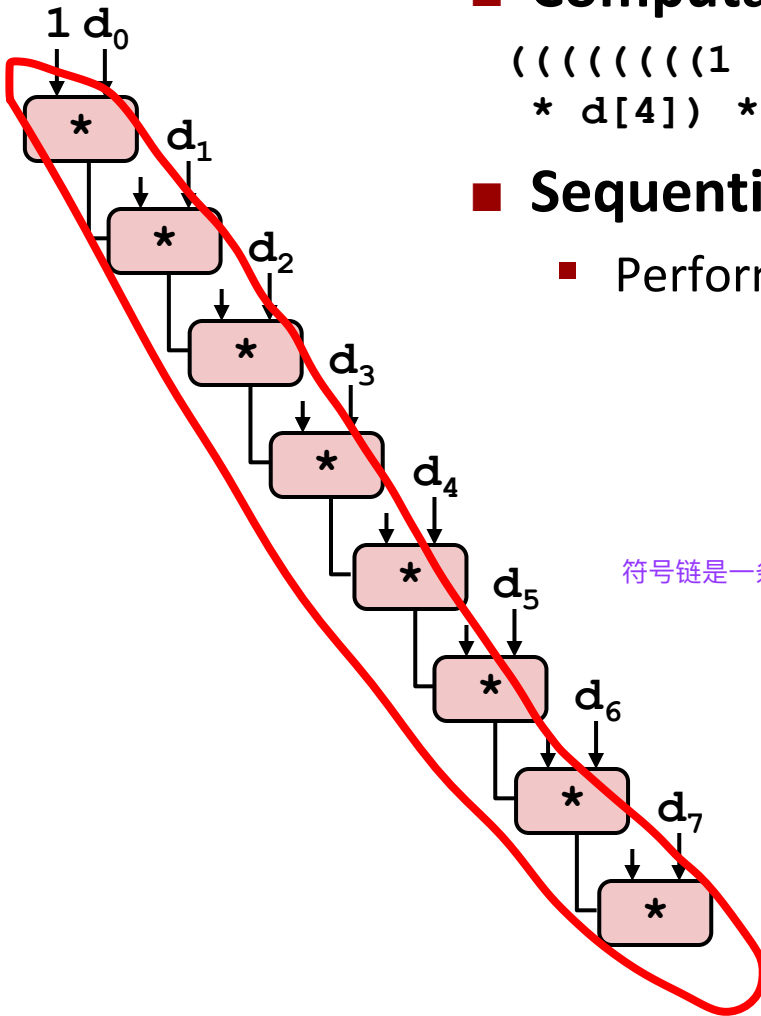
# Combine4 = Serial Computation (OP = \*)

## ■ Computation (length=8)

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

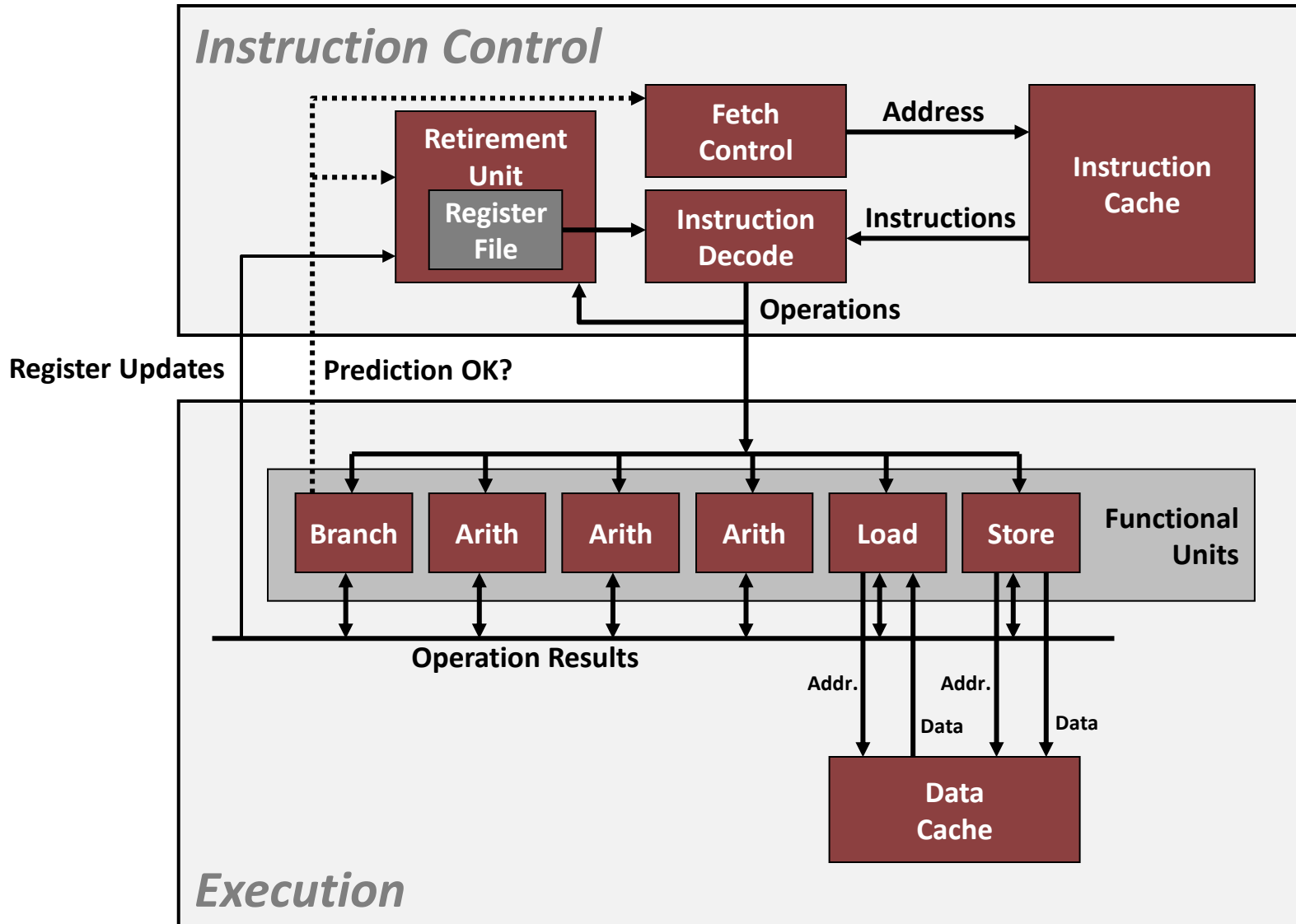
## ■ Sequential dependence

- Performance: determined by latency of OP



符号链是一条，所以CPE 是 D/1

# Modern CPU Design



超标量 (superscalar) CPU架构是指在一颗处理器内核中实行了指令级并发的一类并发运算

<https://zh.wikipedia.org/wiki/%E8%B6%85%E7%B4%94%E9%87%8F>

# Superscalar Processor

- **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

未实现超标量体系结构时，CPU在每个时钟周期仅执行单条指令，因此仅有一个执行单元在工作，其它执行单元空闲。超标量体系结构的CPU在一个时钟周期可以同时分派 (dispatching) 多条指令在不同的执行单元中被执行，这就实现了指令级的并行。超标量体系结构可以视作多指令流多数据流

- **Benefit:** without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have
  - Type 1: Pipeline execution
  - Type 2: Parallel execution unit
- **Most modern CPUs are superscalar.**
  - Intel: since Pentium (1993)

# Instruction Level Parallelism

- **Type 1: Some instructions take  $> 1$  cycle, but can be pipelined**

<i>Instruction</i>	<i>Latency</i>	<i>Cycles/Issue</i>
Load / Store	4	1
Integer Multiply	3	1
<b>Integer/Long Divide</b>	<b>3-30</b>	<b>3-30</b>
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
<b>Single/Double FP Divide</b>	<b>3-15</b>	<b>3-15</b>

- **Type 2: Multiple instructions can execute in parallel**

- Haswell

2 load, with address computation

1 store, with address computation

4 integer

2 FP multiply

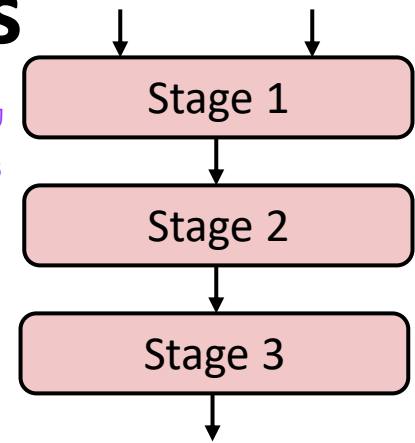
1 FP add

1 FP divide

# Type 1: Pipelined Functional Units

```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2; need P1 and P2 are all over!
    return p3;
}
```

Stage需要  
三个是因为  
乘法的  
latency = 3



	Time						
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

- Divide computation into stages
- Pass **partial computations from stage to stage**
- Stage i can start on new computation once values passed to i+1
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

原来:  $3 \times 3 = 9$   
现在: pipeline  $\Rightarrow 7$

# Throughput Bound

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
<i>Latency Bound</i>	<i>1.00</i>	<i>3.00</i>	<i>3.00</i>	<i>5.00</i>
<i>Throughput Bound</i>	<i>0.50</i>	<i>1.00</i>	<i>1.00</i>	<i>0.50</i>

4 func. units for int +,  
**2 func. units for load**  
 Why Not .25?

1 func. unit for FP +  
 3-stage pipelined FP +

2 func. units for FP \*,  
 2 func. units for load  
 5-stage pipelined FP \*

# Loop Unrolling with Reassociation (2x1a)

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Compare to before

$x = (x \text{ OP } d[i]) \text{ OP } d[i+1];$

# Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
<i>Latency Bound</i>	<i>1.00</i>	<i>3.00</i>	<i>3.00</i>	<i>5.00</i>
<i>Throughput Bound</i>	<i>0.50</i>	<i>1.00</i>	<i>1.00</i>	<i>0.50</i>

## ■ Nearly 2x speedup for Int \*, FP +, FP \*

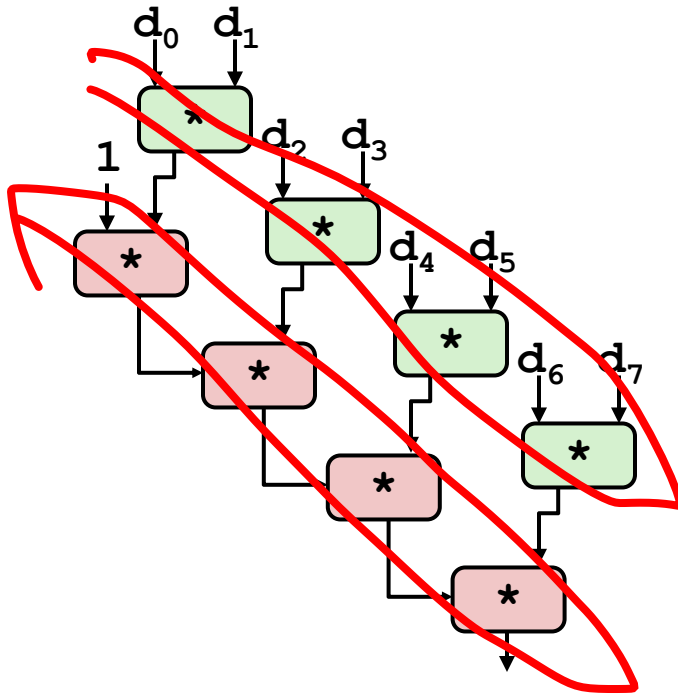
- Reason: Breaks sequential dependency

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



# Reassociated Computation

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



## What changed:

- Ops in the next iteration can be started early (no dependency)

## Overall Performance

- N elements, D cycles latency/op
- $(N/2+1)*D$  cycles:  
 **$CPE = D/2$**

?

符号链是2条，所以CPE是D/2

# Loop Unrolling with Separate Accumulators (2x2)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- Different form of reassociation

# Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
<i>Latency Bound</i>	<i>1.00</i>	<i>3.00</i>	<i>3.00</i>	<i>5.00</i>
<i>Throughput Bound</i>	<i>0.50</i>	<i>1.00</i>	<i>1.00</i>	<i>0.50</i>

- Int + makes use of two load units

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

- 2x speedup (over unroll2) for Int \*, FP +, FP \*

# Separate Accumulators

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

## ■ What changed:

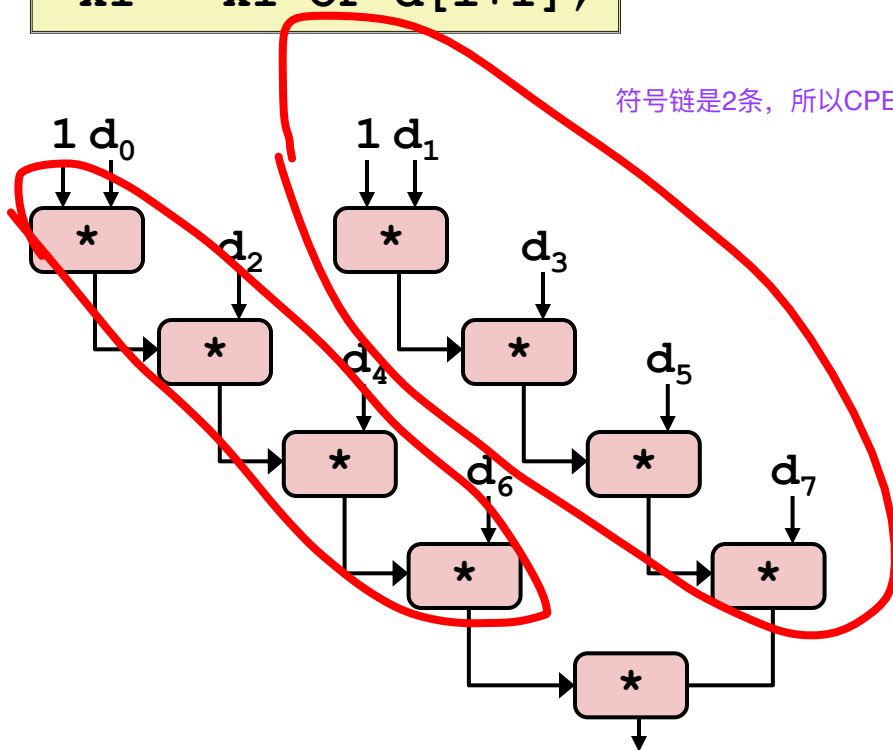
- Two independent “streams” of operations

## ■ Overall Performance

- N elements, D cycles latency/op
- Should be  $(N/2+1)*D$  cycles:  
 **$CPE = D/2$**
- **CPE matches prediction!**

***What Now?***

符号链是2条，所以CPE此时是D/2



# Unrolling & Accumulating

## ■ Idea

- Can unroll to any degree  $L$
- Can accumulate  $K$  results in parallel
- $L$  must be multiple of  $K$

## ■ Limitations

- Diminishing returns
  - Cannot go beyond throughput limitations of execution units
- Large overhead for short lengths
  - Finish off iterations sequentially

# Unrolling & Accumulating: Double \*

## ■ Case

- Intel Haswell
- Double FP Multiplication
- Latency bound: 5.00. Throughput bound: 0.50

<i>Accumulators</i>	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	
	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

# Achievable Performance

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

- Limited only by throughput of functional units
- Up to 42X improvement over original, unoptimized code

*Can we do even better?*

# Programming with AVX2

## YMM Registers

■ 16 total, each 32 bytes

■ 32 single-byte integers



■ 16 16-bit integers



■ 8 32-bit integers



■ 8 single-precision floats



■ 4 double-precision floats



■ 1 single-precision float



■ 1 double-precision float

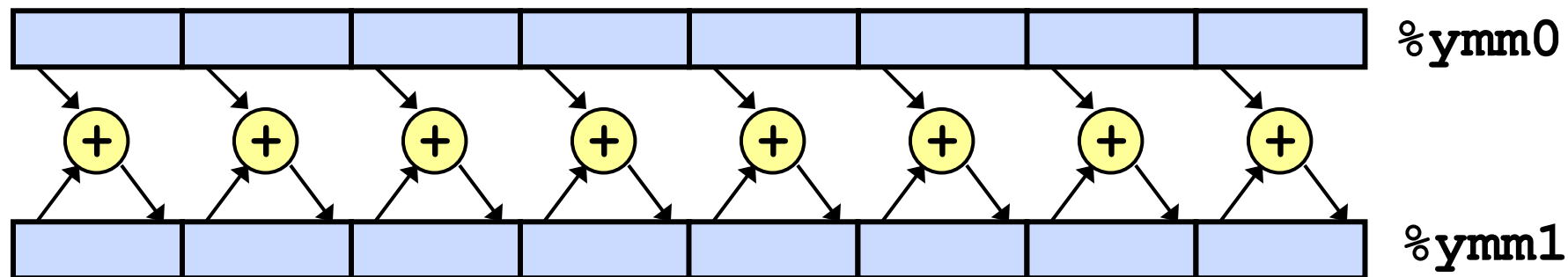




# SIMD Operations

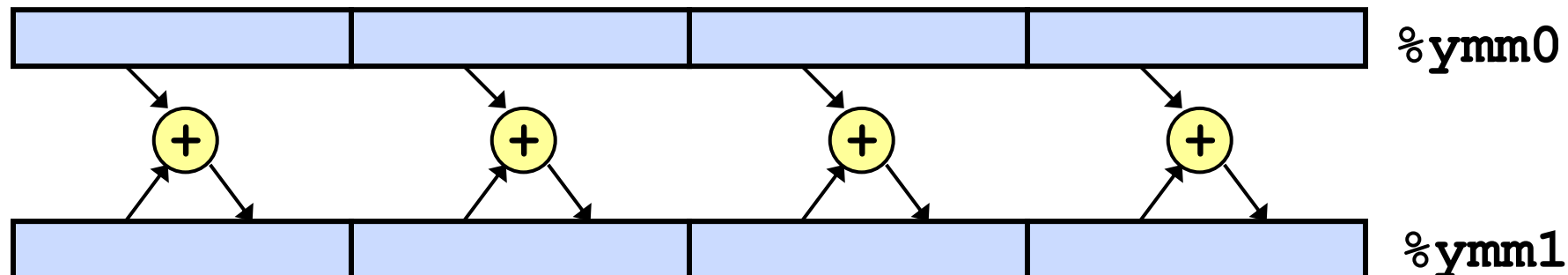
## ■ SIMD Operations: Single Precision

**vaddps %ymm0, %ymm1, %ymm1**



## ■ SIMD Operations: Double Precision

**vaddpd %ymm0, %ymm1, %ymm1**



# Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
<i>Latency Bound</i>	<i>0.50</i>	<i>3.00</i>	<i>3.00</i>	<i>5.00</i>
<i>Throughput Bound</i>	<i>0.50</i>	<i>1.00</i>	<i>1.00</i>	<i>0.50</i>
<i>Vec Throughput Bound</i>	<i>0.06</i>	<i>0.12</i>	<i>0.25</i>	<i>0.12</i>

## ■ Make use of AVX Instructions

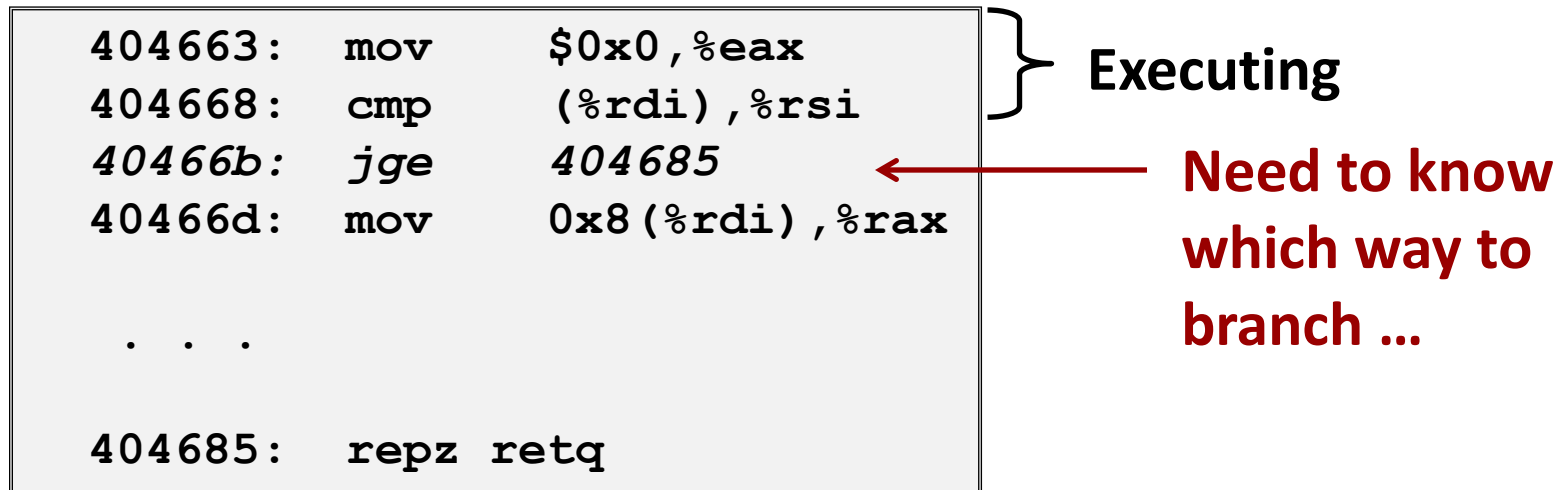
- Parallel operations on multiple data elements
- See Web Aside OPT:SIMD on CS:APP web page

# Today

- Machine-Dependent Optimizations
- Instruction-Level Parallelism
- **Branch Predictions**

# Branches Are A Challenge

- **Instruction Control Unit** must work well ahead of **Execution Unit** to generate enough operations to keep EU busy



If the CPU has to wait for the result of the `cmp` before continuing to fetch instructions, may waste tens of cycles doing nothing!

# Branch Prediction

## ■ *Guess which way branch will go*

- Begin executing instructions at predicted position
- But don't actually modify register or memory data

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

**Predict Taken**

**Continue  
Fetching  
Here**

# Branch Prediction Through Loop

```

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

```

***i = 98***

Assume  
array length = **100**

Predict Taken (OK)

```

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

```

***i = 99***

Predict Taken  
(Oops)

```

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

```

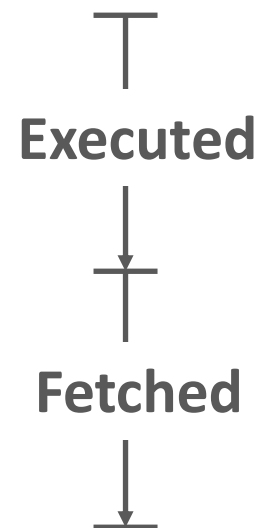
***i = 100***

Read  
invalid  
location

```

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

```

***i = 101***

# Branch Misprediction Invalidation

```

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

```

*i = 98*

Assume  
array length = **100**

Predict Taken (OK)

```

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

```

*i = 99*

Predict Taken  
(Oops)

```

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

```

*i = 100*

Invalidate

```

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

```

*i = 101*

# Branch Misprediction Recovery

```
401029:  mulsd    (%rdx), %xmm0, %xmm0
```

```
40102d:  add      $0x8, %rdx
```

```
401031:  cmp      %rax, %rdx
```

```
401034:  jne      401029
```

```
401036:  jmp      401040
```

```
. . .
```

```
401040:  movsd    %xmm0, (%r12)
```

*i = 99*

Definitely not taken

Reload  
Pipeline

## ■ Performance Cost

- Multiple clock cycles on modern processor
- Can be a major performance limiter



# Branch Prediction Numbers

## ■ A simple heuristic:

- Backwards branches are often loops, so predict taken
- Forwards branches are often ifs, so predict not taken
- >95% prediction accuracy just with this!

## ■ Fancier algorithms track behavior of each branch

- Subject of ongoing research
- 2011 record (<https://www.jilp.org/jwac-2/program/JWAC-2-program.htm>): 34.1 mispredictions per 1000 instructions
- Current research focuses on the remaining handful of “impossible to predict” branches (strongly data-dependent, no correlation with history)
  - e.g. [https://hps.ece.utexas.edu/pub/PruettPatt\\_BranchRunahead.pdf](https://hps.ece.utexas.edu/pub/PruettPatt_BranchRunahead.pdf)

# Optimizing for Branch Prediction

## ■ Loop Unrolling

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

# Optimizing for Branch Prediction

## ■ Transform Branches

```
for (int c=0; c < size; ++c)
{
    if (data[c] >= 128)
        sum += data[c];
}
```

```
.L4:
    mov     rdx, QWORD PTR [rax]
    cmp     rdx, 127
    jbe     .L3
    add     rcx, rdx
    mov     edi, 1
.L3:
    next loop
```

```
int t = (data[c] - 128) >> 31;
sum += ~t & data[c];
```

```
.L3:
    mov     rcx, QWORD PTR [rdx]
    lea     rax, [rcx-128]
    shr     rax, 31
    not     eax
    cdqe
    and     rax, rcx
    add     rsi, rax
    add     rdx, 8
    cmp     rdx, rdi
    jne     .L3
```

# Optimizing for Branch Prediction

## ■ Conditional Moves

```
int absdiff(int x, int y) {
    int result;
    if (x > y) result = x - y;
    else result = y - x;
    return result;
}
```

absdiff:

```
mov    eax, DWORD PTR [rbp-20]
cmp    eax, DWORD PTR [rbp-24]
jle    .L2
mov    eax, DWORD PTR [rbp-20]
sub    eax, DWORD PTR [rbp-24]
mov    DWORD PTR [rbp-4], eax
jmp    .L3
```

.L2:

```
mov    eax, DWORD PTR [rbp-24]
sub    eax, DWORD PTR [rbp-20]
mov    DWORD PTR [rbp-4], eax
```

.L3:

```
mov    eax, DWORD PTR [rbp-4]
```

absdiff:

```
mov    edx, edi
sub    edx, esi
mov    eax, esi
sub    eax, edi
cmp    edi, esi
cmovg  eax, edx
```

Not Always a  
Good Idea

# Optimizing for Branch Prediction

## ■ Make Branch More Predictable

```
for (int c=0; c < size; ++c) {
    if (data[c] >= 128)
        sum += data[c];
}
```

T = branch taken

N = branch not taken

```
data[] = 226, 185, 125, 158, 198, 144, 217, 79, 202, 118, 14, ...
branch =  T,   T,   N,   T,   T,   T,   T,   N,   T,   N,   N, ...

        = TTNTTTTNTNNTTT ...    (completely random)
```

```
data[] = 0, 1, 2, 3, 4, ... 126, 127, 128, 129, 130, ... 250, ...
branch = N  N  N  N  N  ...   N   N   T   T   T   ...   T, ...

        = NNNNNNNNNNNN ... NNNNNNNTTTTTTTTTT ... (easy to predict)
```

# Going Further

- **Compiler optimizations are an easy gain**
  - 20 CPE down to 3-5 CPE
  
- **With careful hand tuning and computer architecture knowledge**
  - 4-16 elements per cycle
  - Newest compilers are closing this gap

# Summary: Getting High Performance

- **Good compiler and flags**
- **Don't do anything sub-optimal**
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers:  
procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)
- **Tune code for machine**
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly