指令级并行

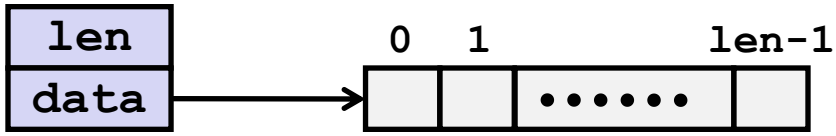
This slide is mostly from csapp course @ CMU.
http://csapp.cs.cmu.edu/3e/instructors.html

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 yield dramatic performance improvement
  - 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

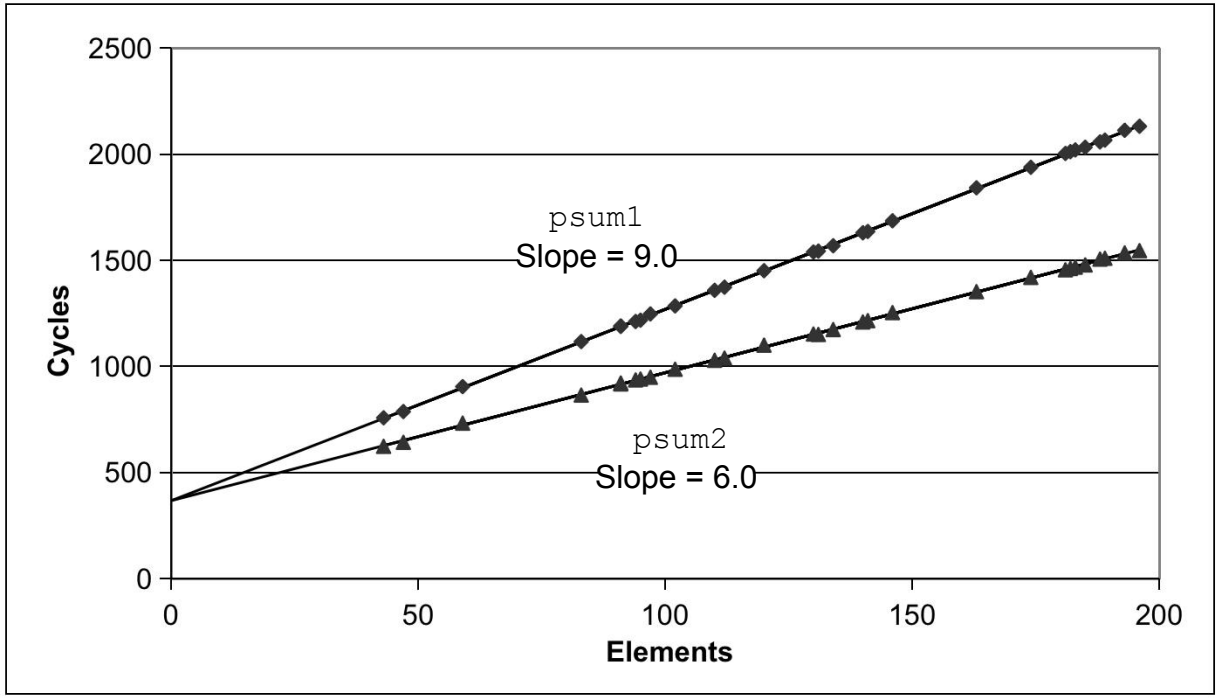
```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

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
- T = CPE*n + Overhead
  - CPE is slope of line



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

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
  - Avoid bounds check on each cycle
- Accumulate in temporary
- Programmer instead of compiler does it

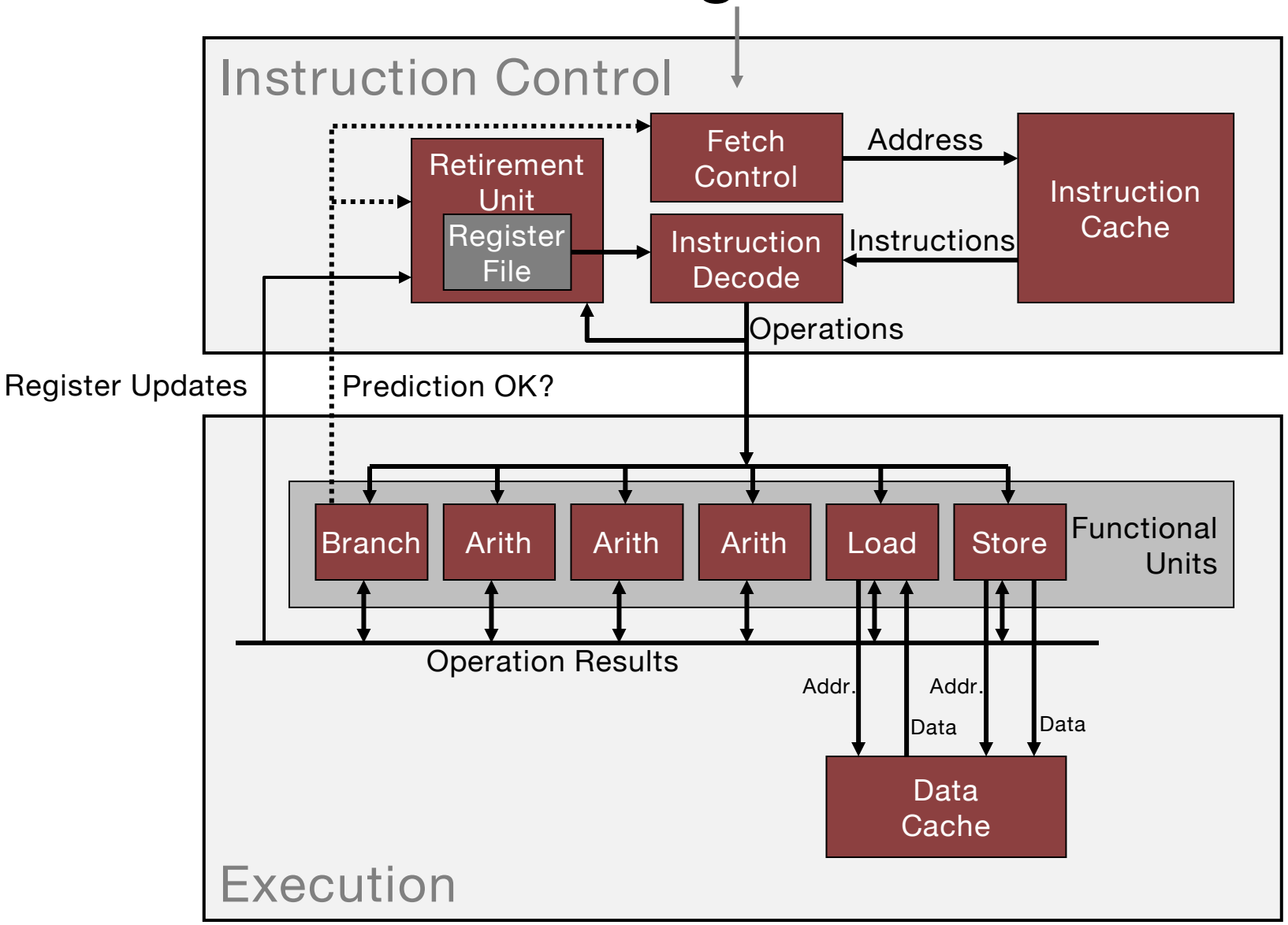
Effect of Basic Optimizations

```
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

- Eliminates sources of overhead in loop

Modern CPU Design



Superscalar Processor

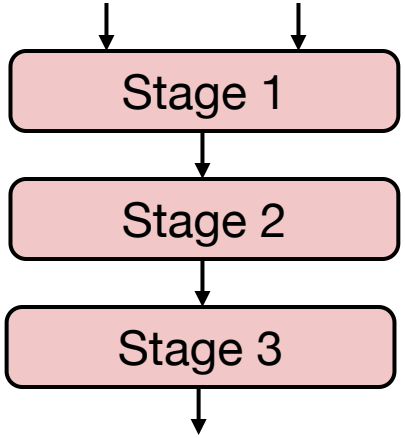
- Definition: A superscalar processor can issue and execute multiple instructions in one cycle. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

- Benefit: without programming effort, superscalar processor can take advantage of the instruction level parallelism that most programs have

- Most modern CPUs are superscalar.
- Intel: since Pentium (1993)

Pipelined Functional Units

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



	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

Haswell CPU

- 8 Total Functional Units

- Multiple instructions can execute in parallel

  2 load, with address computation
  1 store, with address computation
  4 integer
  2 FP multiply
  1 FP add
  1 FP divide

  0：整数运算、浮点乘、整数和浮点数除法、分支
  1：整数运算、浮点加、整数乘、浮点乘
  2：加载、地址计算
  3：加载、地址计算
  4：存储
  5：整数运算
  6：整数运算、分支
  7：存储、地址计算

- Some instructions take > 1 cycle, but can be pipelined

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

Latency bound and throughput bound

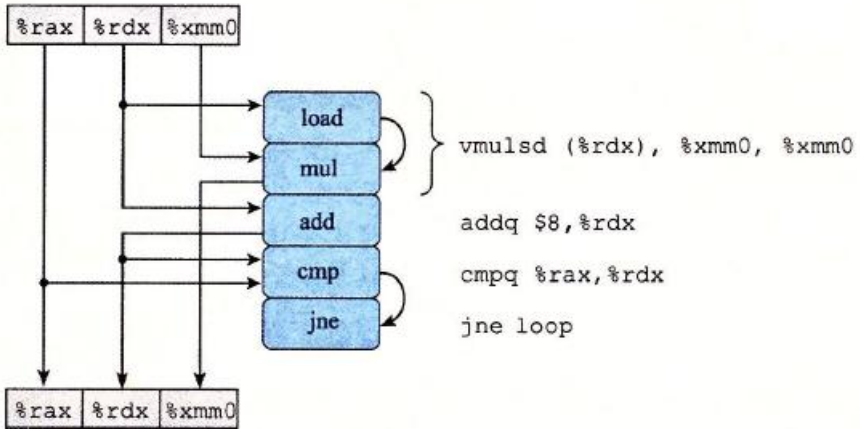
运算	整数			浮点数		
	延迟	发射	容量	延迟	发射	容量
加法	1	1	4	3	1	1
乘法	3	1	1	5	1	2
除法	3 ~ 30	3 ~ 30	1	3 ~ 15	3 ~ 15	1

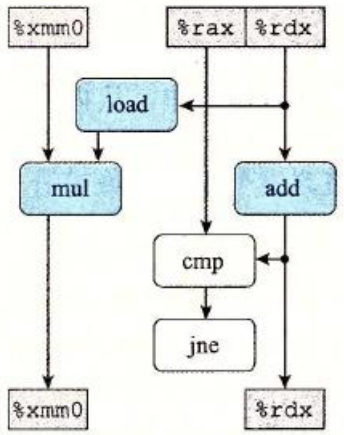
Method	Integer			Double FP		
Operation	Add	Mult	Div	Add	Mult	Div
Combine4	1.27	3.01	--	3.01	5.01	--
Latency Bound	1.00	3.00	16	3.00	5.00	8
Throughput (per cycle)	~~4*1~~ 2*1	1	16	1	1	8
Throughput bound (per op)	0.5	1		1	0.5	

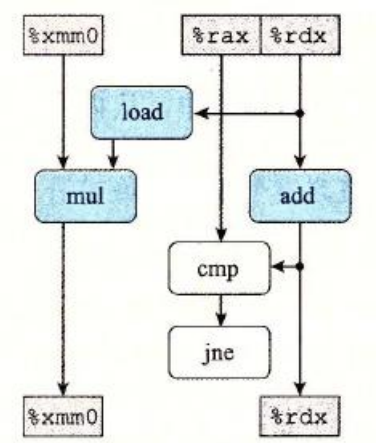
x86-64 Compilation of Combine4

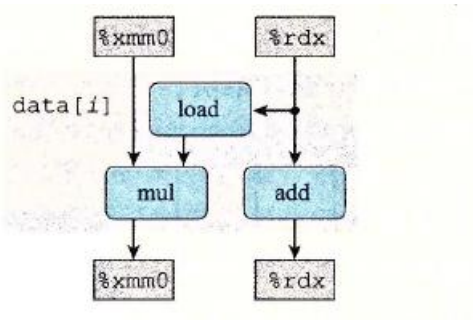
- Inner Loop (Case: float Multiply)

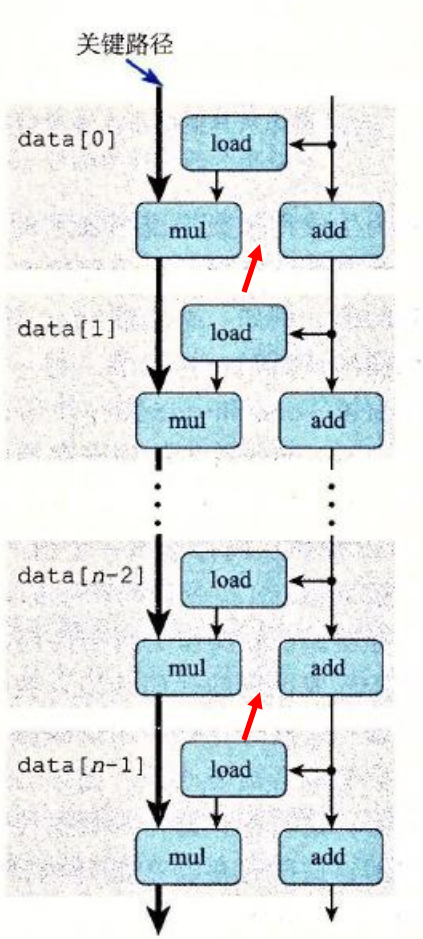
```
.L519:                              # Loop:
  vmulsd (%rdx), %xmm0, %xmm0       # t = t * d[i]
  addq   $8, %rdx                   # increment data+i
  cmpq   %rax, %rdx                 # Compare to data+length
  jg     .L519                      # If !=, goto Loop
```





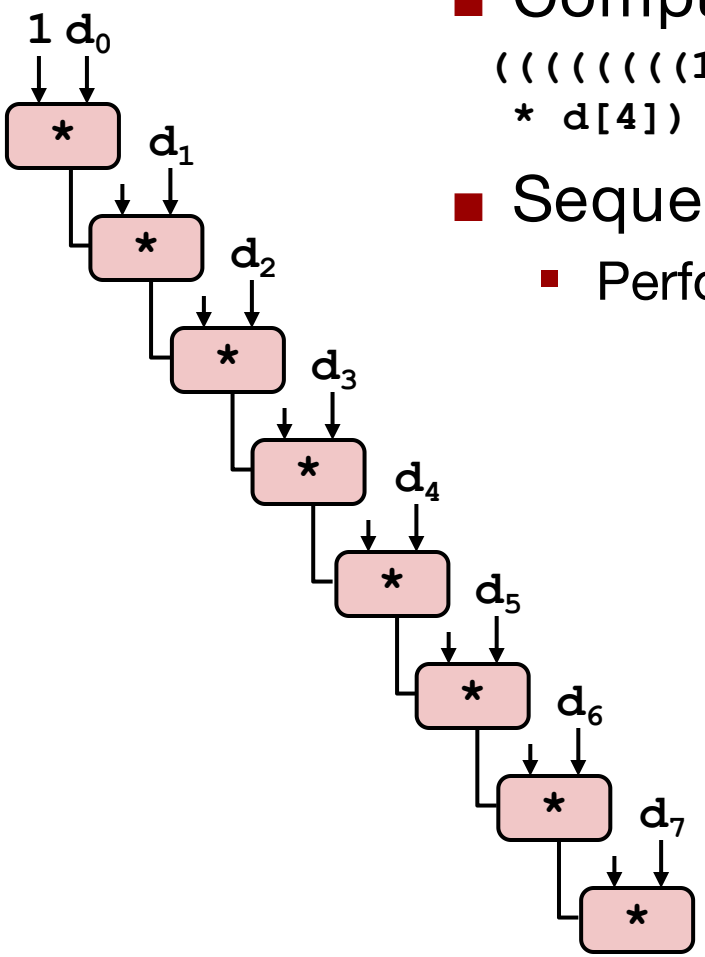

%xmm0
%rax
%rdx
load
mul
add
cmp
jne
%xmm0
%rdx


%xmm0
%rdx
data[i]
load
mul
add
%xmm0
%rdx


关键路径
data[0]
load
mul
add
data[1]
load
mul
add
data[n-2]
load
mul
add
data[n-1]
load
mul
add

	Time													
	1	2	3	4	5	6	7							
Stage 1	load add				mul	load add				mul	load add			
Stage 2		load cmp				mul	load cmp				mul	load cmp		
Stage 3			load jne				mul	load jne				mul	load jne	
Stage 4				load				mul	load				mul	load
Stage 5									mul					mul

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

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;
}
```

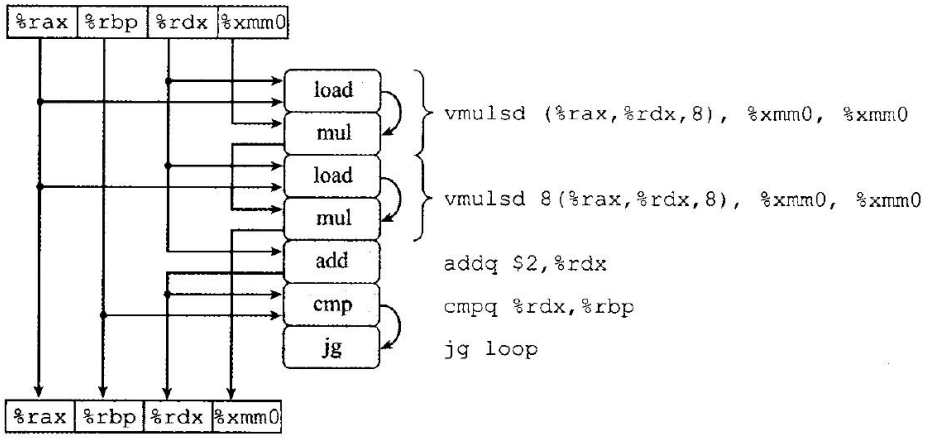
- Perform 2x more useful work per iteration

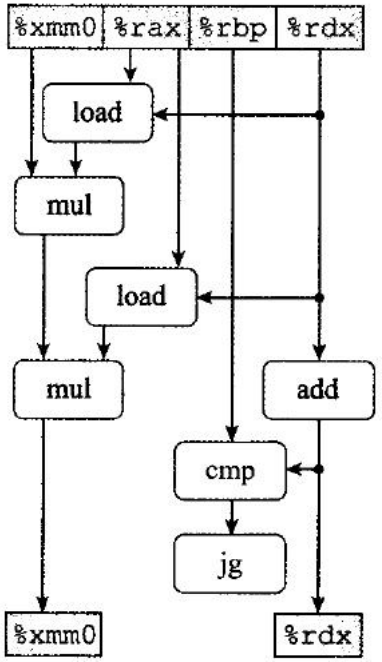
Effect of Loop Unrolling

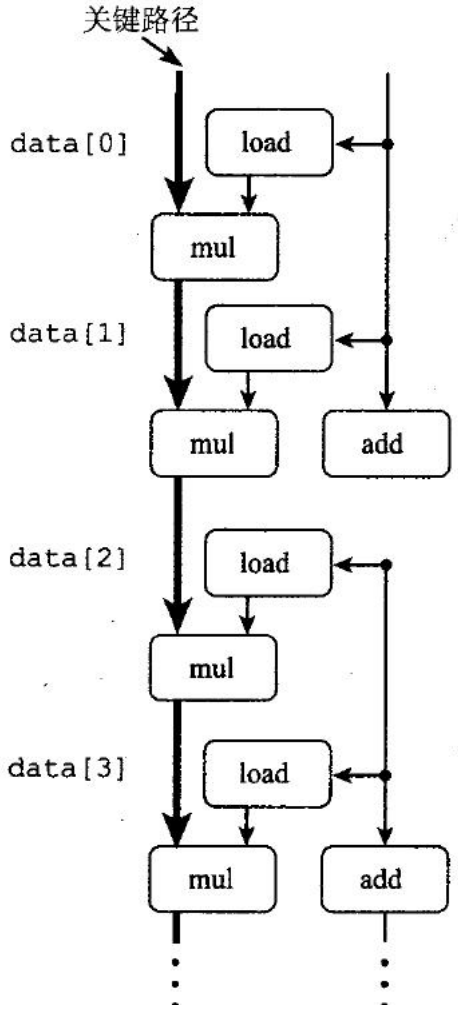
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- Helps integer add
  - Achieves latency bound
- Others don't improve. Why?
  - Still sequential dependency

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


%rax
%rbp
%rdx
%xmm0
load
mul
load
mul
add
cmp
jg
vmulsd (%rax,%rdx,8), %xmm0, %xmm0
vmulsd 8(%rax,%rdx,8), %xmm0, %xmm0
addq $2,%rdx
cmpq %rdx,%rbp
jg loop
%rax
%rbp
%rdx
%xmm0


%xmm0
%rax
%rbp
%rdx
load
mul
load
mul
add
cmp
jg
%xmm0
%rdx


关键路径
data[0]
load
mul
data[1]
load
mul
add
data[2]
load
mul
data[3]
load
mul
add

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 OP d[i]) OP d[i+1];
```

- Can this change the result of the computation?
  - Yes, for FP. Why? => compiler can't do it

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
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

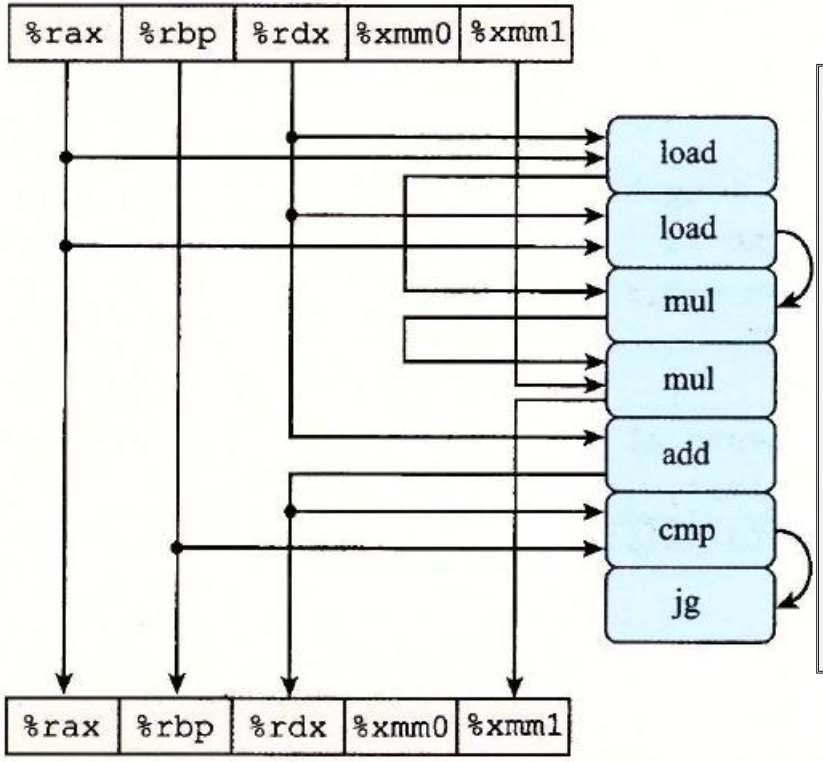
- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential dependency

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

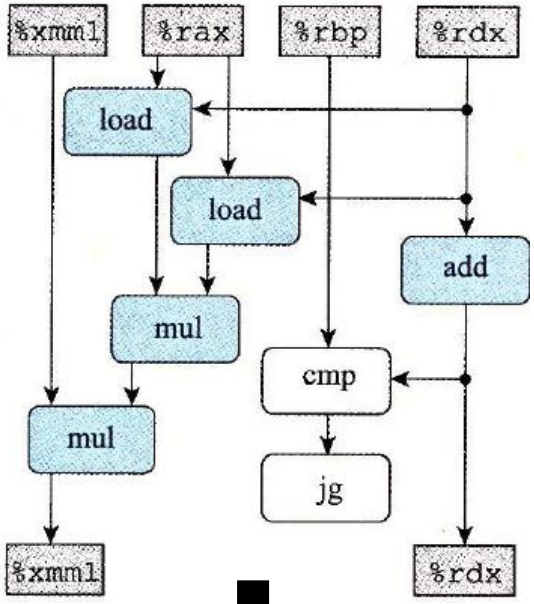
  - Why is that? (next slide)

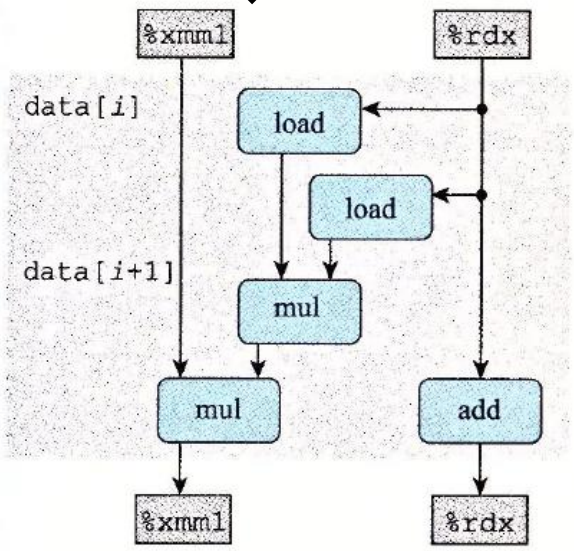
2 func. units for FP
2 func. units for loa

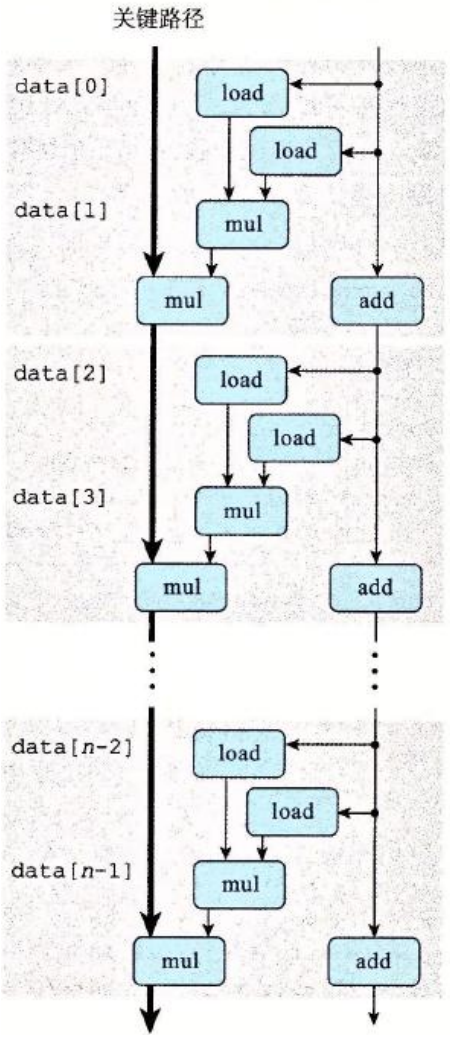
4 func. units for int +
2 func. units for load




```
.L519:
  vmovsd (%rax, %rdx, 8), %xmm0

  vmulsd 8(%rax, %rdx, 8), %xmm0, %xmm0

  vmulsd  %xmm0, %xmm1, %xmm1

  addq   $2, %rdx

  cmpq   %rdx, %rbp

  jg     .L519
```


%xmm1
%rax
%rbp
%rdx
load
load
add
mul
cmp
mul
jg
%xmm1
%rdx


%xmm1
%rdx
data[i]
load
load
data[i+1]
mul
mul
add
%xmm1
%rdx

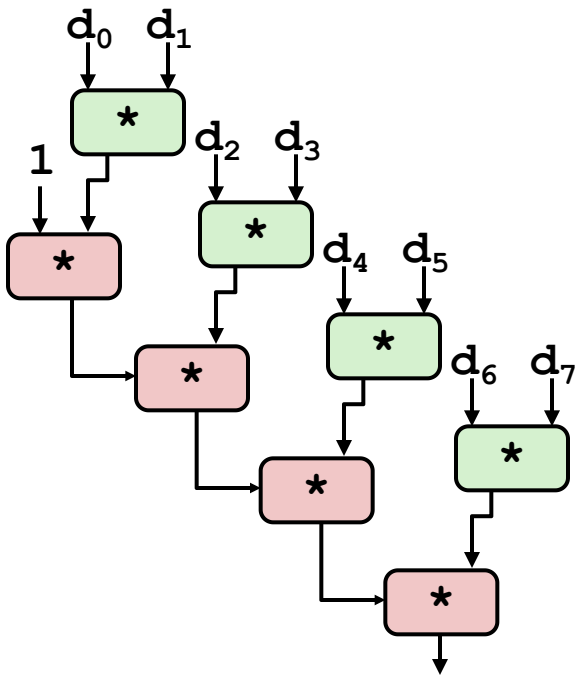

关键路径
data[0]
load
load
data[1]
mul
mul
add
data[2]
load
load
data[3]
mul
mul
add
data[n-2]
load
load
data[n-1]
mul
mul
add

- CPE=5/2=2.5

Time														
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
load add load				mul1	load add load				mul2 mul1'	load add load				mul2' mul1''
	load add load				mul1	load add load				mul2 mul1'	load add load			
		load add load				mul1	load add load				mul2 mul1'	load add load		
			load load				mul1	load load				mul2 mul1'	load load	
								mul1					mul2 mul1'	

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

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
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

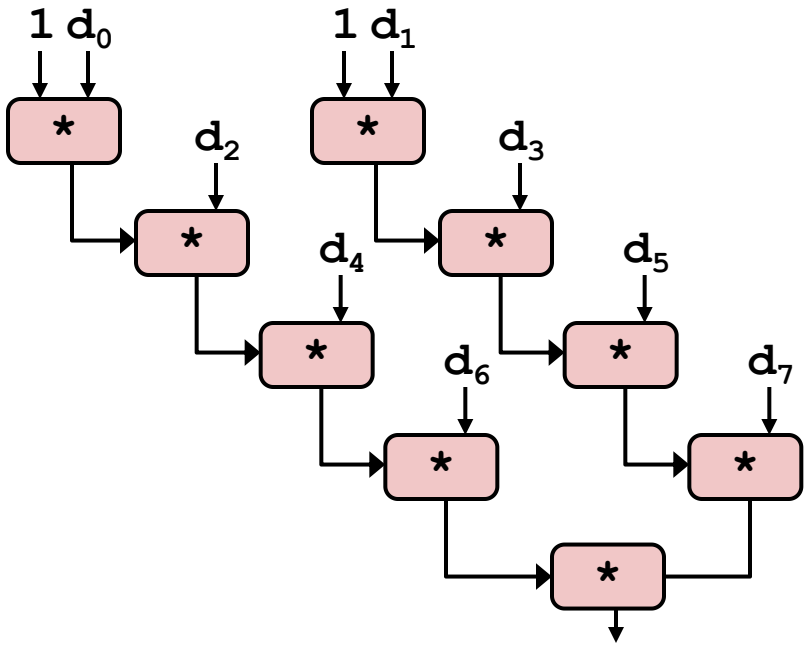
- Int + makes use of two load units

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

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

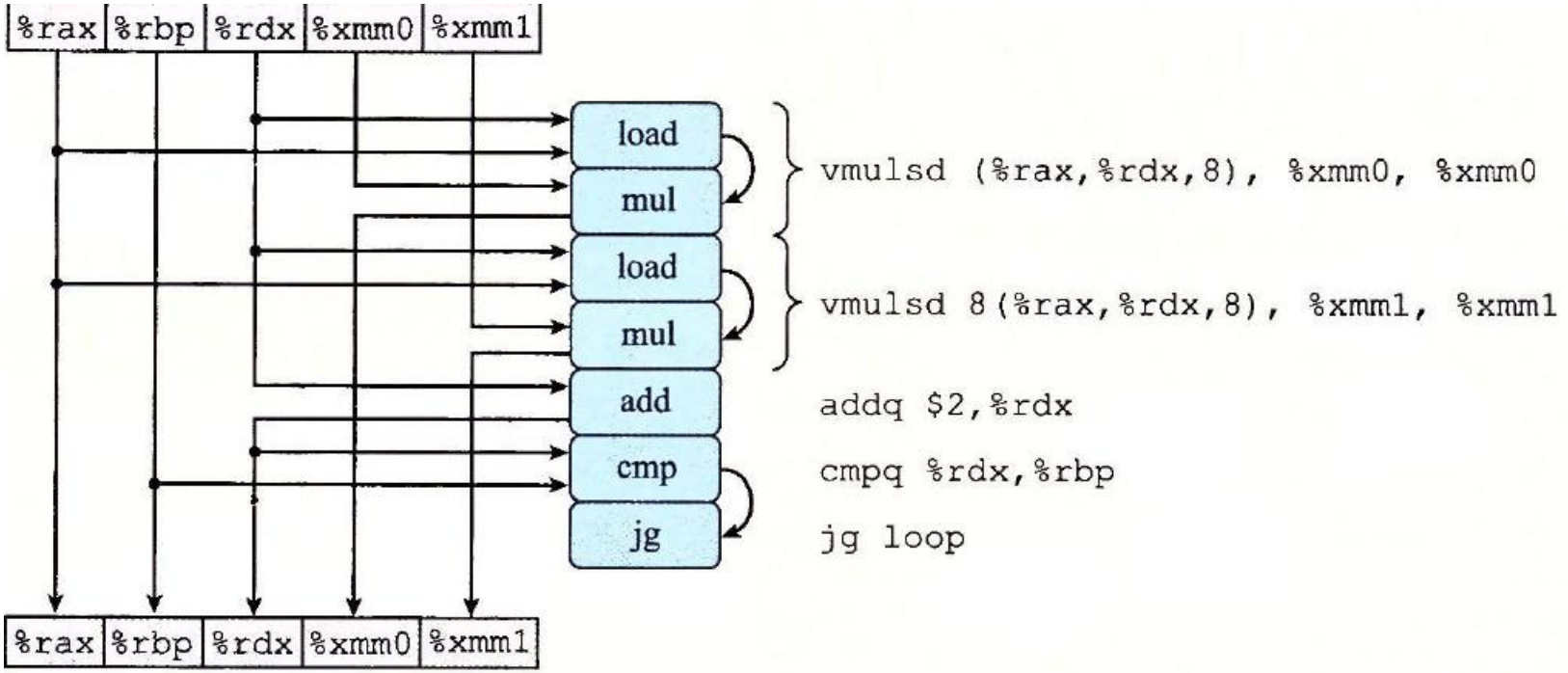
Separate Accumulators

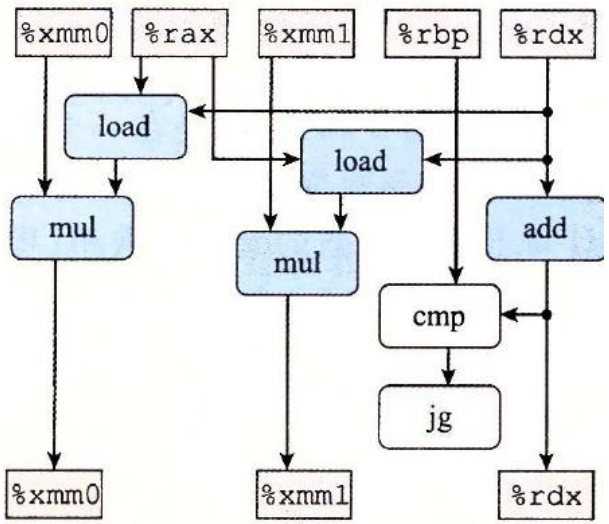
```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

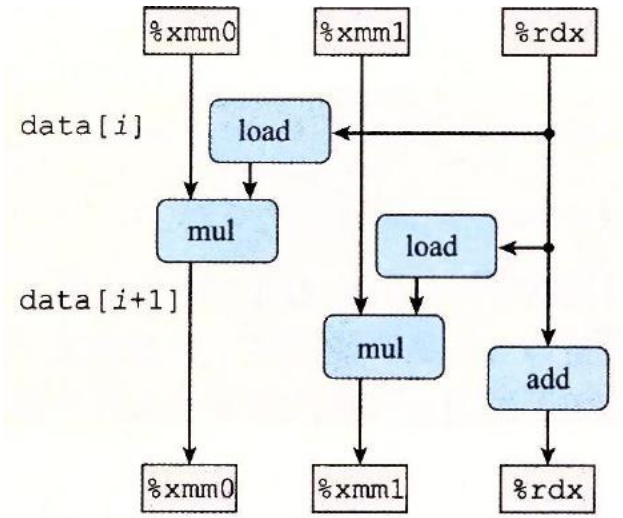


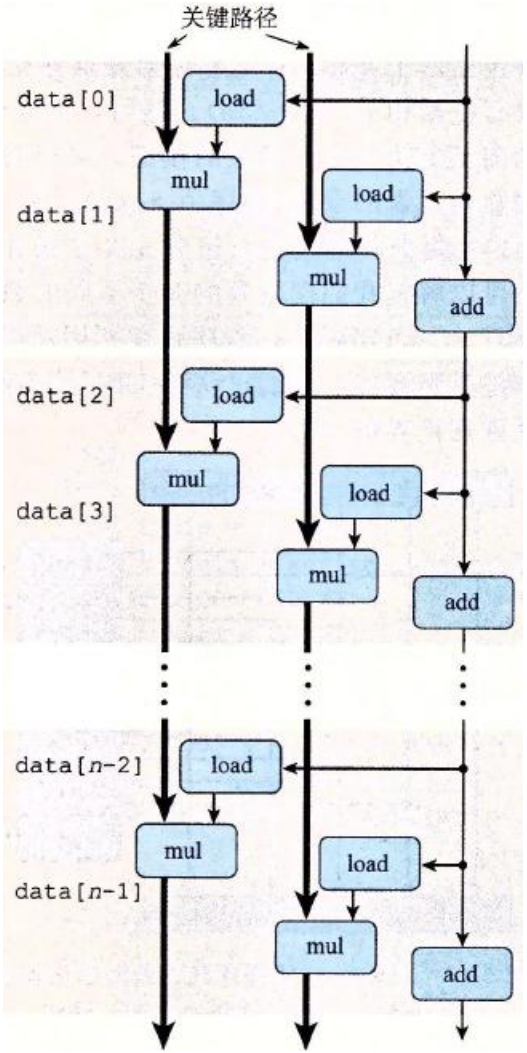
- What changed:
  - Two independent "streams" of operations
- Overall Performance
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: CPE = D/2
  - CPE matches prediction!

What Now?

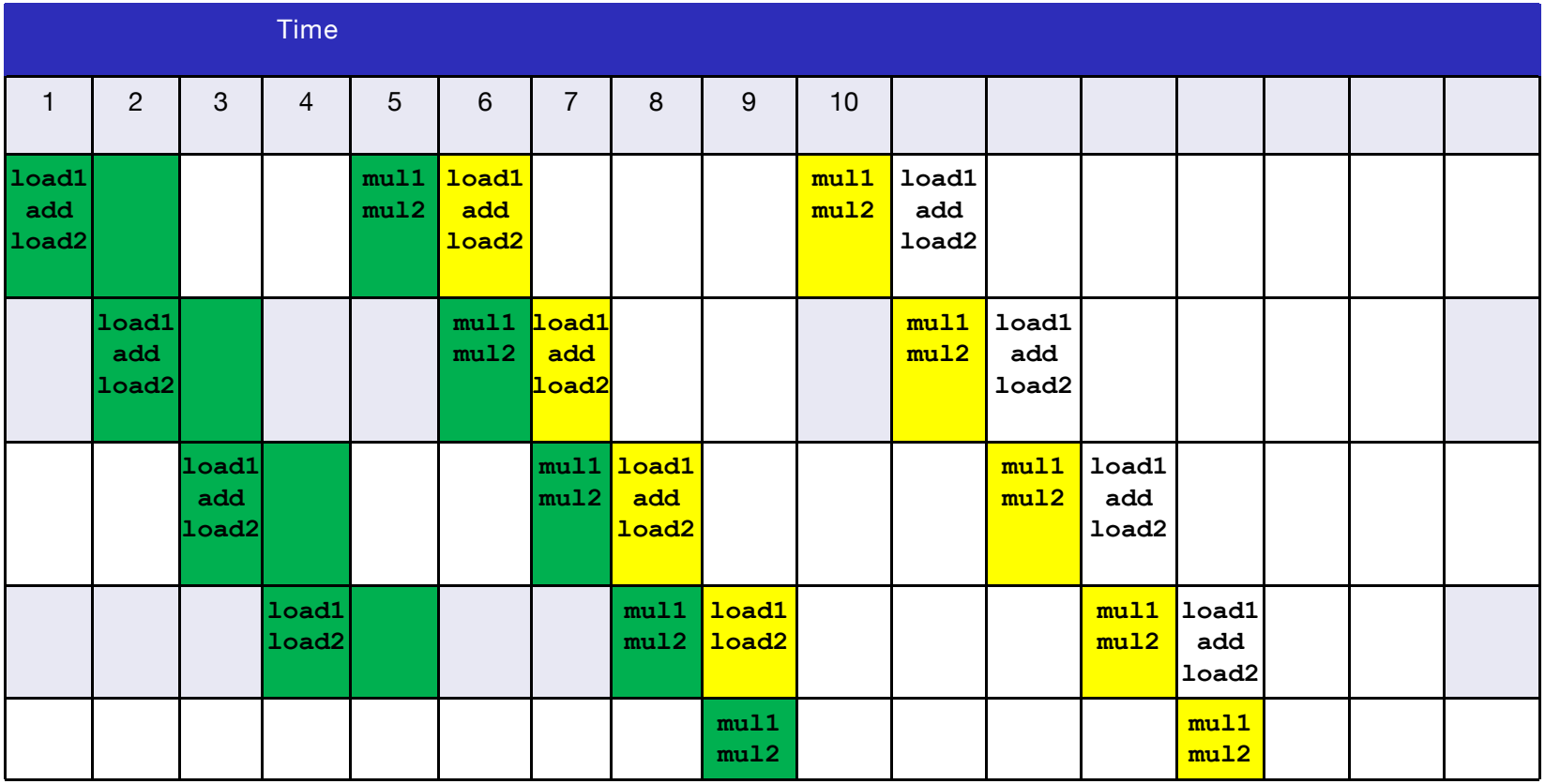

%rax
%rbp
%rdx
%xmm0
%xmm1
load
mul
load
mul
add
cmp
jg
vmulsd (%rax,%rdx,8), %xmm0, %xmm0
vmulsd 8(%rax,%rdx,8), %xmm1, %xmm1
addq $2,%rdx
cmpq %rdx,%rbp
jg loop
%rax
%rbp
%rdx
%xmm0
%xmm1


%xmm0
%rax
%xmm1
%rbp
%rdx
load
load
mul
mul
add
cmp
jg
%xmm0
%xmm1
%rdx


%xmm0
%xmm1
%rdx
data[i]
load
mul
load
data[i+1]
mul
add
%xmm0
%xmm1
%rdx


关键路径
data[0]
load
mul
data[1]
load
mul
add
data[2]
load
mul
data[3]
load
mul
add
data[n-2]
load
mul
data[n-1]
load
mul
add

- CPE=5/2=2.5



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

FP *	Unrolling Factor L							
Accumulators 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

Unrolling & Accumulating: Int +

- Case
  - Intel Haswell
  - Integer addition
  - Latency bound: 1.00. Throughput bound: 1.00

FP *	Unrolling Factor L							
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

Accumulators

Achievable Performance

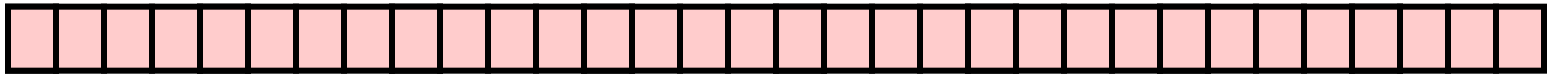
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

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

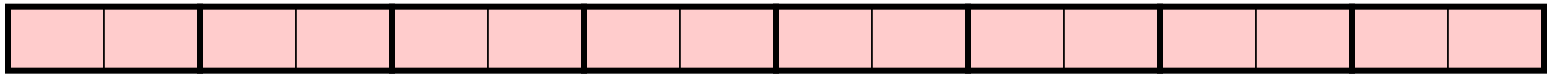
Programming with AVX2

YMM Registers

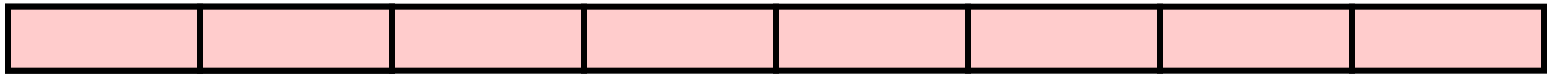
- 16 total, each 32 bytes
- 32 single-byte integers



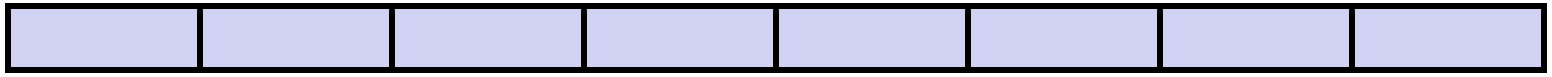
- 16 16-bit integers



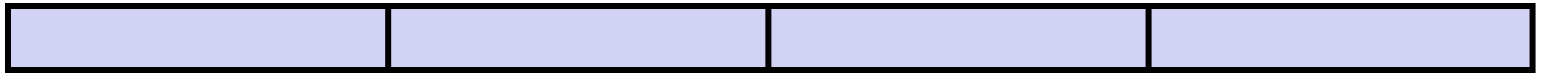
- 8 32-bit integers



- 8 single-precision floats



- 4 double-precision floats

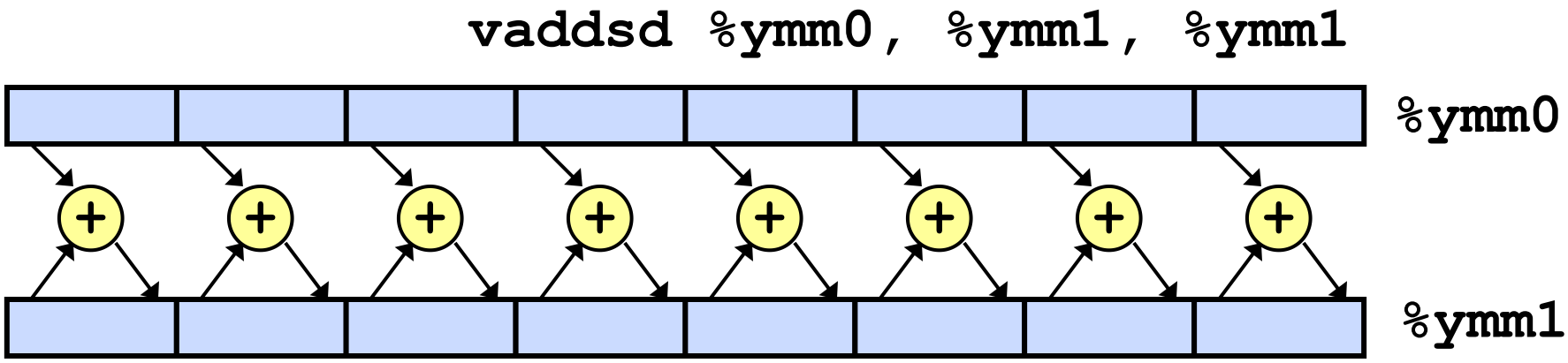


- 1 single-precision float

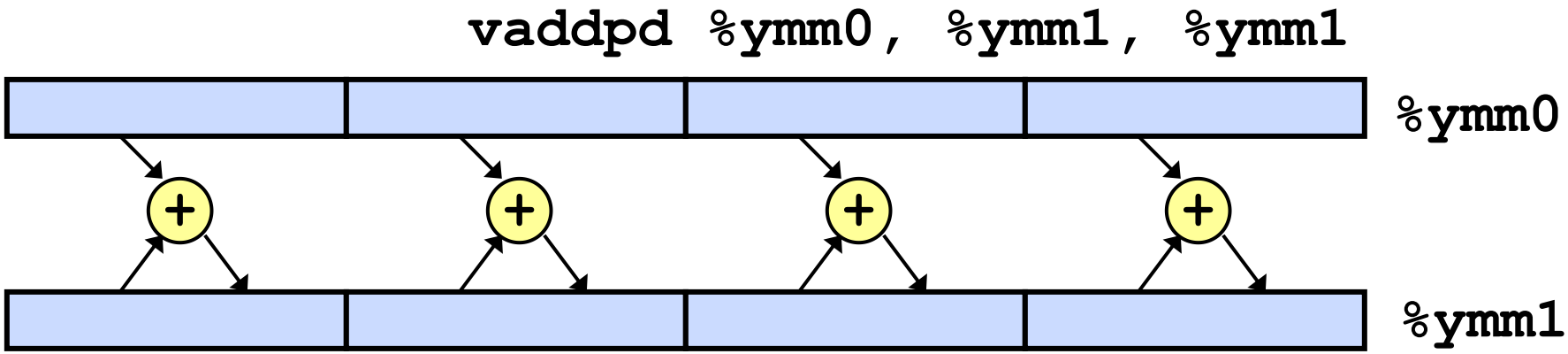
- 1 double-precision float

SIMD Operations

- SIMD Operations: Single Precision



- SIMD Operations: Double Precision



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
Latency Bound	0.50	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50
Vec Throughput Bound	0.06	0.12	0.25	0.12

- Make use of AVX Instructions
  - Parallel operations on multiple data elements
  - See Web Aside OPT:SIMD on CS:APP web page

What About Branches?

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

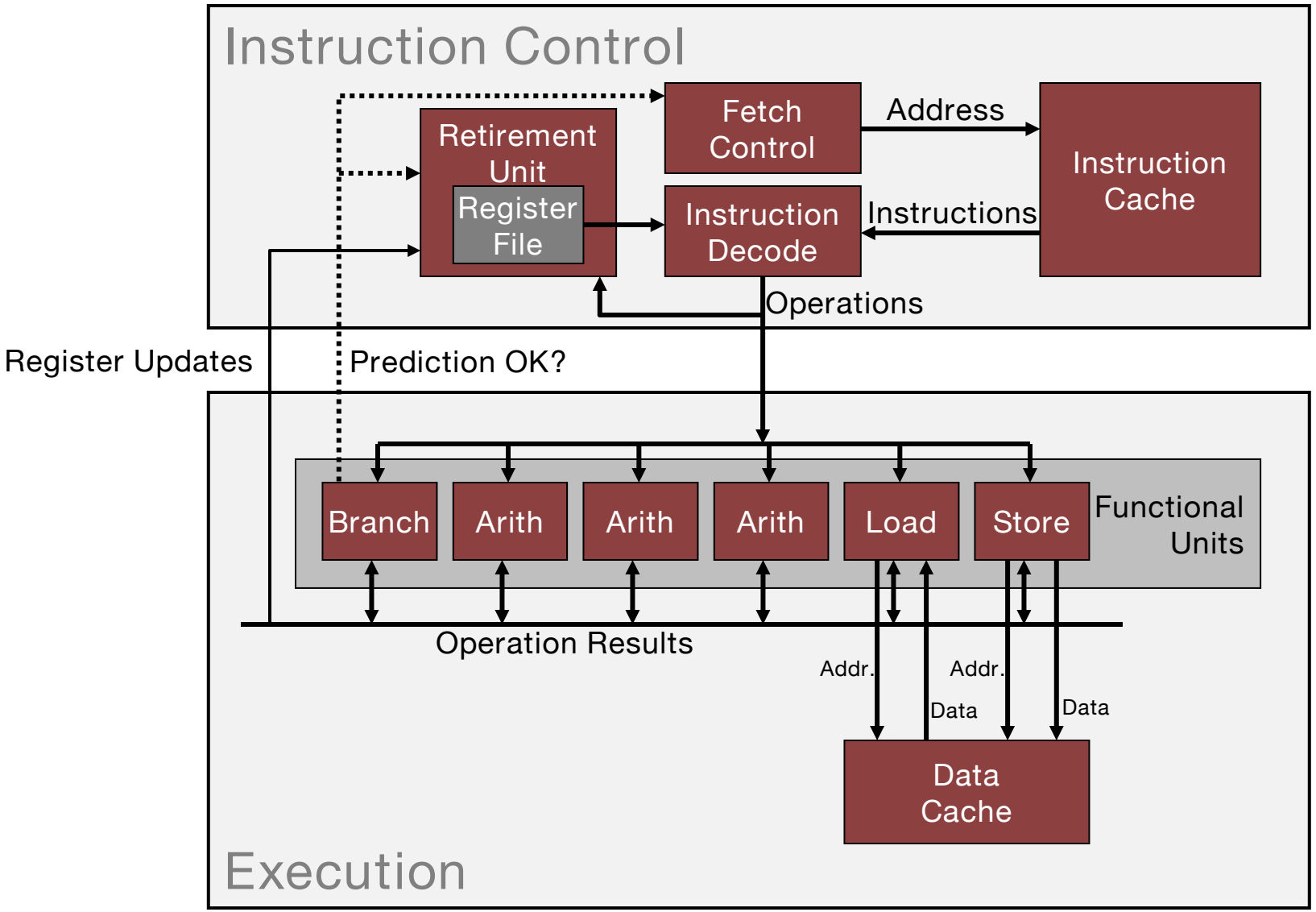
```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

  . . .

404685:   repz retq
```

Executing

How to continue?

  - When encounters conditional branch, cannot reliably determine where to continue fetching

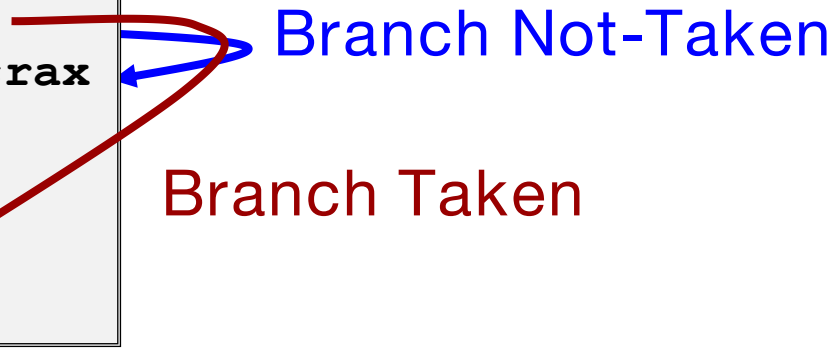
Modern CPU Design



Branch Outcomes

- When encounter conditional branch, cannot determine where to continue fetching
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence
- Cannot resolve until outcome determined by branch/integer unit

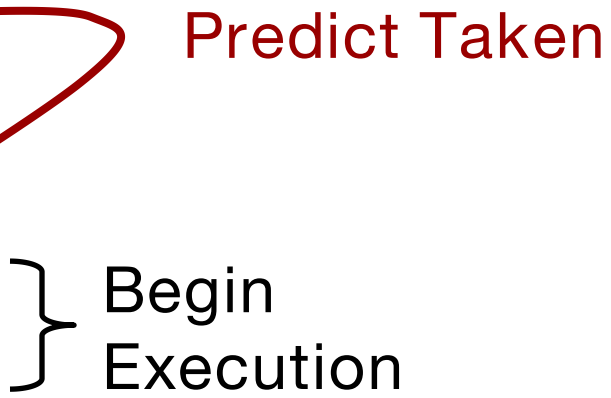
```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

  . . .

404685:   repz retq
```



Branch Prediction

- Idea
  - 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
```



Branch Prediction Through Loop

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```
i = 98

Assume
vector length = 100

Predict Taken (OK)

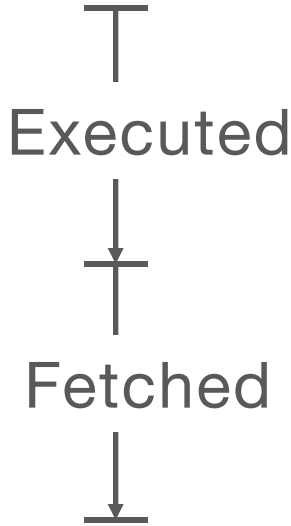
```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```
i = 99

Predict Taken
(Oops)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```
i = 100

Read
invalid
location

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```
i = 101



Branch Misprediction Invalidation

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

i = 98

Assume
vector length = 100

Predict Taken (OK)

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

i = 99

Predict Taken
(Oops)

```
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

Invalidate

Branch Misprediction Recovery

```
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

Definitely not taken

Reload Pipeline

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

Getting High Performance

- Good compiler and flags
- Don't do anything stupid
  - 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 (Covered later in course)