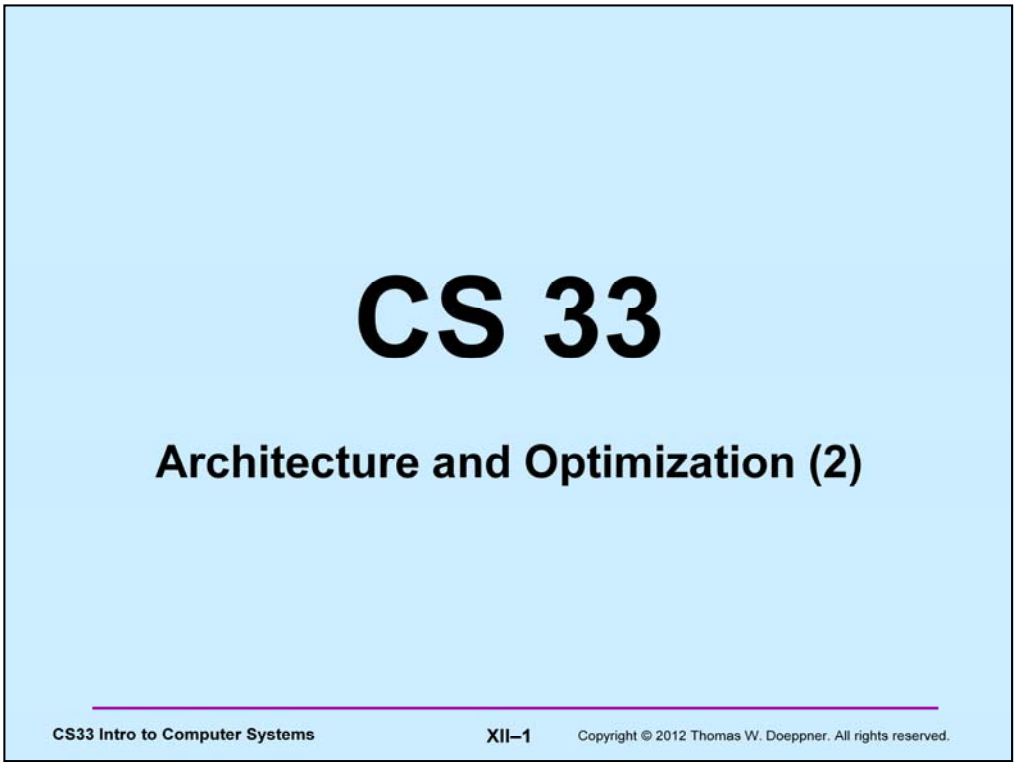
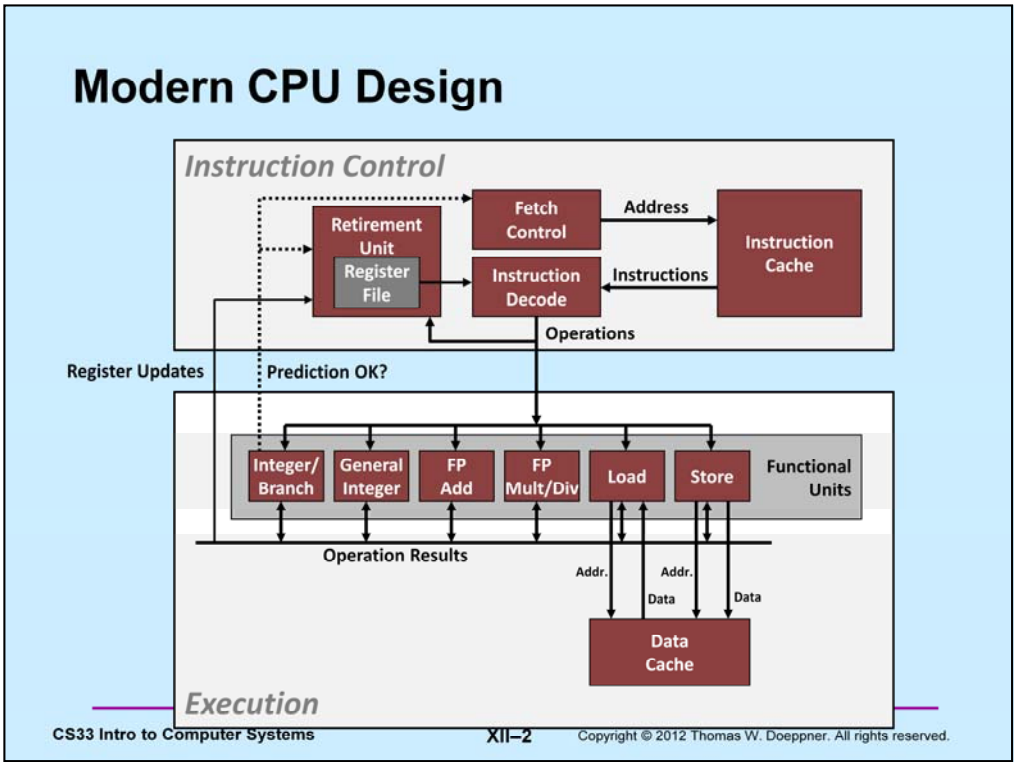
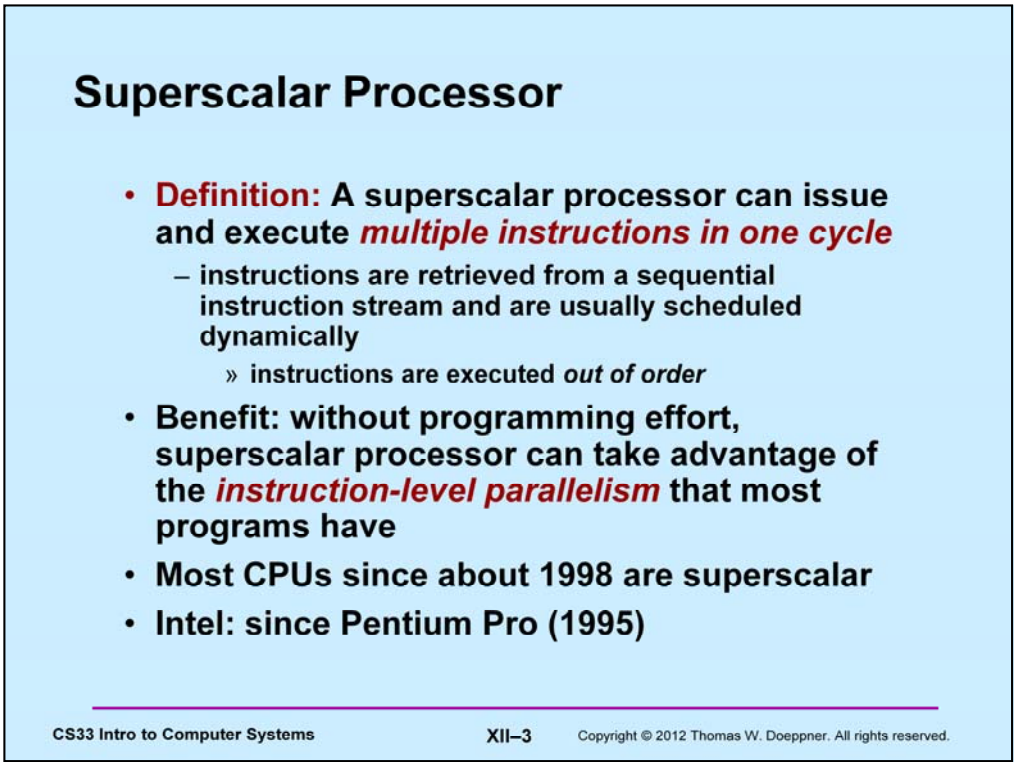


Most of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in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

- addl %eax, %edx
  - a single operation
- addl %eax, 4(%edx)
  - three operations
    - » load value from memory
    - » add to it the contents of %eax
    - » store result in memory

Instruction-Level Parallelism

- `addl 4(%eax), %eax`
  `addl %ebx, %edx`
  - can be executed simultaneously: completely independent
- `addl 4(%eax), %ebx`
  `addl %ebx, %edx`
  - can also be executed simultaneously, but some coordination is required

Out-of-Order Execution

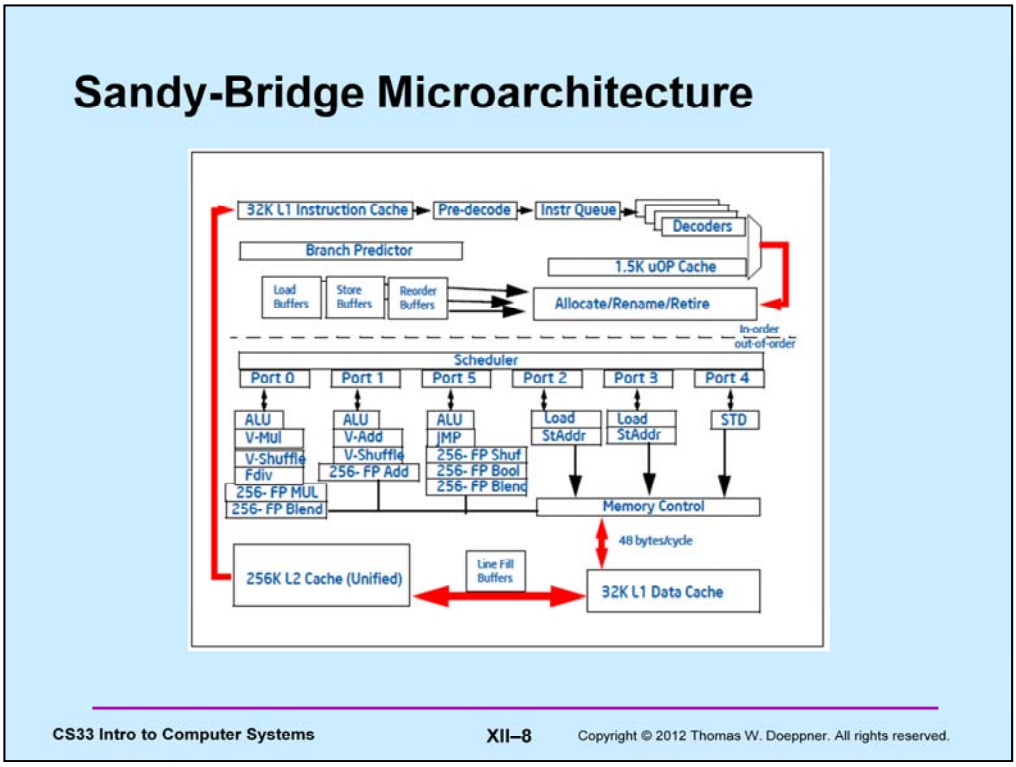
```
• movss    (%rbp), %xmm0
  mulss    (%rax, %rdx, 4), %xmm0
  movss    %xmm0, (%rbp)
  addq     %r8d, %r9d
  imul     q    %rcx, %r12d
  addq     $1, %rdx
```

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

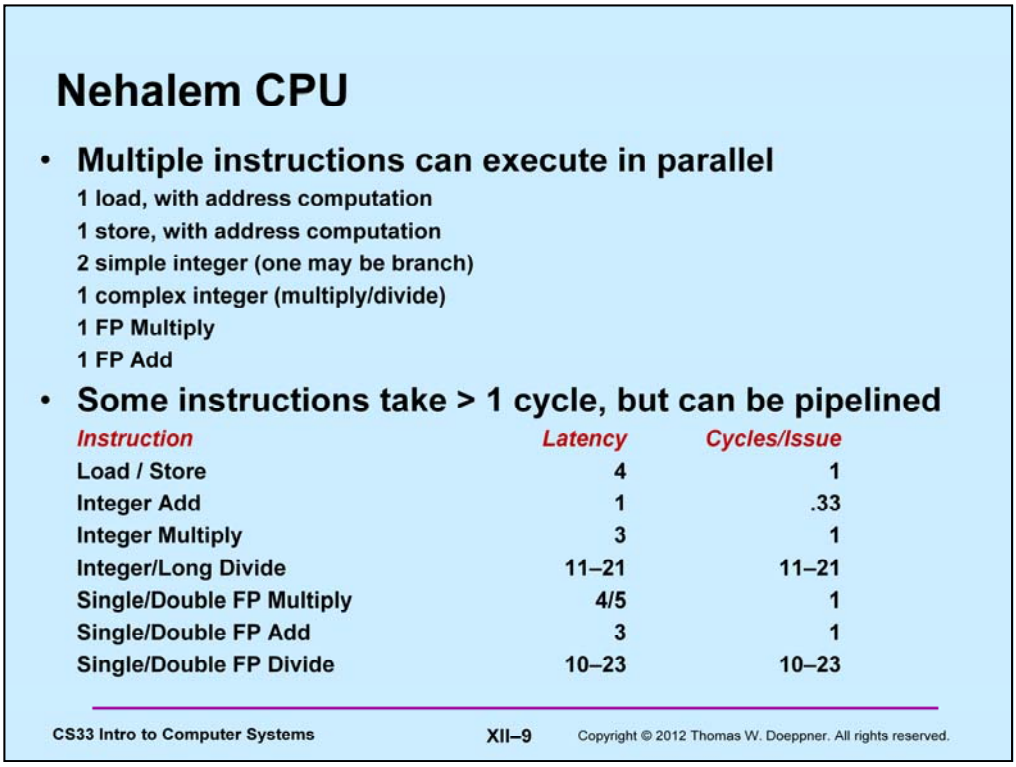
Speculative Execution

```
80489f3:    movl   $0x1,%ecx
80489f8:    xorl   %edx,%edx
80489fa:    cmpl   %esi,%edx
80489fc:    jnl    8048a25
80489fe:    movl   %esi,%esi
8048a00:    imull  (%eax,%edx,4),%ecx
```

perhaps execute these instructions



This is Figure 2-1 from *Intel® 64 and IA-32 Architectures Optimization Reference Manual,* published in April 2012. The document can be found at http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf. It shows the microarchitecture of Intel's "Sandy Bridge" architecture, the successor to the Nehalem architecture.



Supplied by CMU.

"Nehalem" is Intel's code name for its Core I7 processor design.

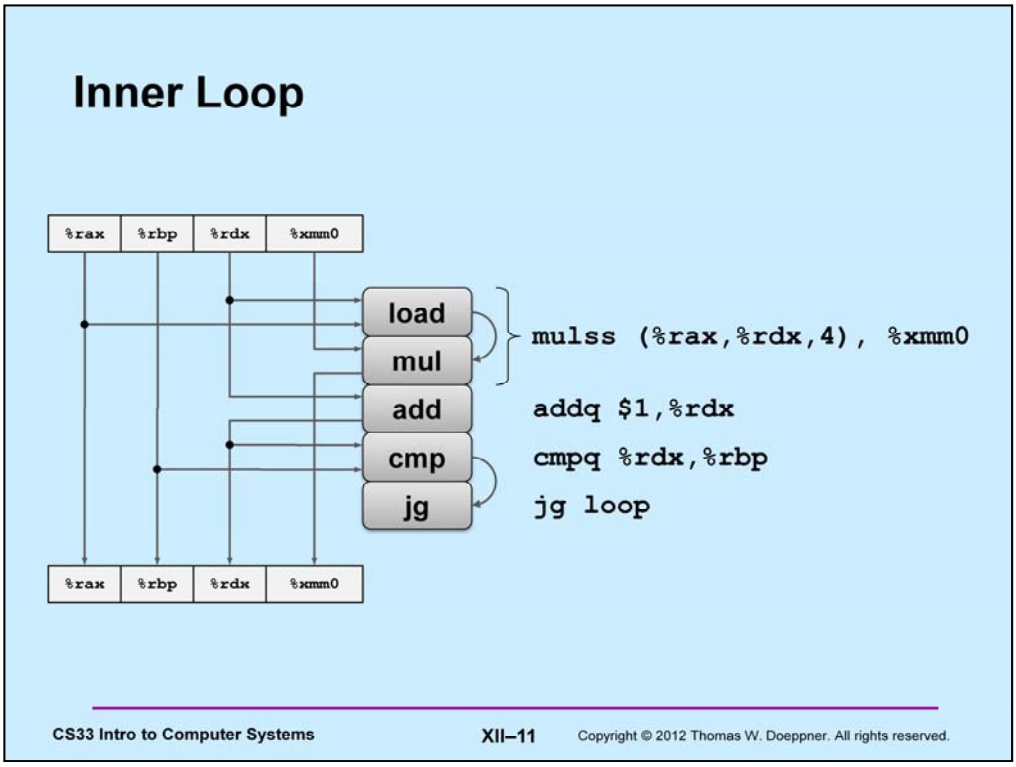
x86-64 Compilation of Combine4

- Inner loop (case: integer multiply)

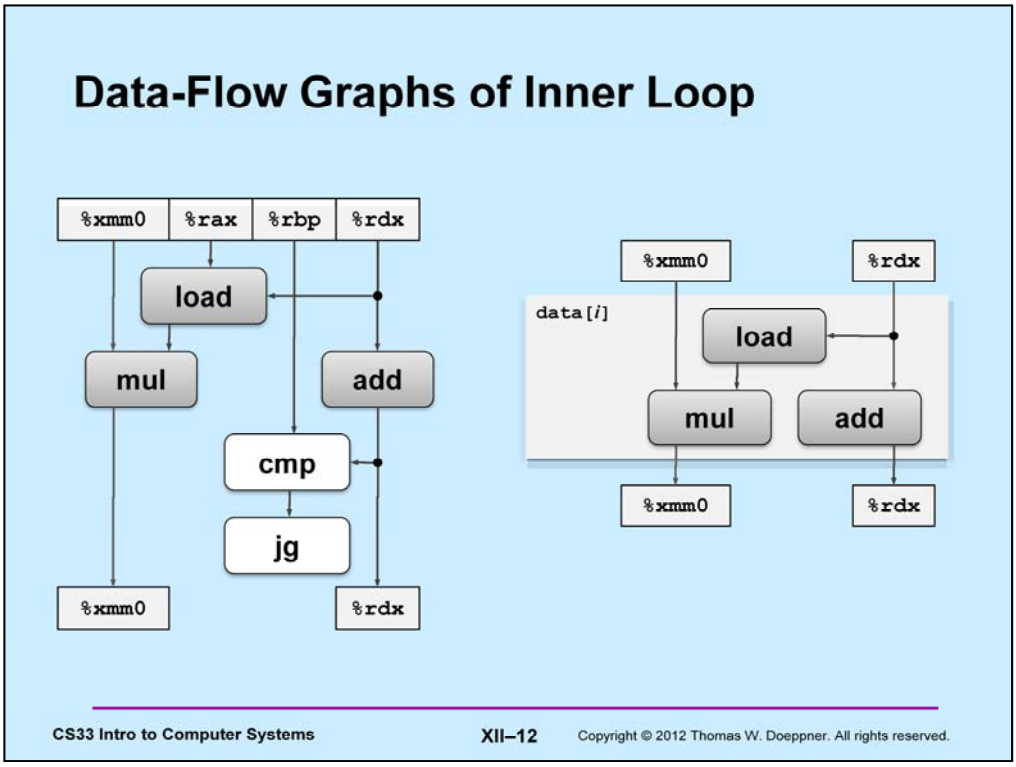
```
.L519:                          # Loop:
  imull  (%rax,%rdx,4), %ecx    # t = t * d[i]
  addq   $1, %rdx               # i++
  cmpq   %rdx, %rbp             # Compare length:i
  jg     .L519                  # If >, goto Loop
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Latency bound	1.0	3.0	3.0	5.0
Thoughput bound	1.0	1.0	1.0	1.0

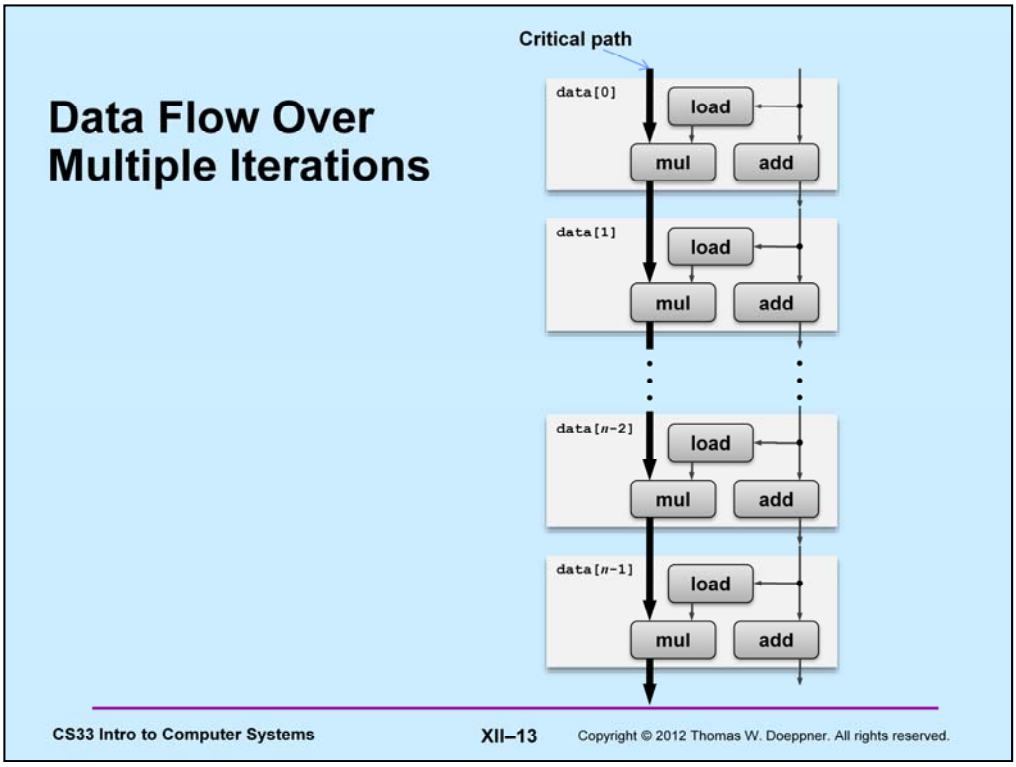
Supplied by CMU.



This is Figure 5.13 of Bryant and O'Hallaron.

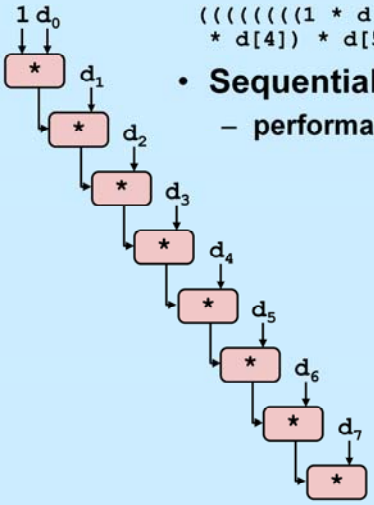


These are Figures 5.14 a and b of Bryant and O'Hallaron.



This is Figure 5.15 of Bryant and O'Hallaron.

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**

Supplied by CMU.

Loop 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;
}
```

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

Supplied by CMU.

Effect of Loop Unrolling

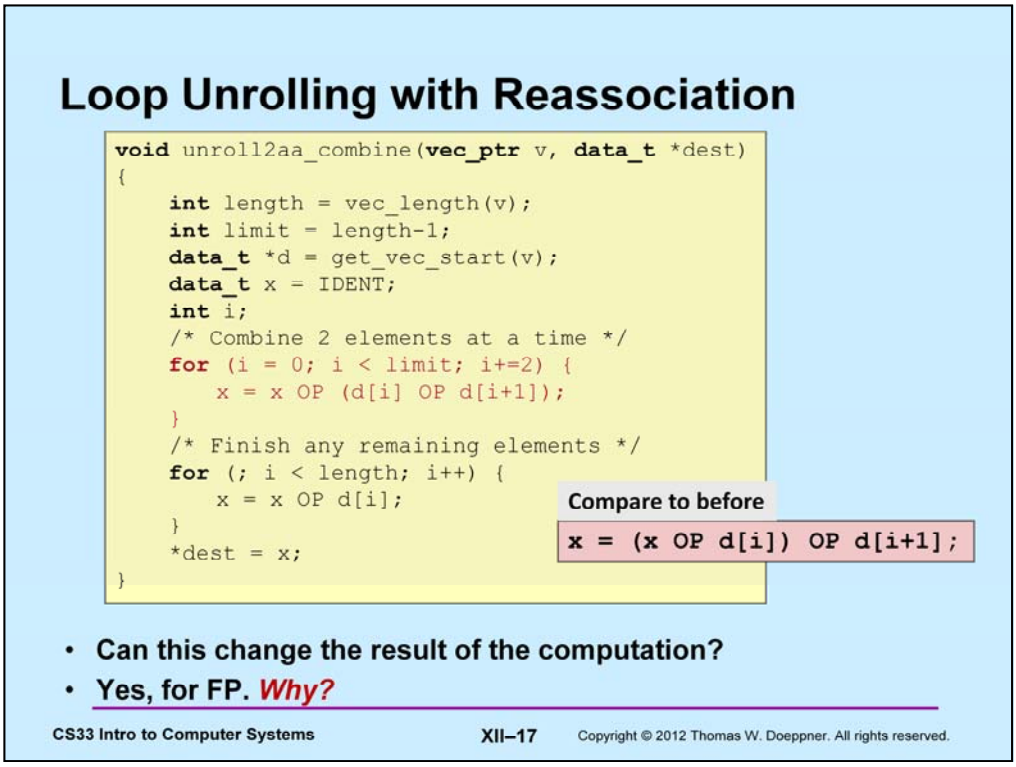
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	1.0	1.0	1.0	1.0

- Helps integer multiply
  - below latency bound
  - compiler does clever optimization
- Others don't improve. *Why?*
  - still sequential dependency

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


Supplied by CMU.

What the compiler does for the case of integer multiplication is to apply reassociation, discussed in the next slide.



Supplied by CMU.

Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Unroll 2x, reassociate	2.0	1.5	1.5	3.0
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	1.0	1.0	1.0	1.0

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

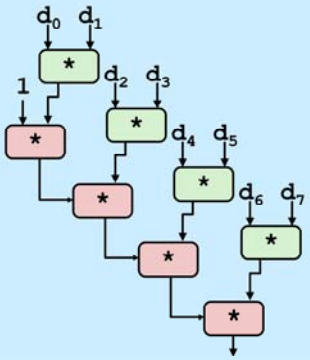
```
x = x OP (d[i] OP d[i+1]);
```

  - 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 FP mult

Supplied by CMU.

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;
}
```

- **Different form of reassociation**

Supplied by CMU.

Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Unroll 2x, reassociate	2.0	1.5	1.5	3.0
Unroll 2x parallel 2x	1.5	1.5	1.5	2.5
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	1.0	1.0	1.0	1.0

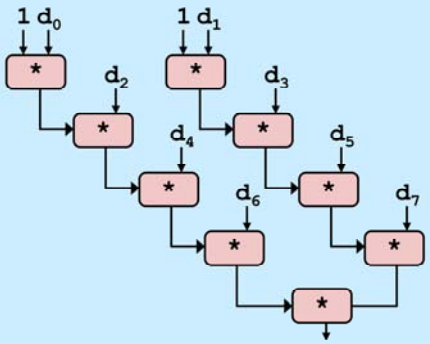
- 2x speedup (over unroll2) for int *, FP +, FP *
  - breaks sequential dependency in a "cleaner," more obvious way

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


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**
  - CPE matches prediction!

What Now?

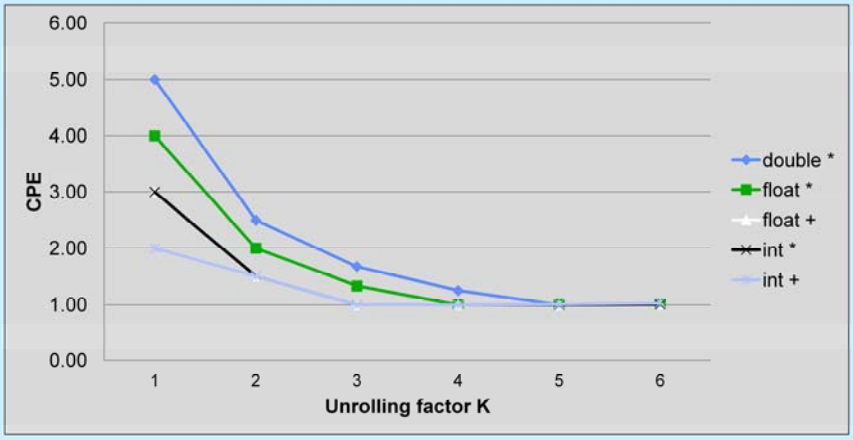
Supplied by CMU.

Unrolling & Accumulating

- Idea
  - can unroll to any degree L
  - can accumulate K results in parallel
  - L must be multiple of K
- Limitations
  - diminishing returns
    - » cannot go beyond throughput limitations of execution units
  - large overhead for short lengths
    - » finish off iterations sequentially

Supplied by CMU.

Performance



- K-way loop unrolling with K accumulators

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar optimum	1.00	1.00	1.00	1.00
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	1.00	1.00	1.00	1.00

- Limited only by throughput of functional units
- Up to 29X improvement over original, unoptimized code

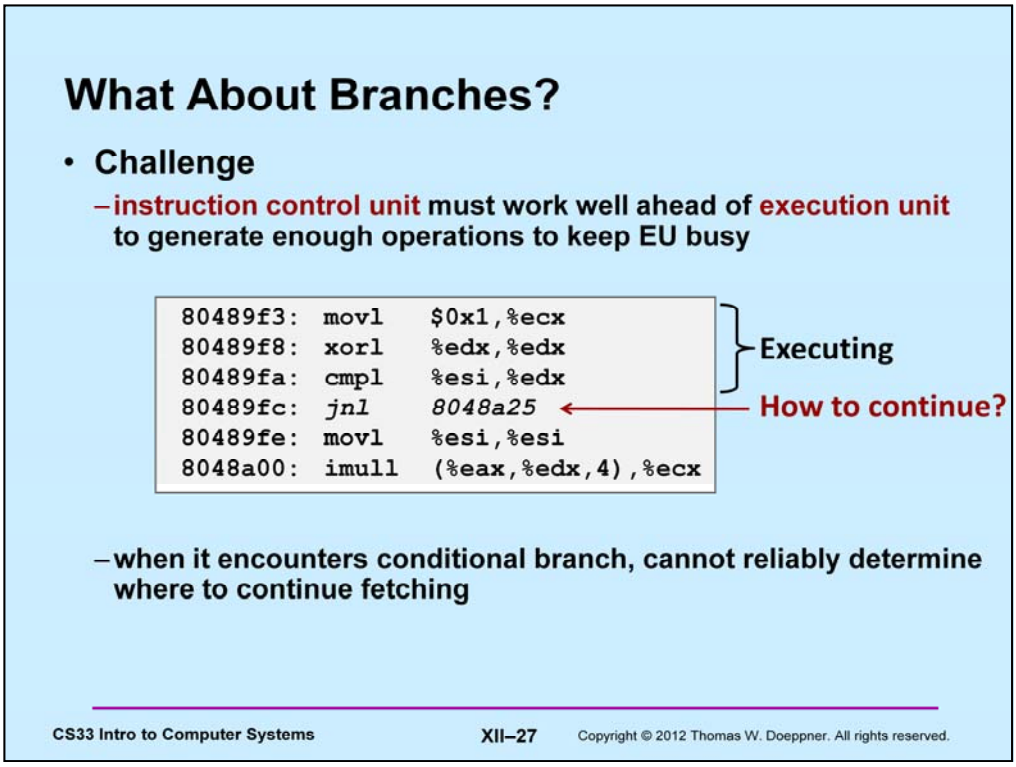
Supplied by CMU.

Using Vector Instructions

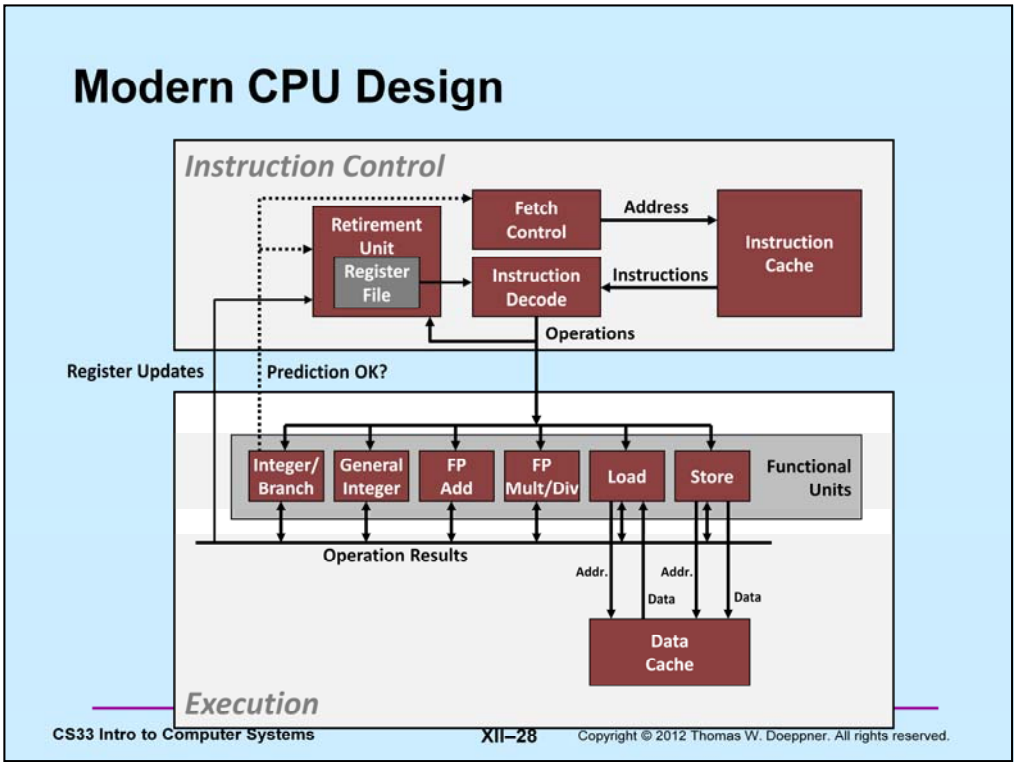
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar optimum	1.00	1.00	1.00	1.00
Vector optimum	0.25	0.53	0.53	0.57
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	1.00	1.00	1.00	1.00
Vec throughput bound	0.25	0.50	0.50	0.50

- Make use of SSE Instructions
  - parallel operations on multiple data elements
  - see Web Aside OPT:SIMD on CS:APP web page

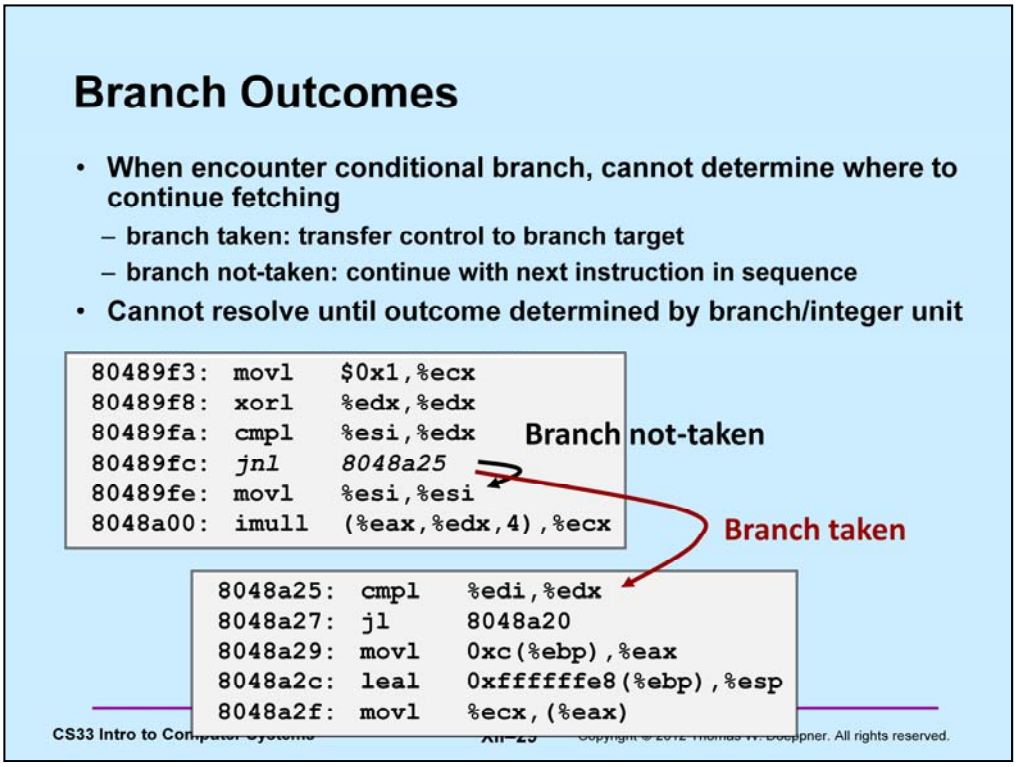
Supplied by CMU.



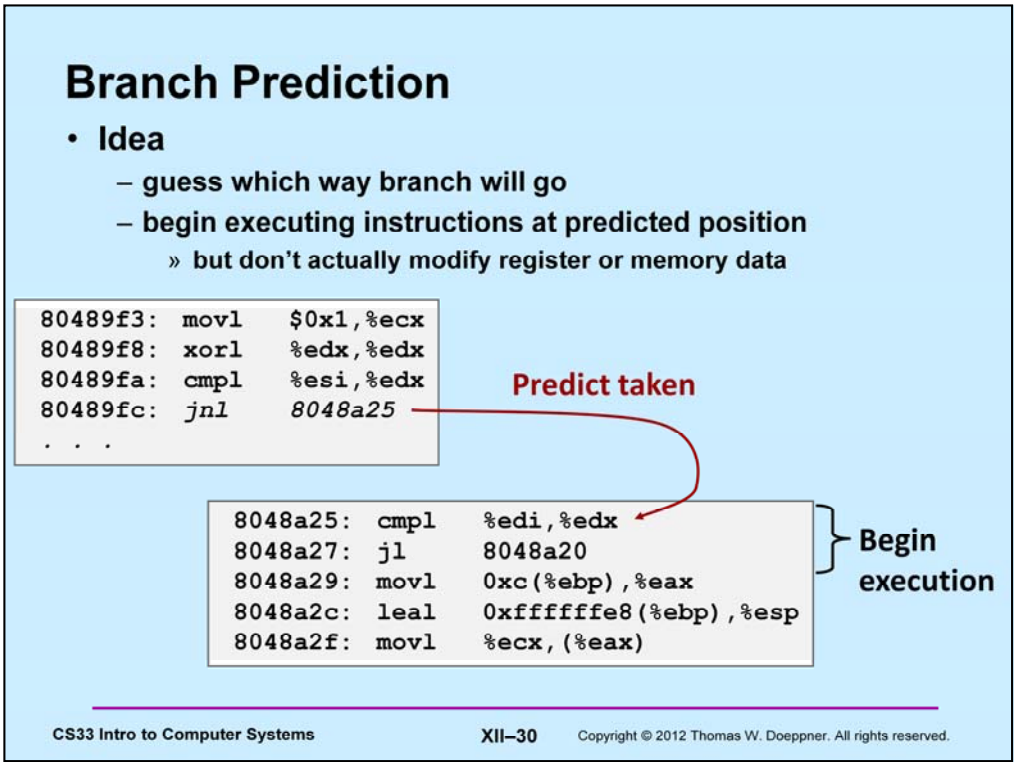
Supplied by CMU.



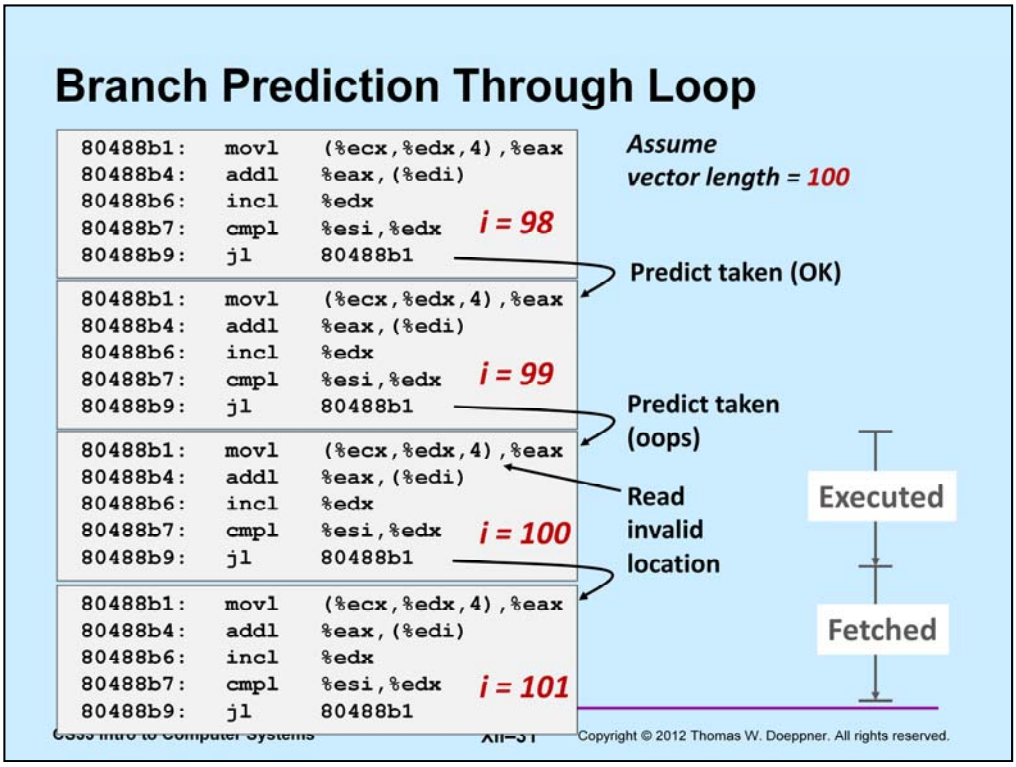
Supplied by CMU.



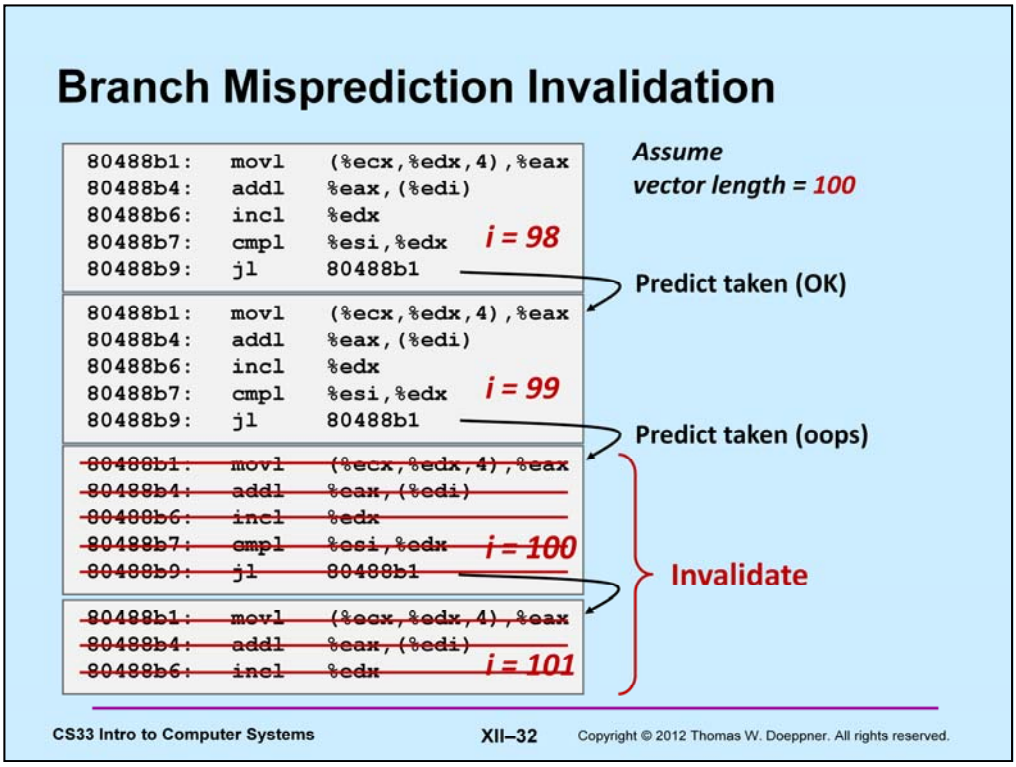
Supplied by CMU.



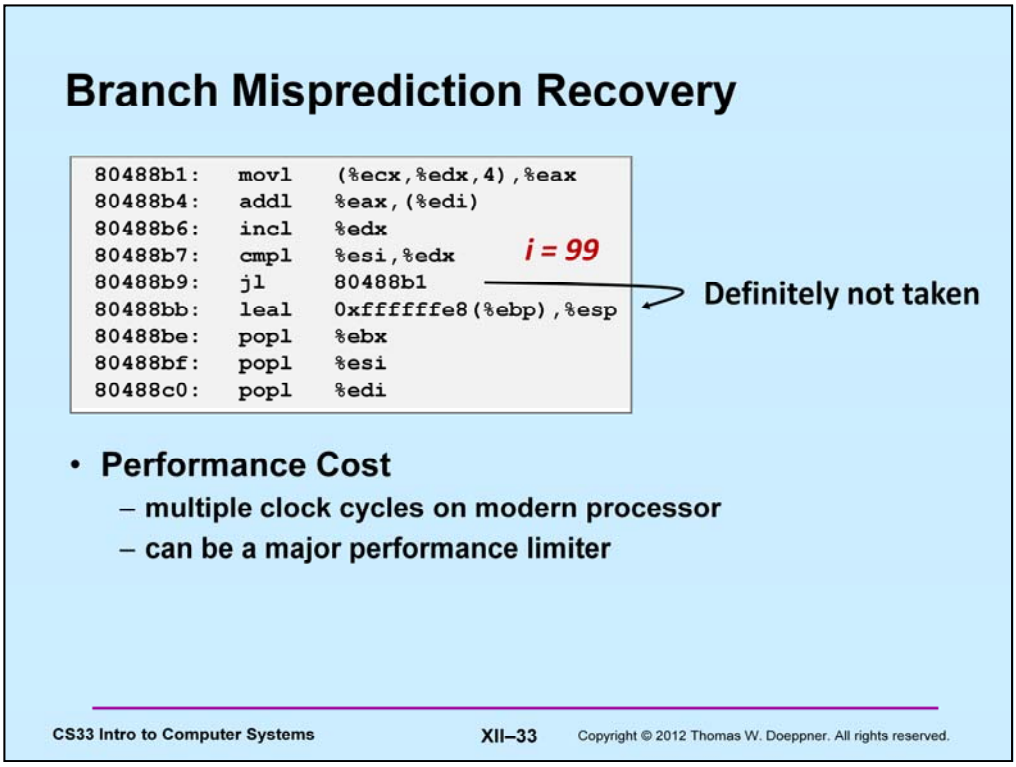
Supplied by CMU.



Supplied by CMU.



Supplied by CMU.



Supplied by CMU.

Conditional Moves

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

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

- Compiled code uses conditional branch
  - 14.5 CPE for random data
  - 2.0 – 4.0 CPE for predictable data
- Compiled code uses conditional move instruction
  - 5.0 CPE regardless of data's pattern

This example is from the textbook.

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.