# CS 33

## Architecture and Optimization (2)

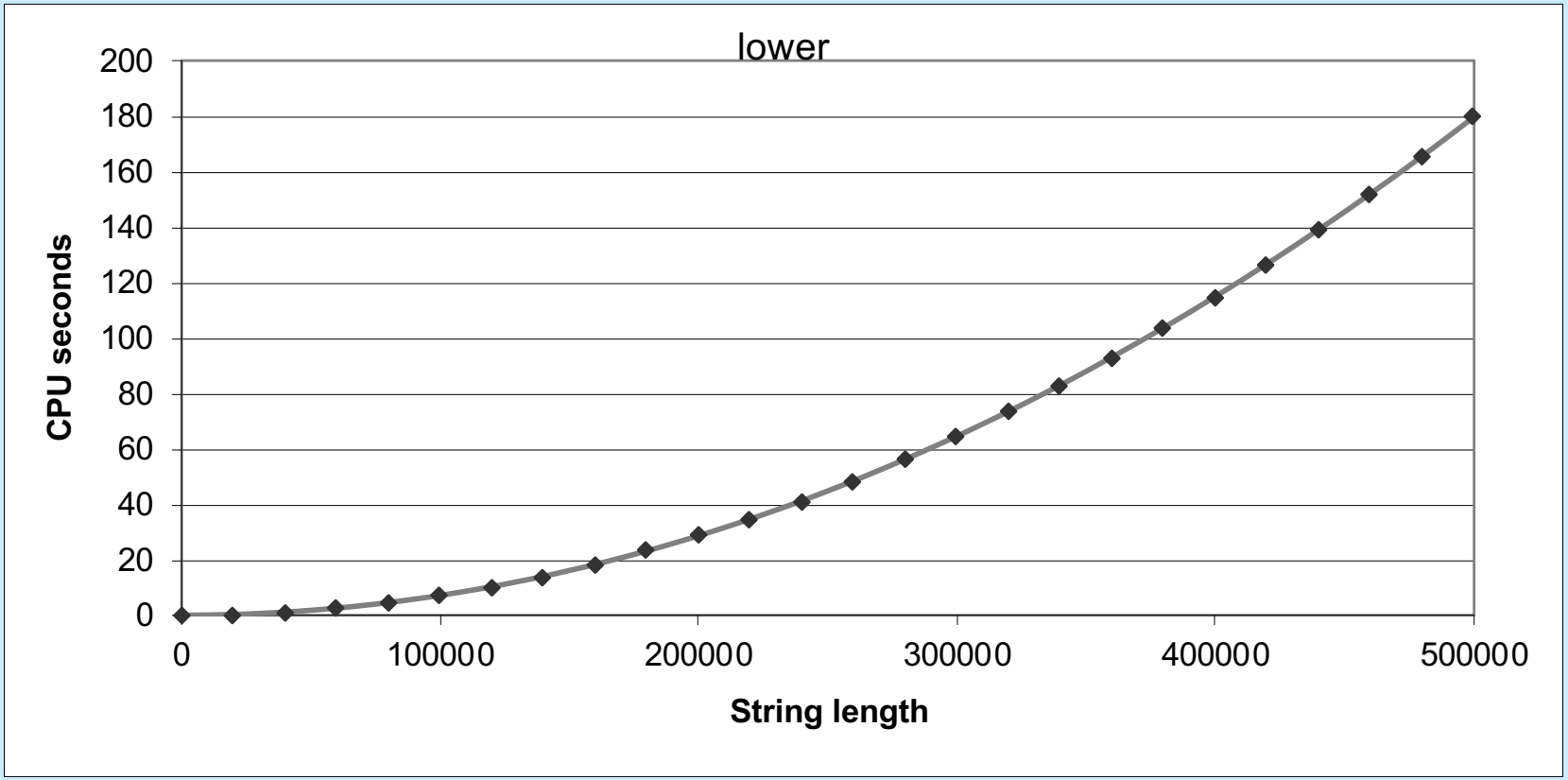
# Optimization Blocker #1: Function Calls

- **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');
}
```

# Lower Case Conversion Performance

- Time quadruples when string length doubles
- Quadratic performance



# Convert Loop To Goto Form

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

- **`strlen` executed every iteration**

# Calling Strlen

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

- **strlen performance**
  - **only way to determine length of string is to scan its entire length, looking for null character**
- **Overall performance, string of length N**
  - **N calls to strlen**
  - **overall $O(N^2)$ performance**

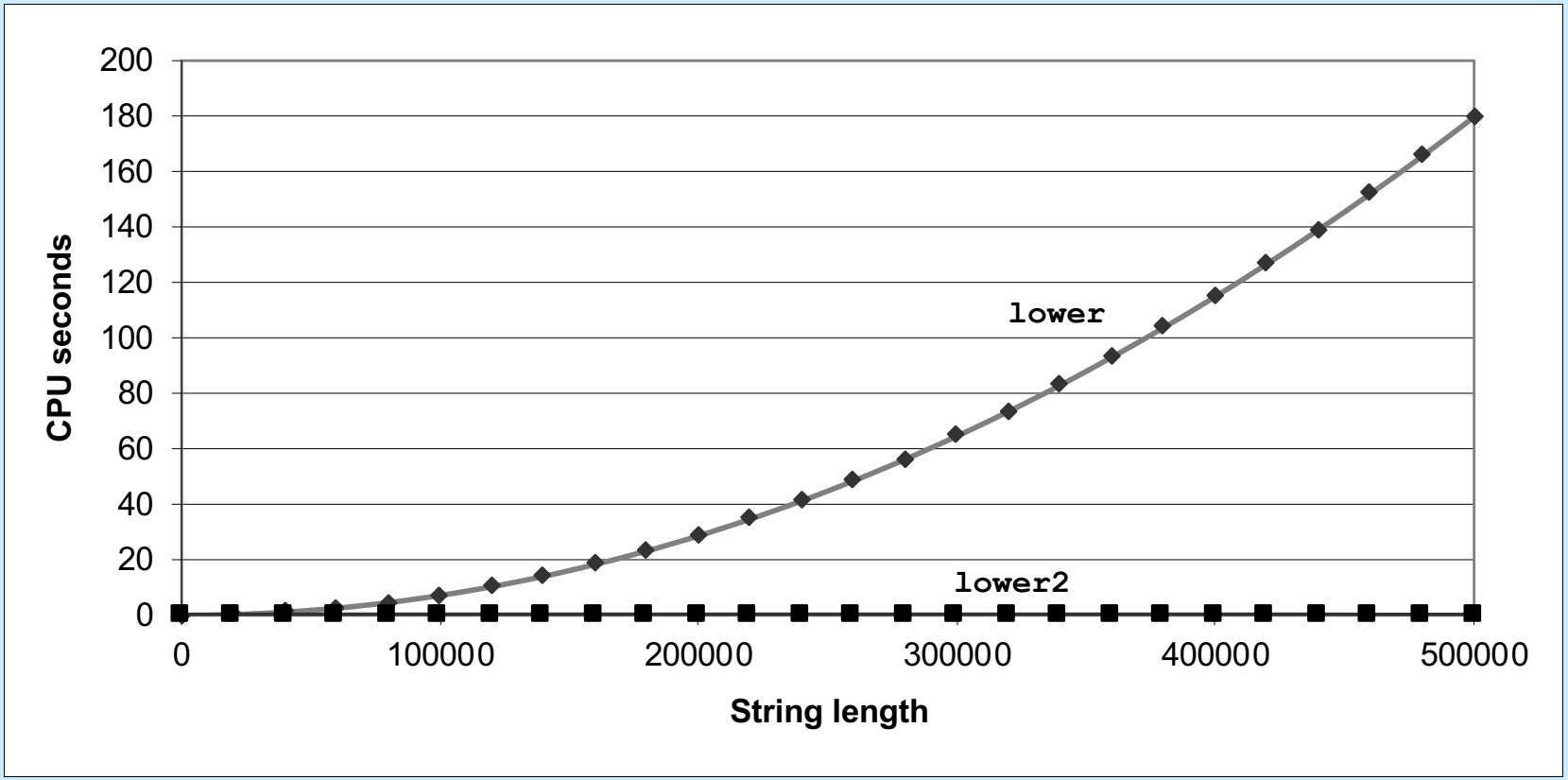
# Improving Performance

```
void lower2(char *s){
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

- **Move call to `strlen` outside of loop**
  - **since result does not change from one iteration to another**
  - **form of code motion**

# Lower-Case Conversion Performance

- **Time doubles when string-length doubles**
  - **linear performance of lower2**



# Optimization Blocker: Function Calls

- *Why couldn't compiler move* `strlen` *out of inner loop?*
  - function 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
    - » function `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

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

# Memory Matters

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

```
# sum_rows1 inner loop
.L3:
   movq    (%r8,%rax,8), %rcx    # rcx = a[i][j]
   addq    %rcx, (%rdx)          # b[i] += rcx
   addq    $1, %rax              # j++
   cmpq    %rax, %rdi            # if i<n
   jne     .L3                   # goto .L3
```

- **Code updates `b[i]` on every iteration**
- **Why couldn't compiler optimize this away?**

# Memory Aliasing

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

```
int A[3][3] =
  {{ 0,   1,   2},
   { 4,   8,  16},
   {32,  64, 128}};

int *B = &A[1][0];

sum_rows1(3, A, B;
```

**Value of `B`:**

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

```
i = 2: [3, 22, 224]
```

- **Code updates `b[i]` on every iteration**
- **Must consider possibility that these updates will affect program behavior**

# Removing Aliasing

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

```
# sum_rows2 inner loop
.L4:
   addq    (%r8, %rax, 8)), %rcx
   addq    $1, %rdi
   cmpq    %rcx, %rdi
   jne     .L4
```

- **No need to store intermediate results**

# Optimization Blocker: Memory Aliasing

- Aliasing
  - two different memory references specify single location
  - easy to have happen in C
    - since allowed to do address arithmetic
    - direct access to storage structures
  - get in habit of introducing local variables
    - accumulating within loops
    - your way of telling compiler not to check for aliasing

# C99 to the Rescue

- **New attribute**
  - **restrict**
    - » **applied to a pointer, tells the compiler that the object pointed to will be accessed only via this pointer**
    - » **compiler thus doesn't have to worry about aliasing**
    - » **but the programmer does ...**
    - » **syntax**

```
int *restrict pointer;
```

# Pointers and Arrays

- `long a[][n]`
  - a is a 2-D array of longs, the size of each row is n
- `long (*b)[n]`
  - b is a pointer to a 1-D array of size n

- a and b are of the same type

# Memory Matters, Fixed

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

```
# sum_rows1 inner loop
.L3:
   addq    (%rdi), %rax
   addq    $8, %rdi
   cmpq    %rcx, %rdi
   jne     .L3
```

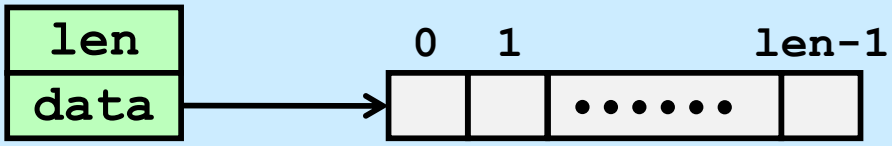
- **Code doesn't update `b[i]` on every iteration**

# 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 have dramatic performance improvement
  - compilers often cannot make these transformations
  - lack of associativity and distributivity in floating-point arithmetic

# Benchmark Example: Datatype for Vectors

```
/* data structure for vectors */
typedef struct{
    int len;
    data_t *data;
} vec_t, *vec_ptr_t;
```




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

/* return length of vector */
int vec_length(vec_ptr_t v) {
     return v->len;
}
```

# Benchmark Computation

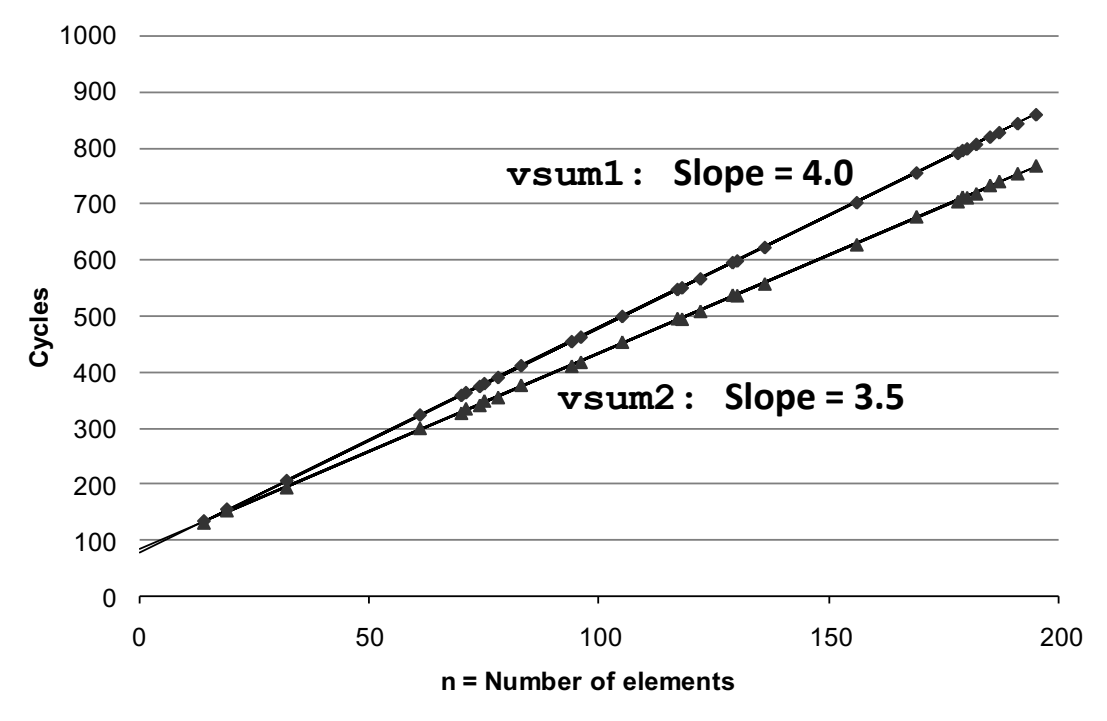
```
void combine1(vec_ptr_t v, data_t *dest){
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

**Compute sum or product of vector elements**

- **Data Types**
  - **use different declarations for `data_t`**
    - » **`int`**
    - » **`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
- T = CPE*n + Overhead
  - CPE is slope of line



# Benchmark Performance

```
void combine1(vec_ptr_t 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 | 29.0 | 29.2 | 27.4 | 27.9 |
| Combine1 –O1 | 12.0 | 12.0 | 12.0 | 13.0 |

# Move vec_length

```
void combine2(vec_ptr_t v, data_t *dest){
    long int i;
    long int length = vec_length(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| Combine1 unoptimized | 29.0 | 29.2 | 27.4 | 27.9 |
| Combine1 –O1 | 12.0 | 12.0 | 12.0 | 13.0 |
| Combine2 | 8.03 | 8.09 | 10.09 | 12.08 |

# Eliminate Function Calls

```
void combine3(vec_ptr_t v, data_t *dest){
    long int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
       *dest = *dest OP data[i];
    }
}
```

```
data_t *get_vec_start(
    vec_ptr v) {
  return v->data;
}
```

| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| Combine2 | 8.03 | 8.09 | 10.09 | 12.08 |
| Combine3 | 6.01 | 8.01 | 10.01 | 12.02 |

# Eliminate Unneeded Memory References

```
void combine4(vec_ptr_t 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;
}
```

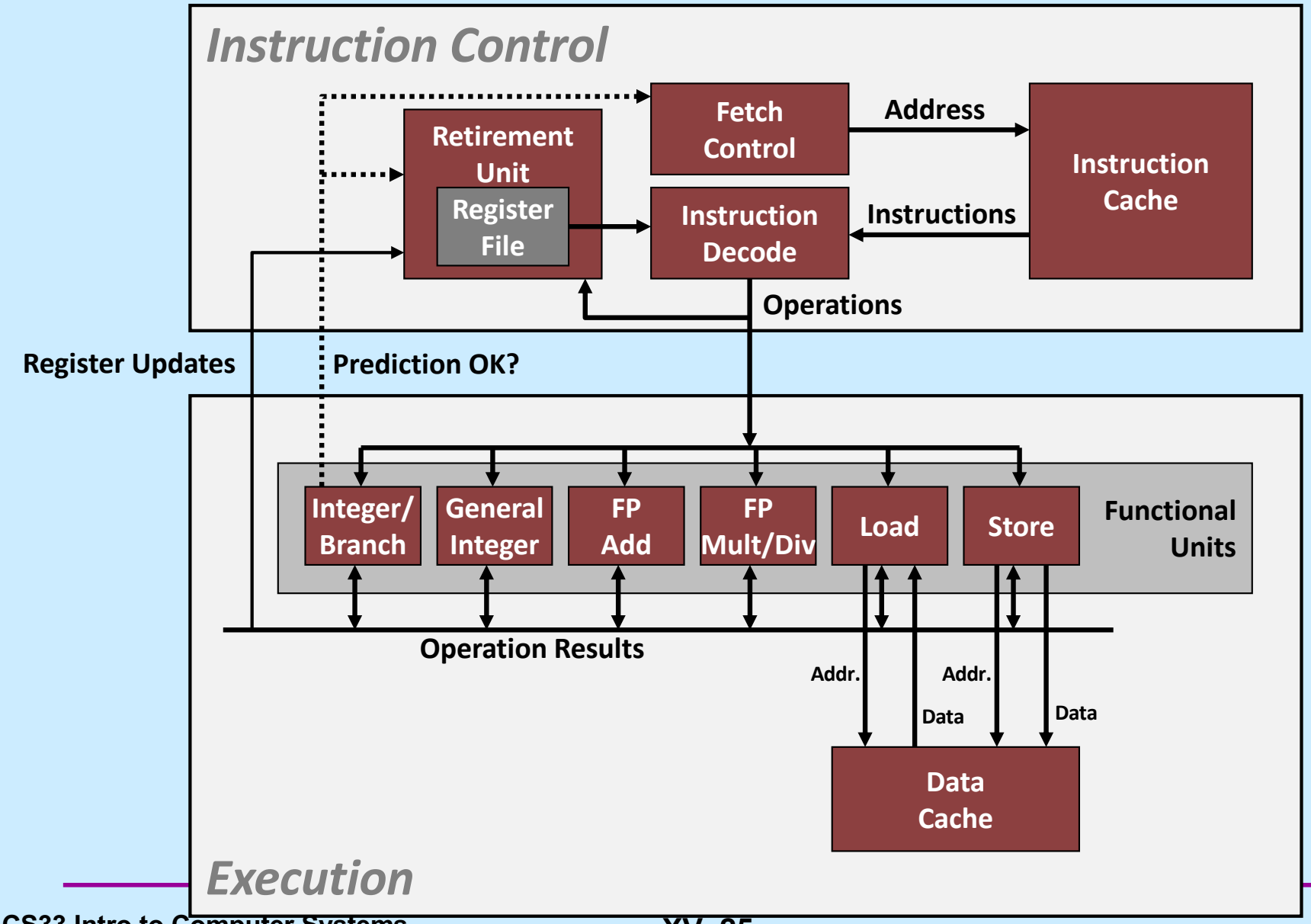
| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| Combine1 –O1 | 12.0 | 12.0 | 12.0 | 13.0 |
| Combine4 | 2.0 | 3.0 | 3.0 | 5.0 |

# Quiz 1

Combine4 is pretty fast; we've done all the "obvious" optimizations. How much faster will we be able to make it? (Hint: it involves taking advantage of pipelining and multiple functional units on the chip.)

a) 1× (it's already as fast as possible)
b) 2× – 4×
c) 16× – 64×

# Modern CPU Design



# Superscalar Processor

- **Definition:** A superscalar processor can issue and execute ***multiple instructions in one cycle***
  - instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically
    - » instructions may be executed *out of order*
- Benefit: without programming effort, superscalar processors can take advantage of the ***instruction-level parallelism*** that most programs have
- Most CPUs since about 1998 are superscalar
- Intel: since Pentium Pro (1995)

# Multiple Operations per Instruction

- addq %rax, %rdx
  - a single operation
- addq %rax, 8(%rdx)
  - three operations
    - » load value from memory
    - » add to it the contents of %rax
    - » store result in memory

# Instruction-Level Parallelism

- `addq 8(%rax), %rax`
  `addq %rbx, %rdx`
  - can be executed simultaneously: completely independent
- `addq 8(%rax), %rbx`
  `addq %rbx, %rdx`
  - can also be executed simultaneously, but some coordination is required

# Out-of-Order Execution

- 
```
movss    (%rbp), %xmm0
mulss    (%rax, %rdx, 4), %xmm0
movss    %xmm0, (%rbp)
addq     %r8, %r9
imulq    %rcx, %r12
addq     $1, %rdx
```

these can be executed without waiting for the first three to finish

# Speculative Execution

```
80489f3:    movl    $0x1,%ecx
80489f8:    xorq    %rdx,%rdx
80489fa:    cmpq    %rsi,%rdx
80489fc:    jnl     8048a25
80489fe:    movl    %esi,%edi
8048a00:    imull   (%rax,%rdx,4),%ecx
```

**perhaps execute these instructions**

# Haswell CPU

- **Functional Units**
  1) Integer arithmetic, floating-point multiplication, integer and floating-point division, branches
  2) Integer arithmetic, floating-point addition, integer and floating-point multiplication
  3) Load, address computation
  4) Load, address computation
  5) Store
  6) Integer arithmetic
  7) Integer arithmetic, branches
  8) Store, address computation

# Haswell CPU

- **Instruction characteristics**

| *Instruction* | *Latency* | *Cycles/Issue* | *Capacity* |
|---|---|---|---|
| Integer Add | 1 | 1 | 4 |
| Integer Multiply | 3 | 1 | 1 |
| Integer/Long Divide | 3-30 | 3-30 | 1 |
| Single/Double FP Add | 3 | 1 | 1 |
| Single/Double FP Multiply | 5 | 1 | 2 |
| Single/Double FP Divide | 3-15 | 3-15 | 1 |
| | | | |
| Load | 4 | 1 | 2 |
| Store | - | 1 | 2 |

# Haswell CPU Performance Bounds

| | Integer | | Floating Point | |
|---|---|---|---|---|
| | + | * | + | * |
| Latency | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput | 4.00 | 1.00 | 1.00 | 2.00 |

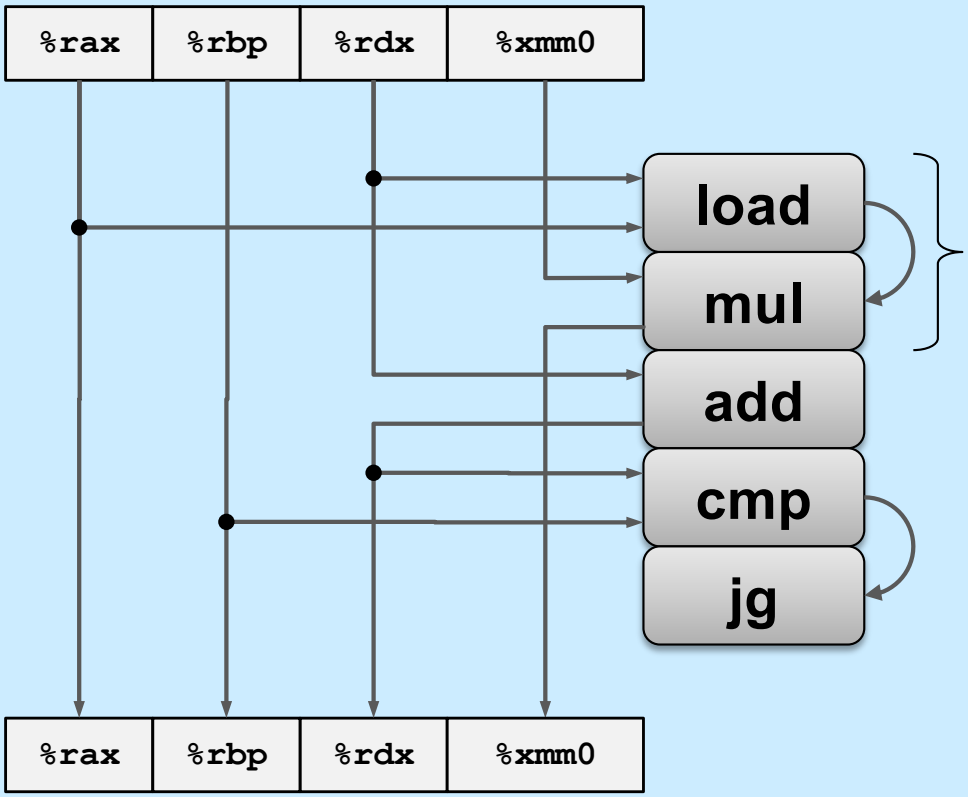
# x86-64 Compilation of Combine4

- **Inner loop (case: SP floating-point multiply)**

```
.L519:                                  # Loop:
  mullss (%rax,%rdx,4), %xmm0 # t = t * d[i]
  addq  $1, %rdx                        # i++
  cmpq  %rdx, %rbp                      # Compare length:i
  jg    .L519                           # If >, goto Loop
```

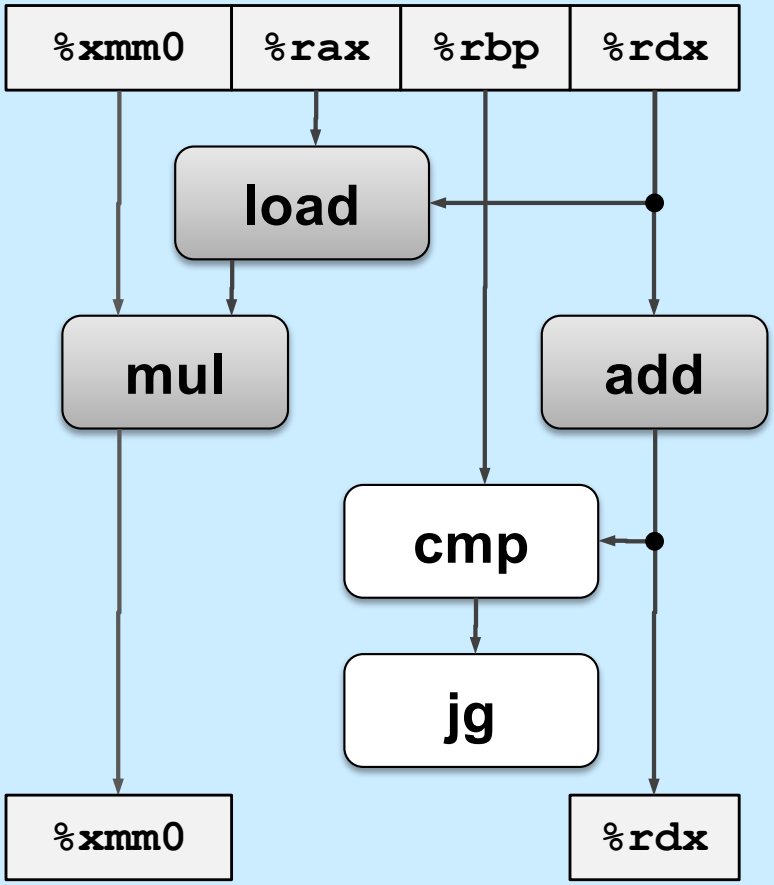
| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| Combine4 | 1.27 | 3.00 | 3.00 | 5.00 |
| Latency bound | 1.00 | 3.00 | 3.00 | 5.0 |
| Throughput bound | 0.25 | 1.00 | 1.00 | 0.50 |

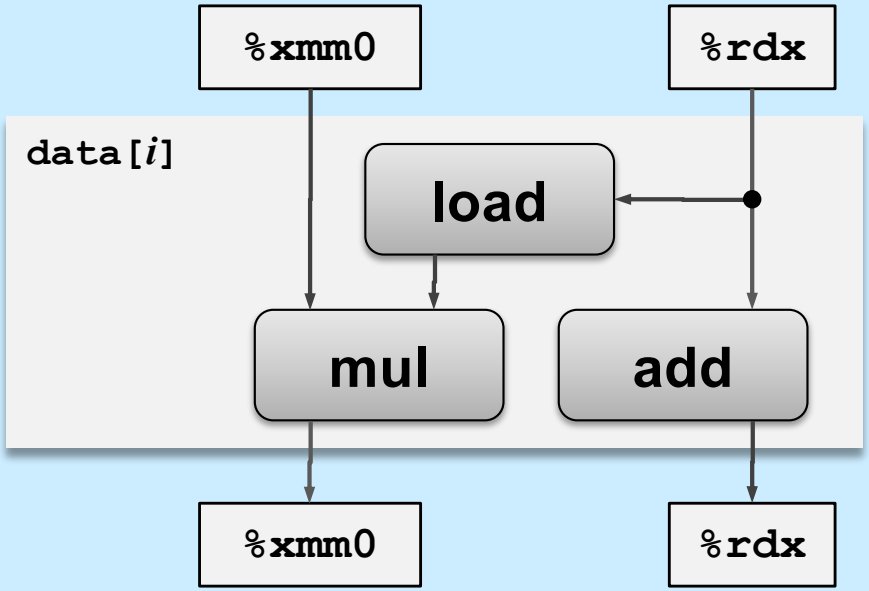
# Inner Loop



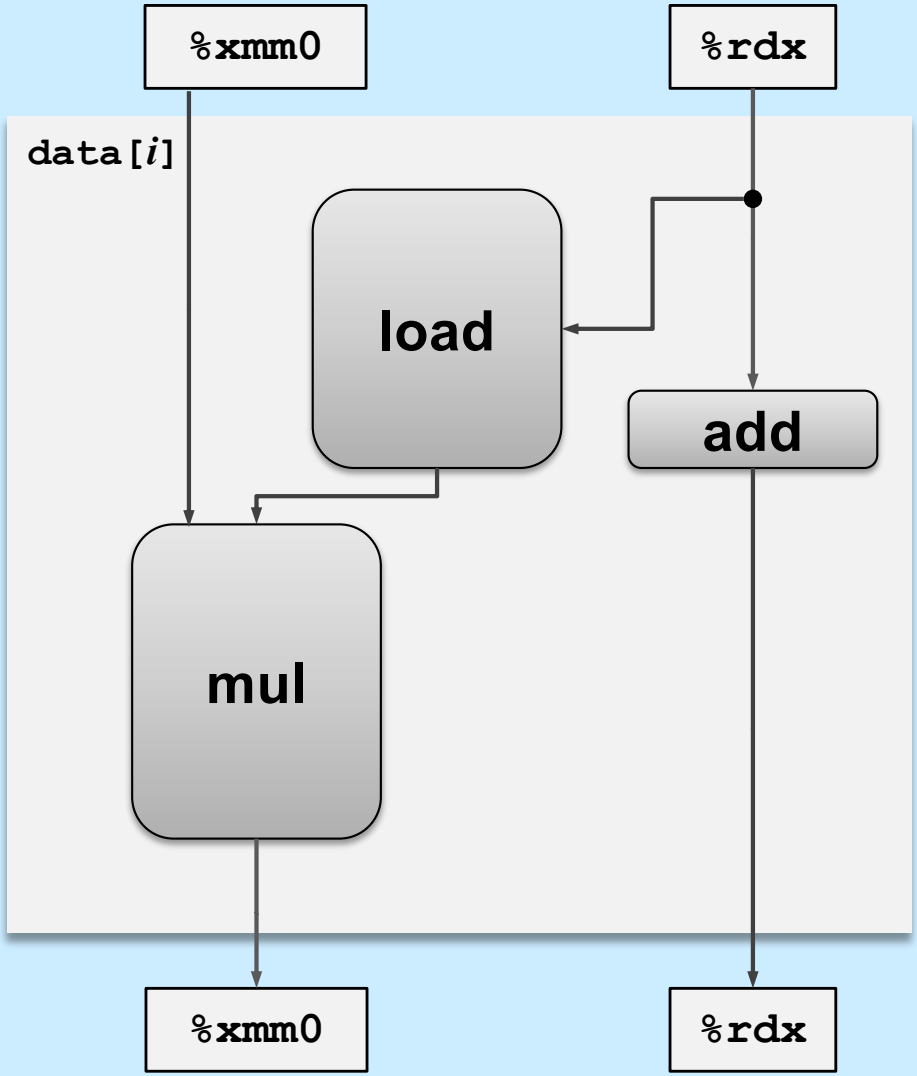

```
mulss (%rax,%rdx,4), %xmm0

addq $1,%rdx

cmpq %rdx,%rbp

jg loop
```

# Data-Flow Graphs of Inner Loop

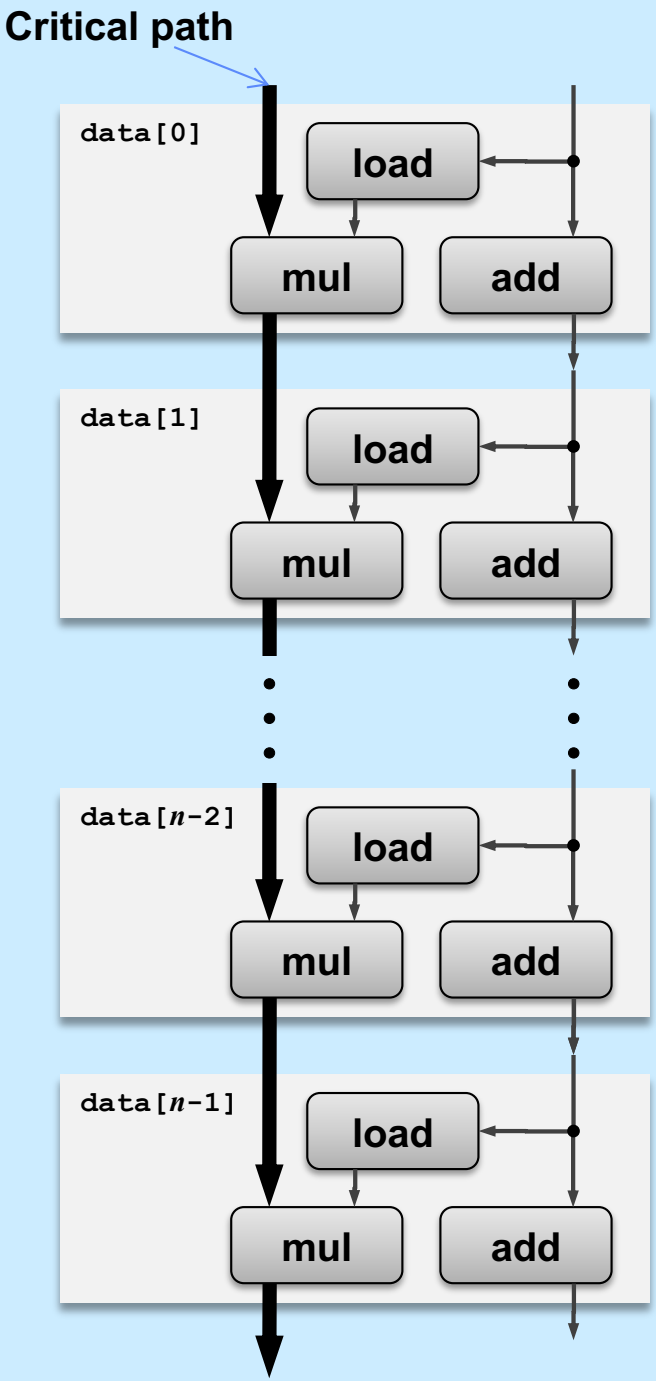


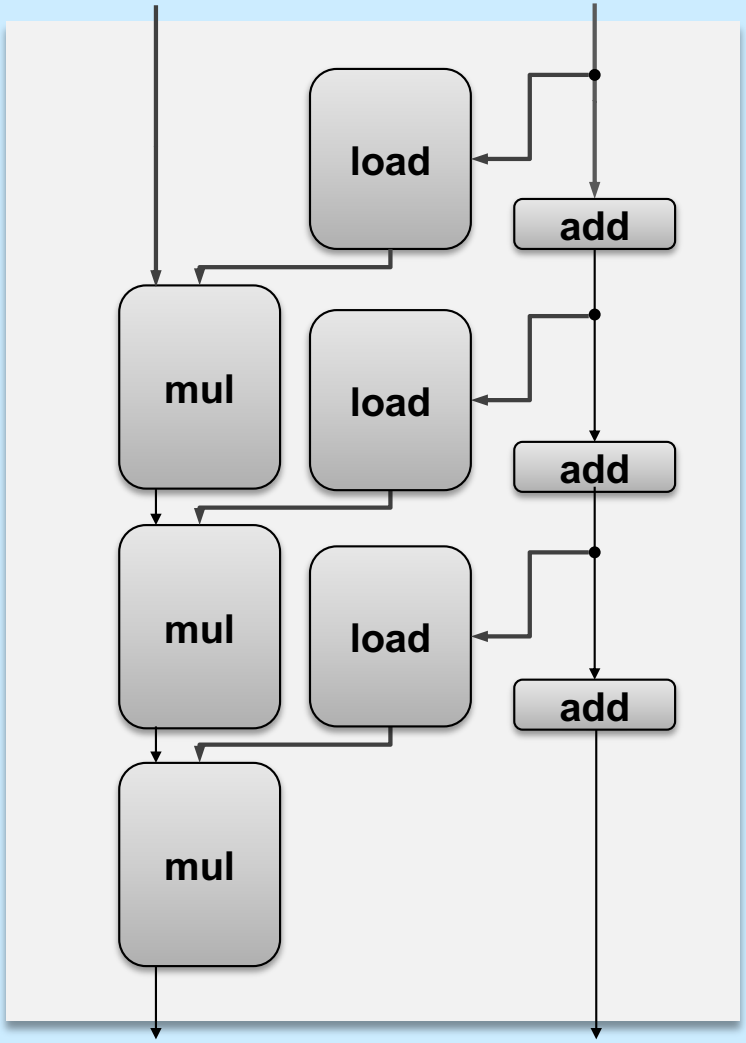
# Relative Execution Times



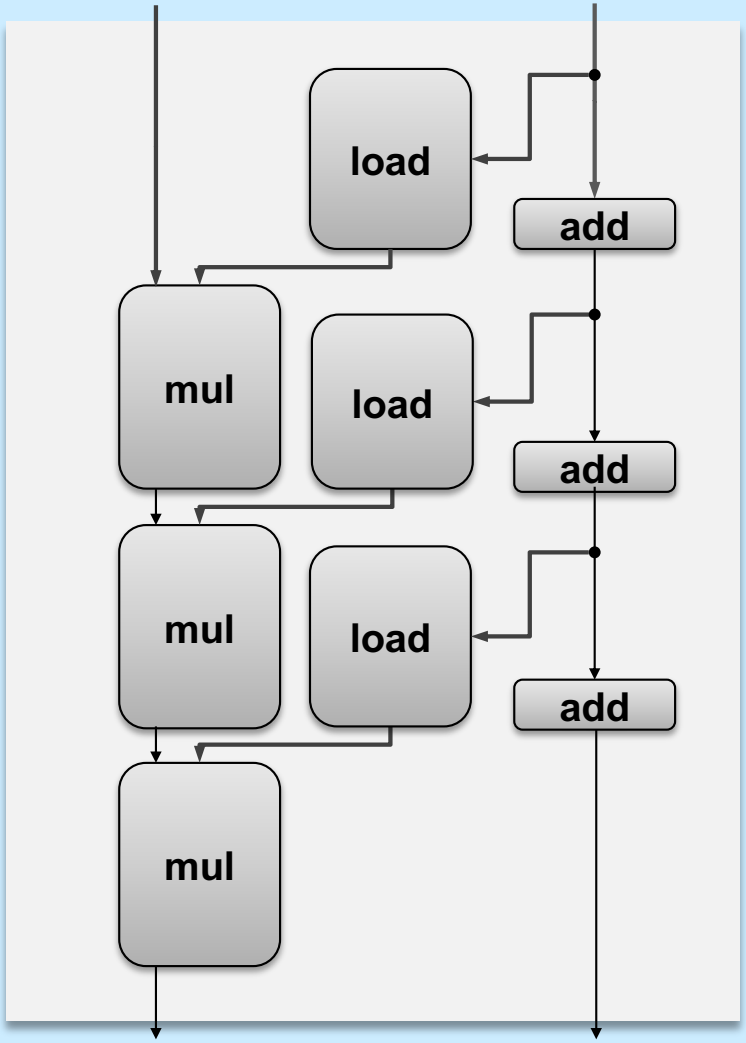
# Data Flow Over Multiple Iterations



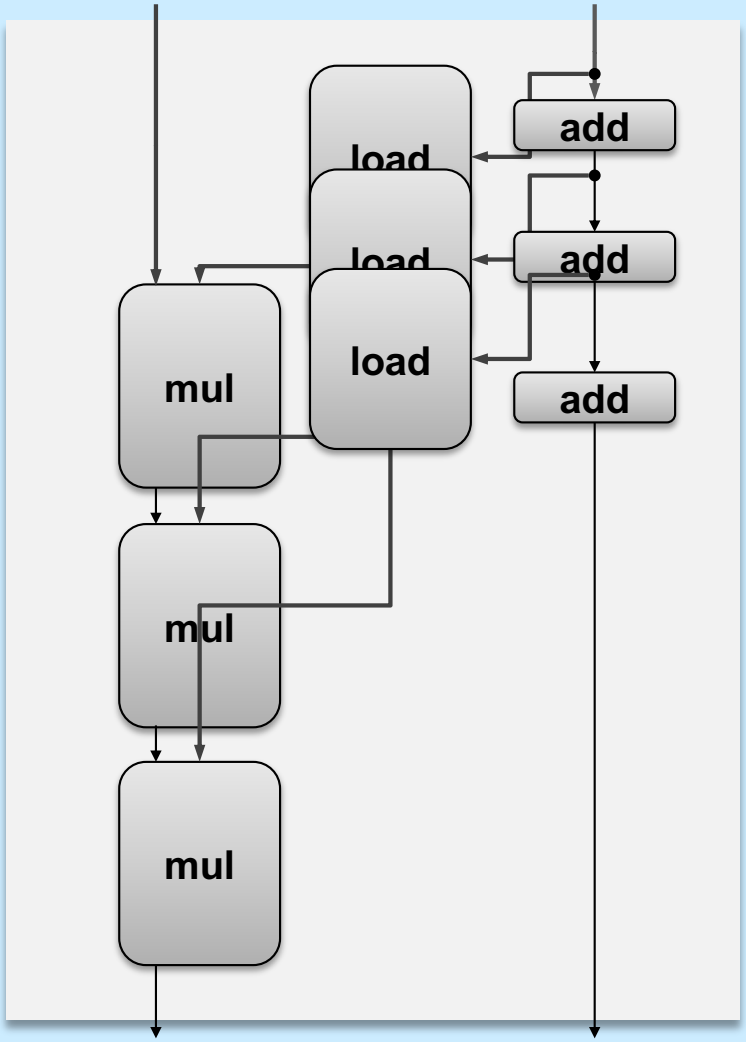
# Pipelined Data-Flow Over Multiple Iterations



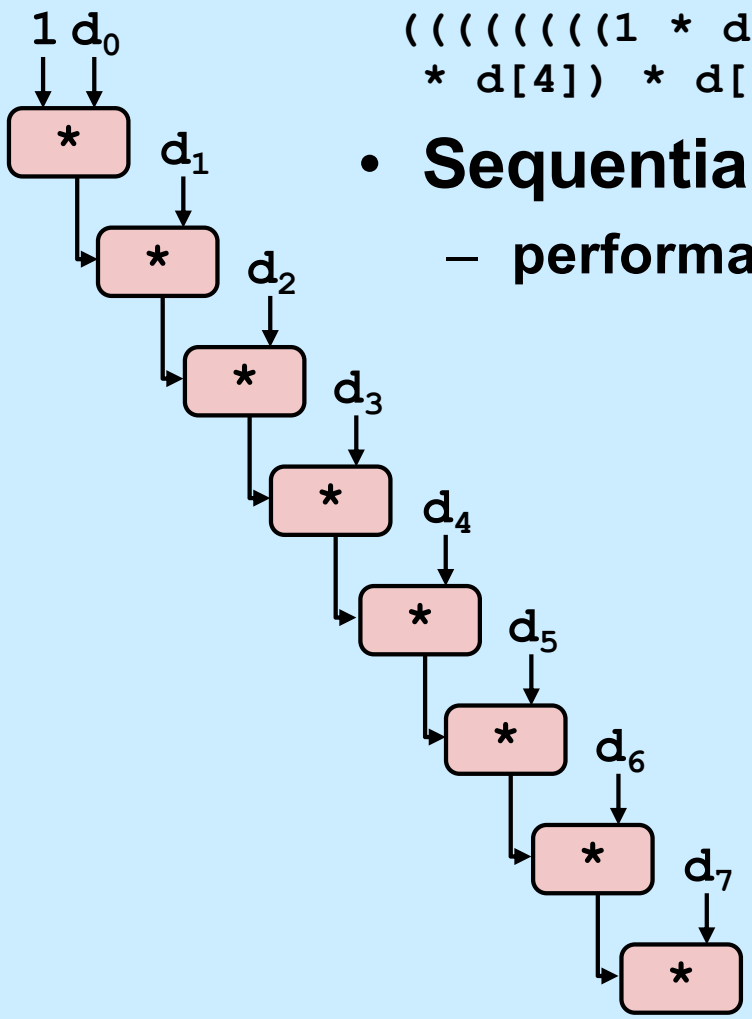
# Pipelined Data-Flow Over Multiple Iterations



# Pipelined Data-Flow Over Multiple Iterations



# 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

```
void unroll2x(vec_ptr_t 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;
}
```

- **Perform 2x more useful work per iteration**

# Effect of Loop Unrolling

| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| Combine4 | 1.27 | 3.00 | 3.00 | 5.00 |
| Unroll 2x | 1.01 | 3.00 | 3.00 | 5.00 |
| Latency bound | 1.0 | 3.0 | 3.0 | 5.0 |
| Throughput bound | 0.25 | 1.0 | 1.0 | 0.5 |

- **Helps integer add**
  - reduces loop overhead
- **Others don't improve. *Why?***
  - still sequential dependency

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

# Loop Unrolling with Reassociation

```
void unroll2xra(vec_ptr_t 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;
}
```

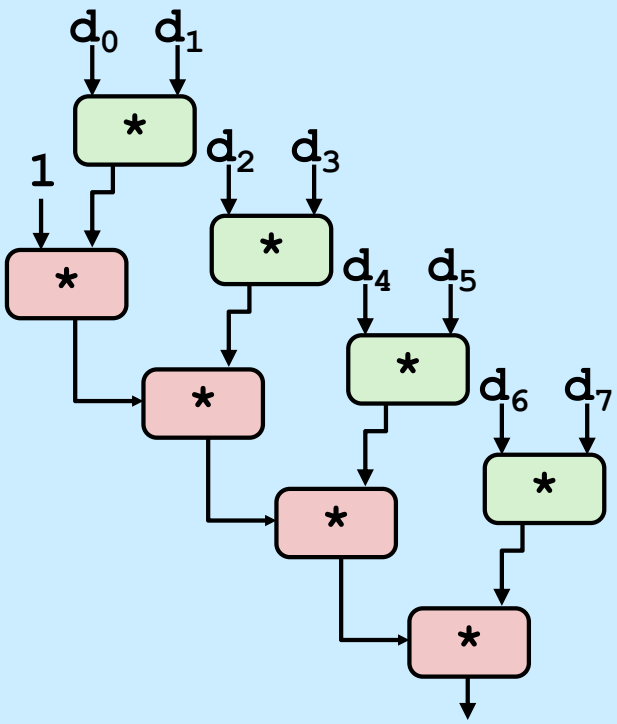
**Compare to before**

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

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

# 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
  - should be (N/2+1)*D cycles: CPE = D/2
  - measured CPE slightly worse for integer addition (there are other things going on)

# Effect of Reassociation

| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| Combine4 | 1.27 | 3.00 | 3.00 | 5.00 |
| Unroll 2x | 1.01 | 3.00 | 3.00 | 5.00 |
| Unroll 2x, reassociate | 1.01 | 1.51 | 1.51 | 2.51 |
| Latency bound | 1.0 | 3.0 | 3.0 | 5.0 |
| Throughput bound | .25 | 1.0 | 1.0 | .5 |

- Nearly 2x speedup for int *, FP +, FP *
  - reason: breaks sequential dependency

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

# Loop Unrolling with Separate Accumulators

```
void unroll2xp2x(vec_ptr_t 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;
}
```

- **Different form of reassociation**

# Effect of Separate Accumulators

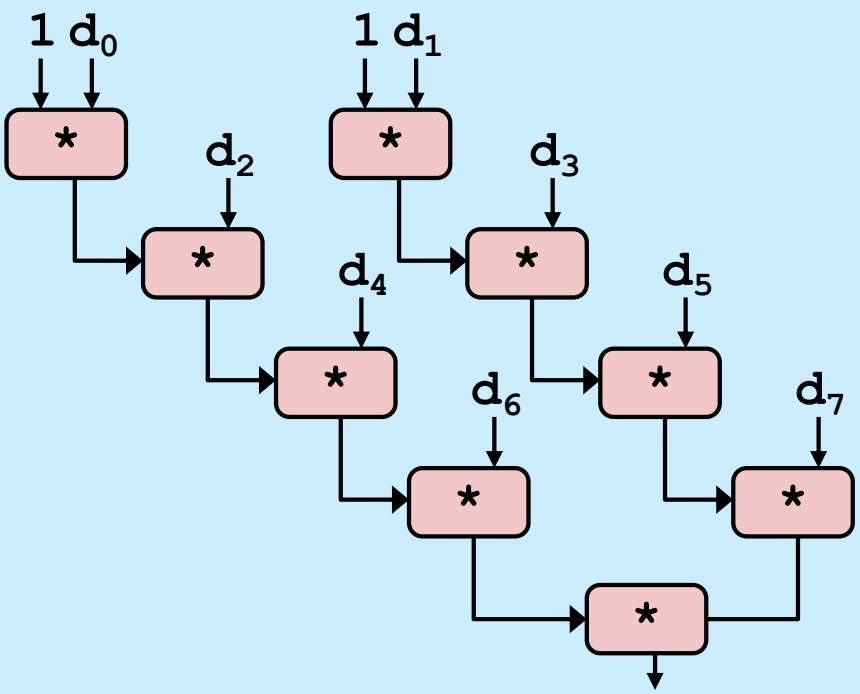
| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| Combine4 | 1.27 | 3.00 | 3.00 | 5.00 |
| Unroll 2x | 1.01 | 3.00 | 3.00 | 5.00 |
| Unroll 2x, reassociate | 1.01 | 1.51 | 1.51 | 2.01 |
| Unroll 2x parallel 2x | .81 | 1.51 | 1.51 | 2.51 |
| Latency bound | 1.0 | 3.0 | 3.0 | 5.0 |
| Throughput bound | .25 | 1.0 | 1.0 | .5 |

- 2x speedup (over unroll 2x) for int *, FP +, FP *
  - breaks sequential dependency in a "cleaner," more obvious way

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

# Separate Accumulators

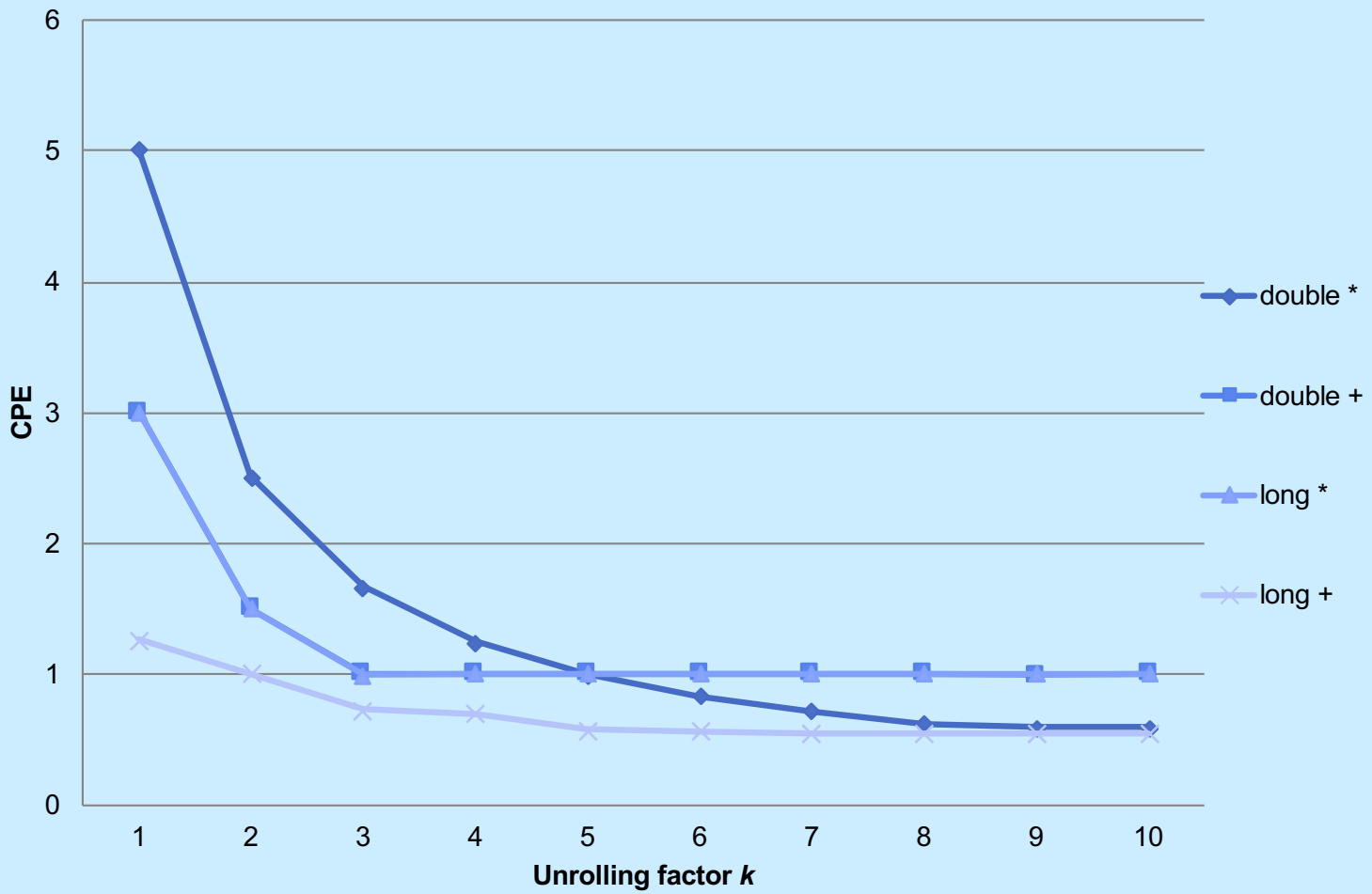
```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```




- **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**
  - Integer addition improved, but not yet at predicted value

***What Now?***

# Performance




- **K-way loop unrolling with K accumulators**
  - **limited by number and throughput of functional units**

# Achievable Performance

| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| Combine4 | 1.27 | 3.0 | 3.0 | 5.0 |
| Achievable scalar | .52 | 1.01 | 1.01 | .54 |
| Latency bound | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput bound | .25 | 1.00 | 1.00 | .5 |

# Using Vector Instructions

| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| Combine4 | 1.27 | 3.0 | 3.0 | 5.0 |
| Achievable Scalar | .52 | 1.01 | 1.01 | .54 |
| Latency bound | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput bound | .25 | 1.00 | 1.00 | .5 |
| Achievable Vector | .05 | .24 | .25 | .16 |
| Vector throughput bound | .06 | .12 | .25 | .12 |

- **Make use of SSE Instructions**
  - parallel operations on multiple data elements

# What About Branches?

- **Challenge**
  - **instruction control unit must work well ahead of execution unit to generate enough operations to keep EU busy**

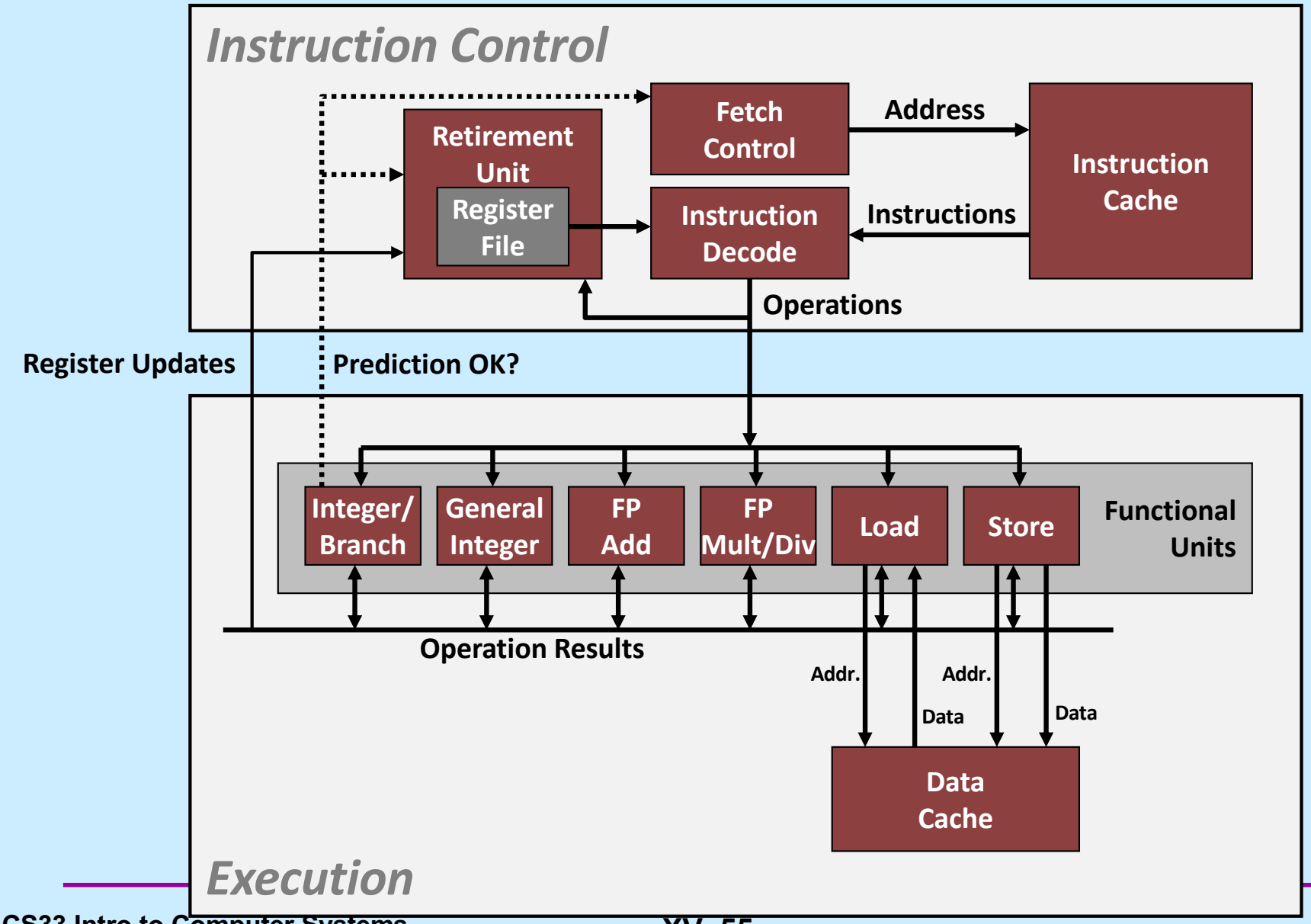
```
80489f3:  movl    $0x1,%ecx
80489f8:  xorq    %rdx,%rdx
80489fa:  cmpq    %rsi,%rdx
80489fc:  jnl     8048a25
80489fe:  movl    %esi,%edi
8048a00:  imull   (%rax,%rdx,4),%ecx
```

**Executing**

**How to continue?**

  - **when it 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

```
80489f3:  movl    $0x1,%ecx
80489f8:  xorq    %rdx,%rdx
80489fa:  cmpq    %rsi,%rdx
80489fc:  jnl     8048a25
80489fe:  movl    %esi,%esi
8048a00:  imull   (%rax,%rdx,4),%ecx
```

Branch not-taken

Branch taken

```
8048a25:  cmpq    %rdi,%rdx
8048a27:  jl      8048a20
8048a29:  movl    0xc(%rbp),%eax
8048a2c:  leal    0xffffffe8(%rbp),%esp
8048a2f:  movl    %ecx,(%rax)
```

# Branch Prediction

- **Idea**
  - guess which way branch will go
  - begin executing instructions at predicted position
    - » but don't actually modify register or memory data

```
80489f3:  movl    $0x1,%ecx
80489f8:  xorq    %edx,%edx
80489fa:  cmpq    %rsi,%rdx
80489fc:  jnl     8048a25
. . .
```

**Predict taken**

```
8048a25:  cmpq    %rdi,%rdx
8048a27:  jl      8048a20
8048a29:  movl    0xc(%rbp),%eax
8048a2c:  leal    0xffffffe8(%rbp),%esp
8048a2f:  movl    %ecx,(%rax)
```

**Begin execution**

# Branch Prediction Through Loop

```
80488b1:   movl    (%rcx,%rdx,4),%eax
80488b4:   addl    %eax,(%rdi)
80488b6:   incl    %edx
80488b7:   cmpl    %esi,%edx
80488b9:   jl      80488b1
```

*i = 98*

*Assume vector length = 100*

Predict taken (OK)

```
80488b1:   movl    (%rcx,%rdx,4),%eax
80488b4:   addl    %eax,(%rdi)
80488b6:   incl    %edx
80488b7:   cmpl    %esi,%edx
80488b9:   jl      80488b1
```

*i = 99*

Predict taken (oops)

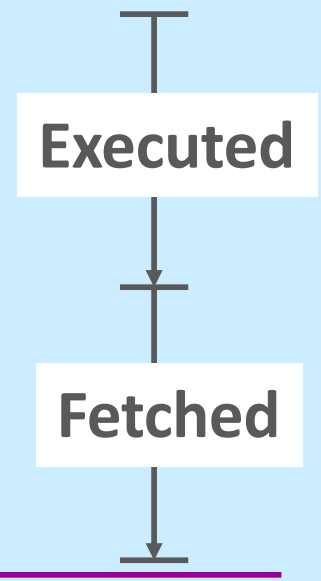
```
80488b1:   movl    (%rcx,%rdx,4),%eax
80488b4:   addl    %eax,(%rdi)
80488b6:   incl    %edx
80488b7:   cmpl    %esi,%edx
80488b9:   jl      80488b1
```

*i = 100*

Read invalid location

```
80488b1:   movl    (%rcx,%rdx,4),%eax
80488b4:   addl    %eax,(%rdi)
80488b6:   incl    %edx
80488b7:   cmpl    %esi,%edx
80488b9:   jl      80488b1
```

*i = 101*



# Branch Misprediction Invalidation

```
80488b1:   movl    (%rcx,%rdx,4),%eax
80488b4:   addl    %eax,(%rdi)
80488b6:   incl    %edx
80488b7:   cmpl    %esi,%edx
80488b9:   jl      80488b1
```

*i = 98*

*Assume*
*vector length = 100*

Predict taken (OK)

```
80488b1:   movl    (%rcx,%rdx,4),%eax
80488b4:   addl    %eax,(%rdi)
80488b6:   incl    %edx
80488b7:   cmpl    %esi,%edx
80488b9:   jl      80488b1
```

*i = 99*

Predict taken (oops)

~~80488b1: movl (%rcx,%rdx,4),%eax~~
~~80488b4: addl %eax,(%rdi)~~
~~80488b6: incl %edx~~
~~80488b7: cmpl %esi,%edx~~
~~80488b9: jl 80488b1~~

*i = 100*

~~80488b1: movl (%rcx,%rdx,4),%eax~~
~~80488b4: addl %eax,(%rdi)~~
~~80488b6: incl %edx~~

*i = 101*

Invalidate

# Branch Misprediction Recovery

```
80488b1:   movl    (%rcx,%rdx,4),%eax
80488b4:   addl    %eax,(%rdi)
80488b6:   incl    %edx
80488b7:   cmpl    %esi,%edx
80488b9:   jl      80488b1
80488bb:   leal    0xffffffe8(%rbp),%esp
80488be:   popl    %ebx
80488bf:   popl    %esi
80488c0:   popl    %edi
```

*i = 99*

**Definitely not taken**

- **Performance Cost**
  - multiple clock cycles on modern processor
  - can be a major performance limiter

# Latency of Loads

```
typedef struct ELE {
  struct ELE *next;
  long data;
} list_ele, *list_ptr;

int list_len(list_ptr ls) {
  long len = 0;
  while (ls) {
    len++;
    ls = ls->next;
  }
  return len;
}
```

```
# len in %rax, ls in %rdi

.L11:                       # loop:
  addq   $1, %rax           # incr len
  movq   (%rdi), %rdi       # ls = ls->next
  testq  %rdi, %rdi         # test ls
  jne    .L11               # if != 0
                            #   go to loop
```

- **4 CPE**

# Clearing an Array ...

```
#define ITERS 100000000
void clear_array() {
  long dest[100];
  int iter;
  for (iter=0; iter<ITERS; iter++) {
    long i;
    for (i=0; i<100; i++)
      dest[i] = 0;
  }
}
```

- **1 CPE**

# Store/Load Interaction

```
void write_read(long *src, long *dest, long n) {
  long cnt = n;
  long val = 0;

  while(cnt--) {
    *dest = val;
    val = (*src)+1;
  }
}
```

# Store/Load Interaction

```
void write_read(long *src,
    long *dest, long n){
  long cnt = n;
  long val = 0;
  while(cnt--) {
    *dest = val;
    val = (*src)+1;
  }
}
```

```
long a[] = {-10, 17};
```

**Example A:** `write_read(&a[0],&a[1],3)`

| | Initial | Iter. 1 | Iter. 2 | Iter. 3 |
|---|---|---|---|---|
| cnt | 3 | 2 | 1 | 0 |
| a | –10, 17 | –10, 0 | –10, –9 | –10, –9 |
| val | 0 | –9 | –9 | –9 |

- **CPE 1.3**

**Example B:** `write_read(&a[0],&a[0],3)`

| | Initial | Iter. 1 | Iter. 2 | Iter. 3 |
|---|---|---|---|---|
| cnt | 3 | 2 | 1 | 0 |
| a | –10, 17 | 0, 17 | 1, 17 | 2, 17 |
| val | 0 | 1 | 2 | 3 |

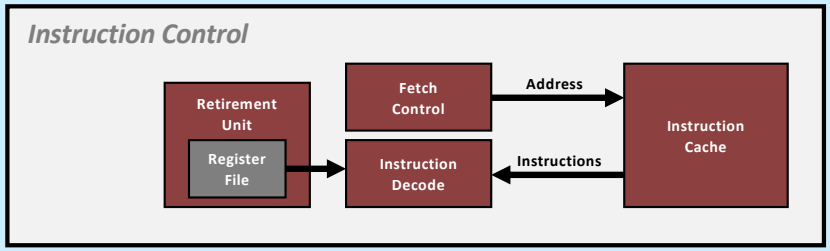
- **CPE 7.3**

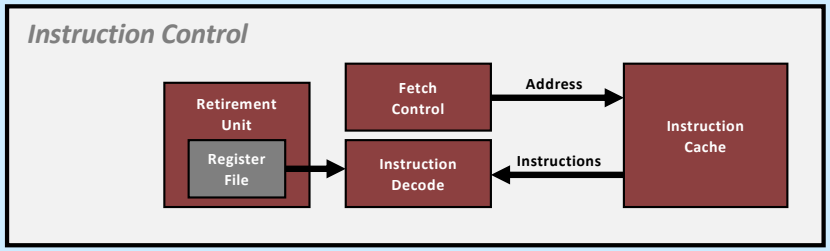
# Getting High Performance

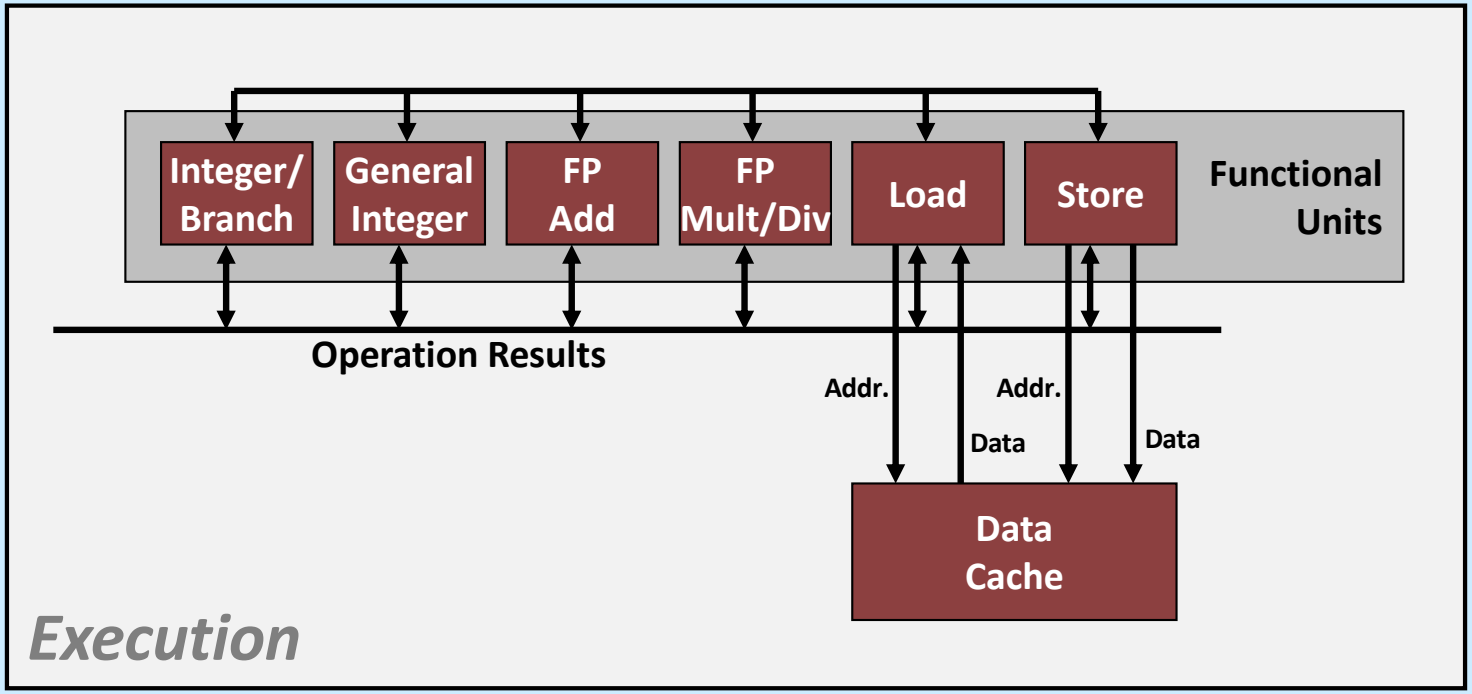
- Good compiler and flags
- Don't do anything stupid
  - watch out for hidden algorithmic inefficiencies
  - write compiler-friendly code
    - » watch out for optimization blockers:
      procedure calls & memory references
  - look carefully at innermost loops (where most work is done)

- Tune code for machine
  - exploit instruction-level parallelism
  - avoid unpredictable branches
  - make code cache friendly (covered soon)

# Hyper Threading



# Multiple Cores