

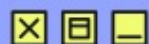
CS 33

Architecture and Optimization (2)



H&W HOURS

BROWNC-HEALTH-AND-WELLNESS.GITHUB.IO



Have questions/concerns about mental/physical health, accessibility, or disability rights within the department?

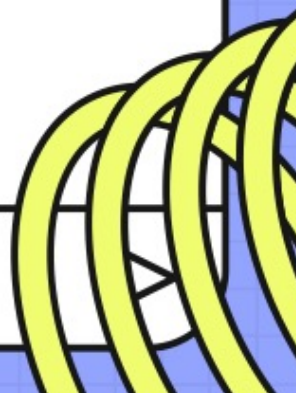
Stop by Health and Wellness Hours!

Monday 5:00 - 6:00 PM (Anna) in CIT 102

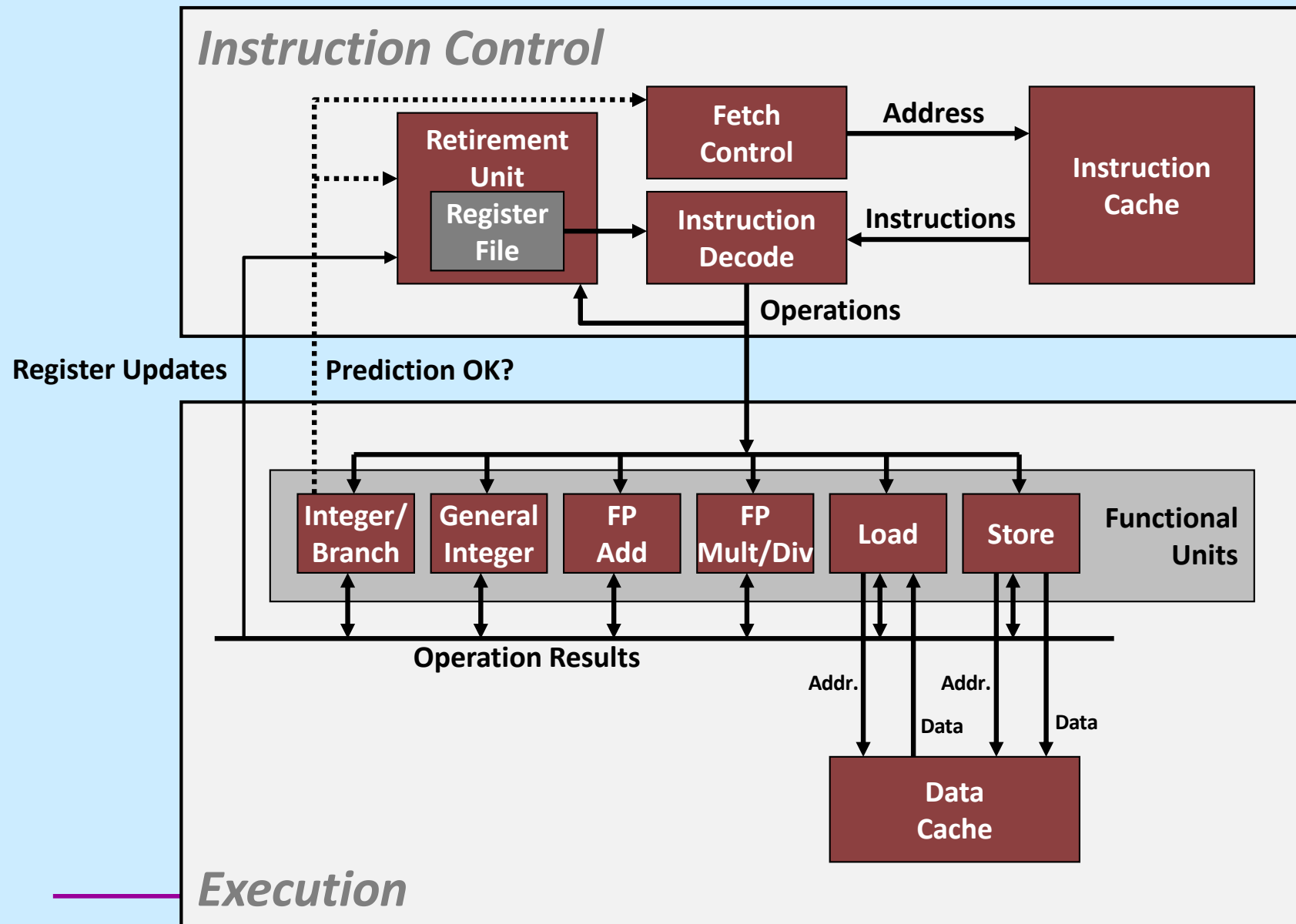
Tuesday 5:00 - 6:00 PM (David) in CIT 102

Wednesday 12:00 - 1:00 PM (Laura) in CIT 221

Thursday 5:00 - 6:00 PM (Cristian) in CIT 102



Modern CPU Design



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**

<i>Instruction</i>	<i>Latency</i>	<i>Cycles/Issue</i>	<i>Capacity</i>
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

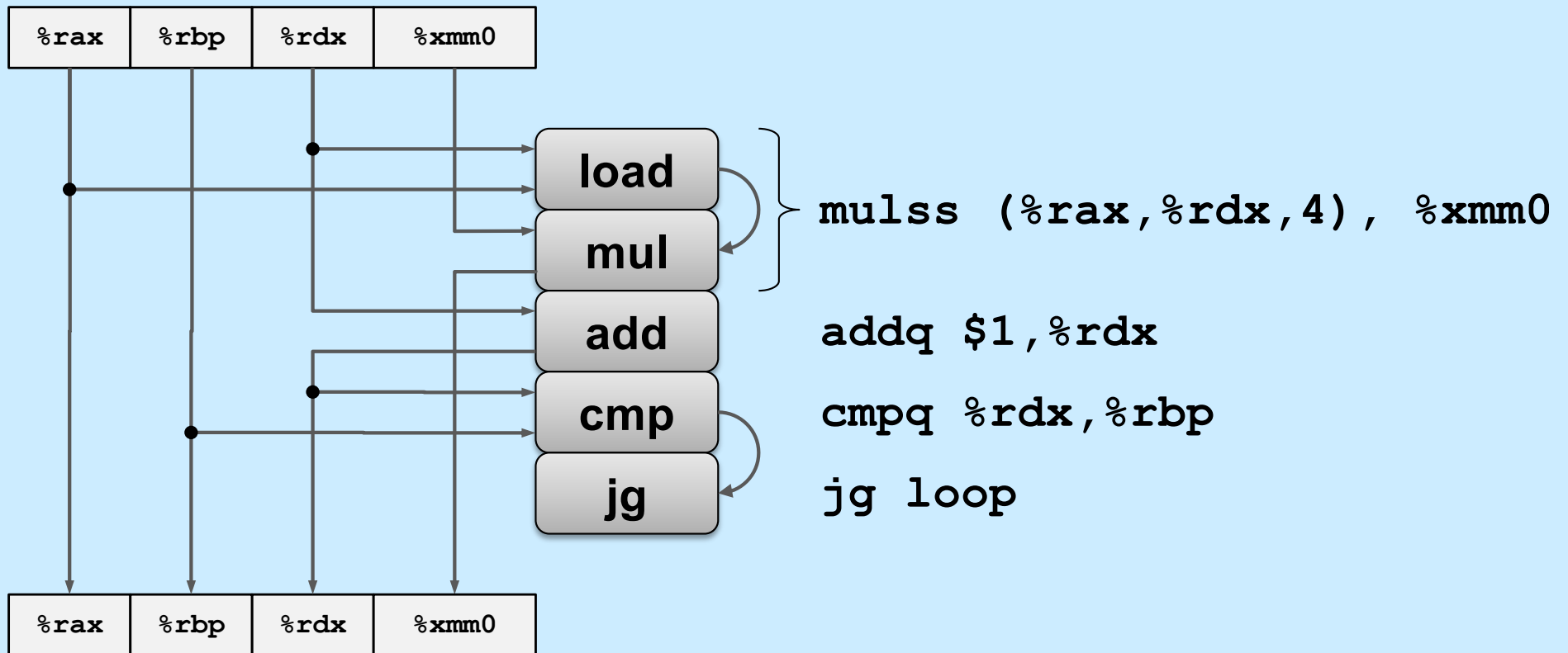
x86-64 Compilation of Combine4

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

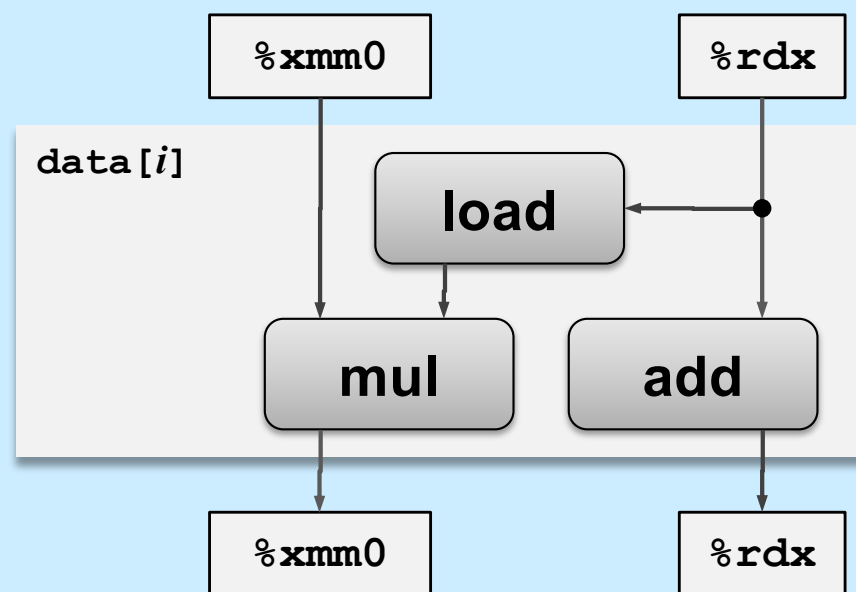
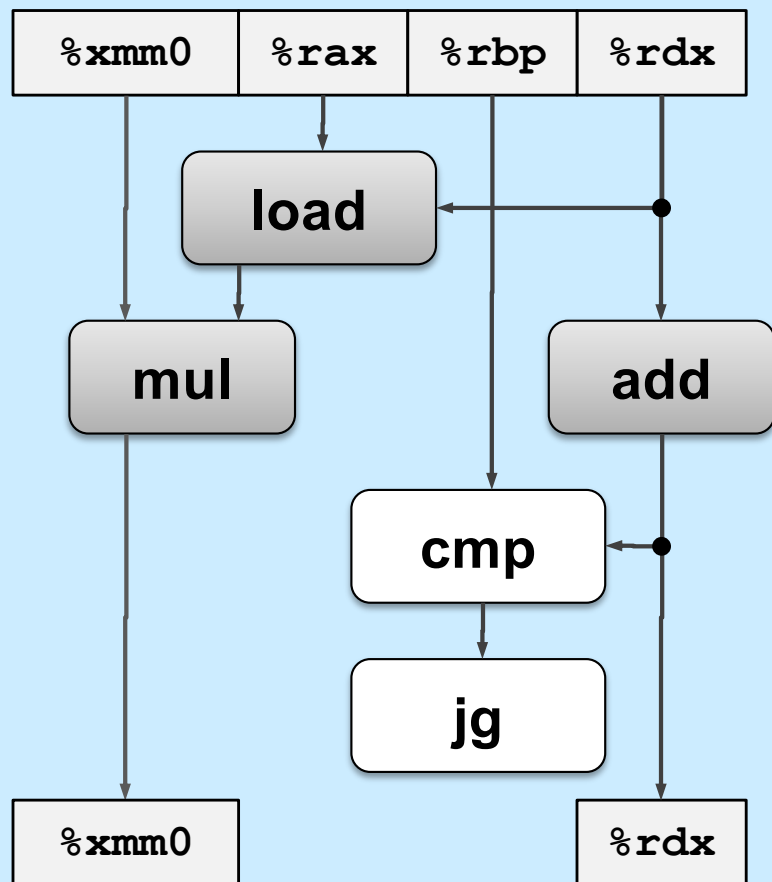
```
.L519:                                # Loop:
    mullss (%rax,%rdx,4), %xmm0      # t = t * d[i]
    addq $1, %rdx                    # i++
    cmpq %rdx, %rbp                  # Compare length:i
    jg     .L519                     # If >, goto Loop
```

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

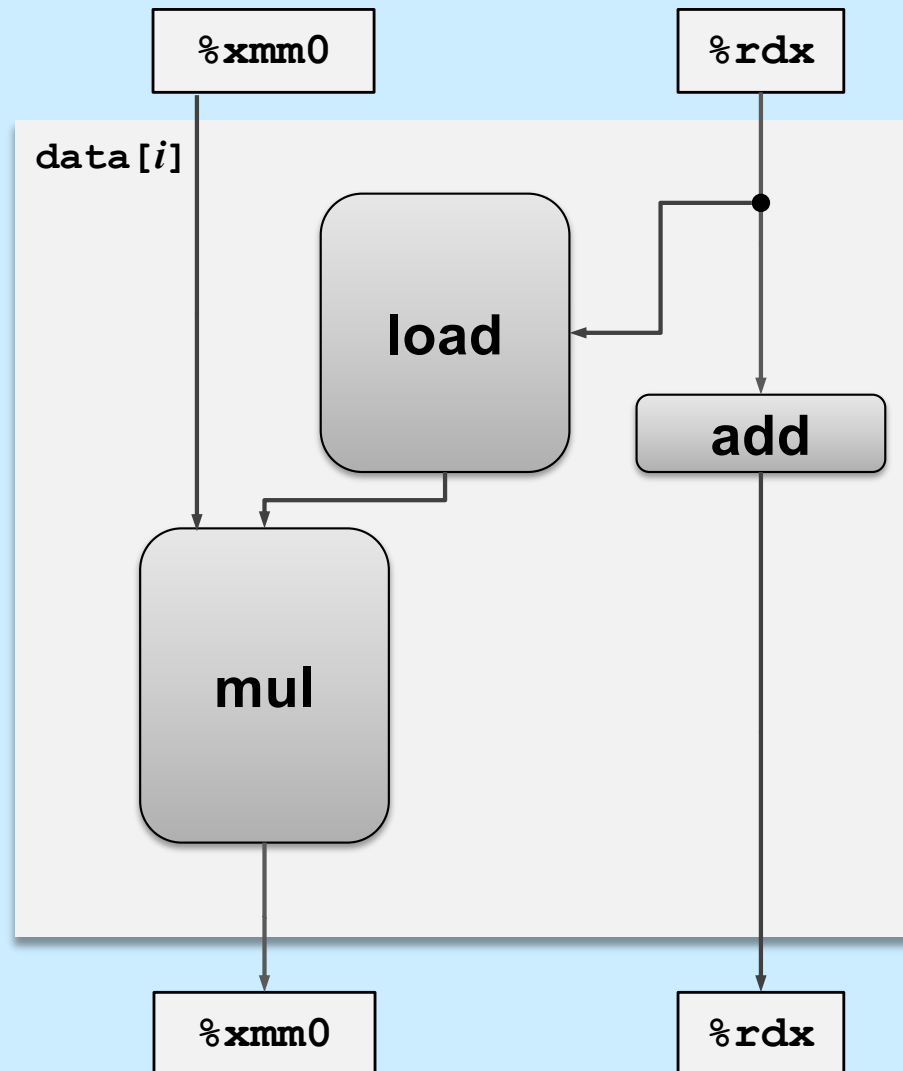
Inner Loop



Data-Flow Graphs of Inner Loop

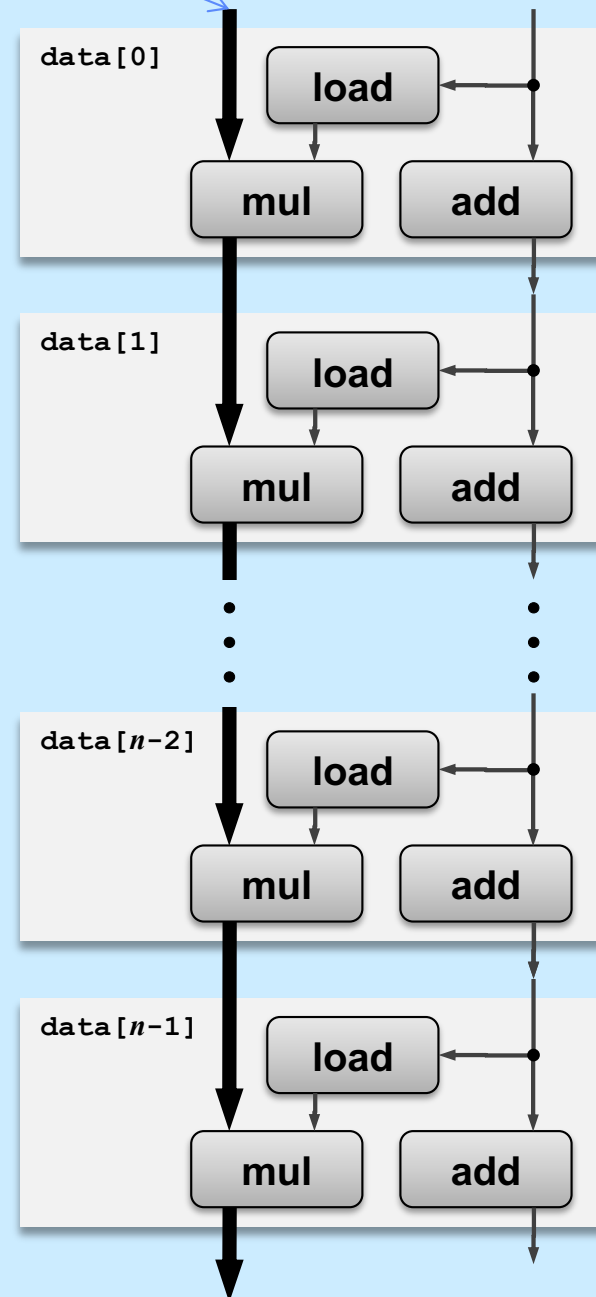


Relative Execution Times

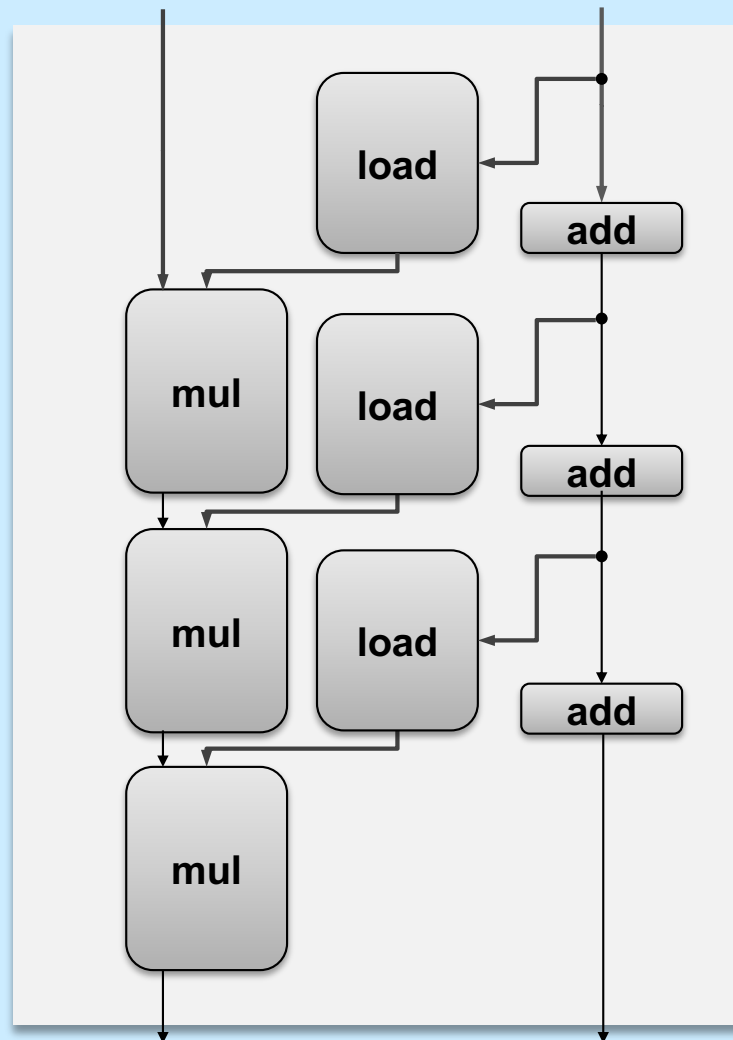


Data Flow Over Multiple Iterations

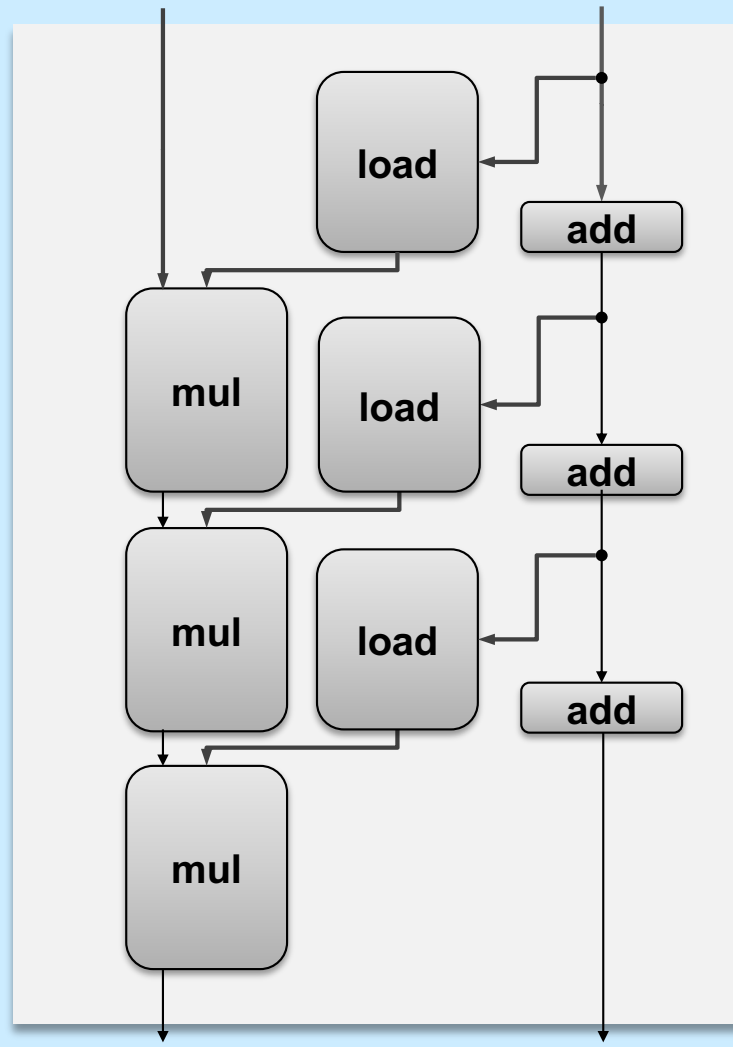
Critical path



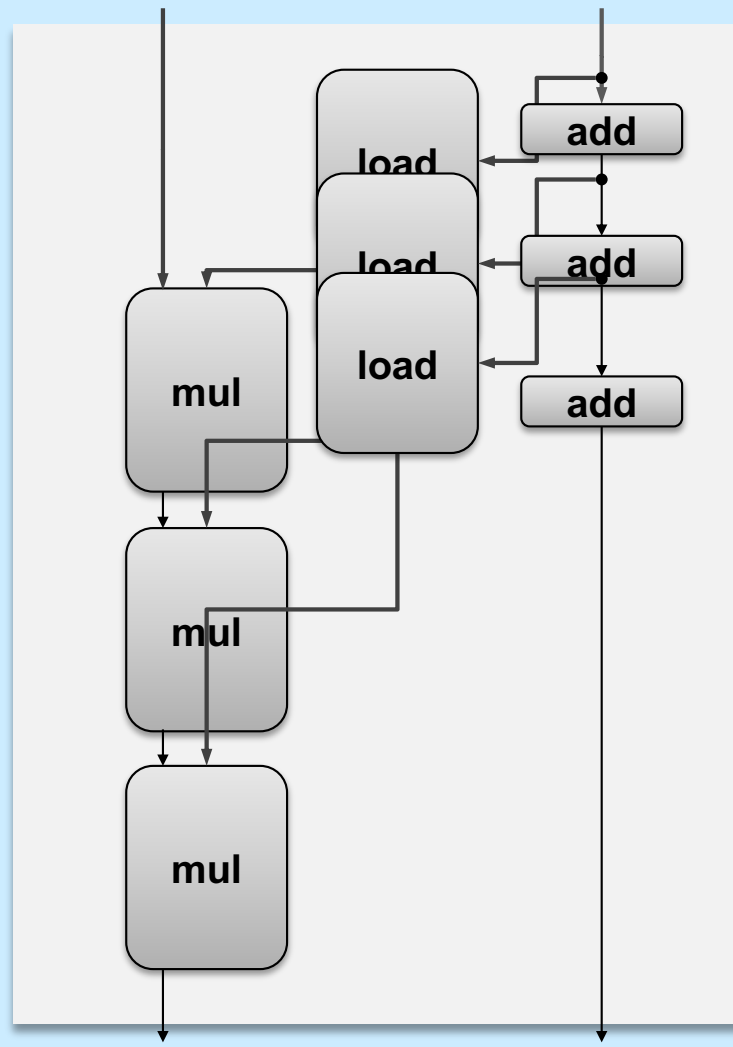
Pipelined Data-Flow Over Multiple Iterations



Pipelined Data-Flow Over Multiple Iterations



Pipelined Data-Flow Over Multiple Iterations



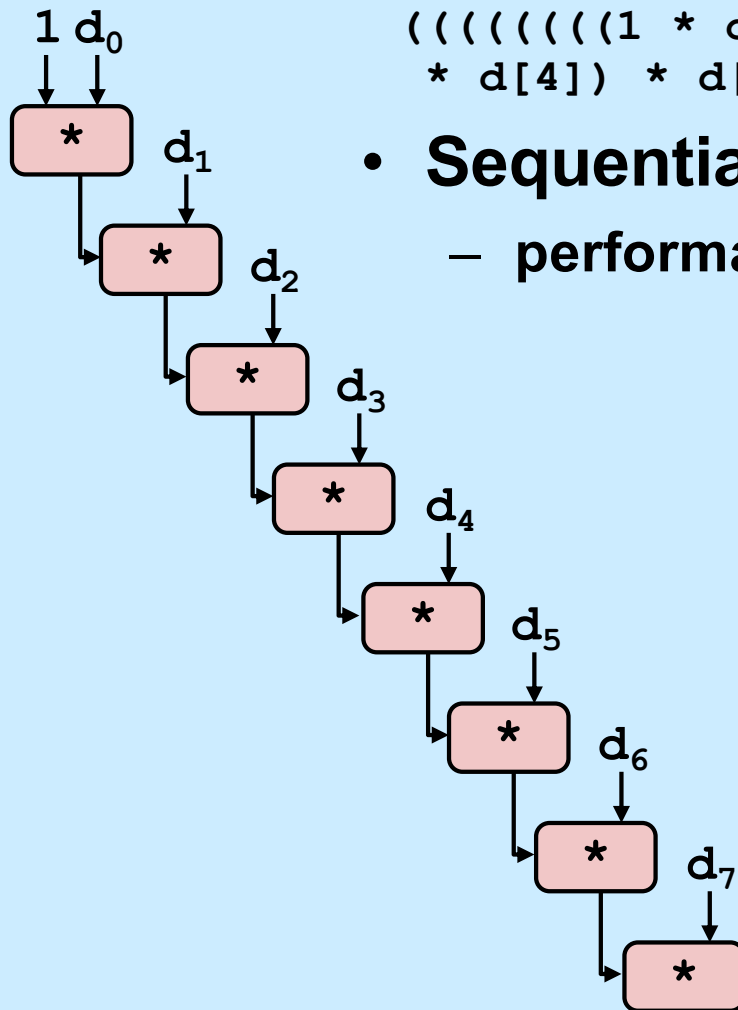
Combine4 = Serial Computation (OP = *)

- **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;
}
```

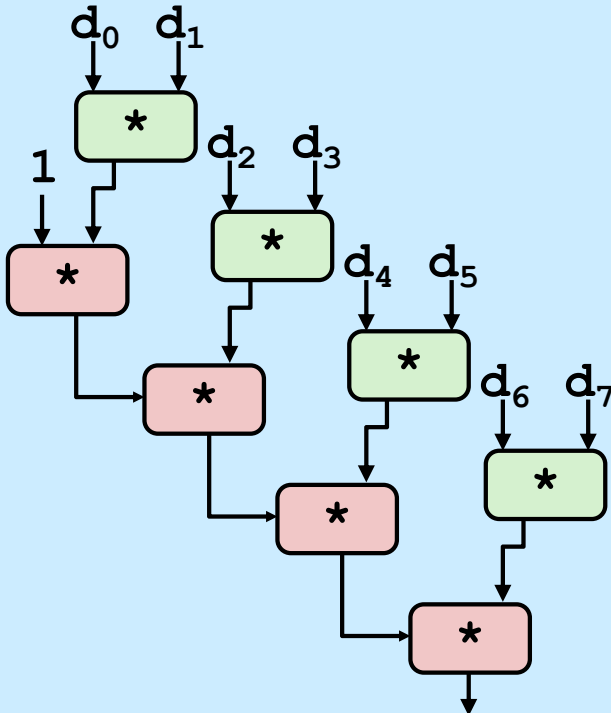
Compare to before

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

- 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

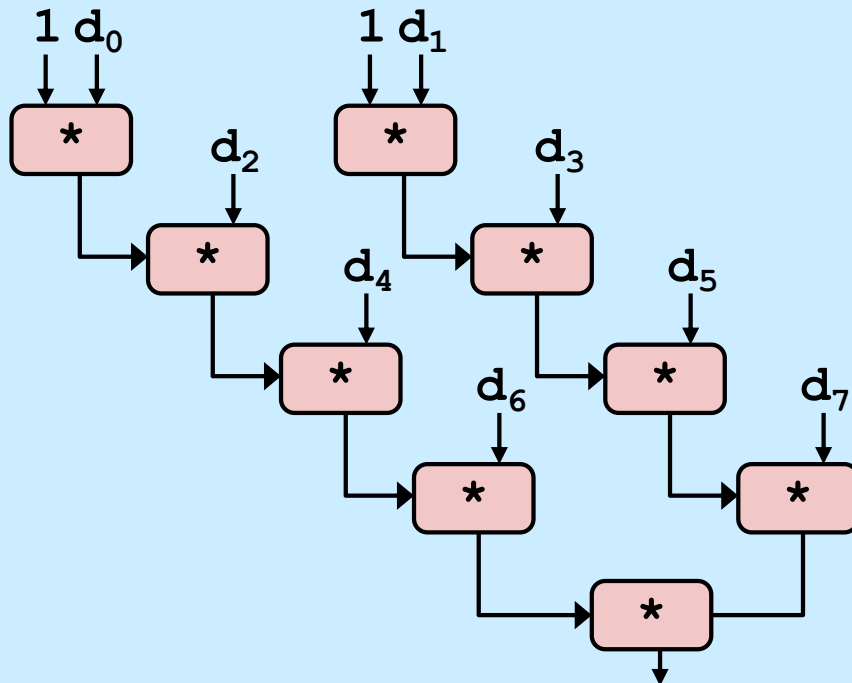
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.00	3.00	5.00
Unroll 2x	1.01	3.00	3.00	5.00
Unroll 2x, reassociate	1.01	1.51	1.51	2.01
Unroll 2x parallel 2x	.81	1.51	1.51	2.51
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	.25	1.0	1.0	.5

- 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

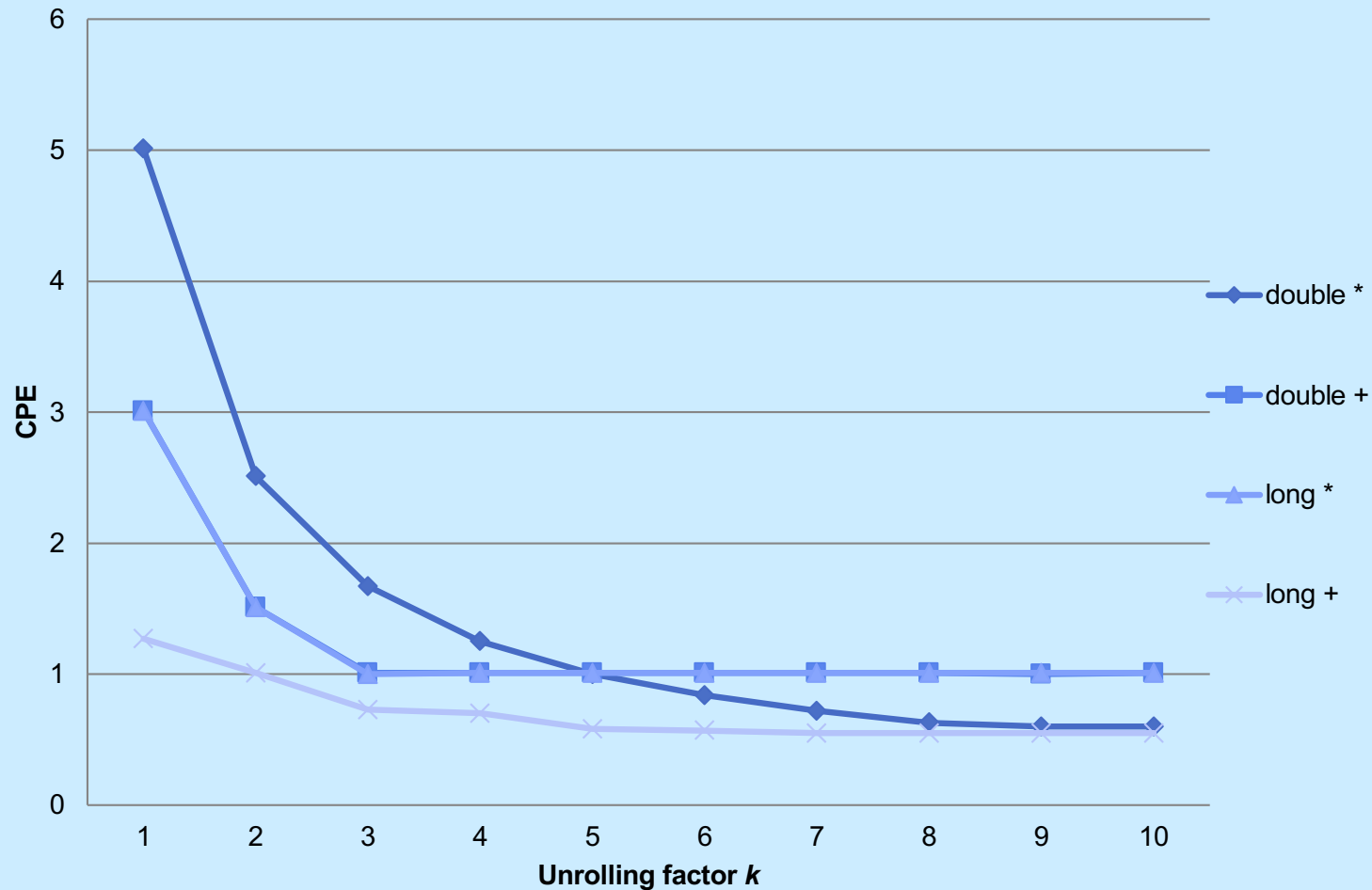
What Now?

Quiz 1

We're making progress. With two accumulators we get a two-fold speedup. With three accumulators, we can get a three-fold speedup. How much better performance can we expect if we add even more accumulators?

- a) It keeps on getting better as we add more and more accumulators**
- b) It's limited by the latency bound**
- c) It's limited by the throughput bound**
- d) It's limited by something else**

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

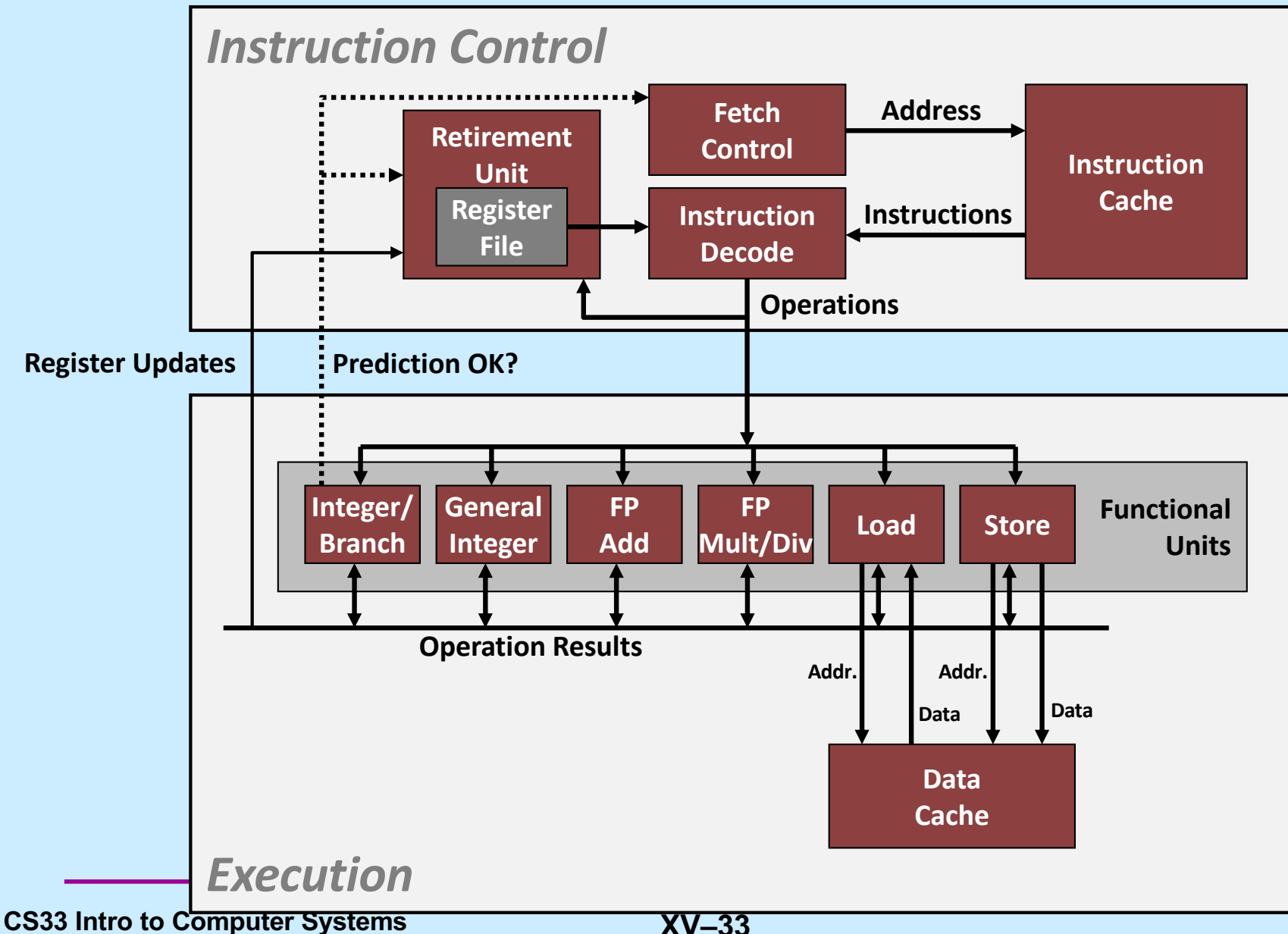
- **instruction control unit** must work well ahead of **execution unit** to generate enough operations to keep EU busy

```
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    0xfffffffffe8(%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    0xfffffffffe8(%rbp),%esp
8048a2f: movl    %ecx, (%rax)
```

} **Begin
execution**

Branch Prediction Through Loop

Assume
vector length = **100**

```
80488b1:  movl    (%rcx,%rdx,4),%eax
80488b4:  addl    %eax, (%rdi)
80488b6:  incl    %edx
80488b7:  cmpl    %esi,%edx    i = 98
80488b9:  jnl     80488b1
```

Predict taken (OK)

```
80488b1:  movl    (%rcx,%rdx,4),%eax
80488b4:  addl    %eax, (%rdi)
80488b6:  incl    %edx
80488b7:  cmpl    %esi,%edx    i = 99
80488b9:  jnl     80488b1
```

Predict taken
(oops)

```
80488b1:  movl    (%rcx,%rdx,4),%eax
80488b4:  addl    %eax, (%rdi)
80488b6:  incl    %edx
80488b7:  cmpl    %esi,%edx    i = 100
80488b9:  jnl     80488b1
```

Read
invalid
location

```
80488b1:  movl    (%rcx,%rdx,4),%eax
80488b4:  addl    %eax, (%rdi)
80488b6:  incl    %edx
80488b7:  cmpl    %esi,%edx    i = 101
80488b9:  jnl     80488b1
```

Executed

Fetches

Branch Misprediction Invalidation

Assume
vector length = **100**

```
80488b1:  movl    (%rcx,%rdx,4),%eax
80488b4:  addl    %eax, (%rdi)
80488b6:  incl    %edx
80488b7:  cmpl    %esi,%edx    i = 98
80488b9:  jnl     80488b1
```

Predict taken (OK)

```
80488b1:  movl    (%rcx,%rdx,4),%eax
80488b4:  addl    %eax, (%rdi)
80488b6:  incl    %edx
80488b7:  cmpl    %esi,%edx    i = 99
80488b9:  jnl     80488b1
```

Predict taken (oops)

```
80488b1:  movl    (%rcx,%rdx,4),%eax
80488b4:  addl    %eax, (%rdi)
80488b6:  incl    %edx
80488b7:  cmpl    %esi,%edx    i = 100
80488b9:  jnl     80488b1
```

Invalidate

```
80488b1:  movl    (%rcx,%rdx,4),%eax
80488b4:  addl    %eax, (%rdi)
80488b6:  incl    %edx    i = 101
```

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    0xffffffffe8(%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 1000000000
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-->0) {  
        *dest = val;  
        val = (*src)+1;  
    }  
}
```

Store/Load Interaction

```
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	<table><tr><td>-10</td><td>17</td></tr></table>	-10	17	<table><tr><td>-10</td><td>0</td></tr></table>	-10	0	<table><tr><td>-10</td><td>-9</td></tr></table>	-10	-9	<table><tr><td>-10</td><td>-9</td></tr></table>	-10	-9
-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	<table><tr><td>-10</td><td>17</td></tr></table>	-10	17	<table><tr><td>0</td><td>17</td></tr></table>	0	17	<table><tr><td>1</td><td>17</td></tr></table>	1	17	<table><tr><td>2</td><td>17</td></tr></table>	2	17
-10	17											
0	17											
1	17											
2	17											
val	0	1	2	3								

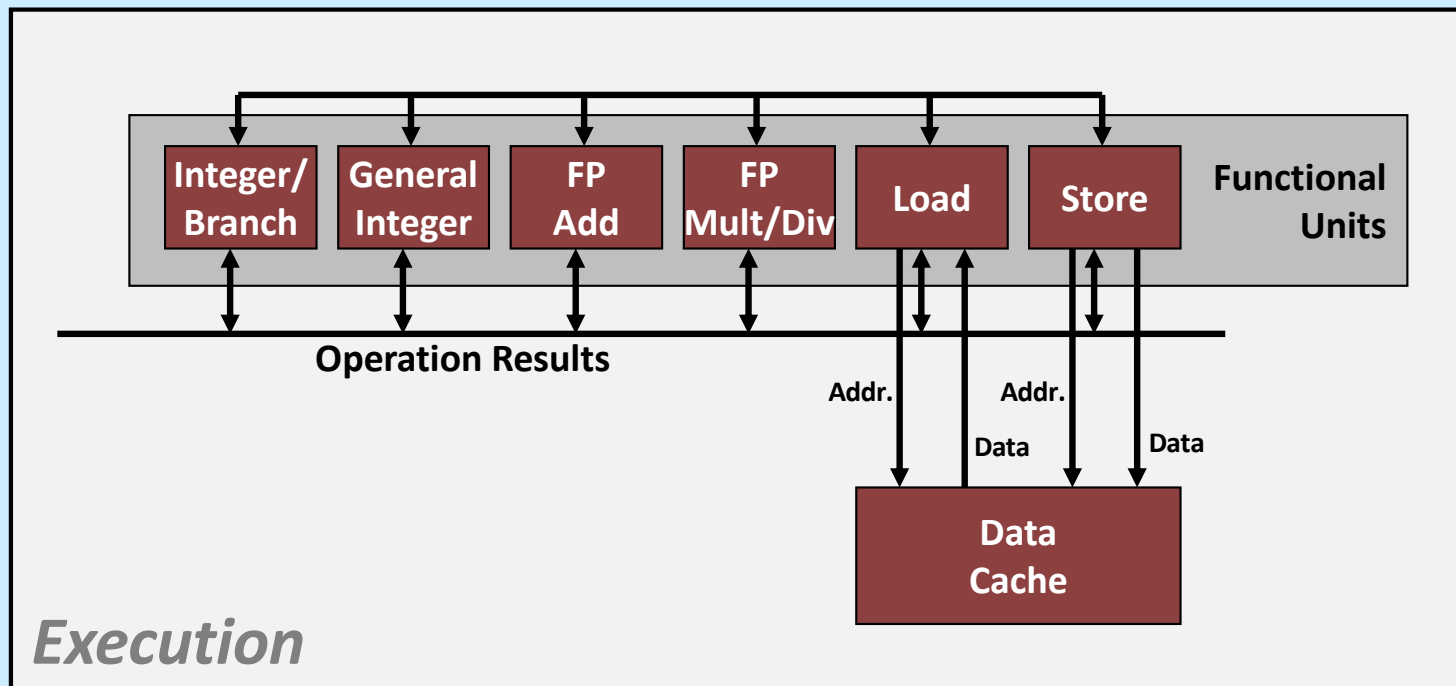
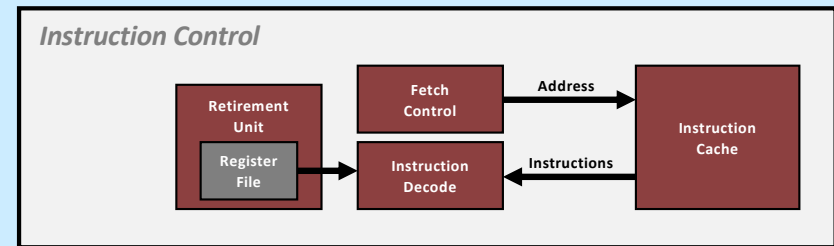
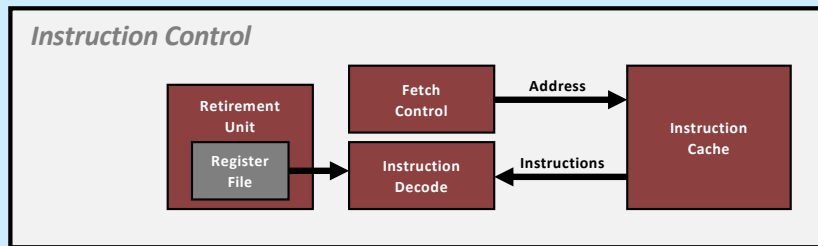
• CPE 7.3

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

Getting High Performance

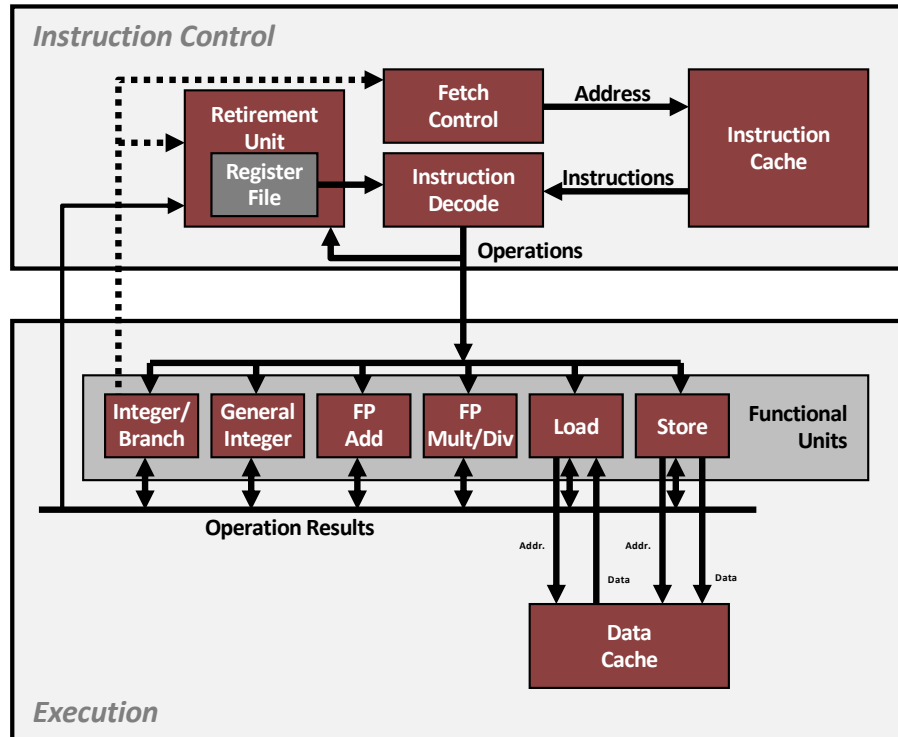
- **Good compiler and flags**
- **Don't do anything stupid**
 - watch out for hidden algorithmic inefficiencies
 - write compiler-friendly code
 - » watch out for optimization blockers:
function calls & memory references
 - look carefully at innermost loops (where most work is done)
- **Tune code for machine**
 - exploit instruction-level parallelism
 - avoid unpredictable branches
 - make code cache friendly (covered soon)

Hyper Threading

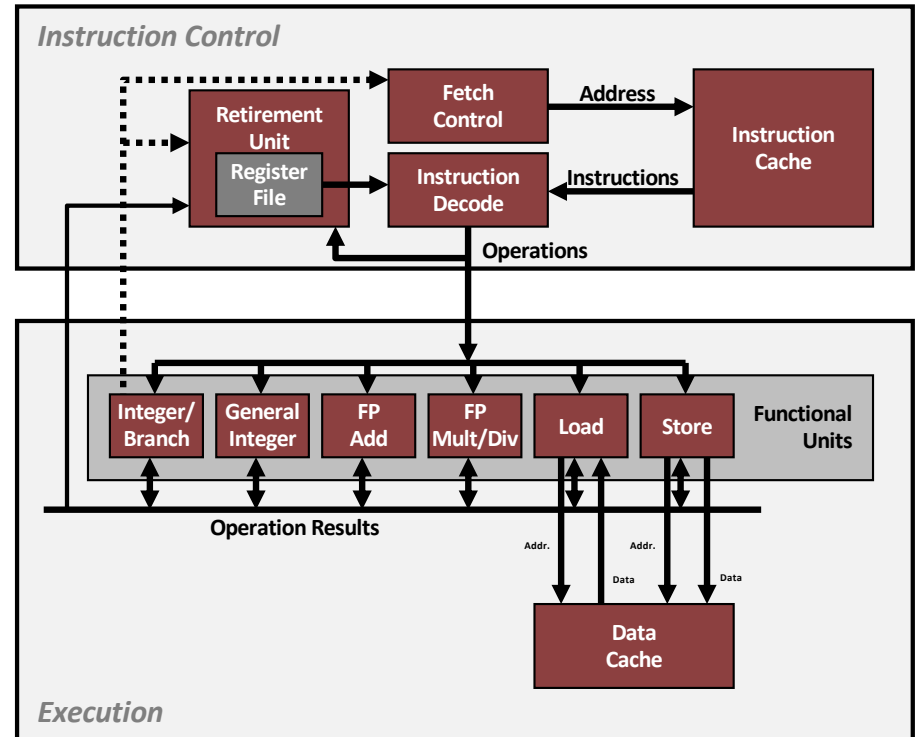


Multiple Cores

Chip



Other Stuff



More
Cache

Other Stuff