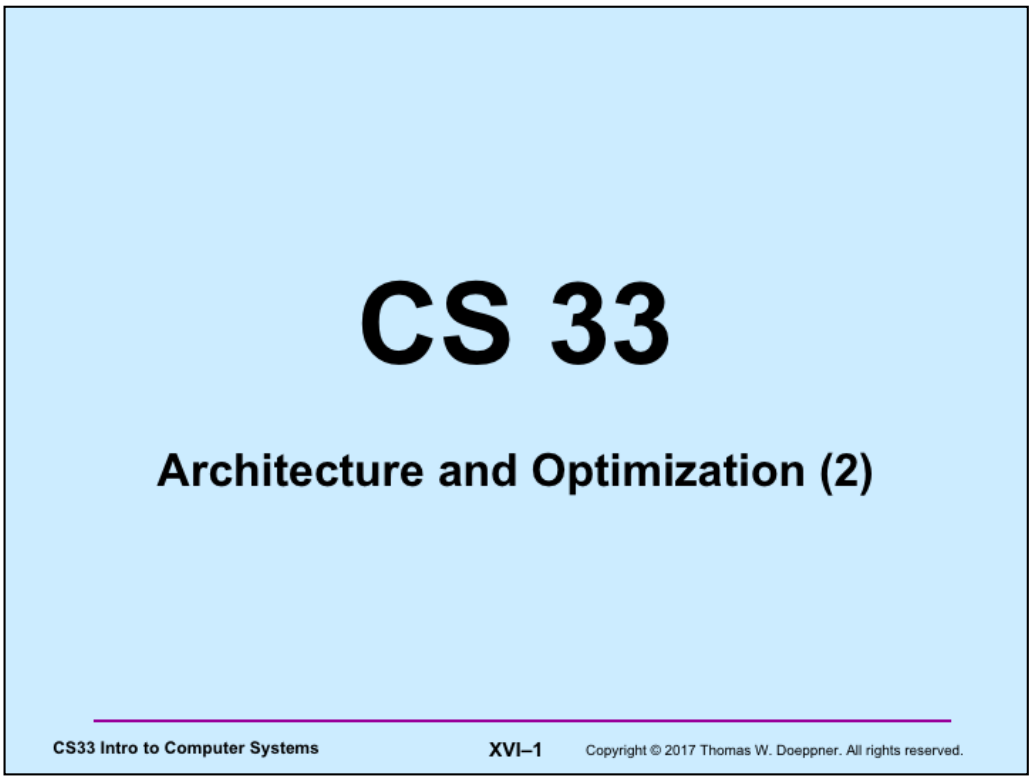
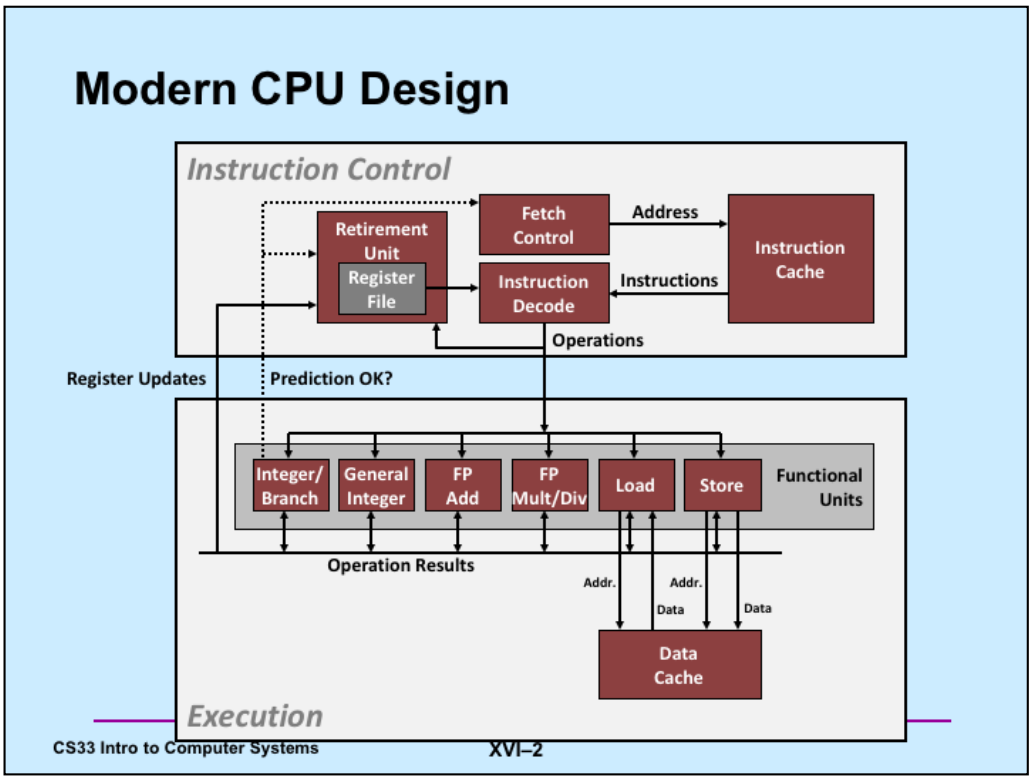
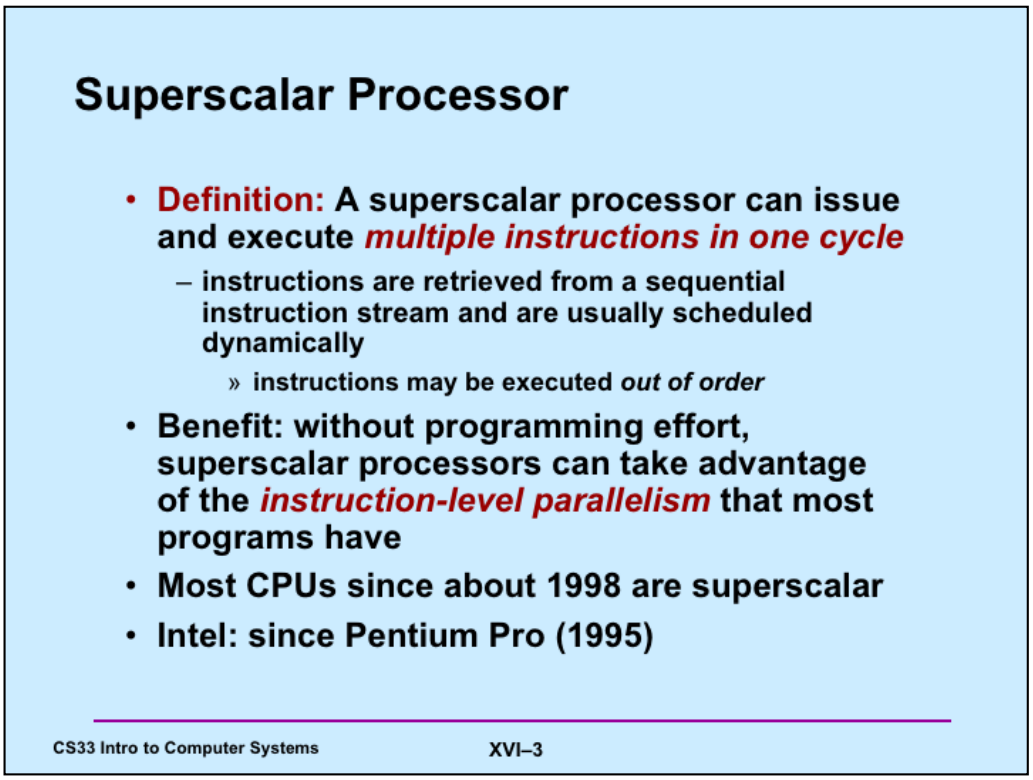


Most of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.



Supplied by CMU.



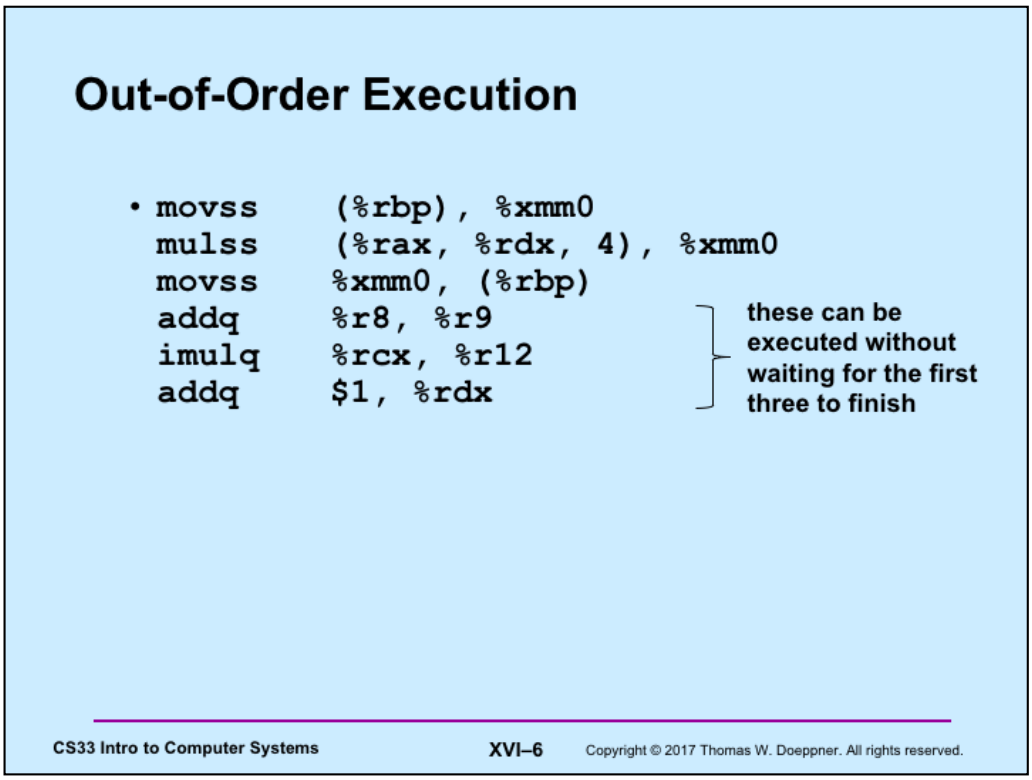
Supplied by CMU.

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

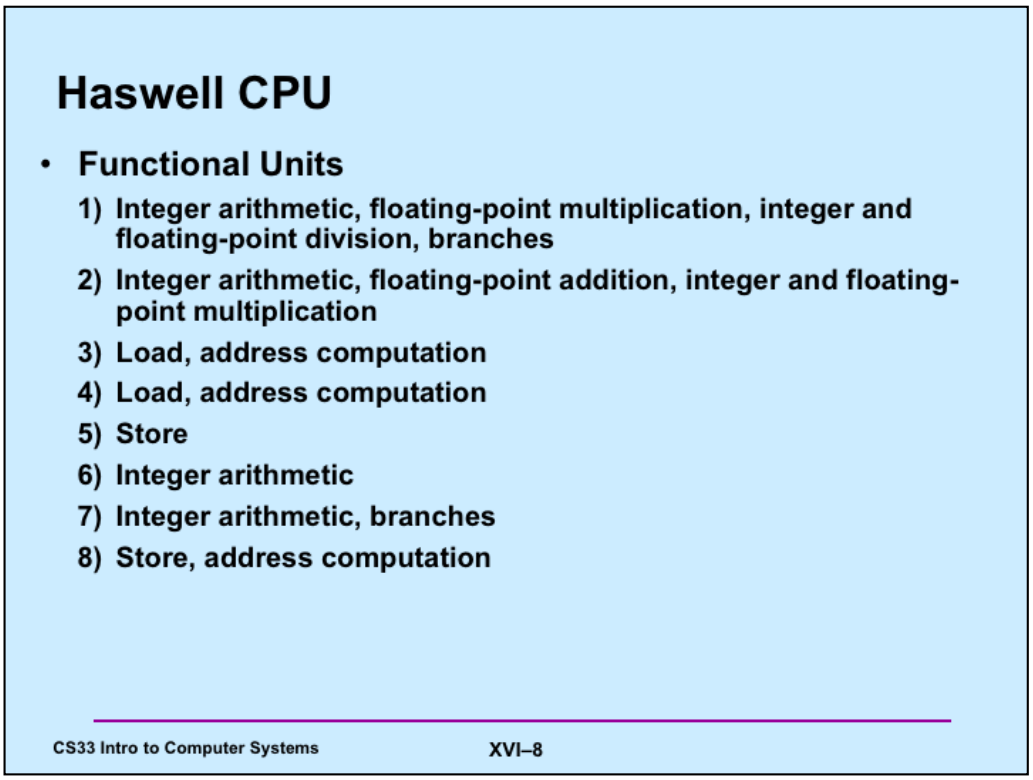


Note that the first three instructions are floating-point instructions, and %xmm0 is a floating-point register.

Speculative Execution

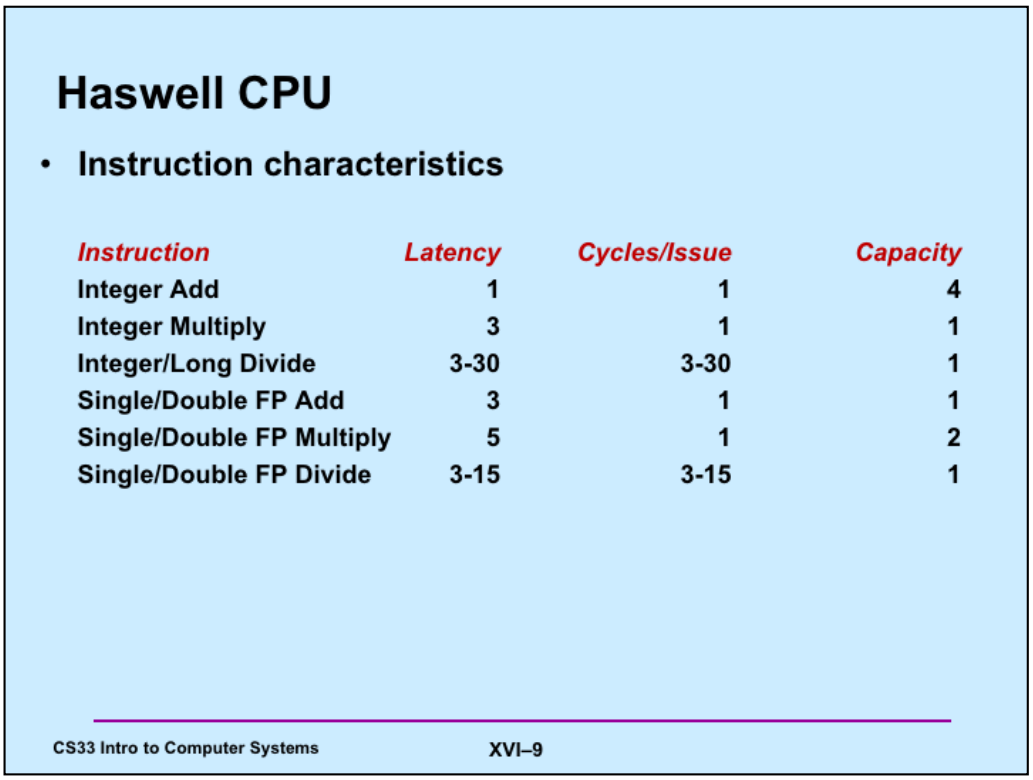
```
80489f3:    movl    $0x1,%ecx
80489f8:    xorq    %rdx,%rdx
80489fa:    cmpq    %rsi,%rdx
80489fc:    jnl     8048a25
80489fe:    movl    %esi,%edi
8048a00:    imull   (%rax,%rdx,4),%ecx
```

perhaps execute these instructions



Supplied by CMU.

“Haswell” is Intel’s code name for recent versions of its Core I7 processor design.



Supplied by CMU.

“Haswell” is Intel’s code name for recent versions of its Core I7 processor design.

Haswell CPU Performance Bounds

	Integer		Floating Point	
	+	*	+	*
Latency	1.00	3.00	3.00	5.00
Throughput	0.50	1.00	1.00	0.50

x86-64 Compilation of Combine4

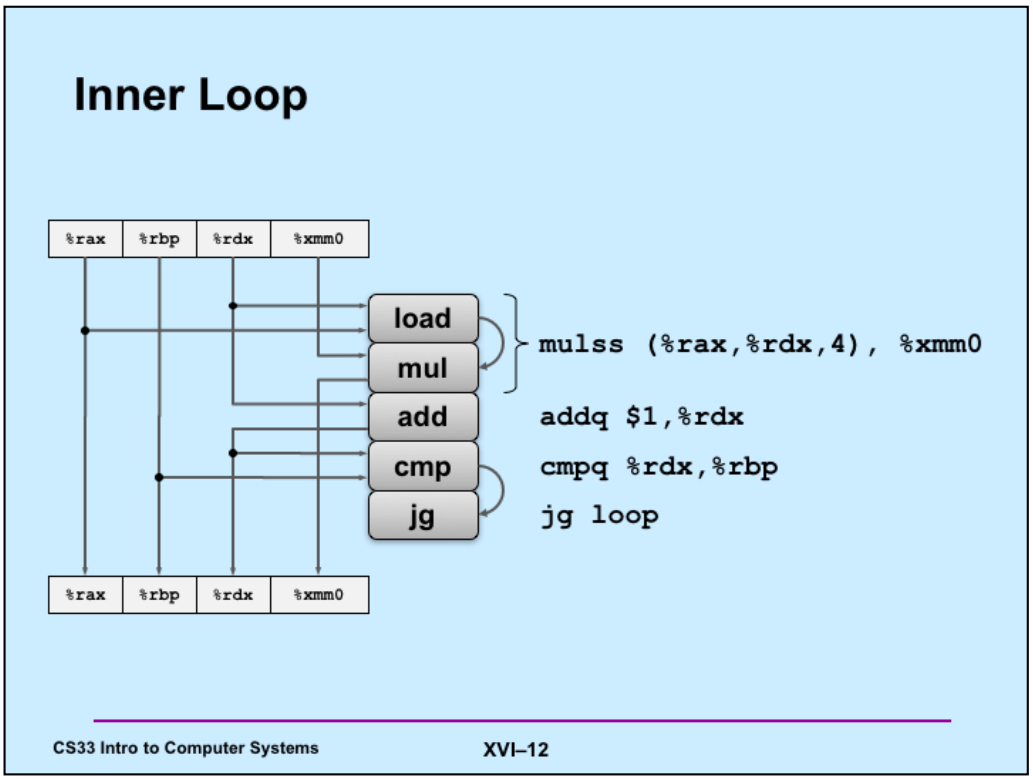
- Inner loop (case: integer multiply)

```
.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
```

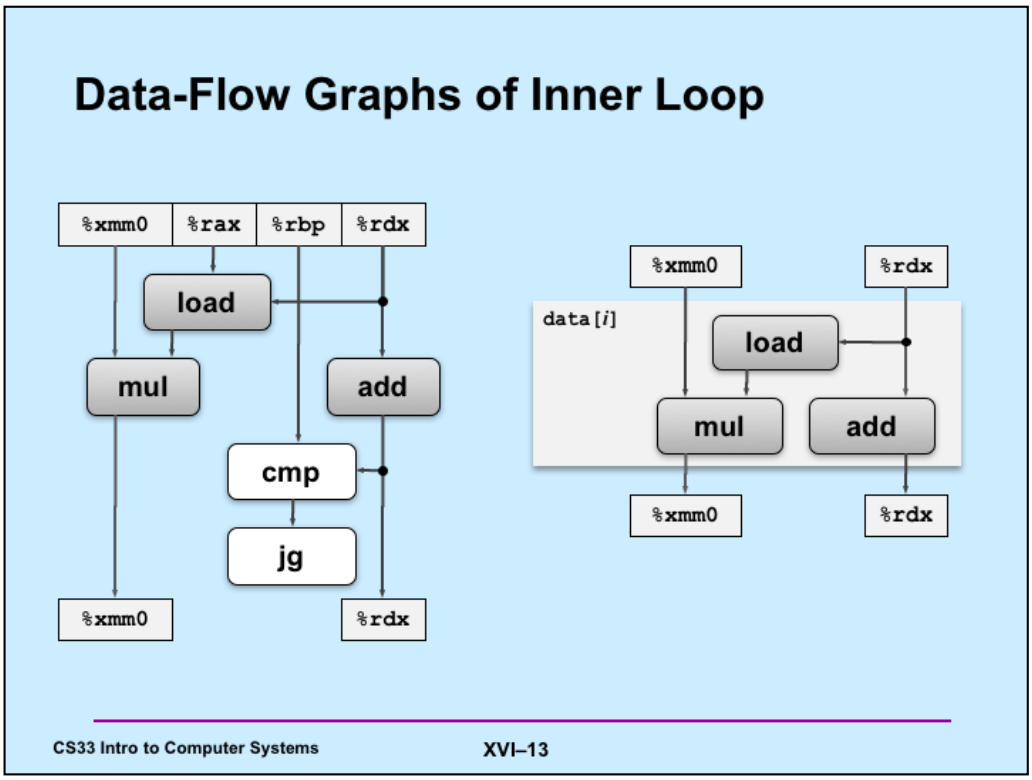
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latency bound	1.00	3.00	3.00	5.0
Throughput bound	0.50	1.00	1.00	0.50

Supplied by CMU.

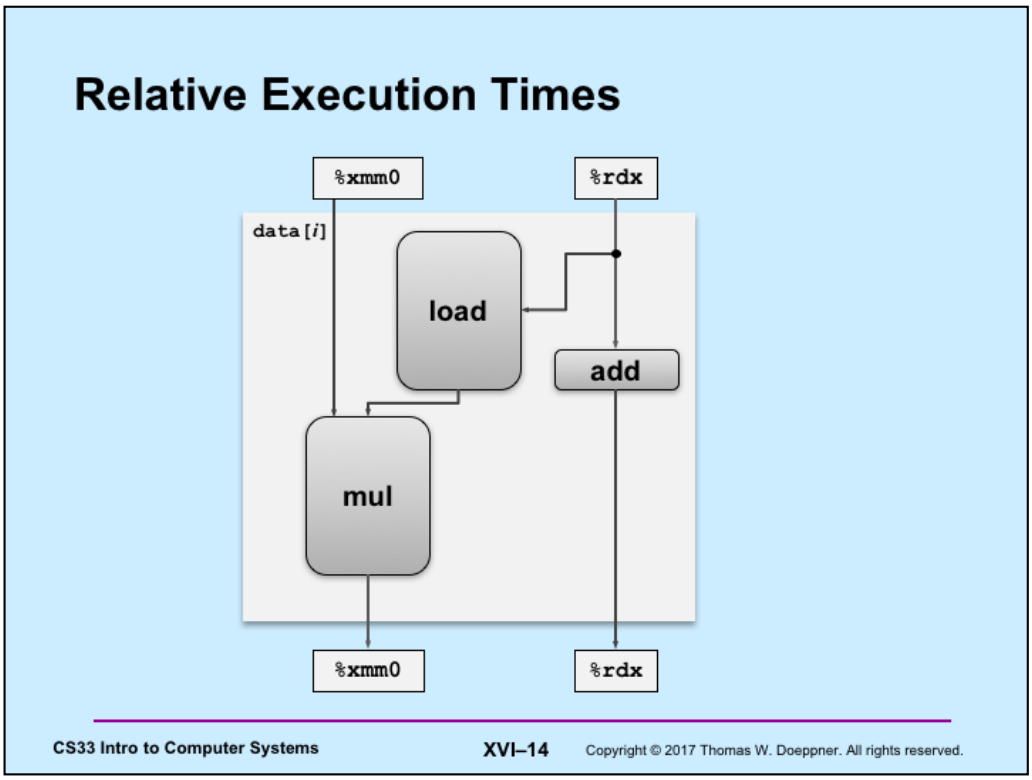
These numbers are for the Haswell CPU.



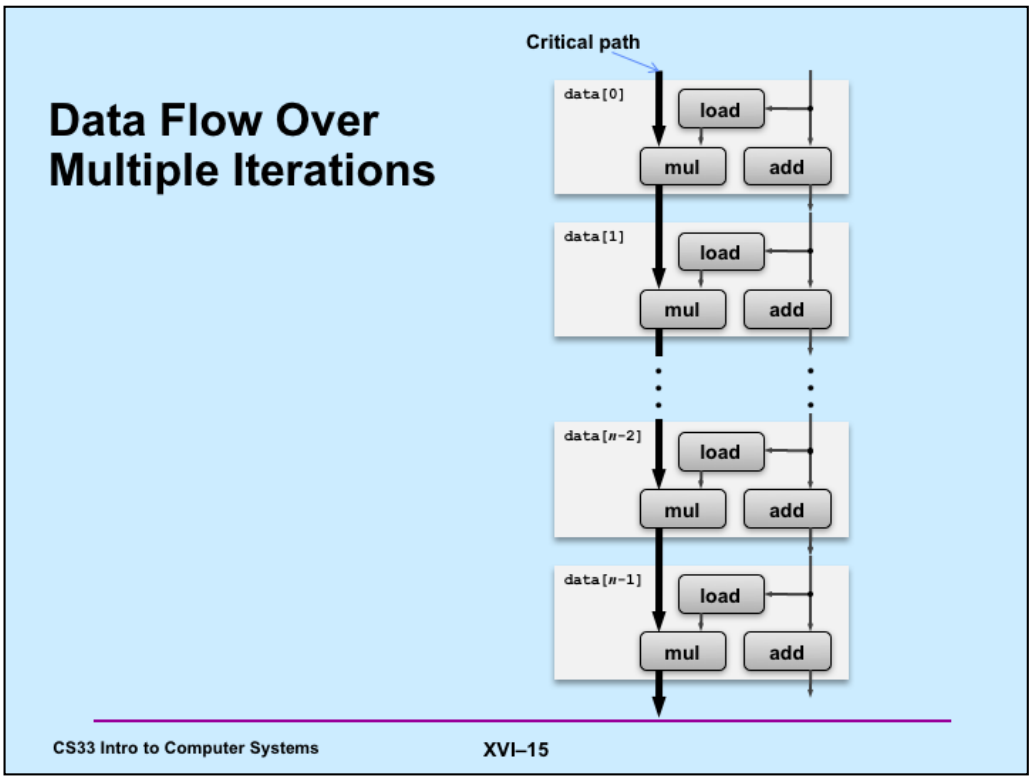
This is Figure 5.13 of Bryant and O'Hallaron. It shows the code for the single-precision floating-point version of our example.



These are Figures 5.14 a and b of Bryant and O'Hallaron.

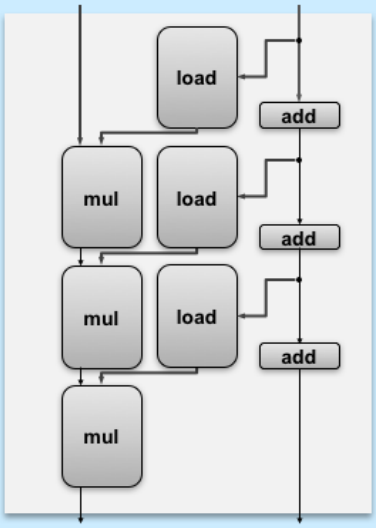


Here we modify the graph of the previous slide to show the relative times required of *mul*, *load*, and *add*.



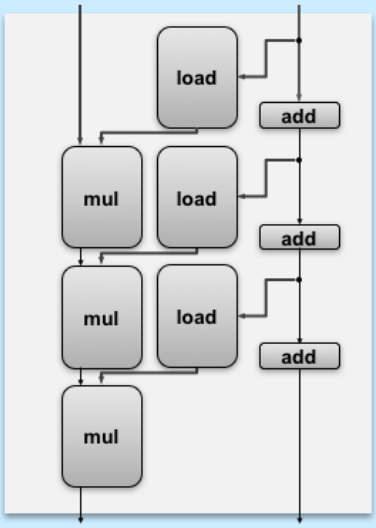
This is Figure 5.15 of Bryant and O'Hallaron.

Pipelined Data-Flow Over Multiple Iterations

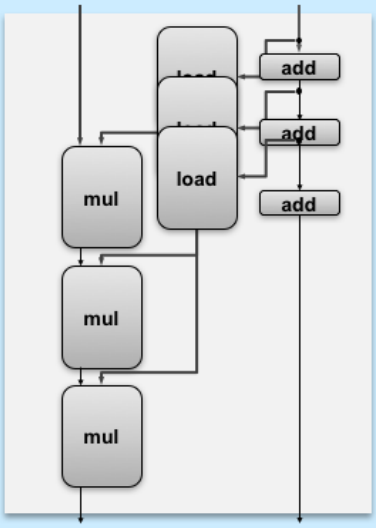


Without pipelining, the data flow would appear as shown in the slide.

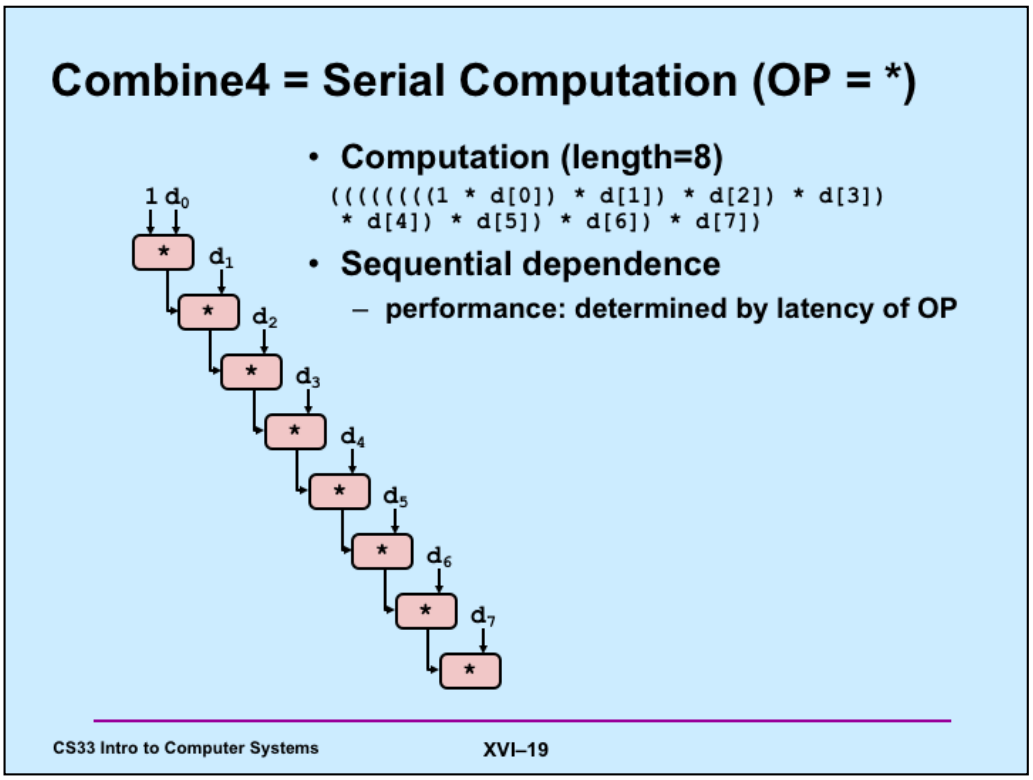
Pipelined Data-Flow Over Multiple Iterations



Pipelined Data-Flow Over Multiple Iterations



Since the loads can be pipelined, it's clear that the multiplies form the critical path. (Note that the multiplies cannot be pipelined since each subsequent multiply depends on the result of the previous.)



Supplied by CMU.

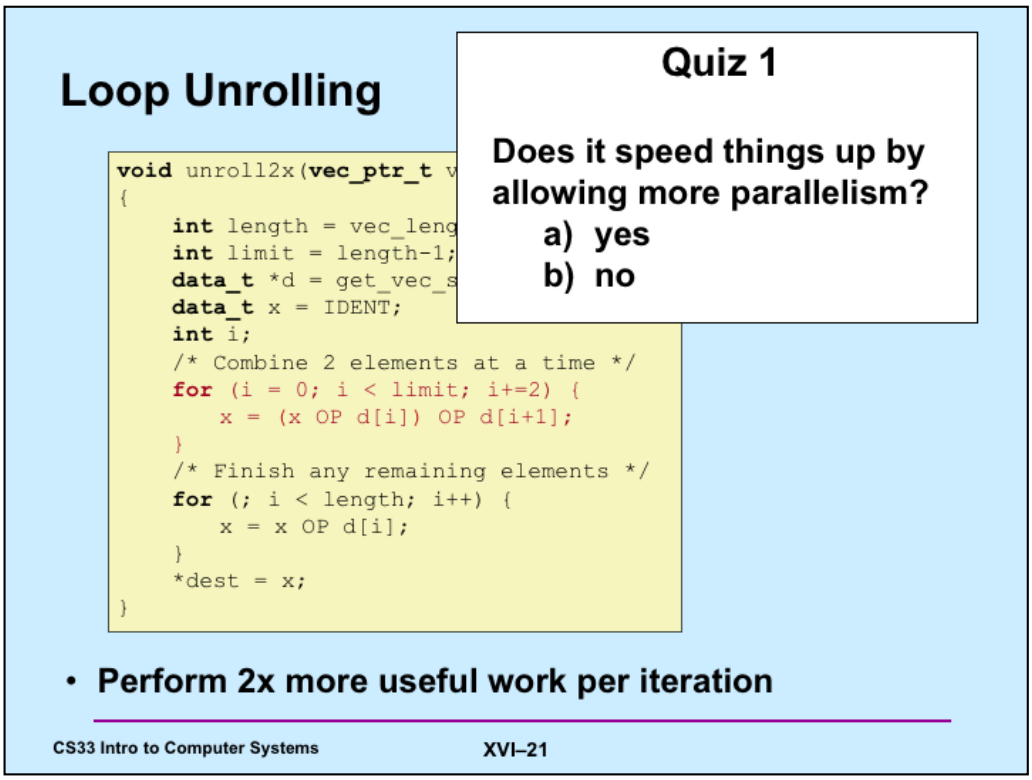
Since the multiplies form the critical path, here we focus only on them.

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**

Supplied by CMU.



Supplied by CMU.

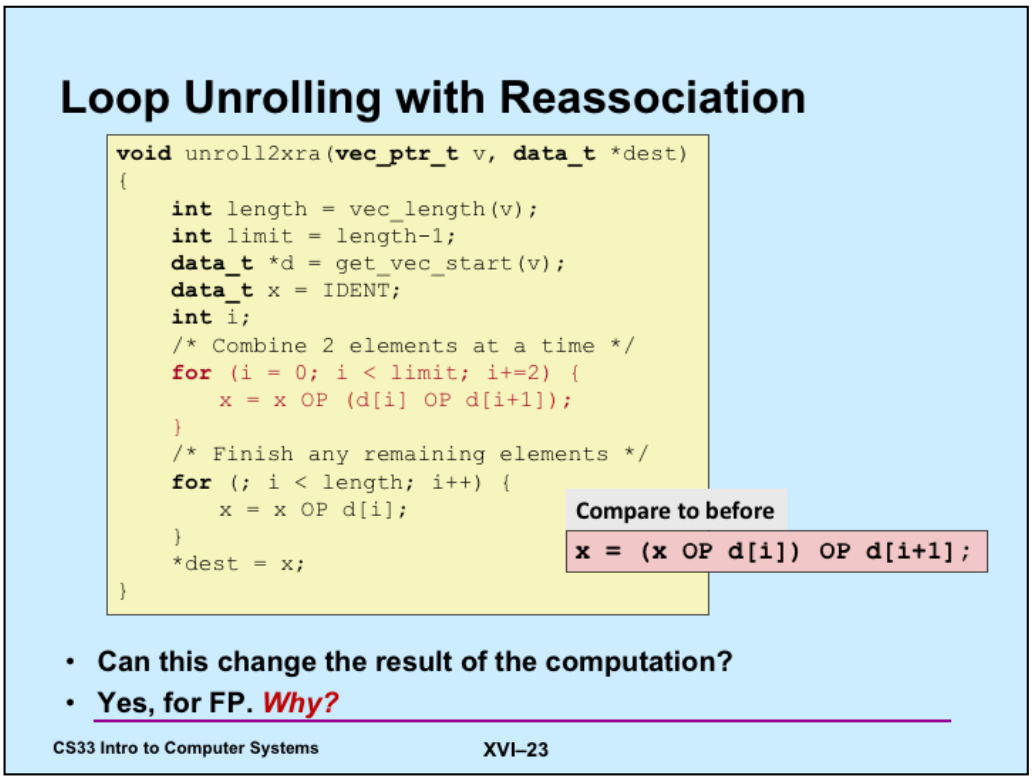
Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x	1.01	3.01	3.01	5.01
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	0.5	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];
```


Supplied by CMU.



Supplied by CMU.

Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x	1.01	3.01	3.01	5.01
Unroll 2x, reassociate	1.01	1.51	1.51	2.51
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	.5	1.0	1.0	.5

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

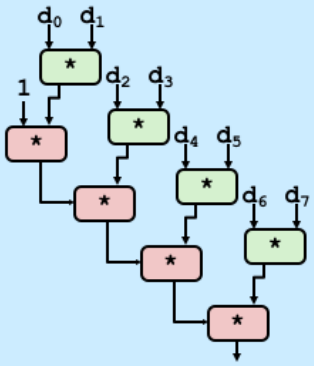
```
x = x OP (d[i] OP d[i+1]);
```

  - why is that? (next slide)

Supplied by CMU.

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

Supplied by CMU.

How much time is required to compute the products shown in the slide? The multiplications in the upper right, directly involving the d_i, could all be done at once, since there are no dependencies; thus computing them can be done in D cycles, where D is the latency required for multiply. The multiplications in the lower left must be done sequentially, since each depends on the previous; thus computing them requires (N/2)*D cycles. Since first of the top right multiplies must be completed before the bottom left multiplies can start, the overall performance has a lower bound of (N/2 + 1)*D.

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**

Supplied by CMU.

Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x	1.01	3.01	3.01	5.01
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	.5	1.0	1.0	.5

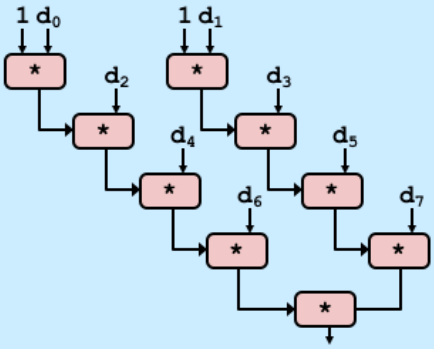
- 2x speedup (over unroll 2x) for int *, FP +, FP *
  - breaks sequential dependency in a "cleaner," more obvious way

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


Supplied by CMU.

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?

Supplied by CMU.

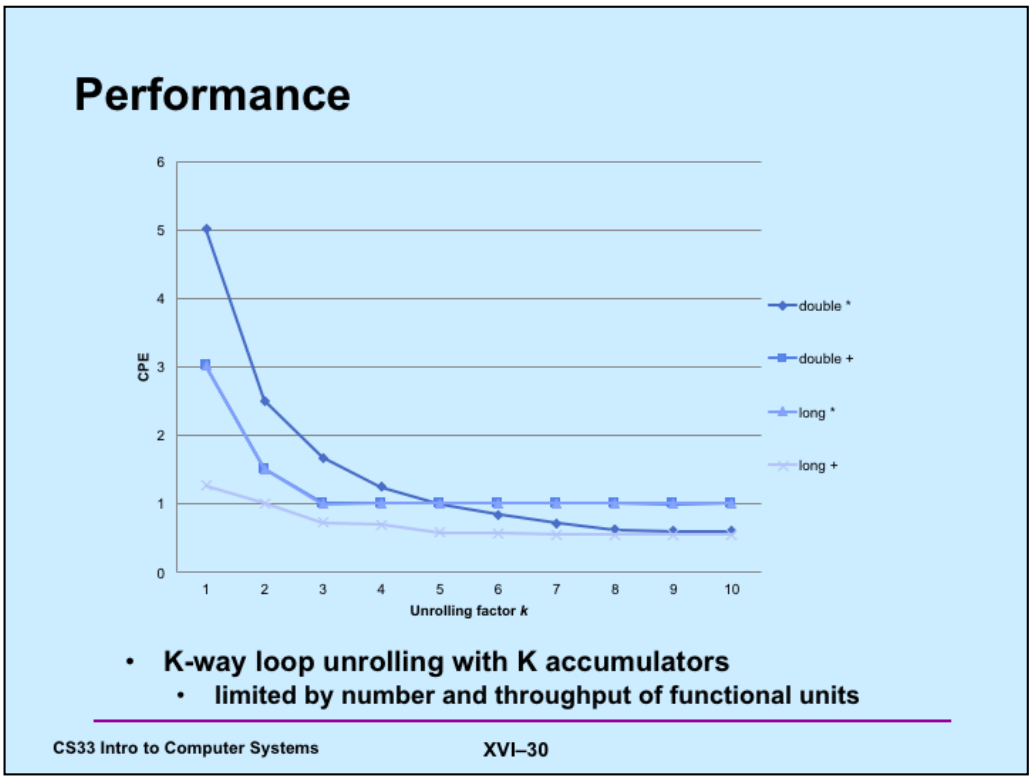
Quiz 2

With 3 accumulators there will be 3 independent streams of instructions; with 4 accumulators 4 independent streams of instructions, etc.

Thus with n accumulators we can have a speedup of O(n), as long as n is no greater than the number of available registers.

a) true

b) false



This is Figure 5.30 from the textbook.

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Achievable scalar	.54	1.01	1.01	.520
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	.5	1.00	1.00	.5

Supplied by CMU.

Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Achievable Scalar	.54	1.01	1.01	.520
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	.5	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**

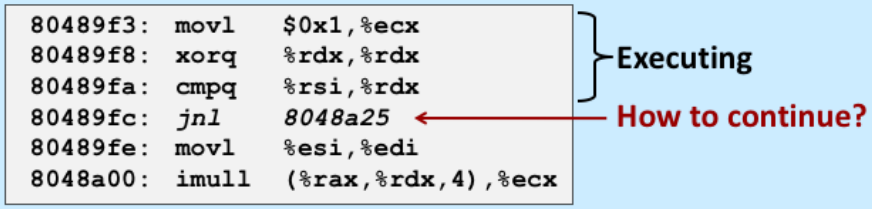
Supplied by CMU.

We'll look at vector instructions in an upcoming lecture.

SSE stands for "streaming SIMD extensions". SIMD stands for "single instruction multiple data" – these are instructions that operate on vectors.

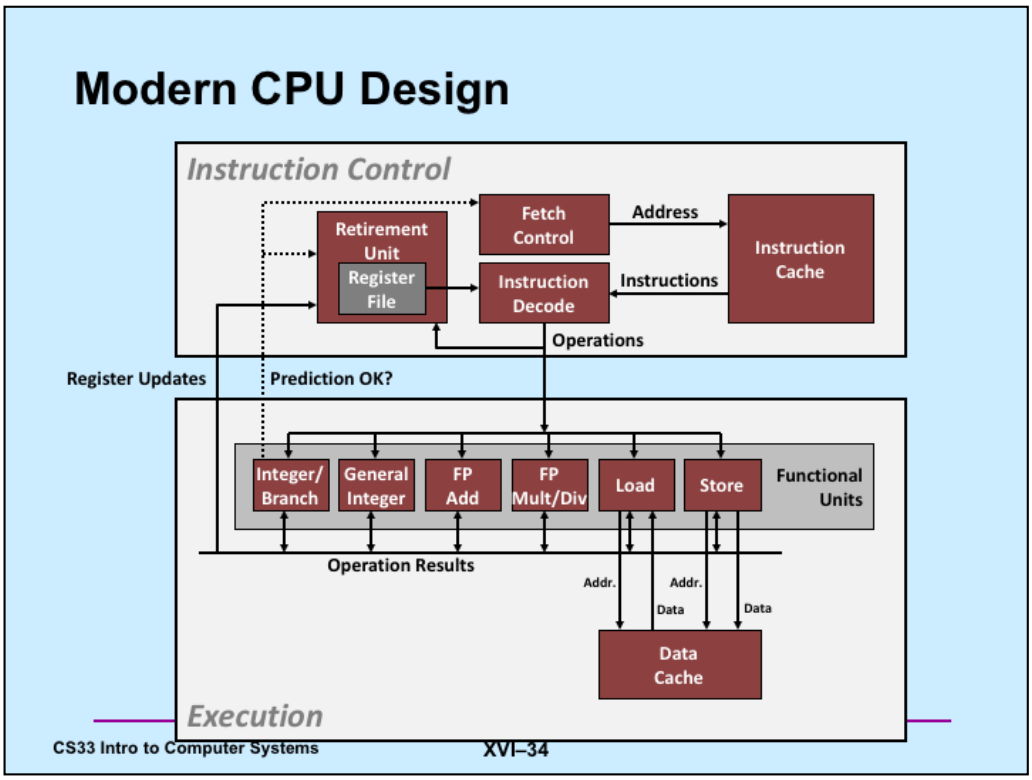
What About Branches?

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

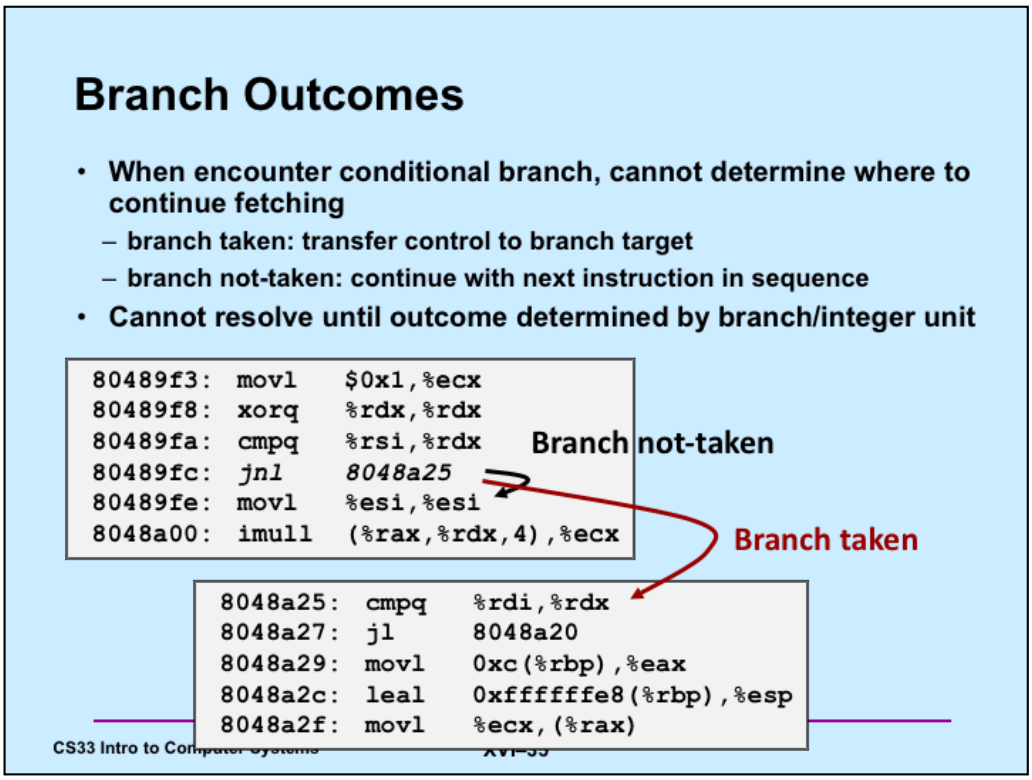


  - **when it encounters conditional branch, cannot reliably determine where to continue fetching**

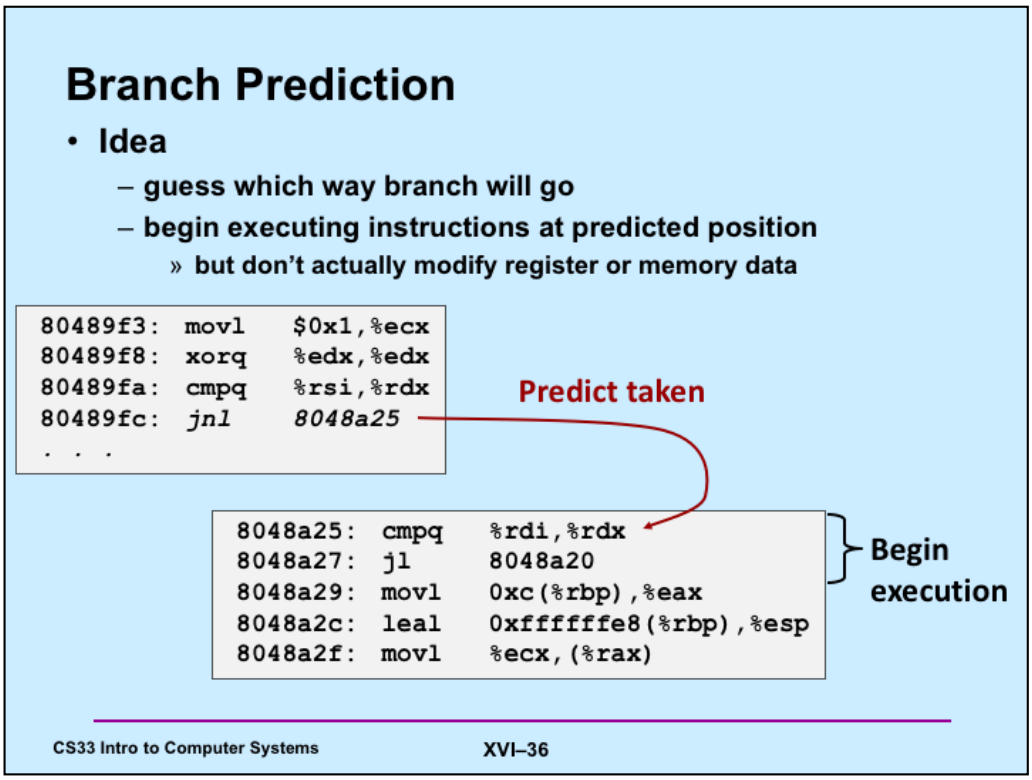
Supplied by CMU, converted to x86-64.



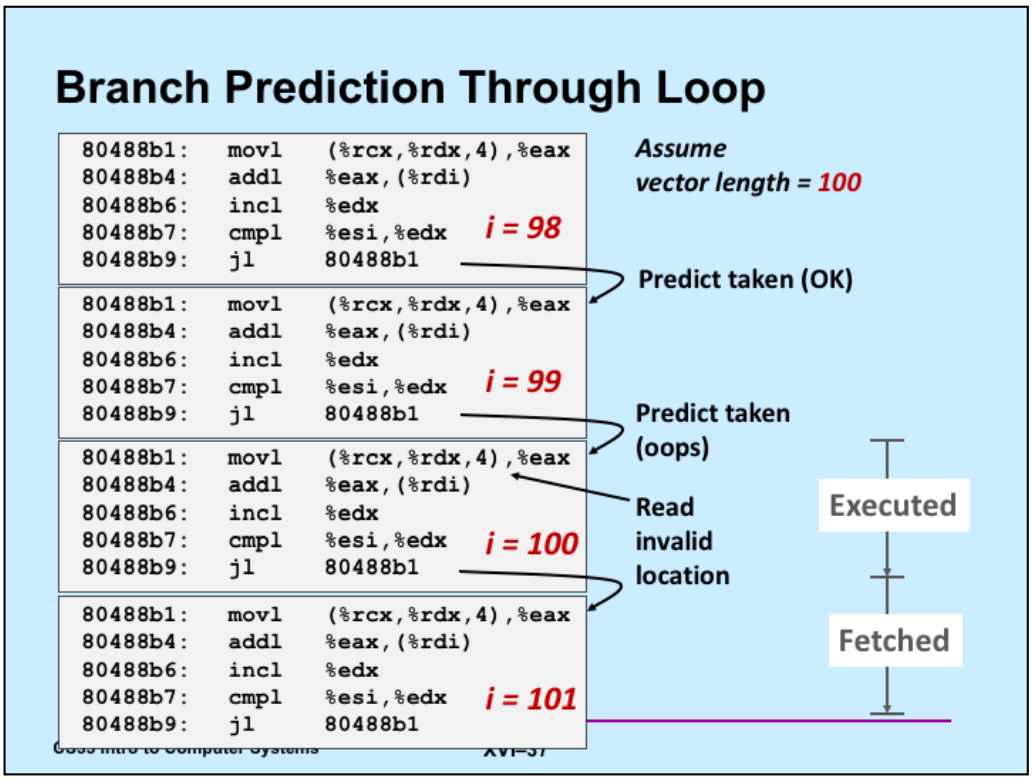
Supplied by CMU.



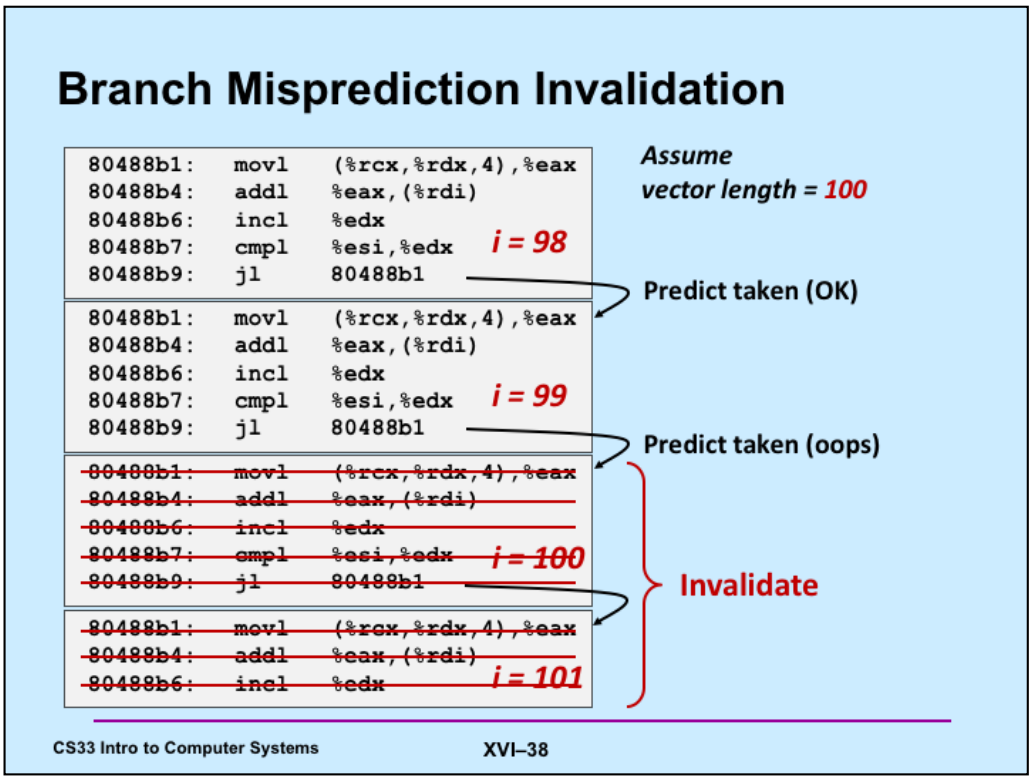
Supplied by CMU.



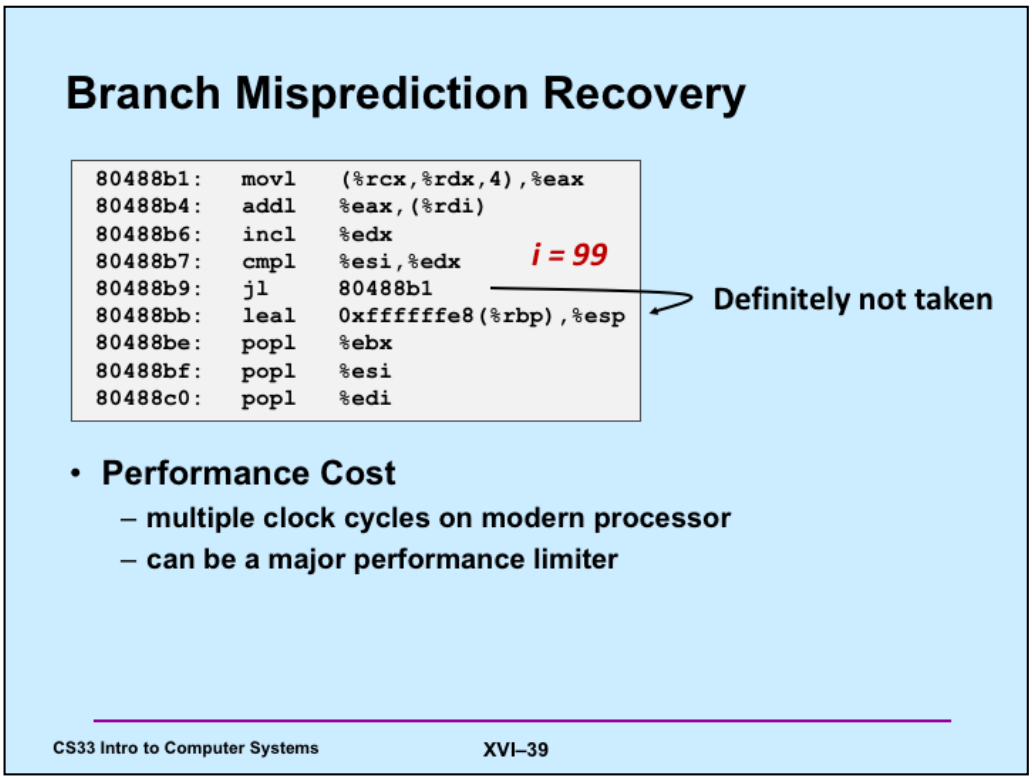
Supplied by CMU.



Supplied by CMU.



Supplied by CMU.



Supplied by CMU.

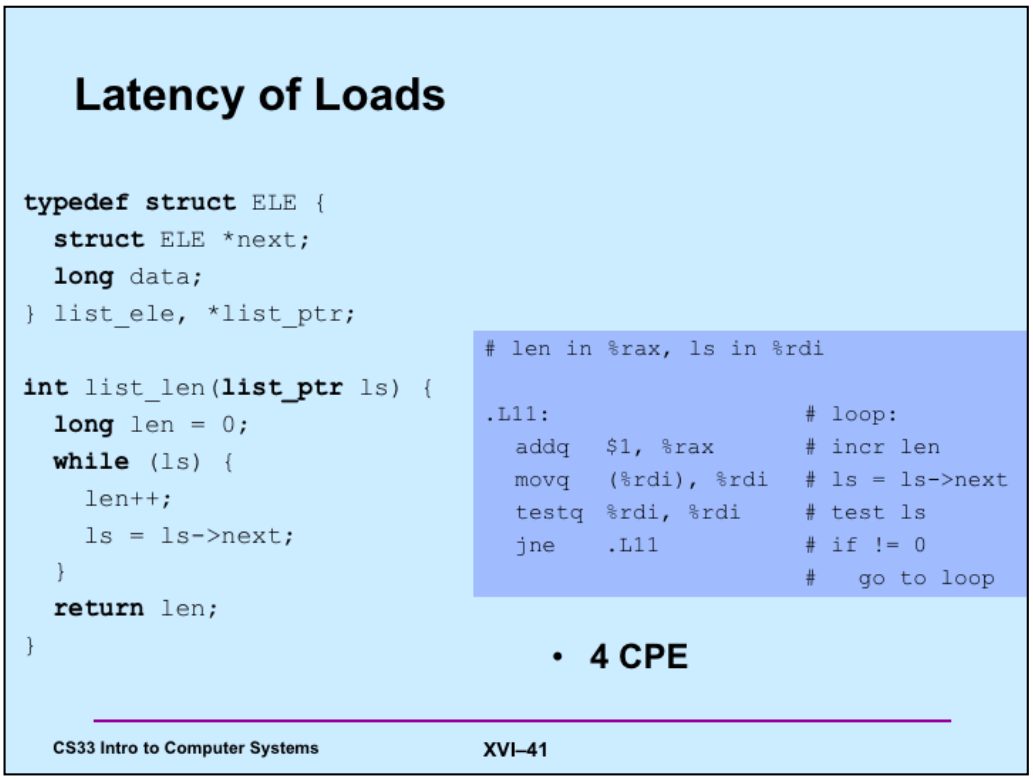
Conditional Moves

```
void minmax1(long *a, long *b,
     long n) {
  long i;
  for (i=0; i<n; i++) {
    if (a[i] > b[i]) {
      long t = a[i];
      a[i] = b[i];
      b[i] = t;
    }
  }
}
```

```
void minmax2(long *a, long *b,
     long n) {
  long i;
  for (i=0; i<n; i++) {
    long min = a[i] < b[i]?
          a[i] : b[i];
    long max = a[i] < b[i]?
          b[i] : a[i];
    a[i] = min;
    b[i] = max;
  }
}
```

- Compiled code uses conditional branch
  - 13.5 CPE for random data
  - 2.5 – 3.5 CPE for predictable data
- Compiled code uses conditional move instruction
  - 4.0 CPE regardless of data's pattern

This example is from the textbook. Note that in *minmax1*, a conditional move cannot be used, since the compiler does not know whether *a* and *b* are aliased. In *minmax2*, since both *min* and *max* are computed, the compiler is assured that aliasing doesn't matter.



This example is from the textbook (Figure 5.31). Here we can't execute the loads in parallel, since each load is dependent on the result of the previous load. The point is that loads (fetching data from memory) have a latency of 4 cycles.

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**

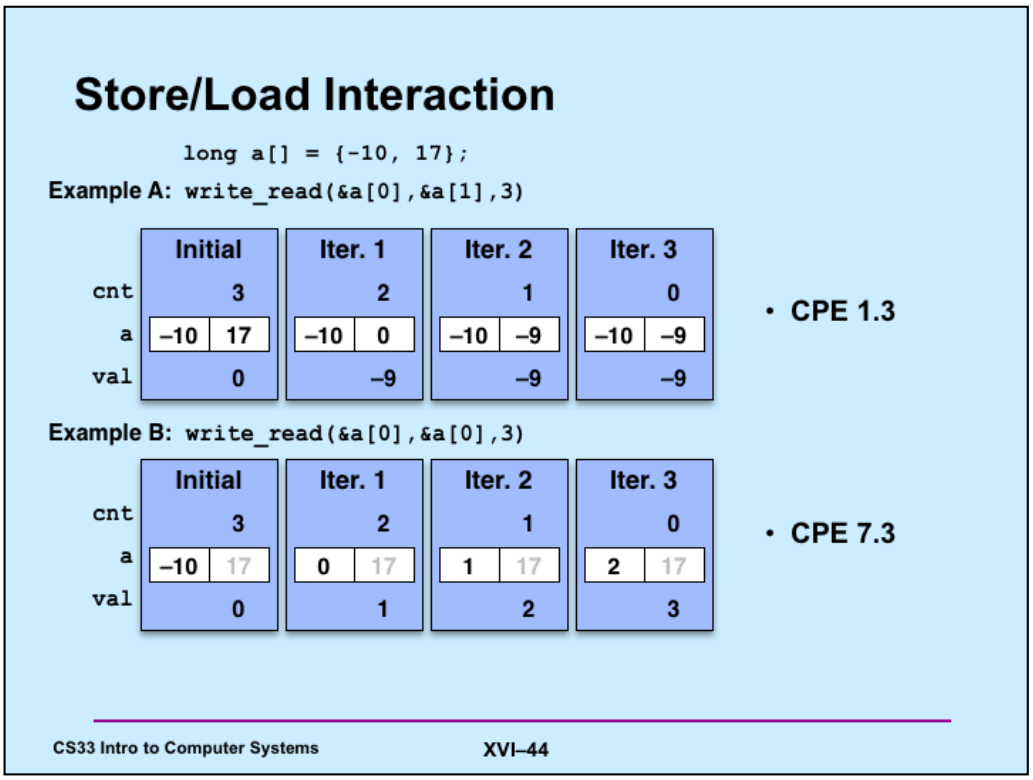
This is adapted from Figure 5.32 of the textbook. There are no data dependencies and thus the stores can be pipelined.

Store/Load Interaction

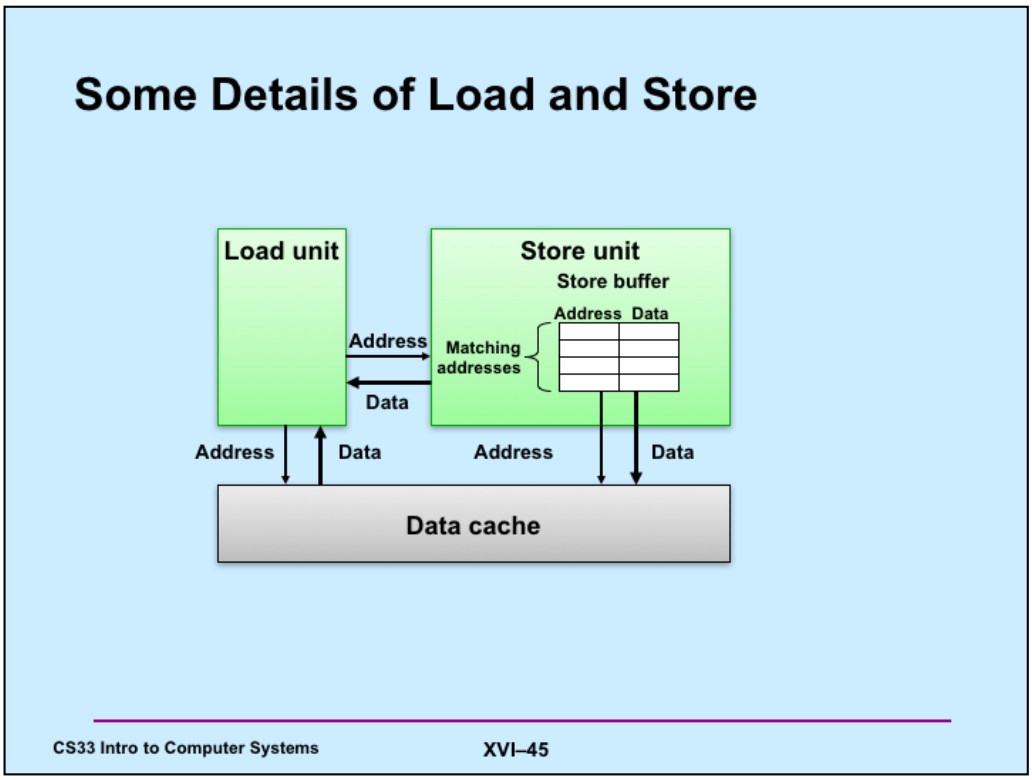
```
void write_read(long *src, long *dest, long n) {
  long cnt = n;
  long val = 0;

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

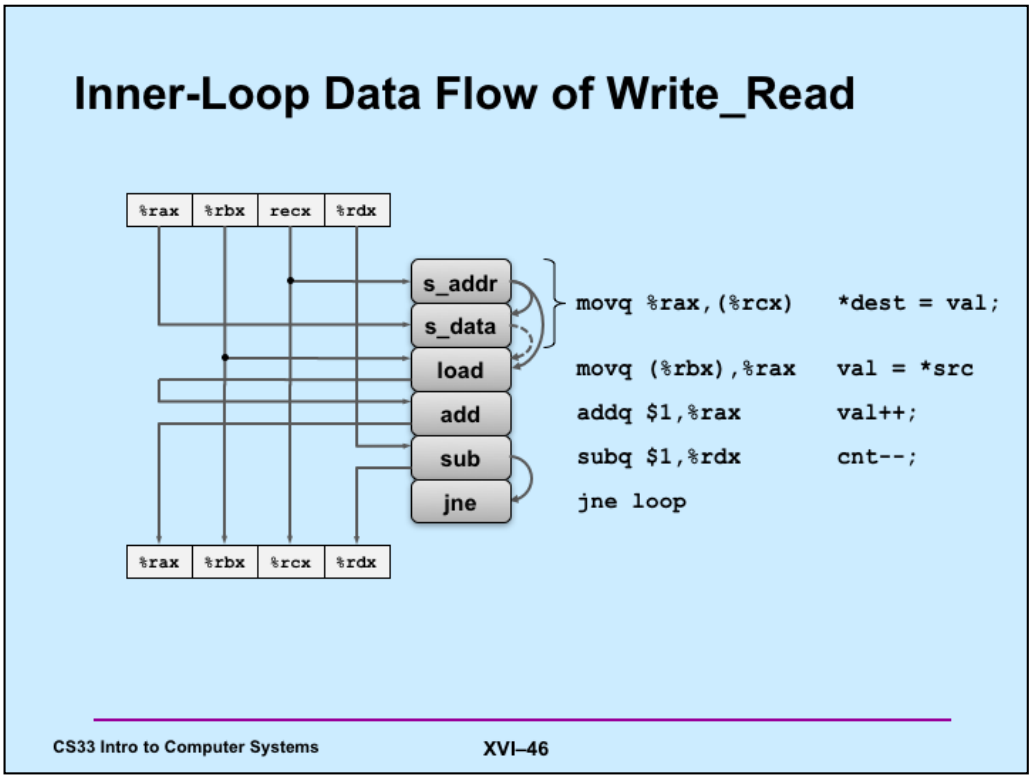

This code is from the textbook.



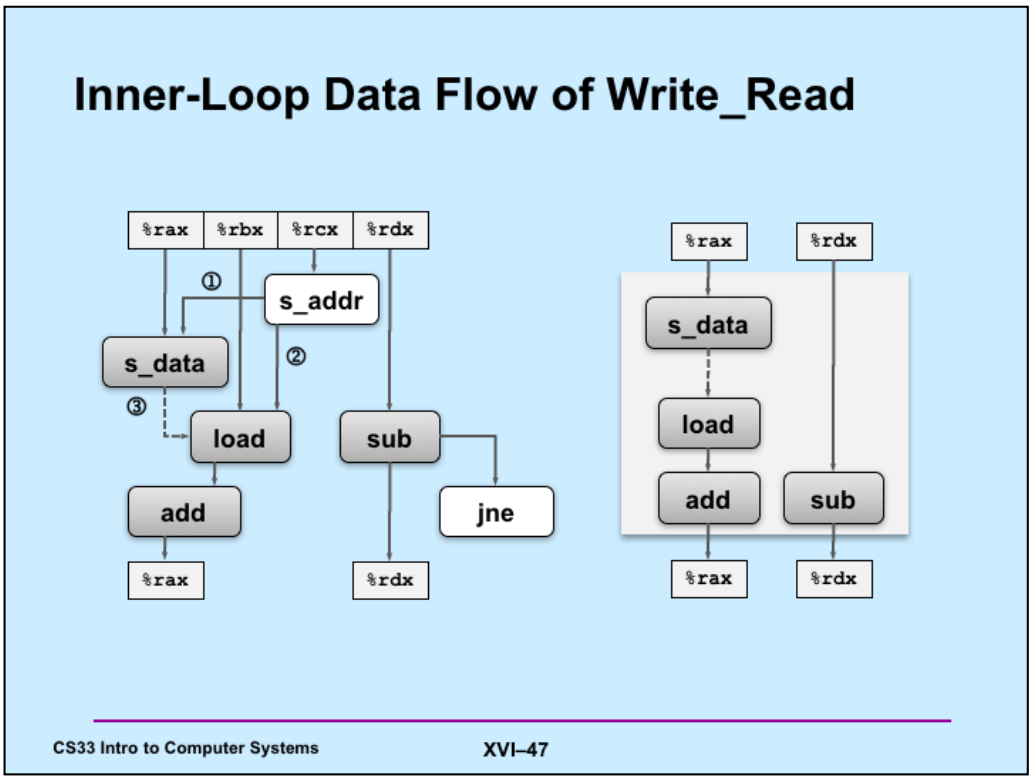
This is Figure 5.33 of the textbook. Performance depends upon whether *src* and *dest* are the same location.



This is Figure 5.34 of the textbook.



This is Figure 5.35 of the textbook.

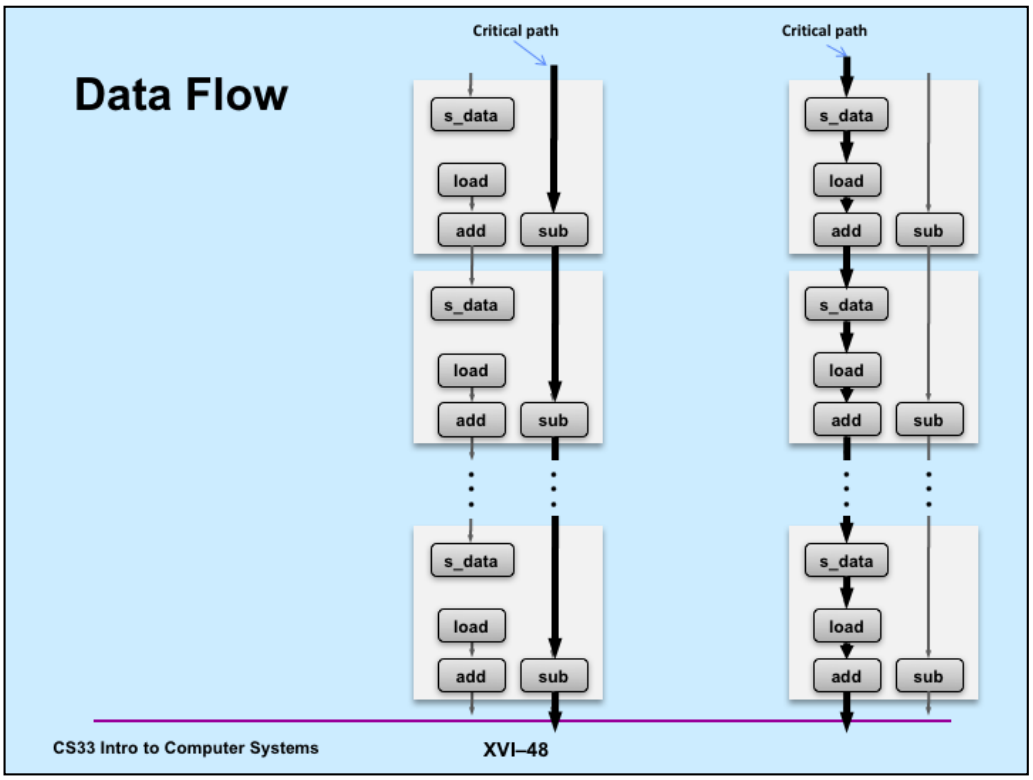


This is Figure 5.36 of the textbook.

The arc labelled 1 represents the dependency that the address of where data must be stored must be computed before the data is stored.

The arc labelled 2 represents the check the processor must make to test whether the store will be to the location of the load.

The arc labelled 3 represents the dependency of the load on the stored data if they use the same address.



This is adapted from Figure 5.37 of the textbook.

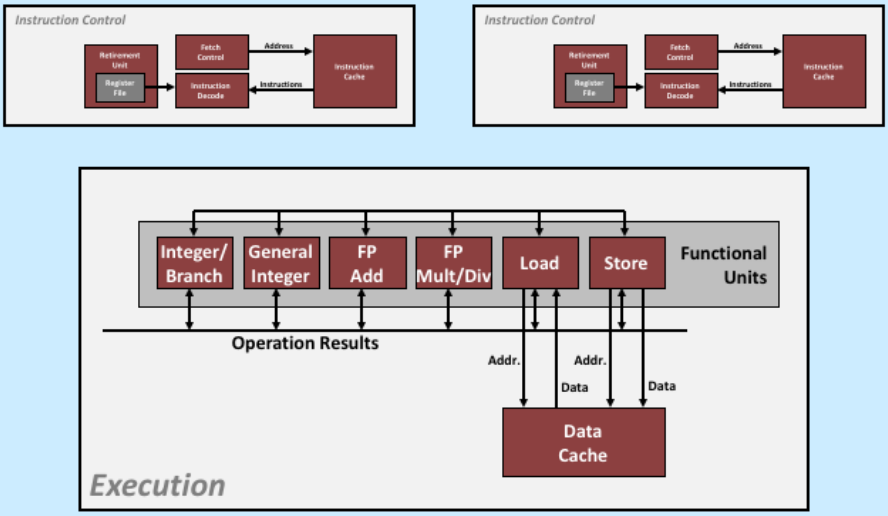
On the left is the case in which the store is to a different location than the store. On the right is the case in which they are to the same location.

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)

Supplied by CMU.

Hyper Threading



Multiple Cores