

Program Optimization

(Chapter 5)

Outline

- **Generally Useful Optimizations**
 - Code motion/precomputation
 - Strength reduction
 - Sharing of common subexpressions
 - Removing unnecessary procedure calls
- **Optimization Blockers**
 - Procedure calls
 - Memory aliasing
- **Exploiting Instruction-Level Parallelism**
- **Dealing with Conditionals**
 - Branch Prediction

Performance Realities

There's more to performance than asymptotic complexity.

- **Constant factors matter too!**

 - Easily see 10:1 performance range depending on how code is written

 - Must optimize at multiple levels:

 - algorithm, data representations, procedures, and loops

- **Must understand system to optimize performance**

 - How programs are compiled and executed

 - How to measure program performance and identify bottlenecks

 - How to improve performance without destroying code modularity and generality

Optimizing Compilers

Provide efficient mapping of program to machine

- register allocation
- code selection and ordering (scheduling)
- dead code elimination
- eliminating minor inefficiencies

Don't (usually) improve asymptotic efficiency

- up to programmer to select best overall algorithm
- Big-O savings are (often) more important than constant factors
 - but constant factors also matter

Have difficulty overcoming “optimization blockers”

- potential memory aliasing
- potential procedure side-effects

Aliasing

“When data in memory can be accessed in more than one way”

Example: Is it safe to keep x in a register?

```
int x;  
int *p;  
...  
*p = 123;  
...
```

What if p points to x?

In general, we cannot know the answer to this question without running the program.

Limitations of Optimizing Compilers

- **Fundamental constraint:**

 - Must not cause any change in program behavior

 - Often prevents it from making optimizations when would only affect behavior under pathological conditions.

- **Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles**

 - e.g., Data ranges may be more limited than variable types suggest

- **Most analysis is performed only within procedures**

 - Whole-program analysis is too expensive in most cases

- **Most analysis is based only on *static* information**

 - Compiler has difficulty anticipating run-time inputs

When in doubt, the compiler must be conservative!

Generally Useful Optimizations

Optimizations that you or the compiler should do regardless of processor / compiler

Machine Independent Optimizations:

- Code Motion
- Reduction in Strength
- Using Registers for frequently accessed variables
- Share Common Subexpressions

Code motion

Reduce frequency that a computation is performed

IF it will always produce the same result

THEN move it out of inner loop

```
for (i = 0; i < n; i++)  
  for (j = 0; j < n; j++)  
    a[n*i + j] = b[j];
```

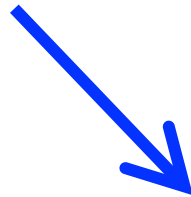

Code motion

Reduce frequency that a computation is performed

IF it will always produce the same result

THEN move it out of inner loop

```
for (i = 0; i < n; i++)  
  for (j = 0; j < n; j++)  
    a[n*i + j] = b[j];
```



```
for (i = 0; i < n; i++) {  
  int ni = n*i;  
  for (j = 0; j < n; j++)  
    a[ni + j] = b[j];  
}
```

Code motion

Most compilers do a good job with array code
and simple loop structures

Code Generated by GCC

```
for (i = 0; i < n; i++)  
  for (j = 0; j < n; j++)  
    a[n*i + j] = b[j];
```

```
for (i = 0; i < n; i++) {  
  int ni = n*i;  
  int *p = a+ni;  
  for (j = 0; j < n; j++)  
    *p++ = b[j];  
}
```

```
testq   %rcx, %rcx           # Test n  
jle     .L1                  # If 0, goto done  
imulq   %rcx, %rdx          # ni = n*i  
leaq    (%rdi,%rdx,8), %rdx  # rowp = A + ni*8  
movl    $0, %eax            # j = 0  
.L3:                                     # loop:  
movsd   (%rsi,%rax,8), %xmm0 # t = b[j]  
movsd   %xmm0, (%rdx,%rax,8) # M[A+ni*8 + j*8] = t  
addq    $1, %rax            # j++  
cmpq    %rcx, %rax          # j:n  
jne     .L3                  # if !=, goto loop  
.L1:                                     # done:  
rep ; ret
```

Reduction in strength

Replace costly operations with simpler ones

Example: Replace multiply & divide with shifts & adds

$$16*x \rightarrow x \ll 4$$

- Depends on cost of multiply or divide instruction
 - Is it worth it? This is “machine dependent”
- Recognize sequence of products and replace with addition

```
for (i = 0; i < n; i++)  
  for (j = 0; j < n; j++)  
    a[n*i + j] = b[j];
```

```
int ni = 0;  
for (i = 0; i < n; i++) {  
  for (j = 0; j < n; j++)  
    a[ni + j] = b[j];  
  ni += n;  
}
```

Using registers

Reading and writing registers is much faster than reading/writing memory!

Limitations

- Compiler not always able to determine whether variable can be held in register
- Possibility of **Aliasing**
 - “Multiple ways of naming/accessing a variable or data item.”
 - There could be a pointer to this variable.
 - Putting it in a registers could be risky.
 - RISKY! It might change the behavior of the program!!!

The performance consequence is huge!

Share common subexpressions

Want to reuse computations where possible

- But compilers often not very sophisticated in exploiting arithmetic properties

```
/* Sum neighbors of i,j */
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up+down+left+right;
```

```
int inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up+down+left+right;
```

3 multiplications: $i*n$, $(i-1)*n$, $(i+1)*n$

```
leaq 1(%rsi), %rax # i+1
leaq -1(%rsi), %r8 # i-1
imulq %rcx, %rsi # i*n
imulq %rcx, %rax # (i+1)*n
imulq %rcx, %r8 # (i-1)*n
addq %rdx, %rsi # i*n+j
addq %rdx, %rax # (i+1)*n+j
addq %rdx, %r8 # (i-1)*n+j
```

1 multiplication: $i*n$

```
imulq %rcx, %rsi # i*n
addq %rdx, %rsi # i*n+j
movq %rsi, %rax # i*n+j
subq %rcx, %rax # i*n+j-n
leaq (%rsi,%rcx), %rcx # i*n+j+n
```

Example: Convert a string to lower case

A function to convert string to lower case:

```
void lower(char *s) {  
    int i;  
    for (i = 0; i < strlen(s); i++)  
        if (s[i] >= 'A' && s[i] <= 'Z')  
            s[i] -= ('A' - 'a');  
}
```

If length of string is n , how does the run-time of this function grow with n ?

- Linear, Quadratic, Cubic, Exponential?

Strlen

```
int lencnt = 0;
size_t strlen(const char *s) {
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```

First call:

Time required = n (i.e., proportional to string length)

Second call:

Another n

Number of times called:

n

Total time:

$$n + n + n + \dots n = n^2$$

$$n^2 + \dots = O(n^2)$$

Example: Convert a string to lower case

A function to convert string to lower case:

```
void lower(char *s) {  
    int i;  
  
    for (i = 0; i < strlen(s); i++)  
        if (s[i] >= 'A' && s[i] <= 'Z')  
            s[i] -= ('A' - 'a');  
}
```

Notice: strlen is executed every iteration!

- Must scan string until finds '\0'
- strlen is linear in length of string
- The loop body is linear in length of string (n)
- The loop body is executed n times.

Overall performance is quadratic... $O(n^2)$

Example: Convert a string to lower case

```
void lower(char *s) {
    int i;

    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

Let's apply **code motion**

Consider the call to **strlen**...

Result does not change from one iteration to another.

Compiler does not know this, though.

Move call to **strlen** outside of loop.

Example: Convert a string to lower case

```
void lower(char *s) {
    int i;
    len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

Let's apply **code motion**

Consider the call to **strlen**...

Result does not change from one iteration to another.

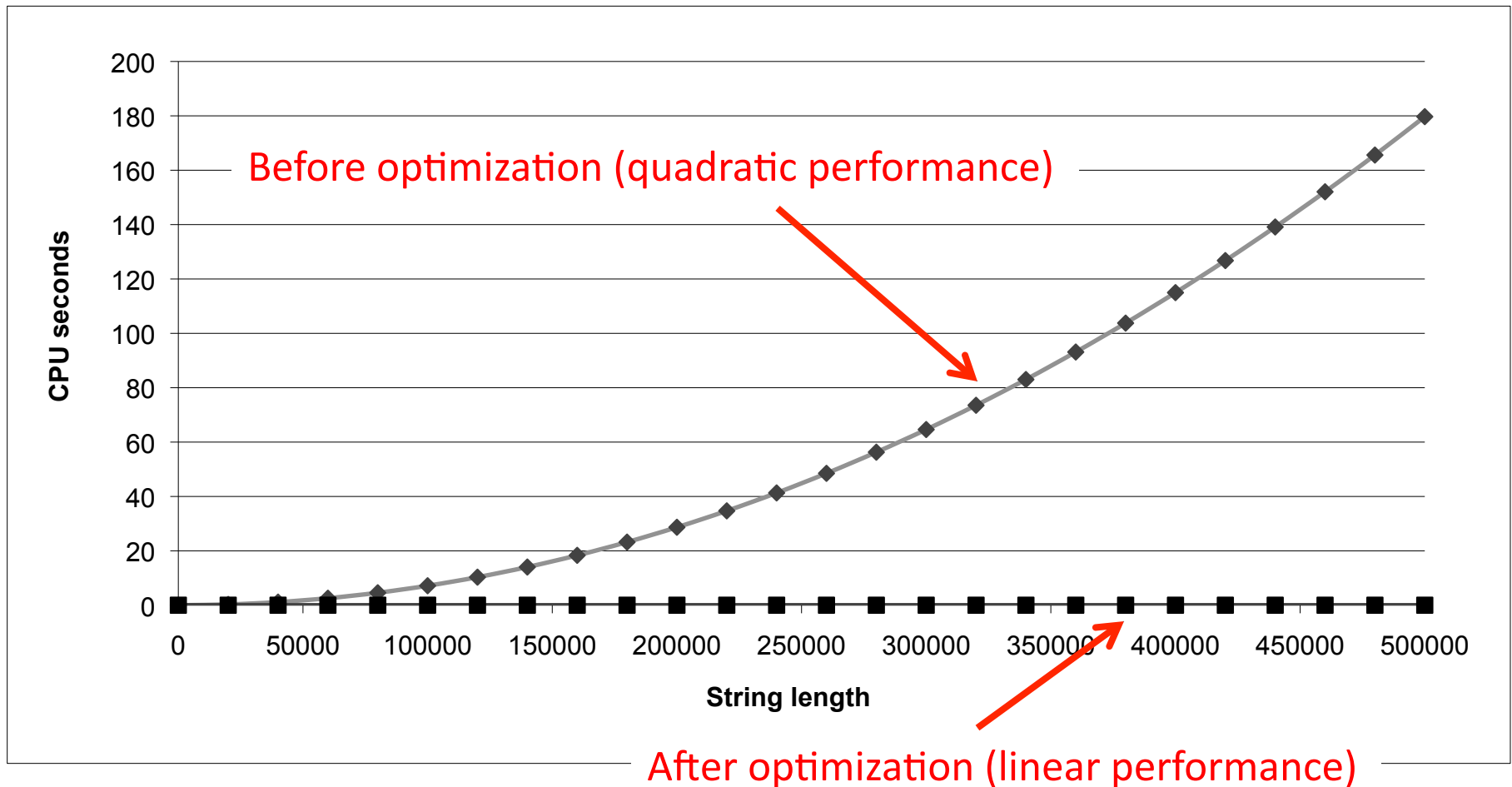
Compiler does not know this, though.

Move call to **strlen** outside of loop.

Example: Convert a string to lower case

Linear Performance $O(n)$: Time doubles when string length doubles

Quadratic Performance $O(n^2)$: Time quadruples when length doubles



Optimization Blocker: Procedure Calls

Why couldn't compiler move `strlen` out of inner loop?

- Procedure may have side effects
 - Alters global state each time called
- Function may not return same value for given arguments
 - Depends on other parts of global state
 - Procedure `lower` could interact with `strlen`

Warning:

Compiler treats procedure call as a black box

Weak optimizations near them

Remedies:

- Use of `inline` functions
 - GCC does this with `-O2`
- Do your own code motion

Memory Aliasing

```
/* Sum the rows in a n X n matrix "a"
   and store in vector "b" */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
# Code for inner loop
Loop:
    movsd    (%rsi,%rax,8), %xmm0    # FP load
    addsd    (%rdi), %xmm0          # FP add
    movsd    %xmm0, (%rsi,%rax,8)   # FP store
    addq     $8, %rdi
    cmpq     %rcx, %rdi
    jne     Loop
```

Code updates **b[i]** on every iteration

Why couldn't compiler optimize this away?

Memory Aliasing

```

/* Sum the rows in a n X n matrix "a"
   and store in vector "b" */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}

```

Must consider possibility that updates will affect program behavior.

```

double A[9] = { 0, 1, 2,
                3, 22, 224,
                32, 64, 128};
sum_rows1(A, A+3, 3);

b[0,1,2] = A[3,4,5] !!!

```

Value of B:

desired: [3, 28, 224]

i = 0: [3, 8, 16]

i = 1: [3, 22, 16]

i = 2: [3, 22, 224]

Removing Aliasing

```
/* Sum the rows in a n X n matrix "a"
   and store in vector "b" */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

```
# Code for inner loop
Loop:
    addsd    (%rdi), %xmm0 # FP load + add
    addq    $8, %rdi
    cmpq    %rax, %rdi
    jne     Loop
```

No need to store intermediate results!

Optimization Blocker: Memory Aliasing

Aliasing:

Two different memory references specify single location

Easy to have happen in C

- Address arithmetic
- Direct access to storage structures

Get in habit of introducing local variables

(e.g., accumulating within loops)

Your way of telling compiler not to check for aliasing

Exploiting Instruction-Level Parallelism

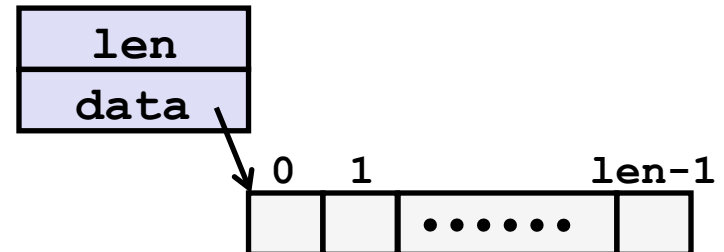
- Need general understanding of modern processor design

Hardware can execute multiple instructions in parallel

- But performance is limited by “data dependencies”
- Simple transformations can have dramatic performance improvement
 - Often, compilers cannot make these transformations
 - Lack of associativity and distributivity in floating-point arithmetic

Example: Data Type for Vectors

```
/* data structure for vectors */  
typedef struct{  
    int len;  
    double *data;  
} vec;
```



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

Benchmark Computation

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

Compute sum or
product of vector
elements

Data Types

Operations

Benchmark Computation

```
void combine1(vec_ptr v, double *dest)
{
    long int i;
    *dest = 1.0;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest * val;
    }
}
```

Compute sum or
product of vector
elements

Data Types

Operations

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 types `data_t`

`int`

`long`

`float`

`double`

Operations

Use different definitions of `OP`

`+` (with `IDENT = 0`)

`*` (with `IDENT = 1`)

Cycles Per Element (CPE)

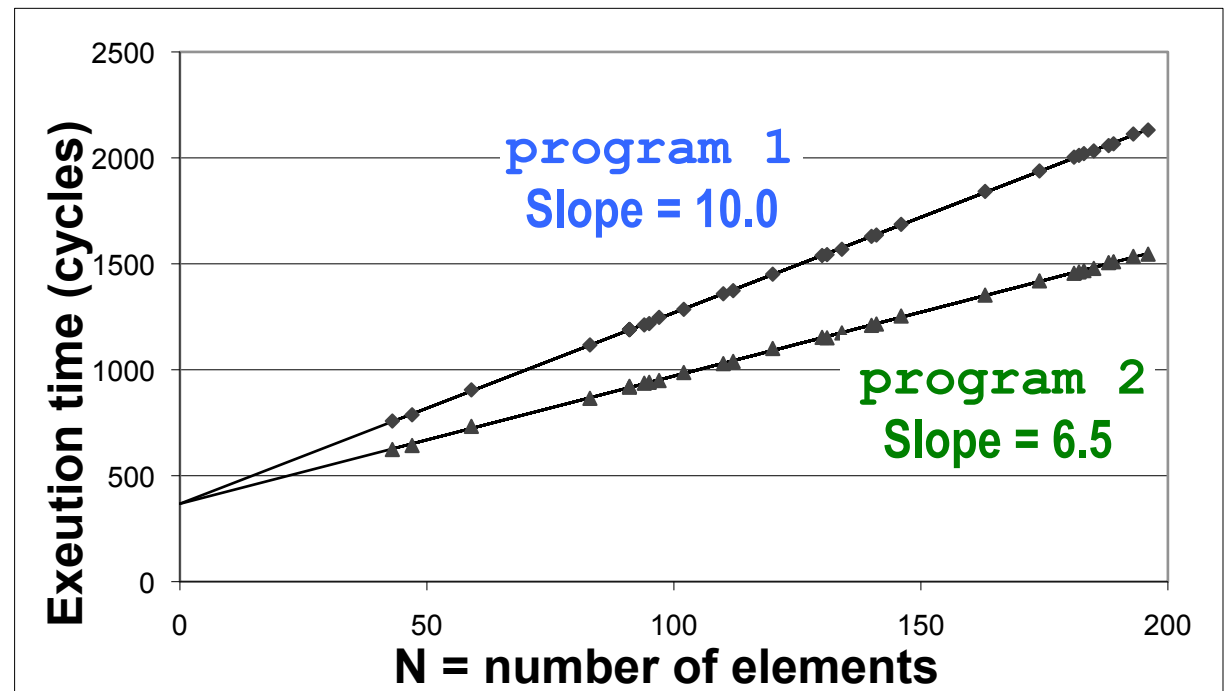
A convenient way to express performance of a program that operates on vectors or lists

n = Length or number of elements to process

In our case:

CPE = cycles per OP

Total Time =
CPE*n + Overhead
CPE =
slope of line



Benchmark Performance: Baseline

```
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

	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

Basic Optimizations

```
void combinel(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;
    }
}
```

- Move `vec_length` out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
    int i;
    int 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

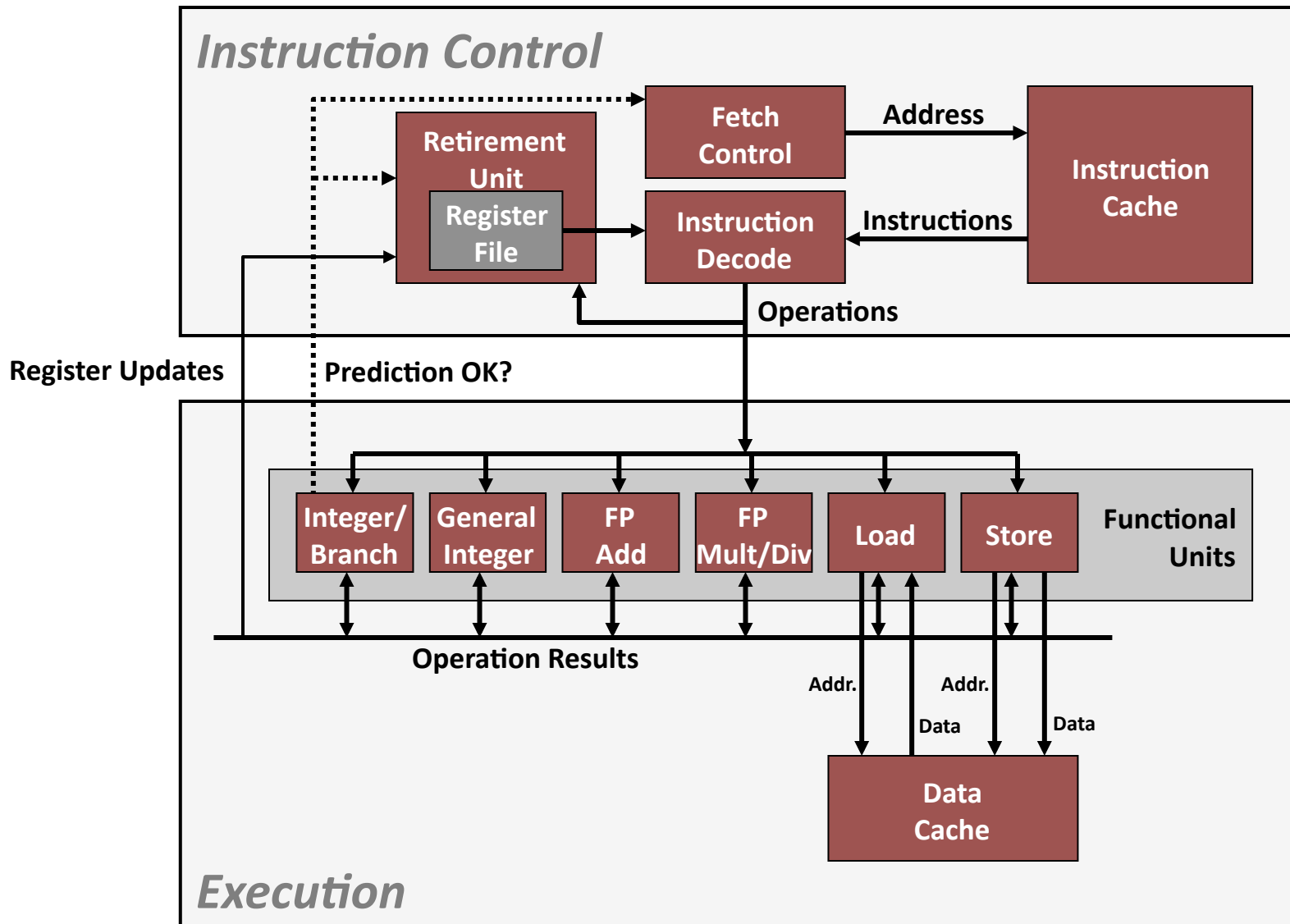
Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
    int i;
    int 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;
}
```

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

This eliminates sources of overhead in loop

Modern CPU Design



Superscalar Processor

- 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.

Without programming effort, a superscalar processor can take advantage of the *instruction level parallelism* that most programs have

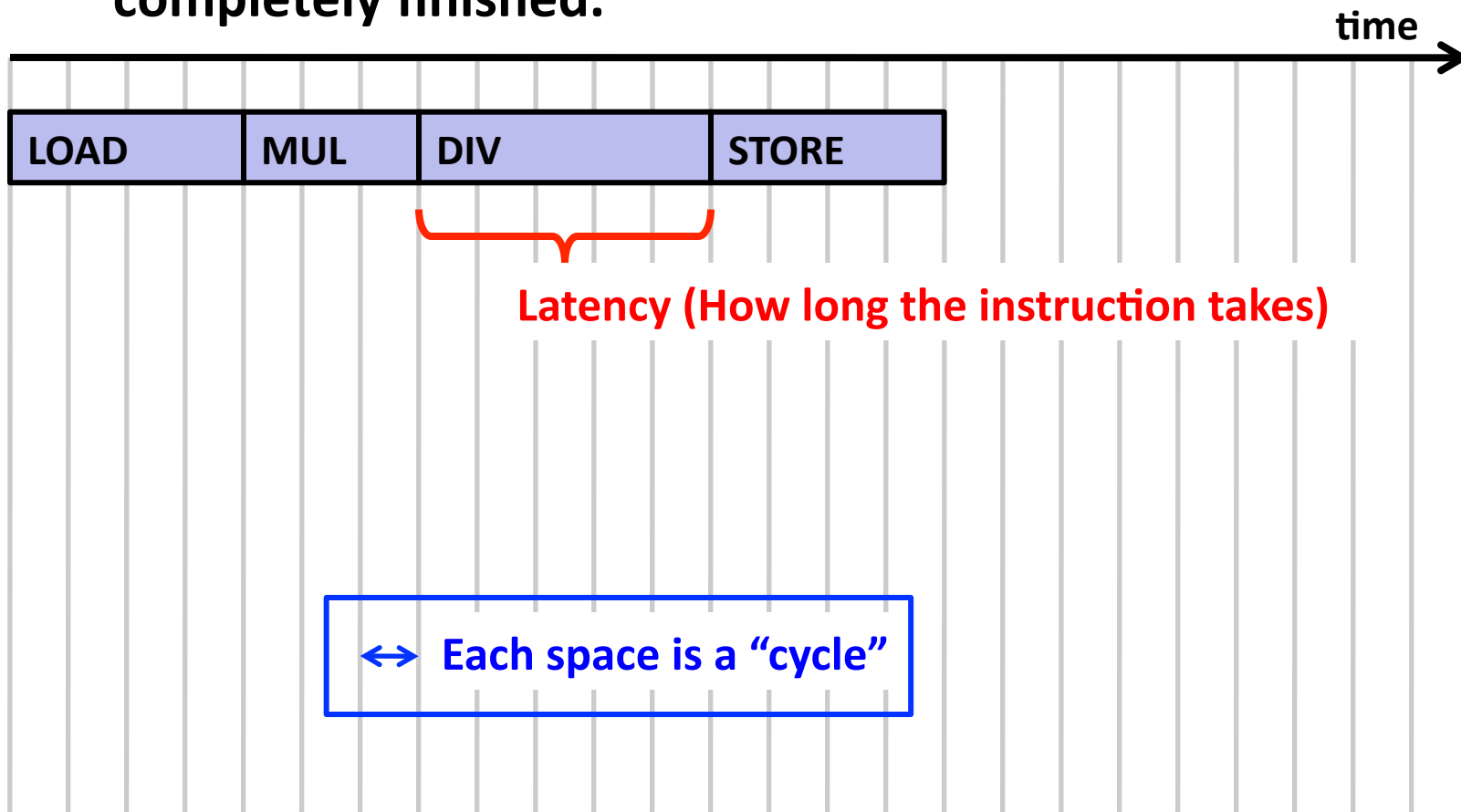
Most CPUs since about 1998 are superscalar.

- Intel: since Pentium Pro

Basic Instruction Execution

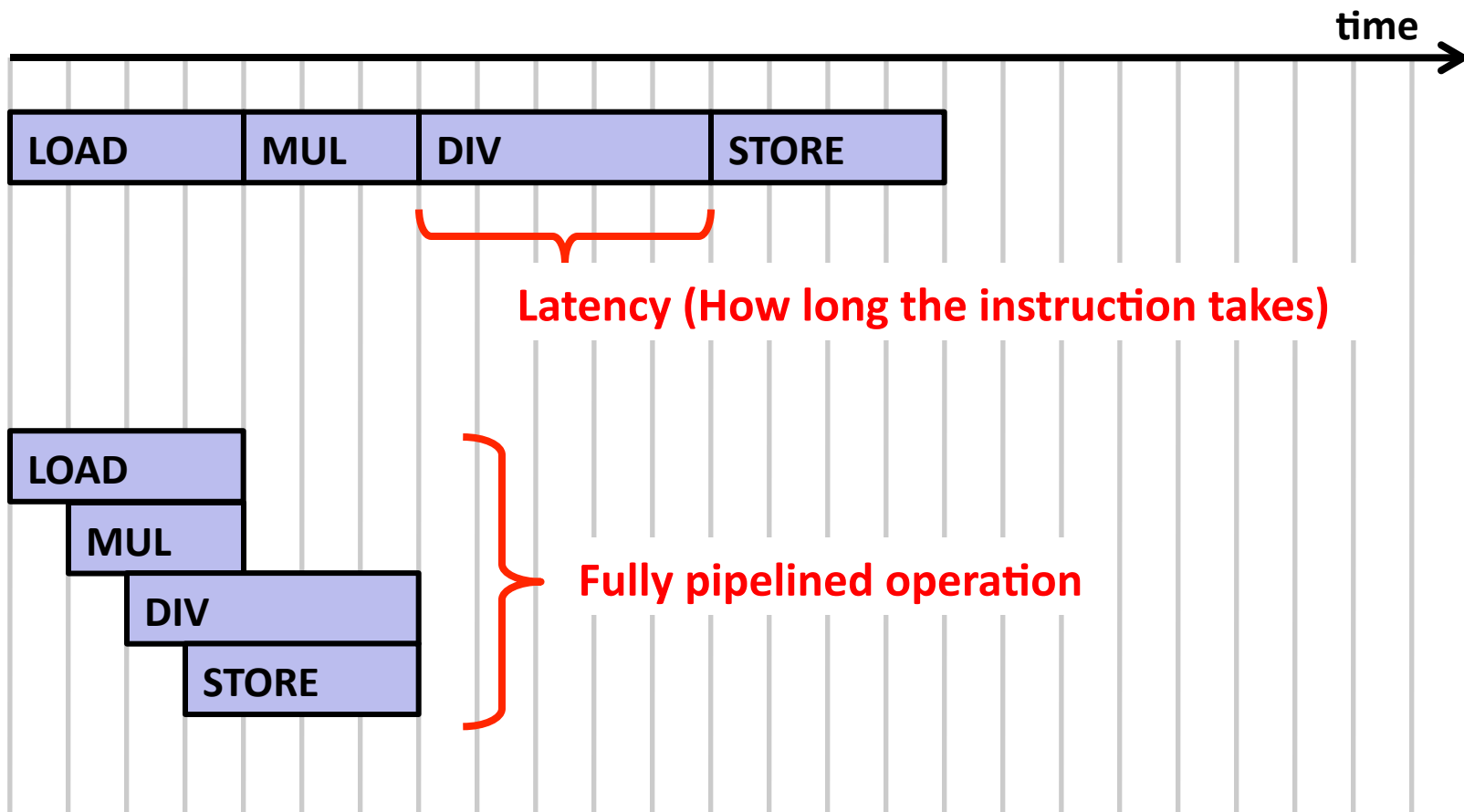
Each instruction takes some time to execute.

We don't start one instruction until the previous one has completely finished.



Pipelined Instruction Execution

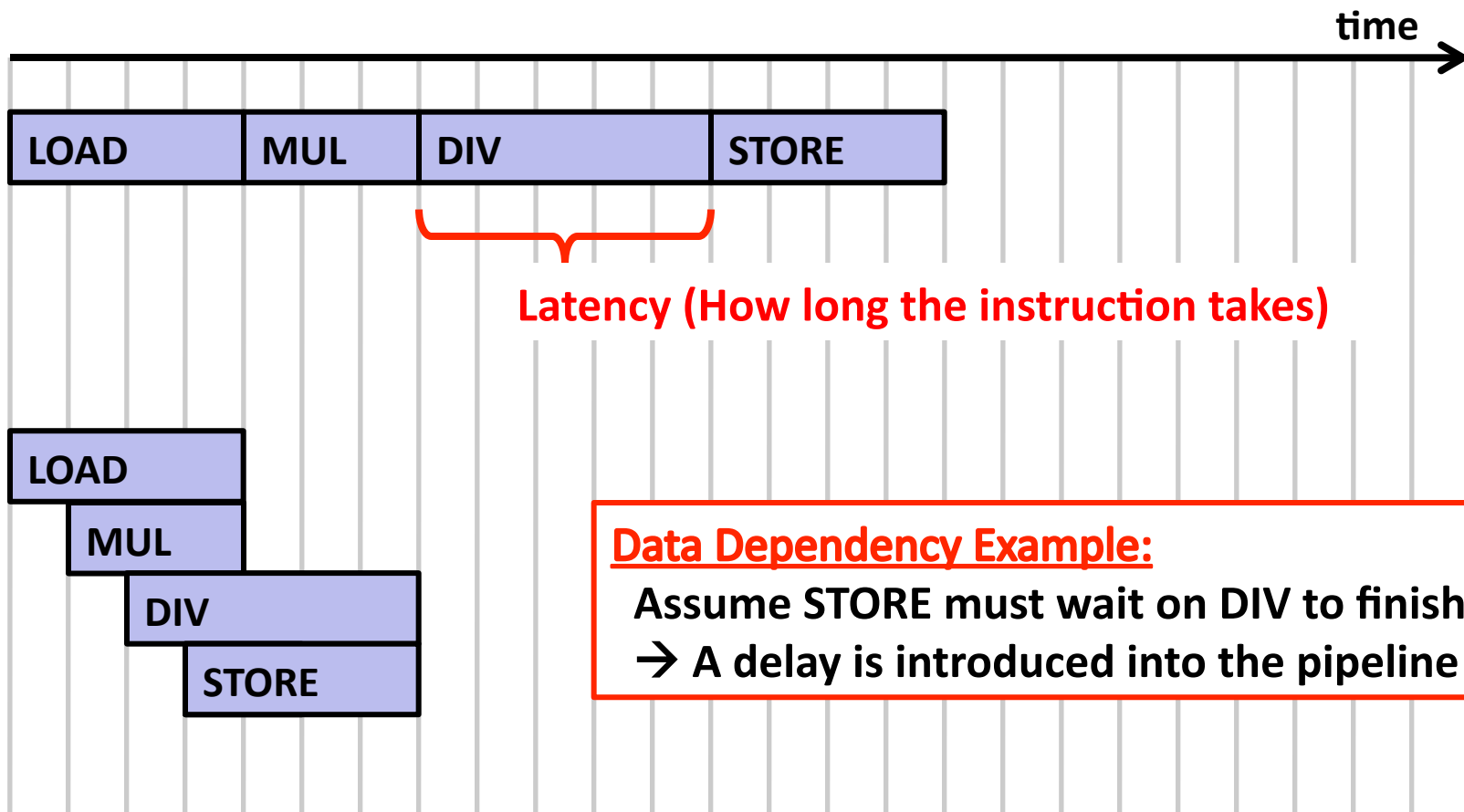
With pipelining, we can start a new instruction every cycle.
We can execute several instructions in parallel!



Pipelined Instruction Execution

Sometimes we cannot start next instruction immediately

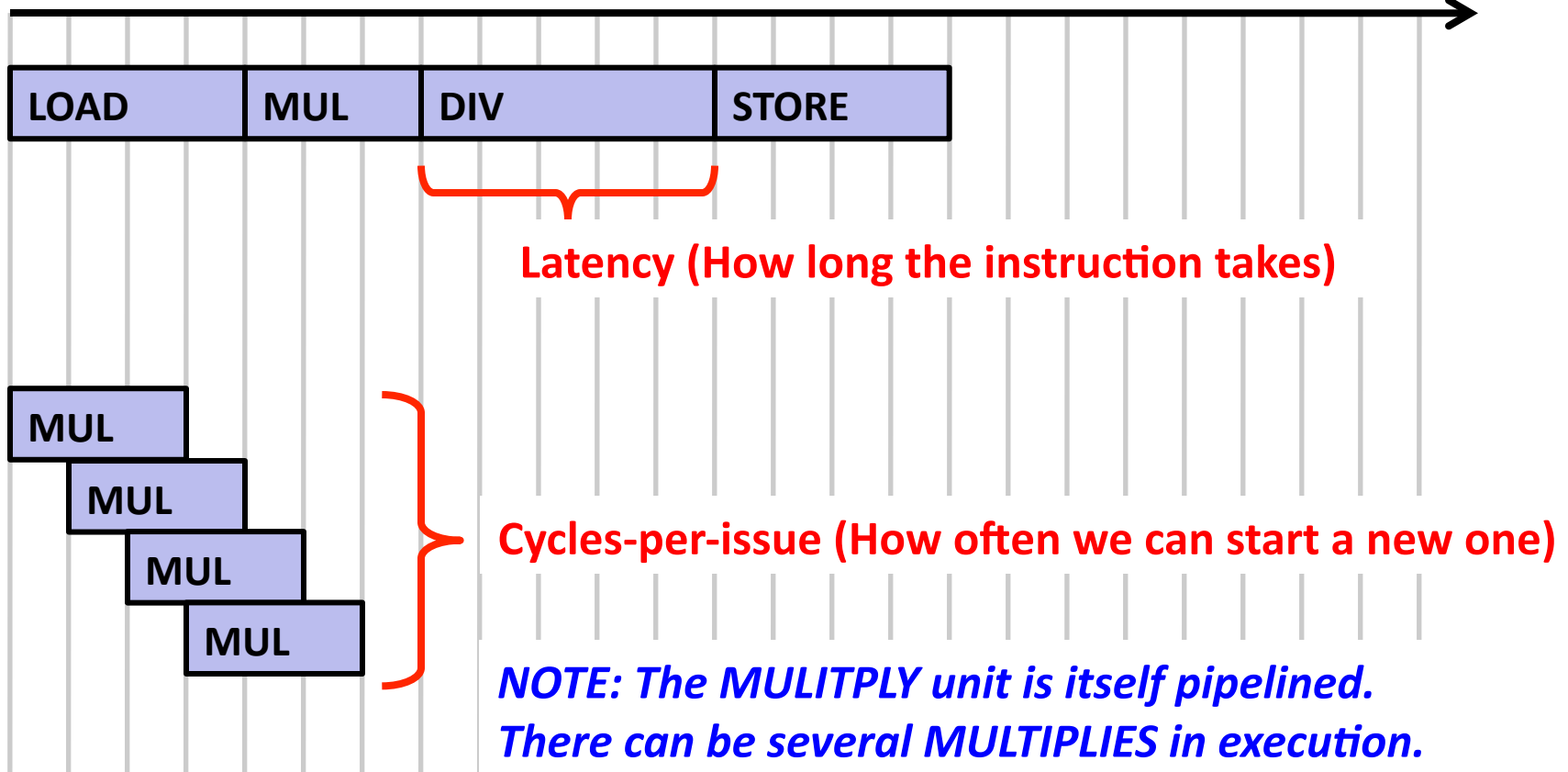
- Data dependencies



Latency and Cycles-Per-Issue

Even though a unit (e.g., MUL) takes several cycles, *it is itself pipelined*.

The “cycles-per-issue” is how often we can start a new one

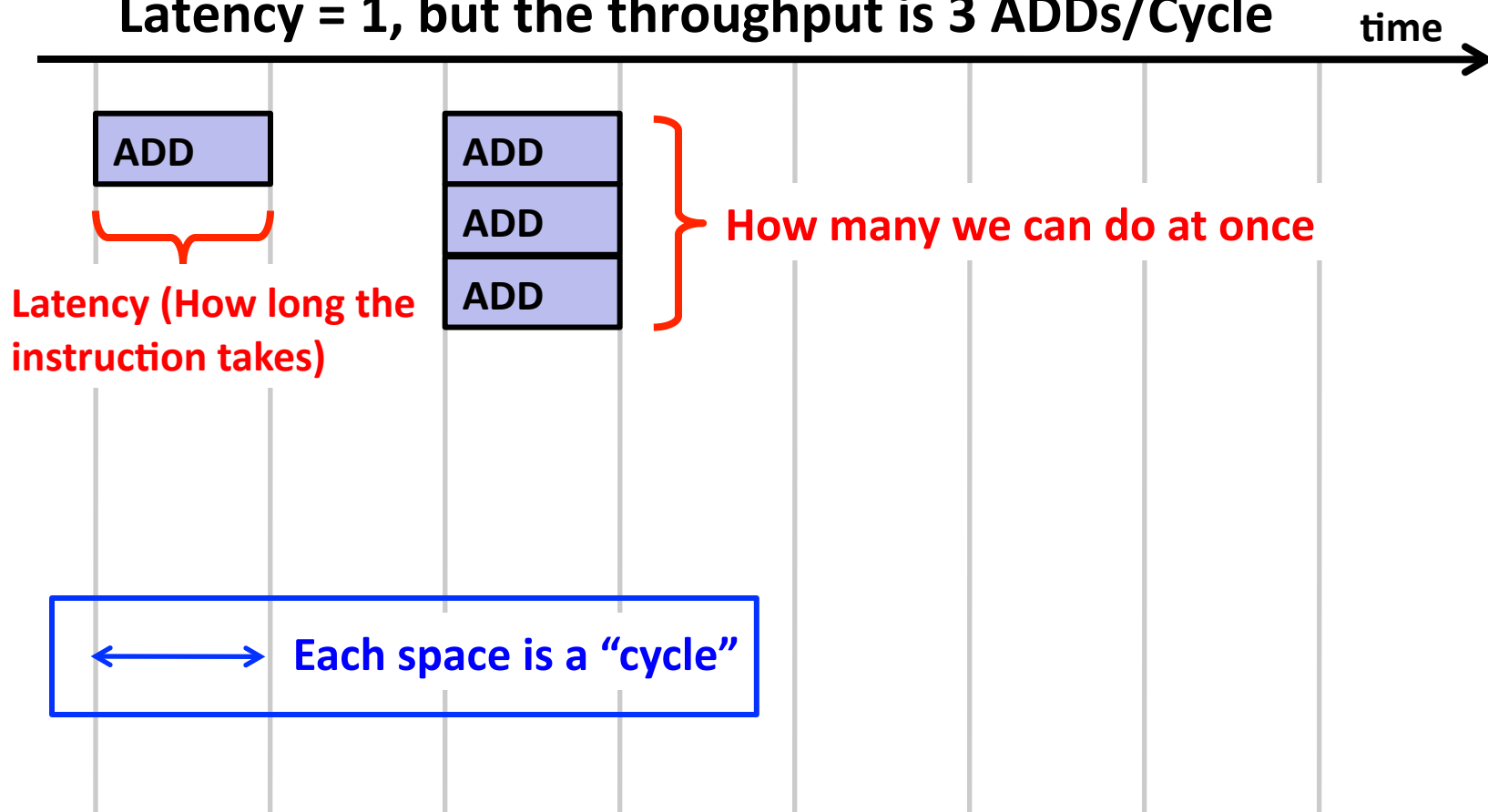


Integer ADD: More than one ADD unit

There are several (e.g., 3) addition units

Three ADDs can be started or executed at once.

Latency = 1, but the throughput is 3 ADDs/Cycle

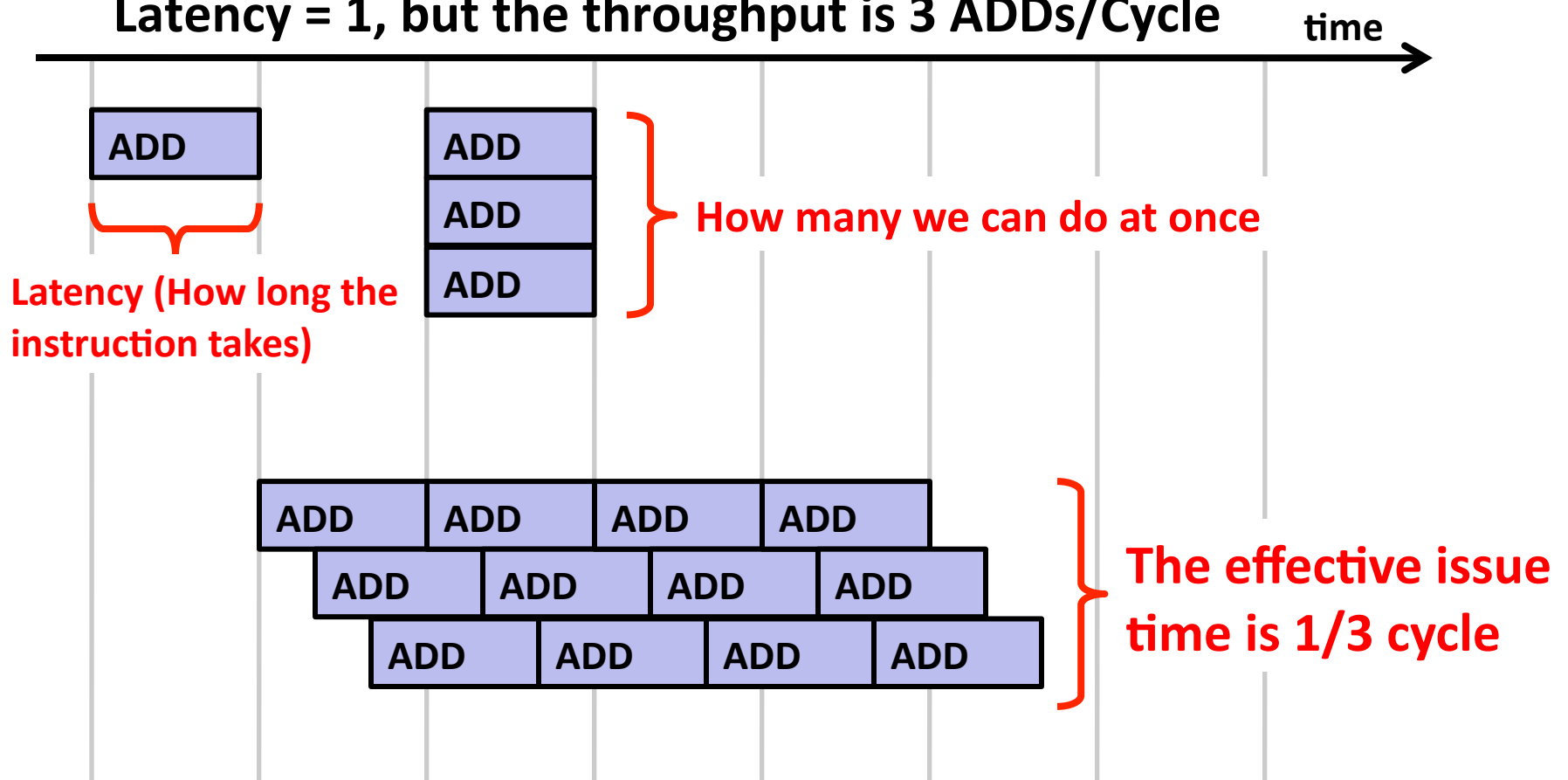


Integer ADD: More than one ADD unit

There are several (e.g., 3) addition units

Three ADDs can be started or executed at once.

Latency = 1, but the throughput is 3 ADDs/Cycle



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 Divide	3-30	3-30
FP Multiply	5	1
FP Add	3	1
FP Divide	3-15	3-15

x86-64 Compilation of Combine4

Look at one case: Integer Multiply

Look at the inner loop.

```
void combine4(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    int *d = get_vec_start(v);
    int t = 1;
    for (i = 0; i < length; i++)
        t = t * d[i];
    *dest = t;
}
```

x86-64 Compilation of Combine4

Look at one case: Integer Multiply

Look at the inner loop.

```
.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
```

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

**It seems limited by the MUL instruction...
Can we make it go any faster?**

Loop Unrolling

Before: Each iteration of the loop executes the loop body 1 time.

Idea: Each iteration of the loop executes the loop body 2 times.

```
for (i = 0; i < n; i++)  
    a[i] = b[i] * c[i];  
}
```

```
for (i = 0; i < n-1; i+=2)  
    a[i]    = b[i]    * c[i];  
    a[i+1] = b[i+1] * c[i+1];  
}
```

Loop Unrolling

Before: Each iteration of the loop executes the loop body 1 time.

Idea: Each iteration of the loop executes the loop body 2 times.

```
for (i = 0; i < n; i++)  
    a[i] = b[i] * c[i];  
}
```

```
for (i = 0; i < n-1; i+=2)  
    a[i] = b[i] * c[i];  
    a[i+1] = b[i+1] * c[i+1];  
}  
if (i < n) {  
    a[i] = b[i] * c[i];  
}
```

n=15

0, 2, 4, 6, 8, 10, 12, 14

14

Loop Unrolling (4 ×)

Before: Each iteration of the loop executes the loop body 1 time.

Idea: Each iteration of the loop executes the loop body 4 times.

```
for (i = 0; i < n; i++)  
    a[i] = b[i] * c[i];  
}
```

```
for (i = 0; i < n-3; i+=4)  
    a[i]    = b[i]    * c[i];  
    a[i+1] = b[i+1] * c[i+1];  
    a[i+2] = b[i+2] * c[i+2];  
    a[i+3] = b[i+3] * c[i+3];  
}  
for (; i < n; i++)  
    a[i] = b[i] * c[i];  
}
```

n=15

0, 4, 8, 12

12, 13, 14

Loop Unrolling (2 × 1 unrolling)

```
void unroll2a_combine(vec_ptr v, data_t *dest) {
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int 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;
}
```

Performs 2× more useful work per iteration

Effect of Loop Unrolling

	Integer		Double FP	
	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

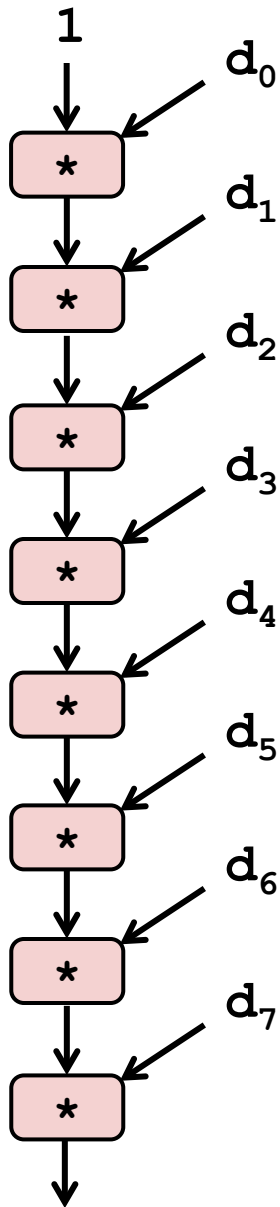
Achieves latency bound

Others don't improve. *Why?*

There is a sequential data dependency

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

What is Combine4 really doing?



Example Computation

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

Note the **sequential dependence**

Performance is limited by **latency** of MUL

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

Reassociating the operations

```
void unroll2aa_combine(vec_ptr v, data_t *dest) {
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int 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;
}
```

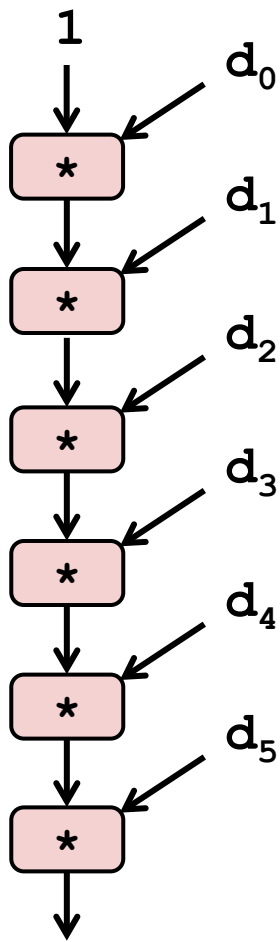
Compare to before

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

Can this change the result of the computation?

Yes, for Floating Point. *Why?*

Reassociated Computation

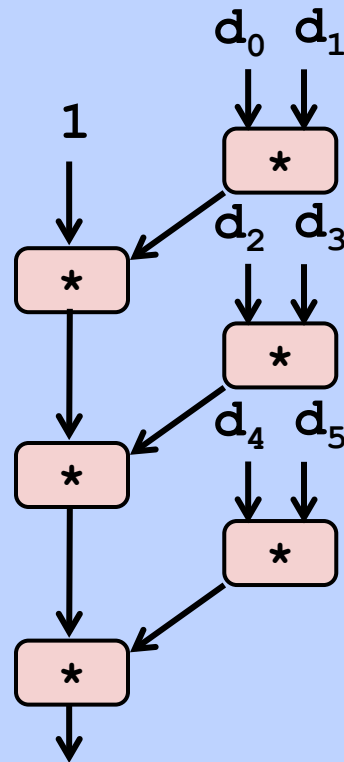


$$x = (x \text{ OP } d[i]) \text{ OP } d[i+1];$$

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

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

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



Effect of Reassociation

	Integer		Double FP	
	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 *

Why? Breaks sequential dependency

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

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

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

Reassociated Computation

What changed?

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

Overall Performance

Number of elements = N

Number of operations = $N/2 + 1$

Latency = D cycles per op

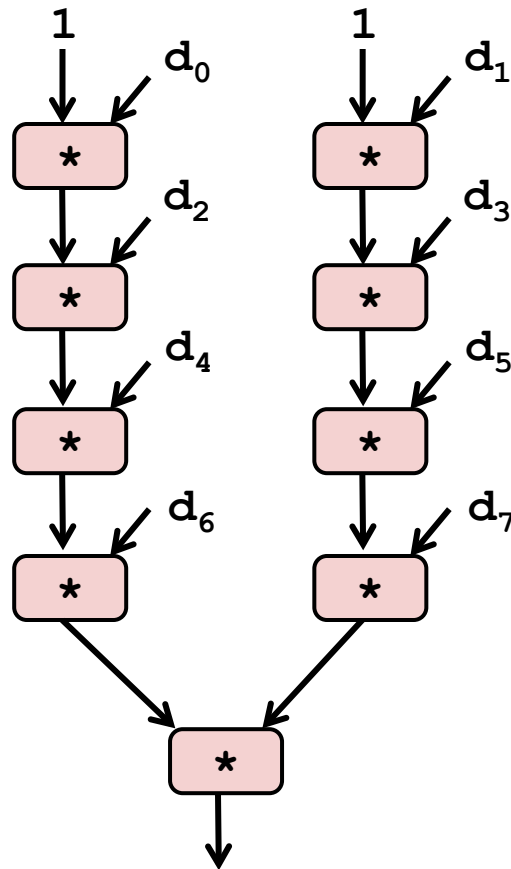
Total cycles = $(N/2 + 1) \times D$ cycles

$\approx N \times D/2$

Measured CPE = $D/2$!!! (for int *, FP +, FP *)

New Idea: Use Separate Accumulators

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



What changed?

Two independent “streams” of operations

Loop Unrolling with Separate Accumulators

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int 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;
}
```

A different form of reassociation

Effect of Separate Accumulators

	Integer		Double FP	
	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

- **2x speedup (over unroll2) for Int *, FP +, FP ***
 - Breaks sequential dependency in a “cleaner,” more obvious way

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

Effect of Separate Accumulators

	Integer		Double FP	
	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

Some improvement...

Theoretical Limit?

The Throughput Bound

Unrolling & Accumulating

Ideas:

We can unroll to any degree L

We 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

- Must finish off iterations sequentially

Effects of Unrolling & Accumulating

Example Case: FP *

Intel Haswell

- Latency bound: 5.00
- Throughput bound: 0.50

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	
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

Accumulators

Achievable Performance

	Integer		Double FP	
	Add	Mult	Add	Mult
Original	22.68	20.02	19.98	20.18
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

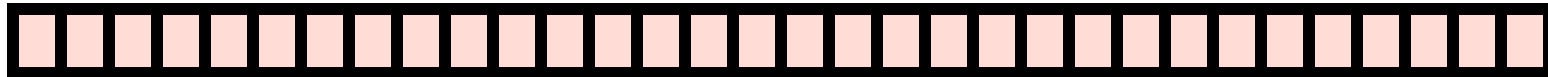
Up to **42× improvement** over original, unoptimized code!

Programming with AVX2

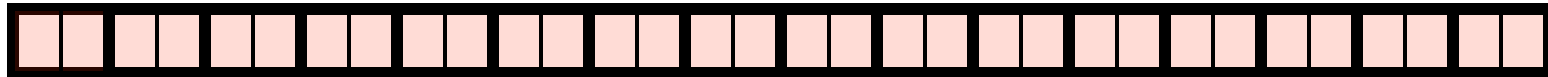
YMM Registers (%ymm0 .. %ymm15)

16 registers, each 32 bytes

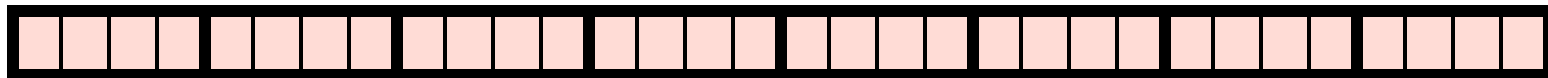
32 single-byte integers



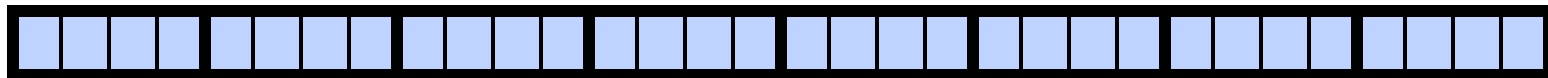
16 16-bit integers



8 32-bit integers



8 single-precision floats



4 double-precision floats



1 single-precision float



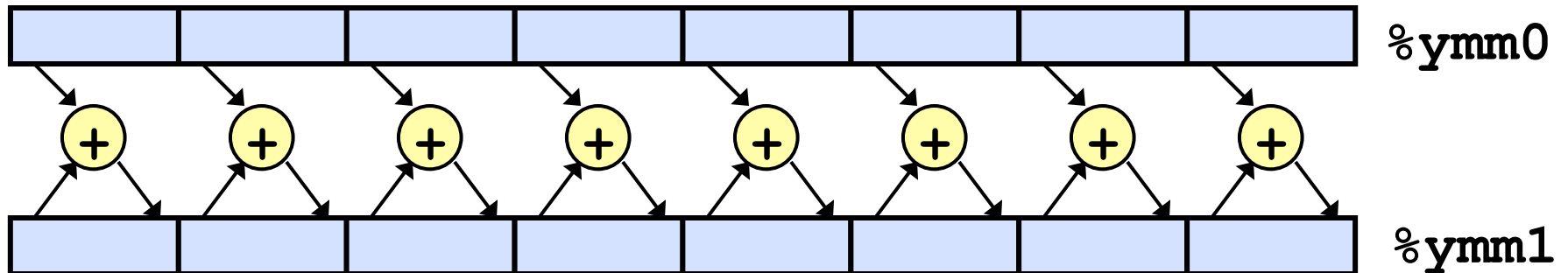
1 double-precision float



SIMD Operations

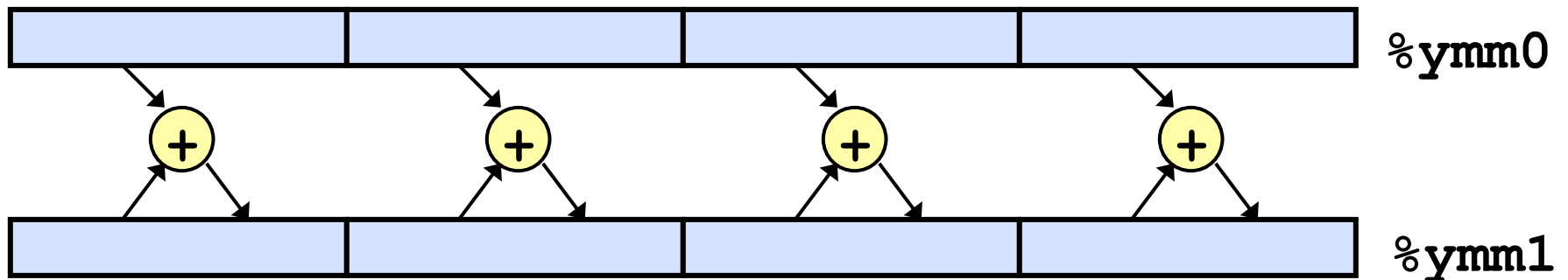
■ Single Precision

`vaddsd` `%ymm0, %ymm1, %ymm1`



■ Double Precision

`vaddpd` `%ymm0, %ymm1, %ymm1`



Using Vector Instructions

	Integer		Double FP	
	Add	Mult	Add	Mult
Best (Scalar)	0.54	1.01	1.01	0.52
Vector Version	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

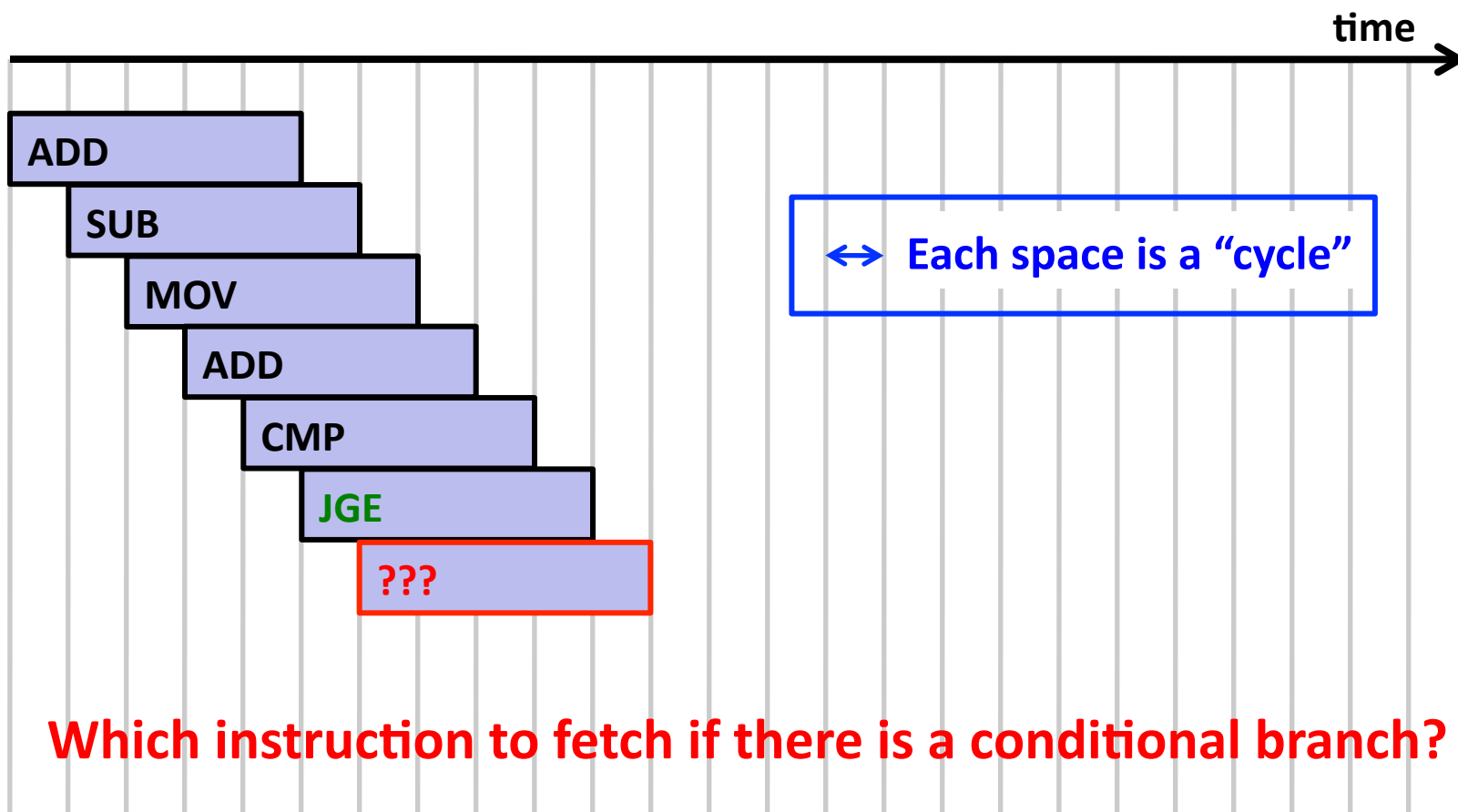
Make use of AVX Instructions

Parallel operations on multiple data elements

See Web Aside OPT:SIMD on CS:APP web page

Pipelined Instruction Execution

Fetch next instruction before previous instruction finishes.



The Pipeline: What About Branches?

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

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

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

. . .

404685:  mov    ...
40468a:  add    ...
40468d:  sub    ...
```

} In execution

← How to continue?

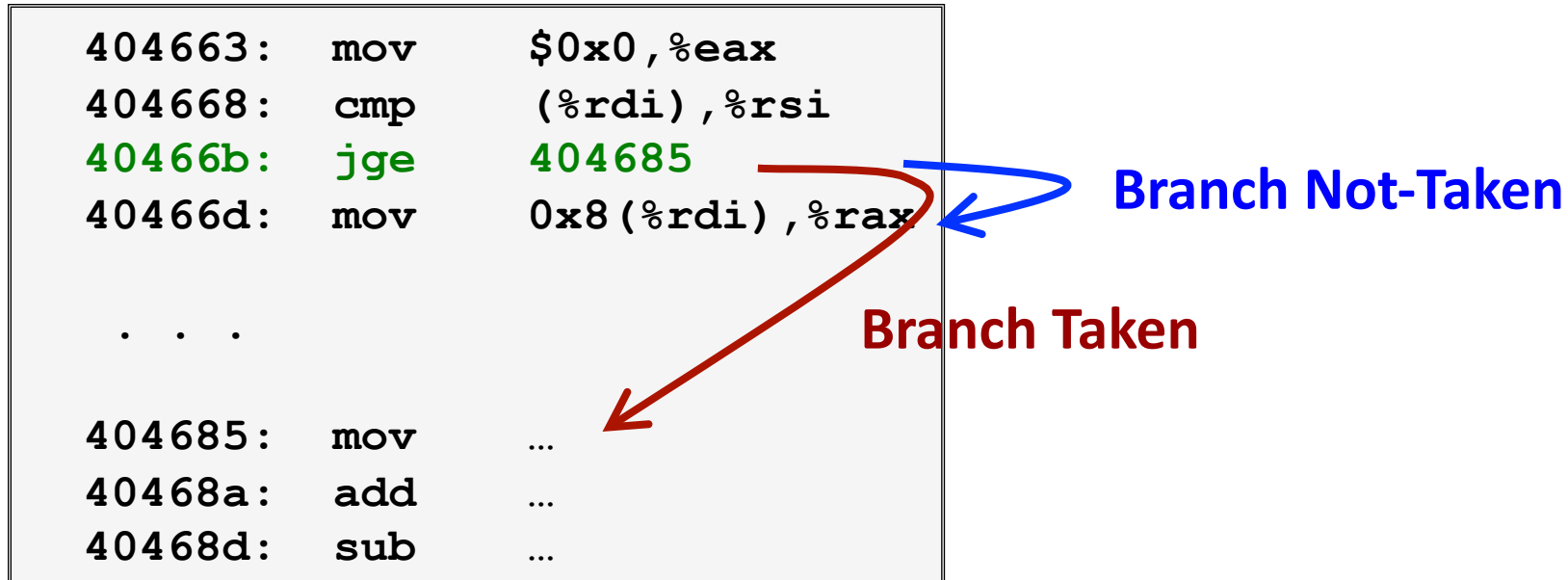
Branch Outcomes

When CPU encounters a conditional branch,
it cannot determine where to continue fetching.

Branch Taken: Transfer control to branch target

Branch Not-Taken: Continue with next instruction in sequence

Can't be sure until the outcome is determined by branch/integer unit



“Branch Prediction”

Guess which way branch will go!

Begin executing instructions at predicted position

...but must not modify register or memory data !

```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax
. . .
404685:  mov    ...
40468a:  add    ...
40468d:  sub    ...
```

Branch Not-Taken

Keep fetching and executing here

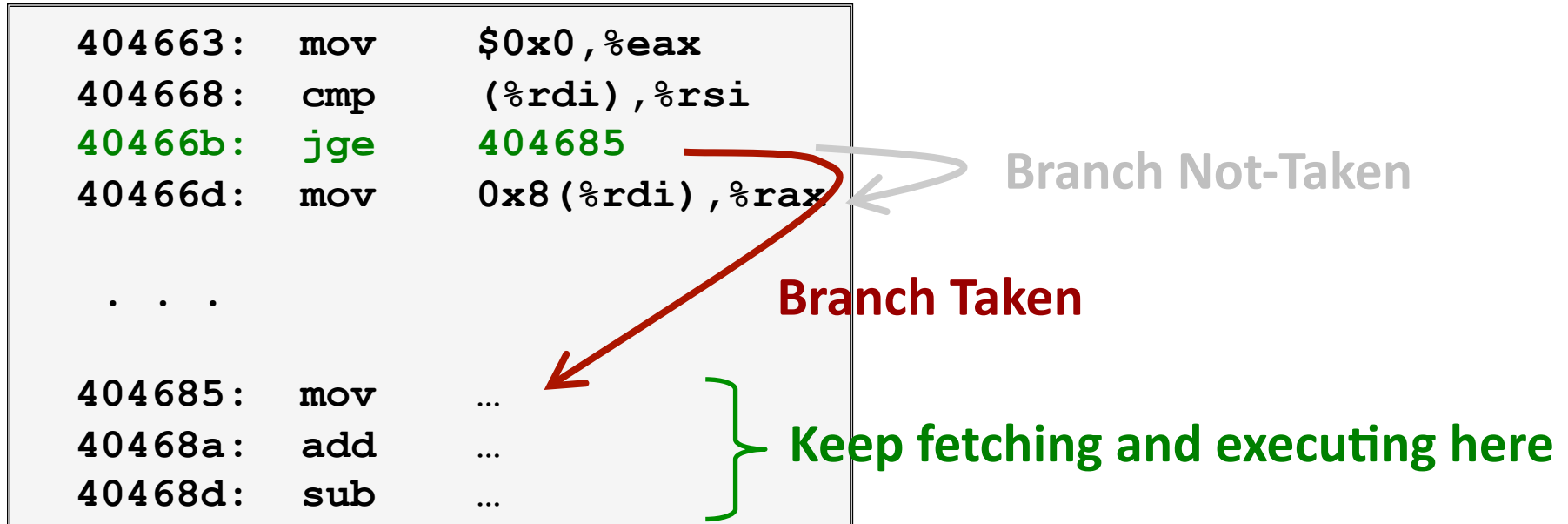
Branch Taken

“Branch Prediction”

Guess which way branch will go!

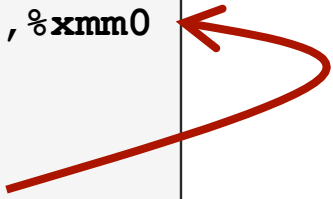
Begin executing instructions at predicted position

...but must not modify register or memory data !



Branch Prediction

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



What is the best guess?

- Jump taken
- Jump not taken

The jump **WILL BE TAKEN**.

Why?

Expanding the Loop

```

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

```

i = 98

Assume
vector length = 100

```

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

```

i = 99

Predict Taken (OK)

```

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

```

i = 100

Predict Taken
(Oops)

Bad updates
to registers
location

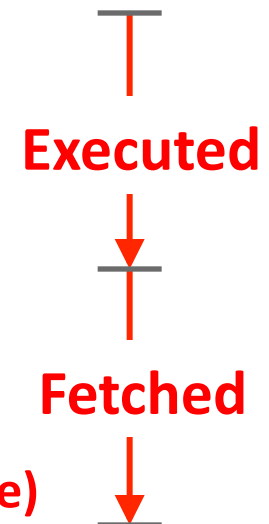
```

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

```

i = 101

Keep going
(still don't know
we made a mistake)



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)

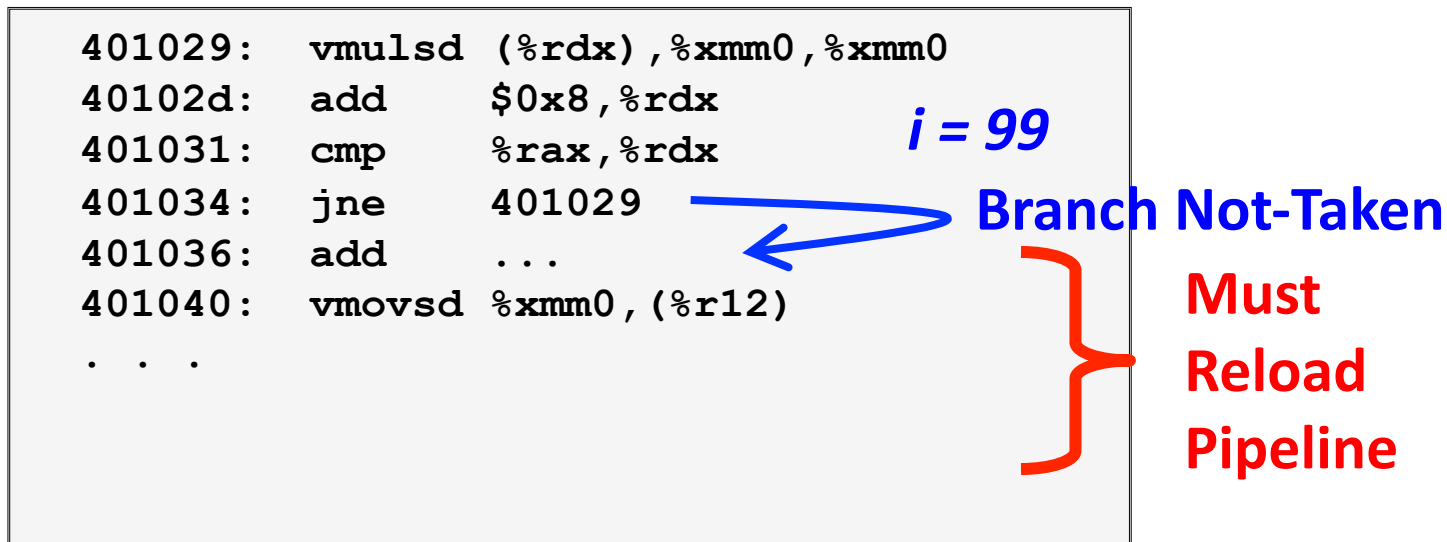
```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029    i = 100
```

Must "Invalidate" these

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

Keep going
(still don't know
we made a mistake)

Branch Misprediction Recovery

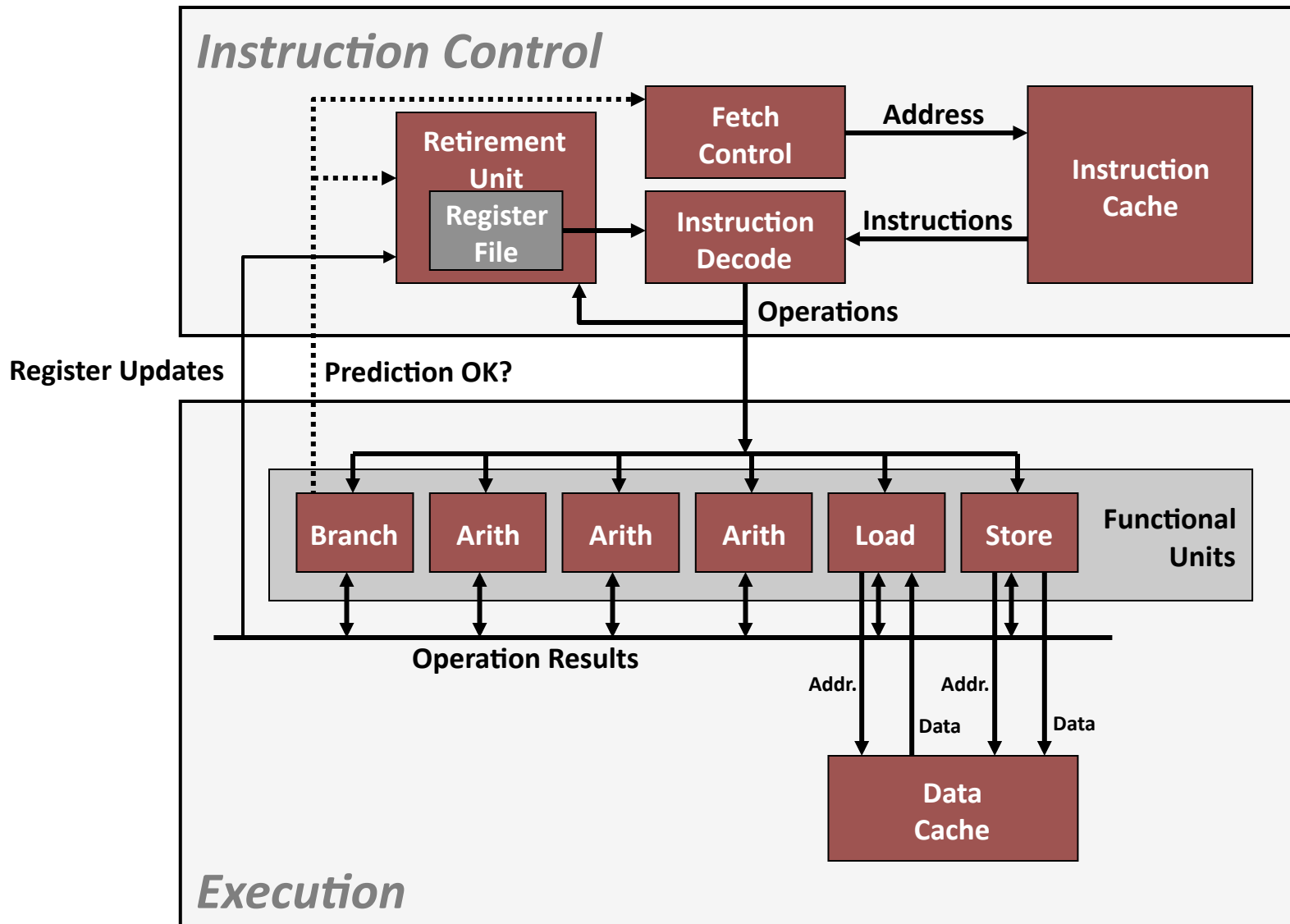


Performance Cost

Can be large (many lost clock cycles)

A major performance limiter

Modern CPU Design



Getting High Performance

- **Use good compiler and the right 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 (to be covered later)