

Program Optimization II

Slides by: Randal E. Bryant and David R. O'Hallaron (CMU)

Presented by Xin Jin and David Hovemeyer for CSF

October 6th, 2023

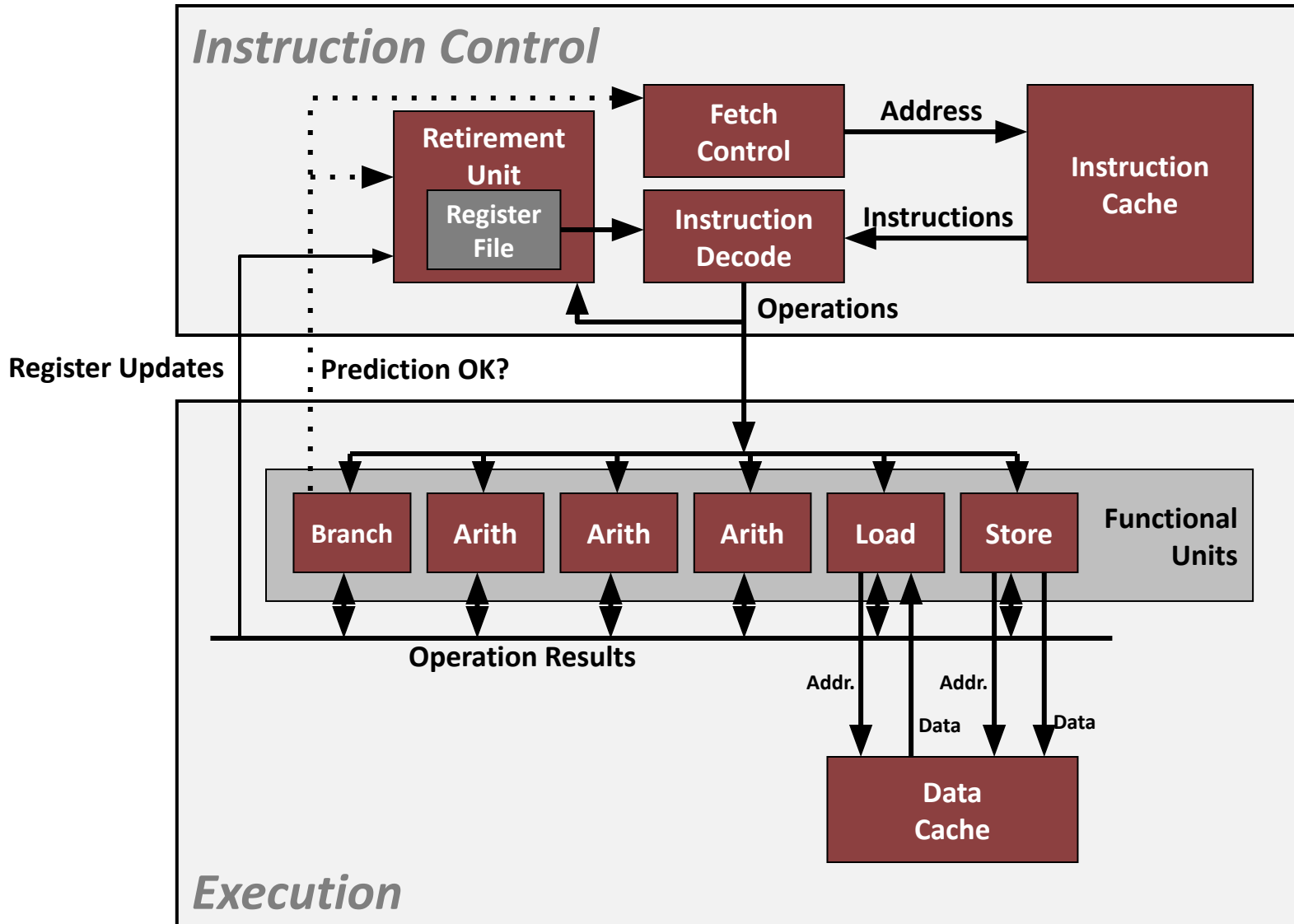
601.229 Computer Systems Fundamentals



Today

- ▶ **Exploiting Instruction-Level Parallelism**
- ▶ **Dealing with Conditionals**

Modern CPU Design

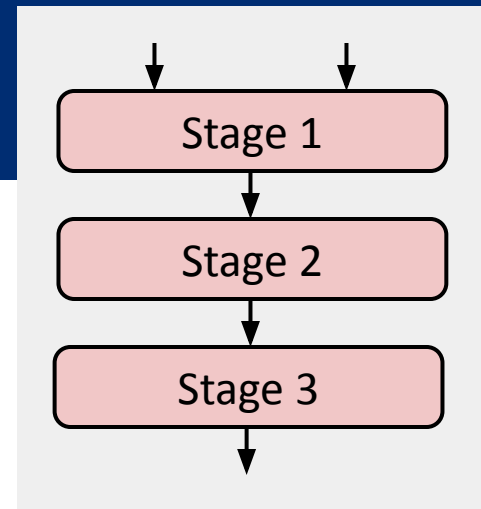


Superscalar Processor

- ▶ **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
- ▶ **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
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

Reminder: combine4 function

```
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;
}
```

- ▶ When IDENT=0 and OP=+: vector sum
- ▶ When IDENT=1 and OP=*: vector multiplication
- ▶ **data_t** can be **int** or **double**

x86-64 Compilation of Combine4

► Inner Loop (Case: Integer Multiply)

```
.L519:      # Loop:
           imull    (%rax,%rdx,4), %ecx    # t = t * d[i]
           addq $1, %rdx # i++
           cmpq %rdx, %rbp    # Compare length:i
           jg  .L519    # If >, goto Loop
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

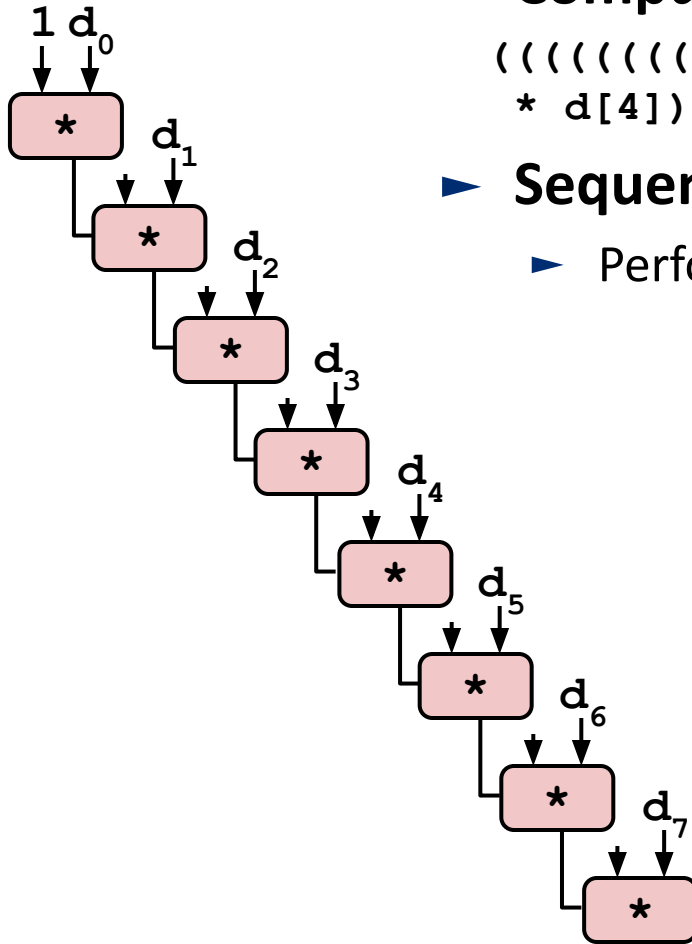
Combine4 = Serial Computation (OP = *)

- ▶ **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



Clicker quiz!

Consider the following x86-64 instruction:

```
addq %r12, %r10
```

Which of these instructions are *independent* of the instruction on the left? (Meaning that they could execute either immediately before or immediately after the instruction on the left without affecting the result of the program.)

- A. `mulq %r12, %r13`**
- B. `mulq %r13, %r12`**
- C. `mulq %r13, %r10`**
- D. `mulq %r10, %r13`**
- E. None of the above**

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

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

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

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

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

- ▶ Can this change the result of the computation?
- ▶ Yes, for FP. *Why?*

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]);
```

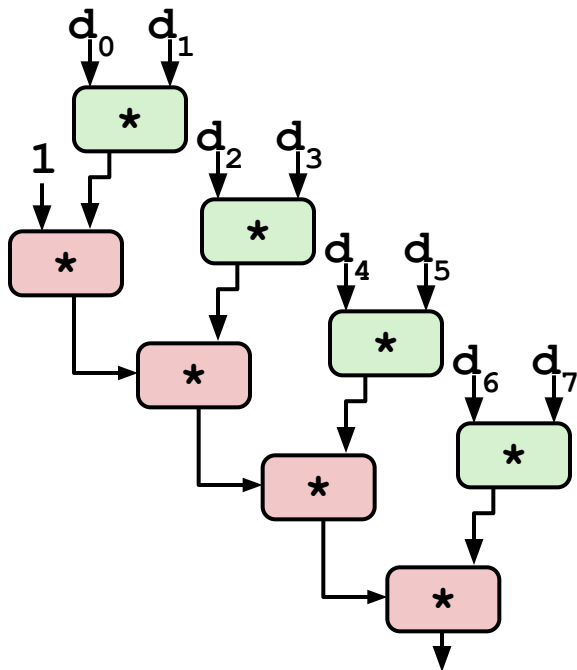
- Why is that? (next slide)

2 func. units for FP *
2 func. units for load

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

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

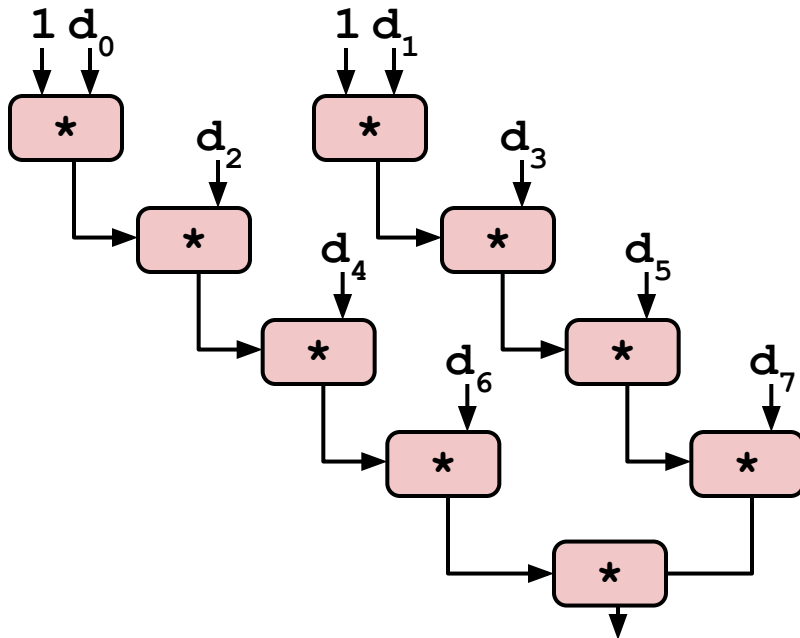
- Int + makes use of two load units

```
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?

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

Accumulators	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

Unrolling & Accumulating: Int +

► Case

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

<i>Accumulators</i>	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

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

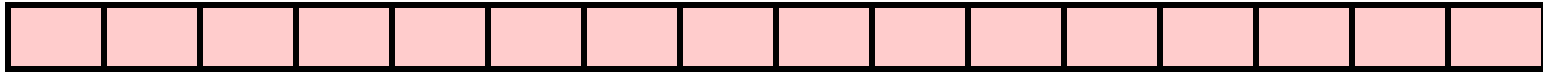
Programming with AVX2

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



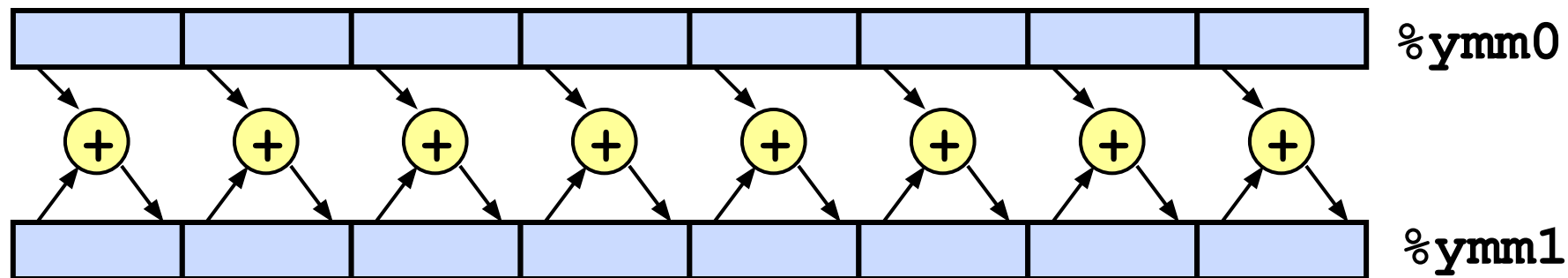
■ 1 double-precision float



SIMD Operations

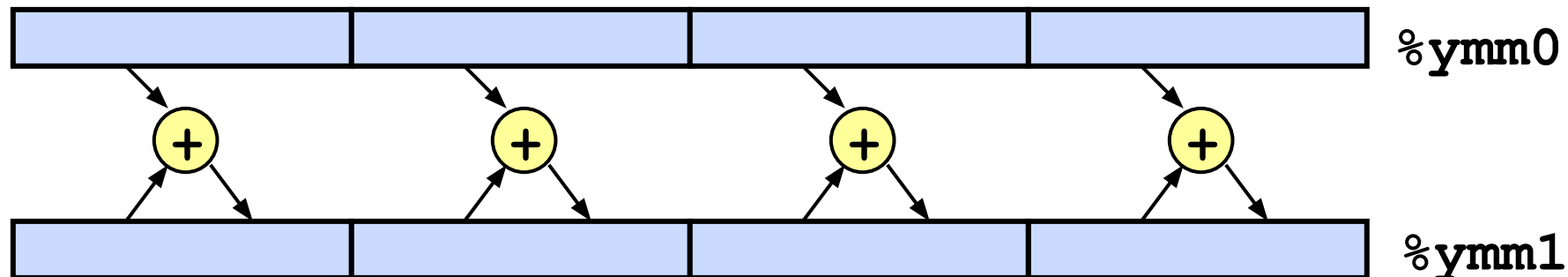
- ▶ SIMD Operations: Single Precision

vaddsd %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
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

. . .

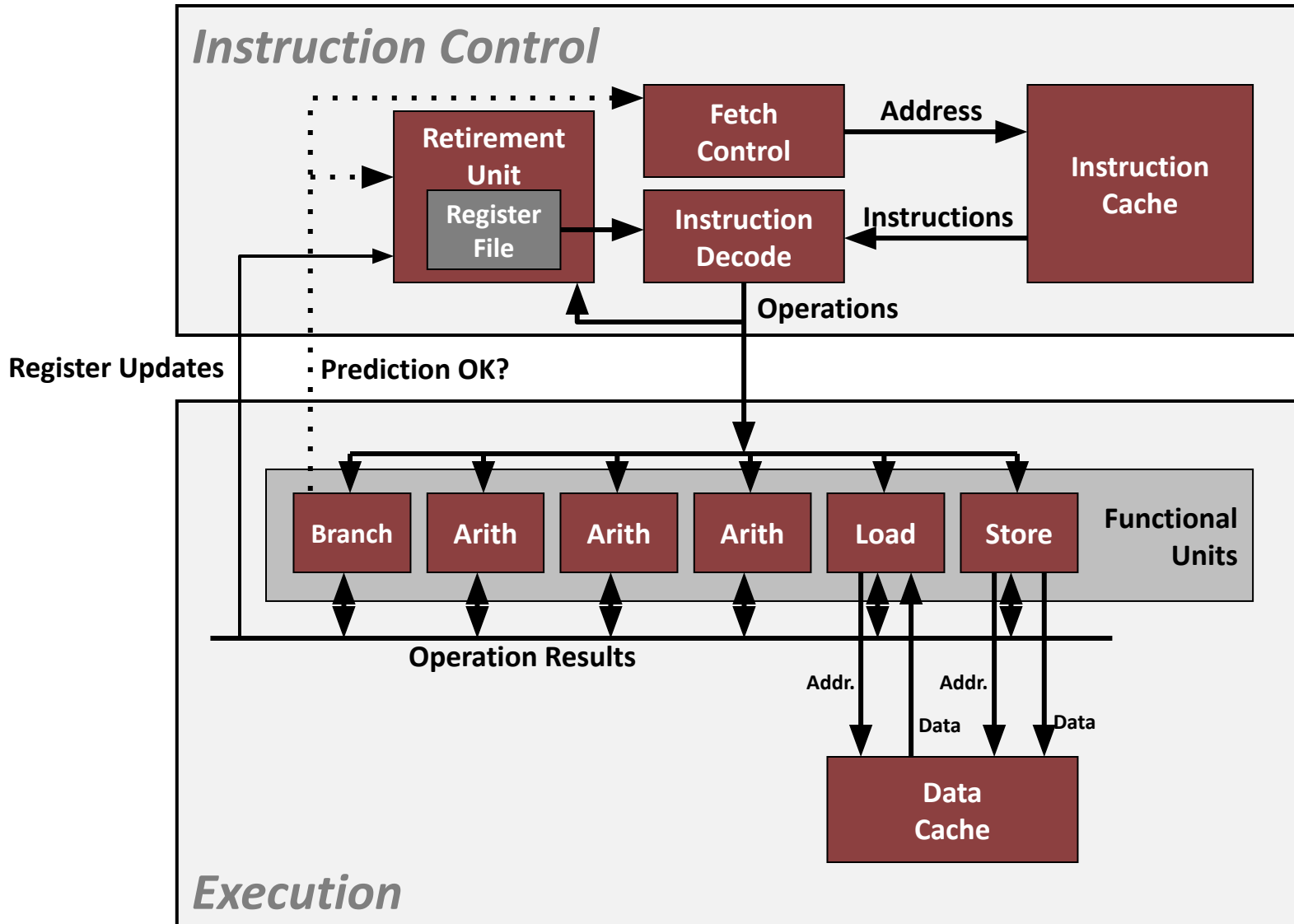
404685:  repz   retq
```

} 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 Not-Taken

Branch
Taken

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
```

**Predict
Taken**

**Begin
Execution**

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)

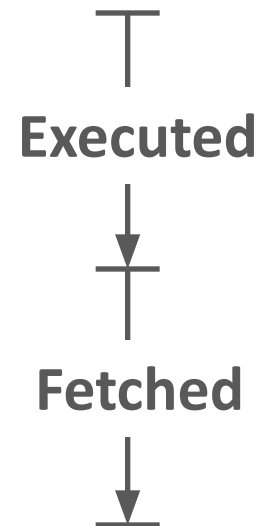
```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

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

Invalidate

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

i = 101

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)