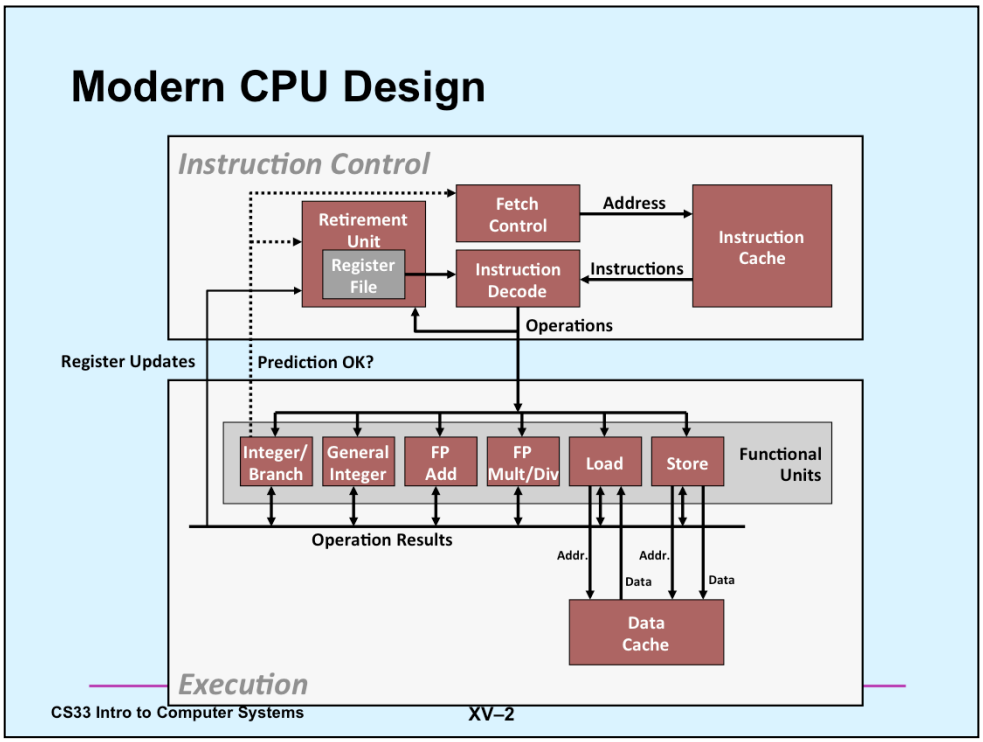
CS 33

Architecture and Optimization (2)

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.

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)

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
  imulq    %rcx, %r12d
  addq     $1, %rdx
```

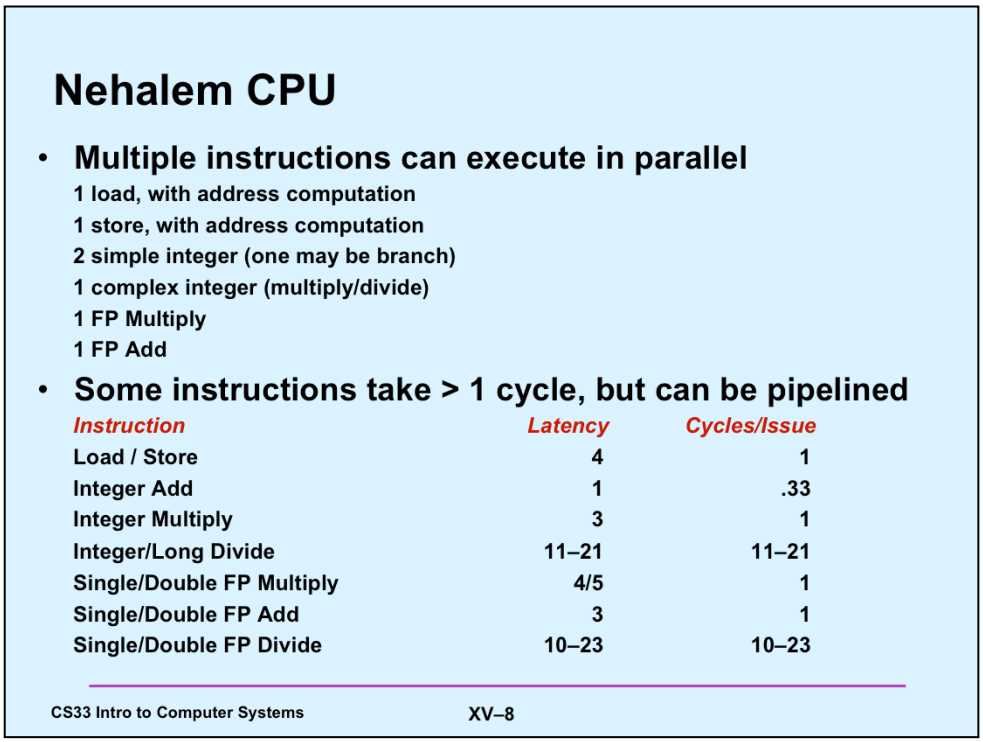
these can be executed without waiting for the first three to finish

Note that the first three instructions are floating-point instructions, and %xmm0 is a floating-point register. We will discuss x86 floating point in an upcoming lecture.

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



Supplied by CMU.

“Nehalem” is Intel’s code name for its Core I7 processor design.

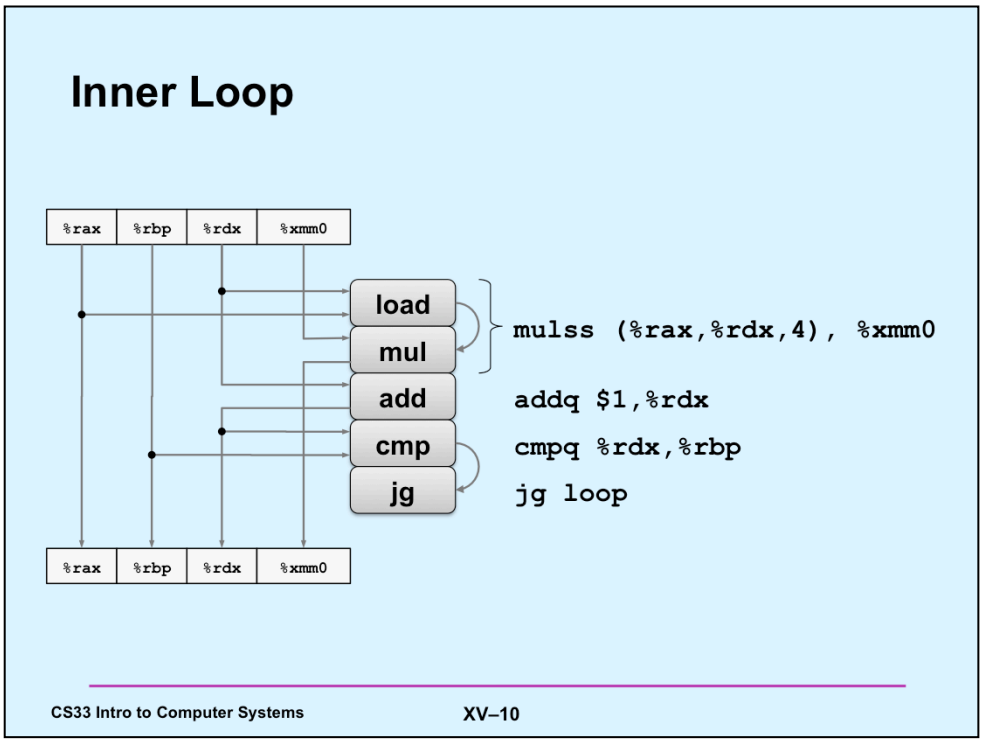
x86-64 Compilation of Combine4

- Inner loop (case: integer multiply)

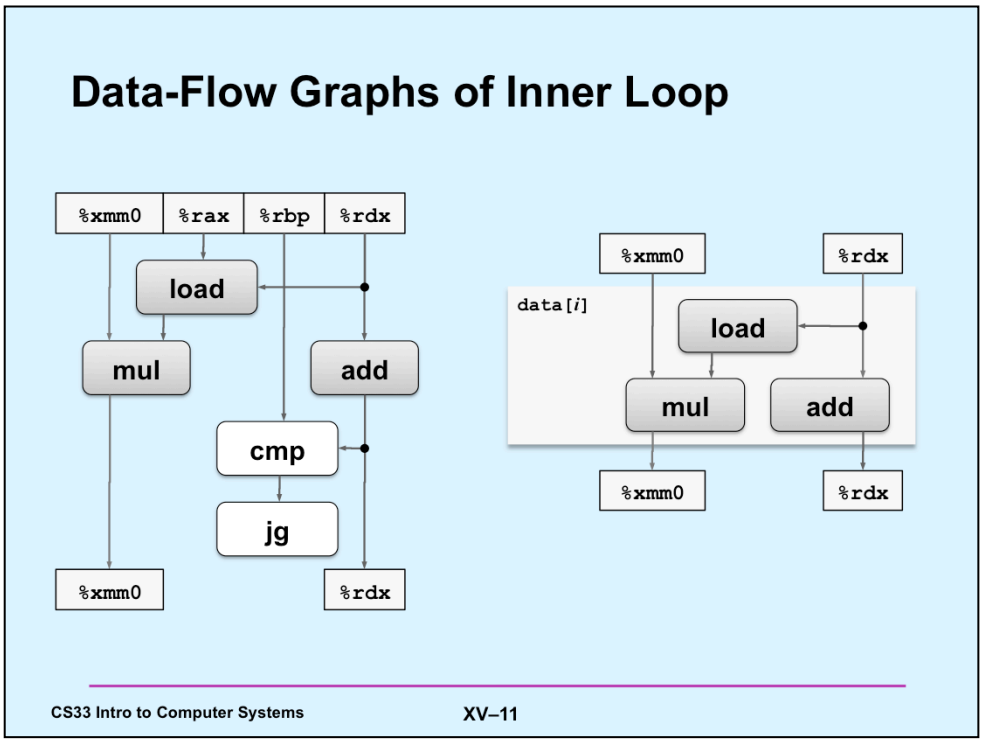
```
.L519:                          # Loop:
  imull  (%rax,%rdx,4), %ecx    # t = t * d[i]
  addq   $1, %rdx               # i++
  cmpq   %rdx, %rbp             # Compare length:i
  jg     .L519                  # If >, goto Loop
```

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
Throughput bound	1.0	1.0	1.0	1.0

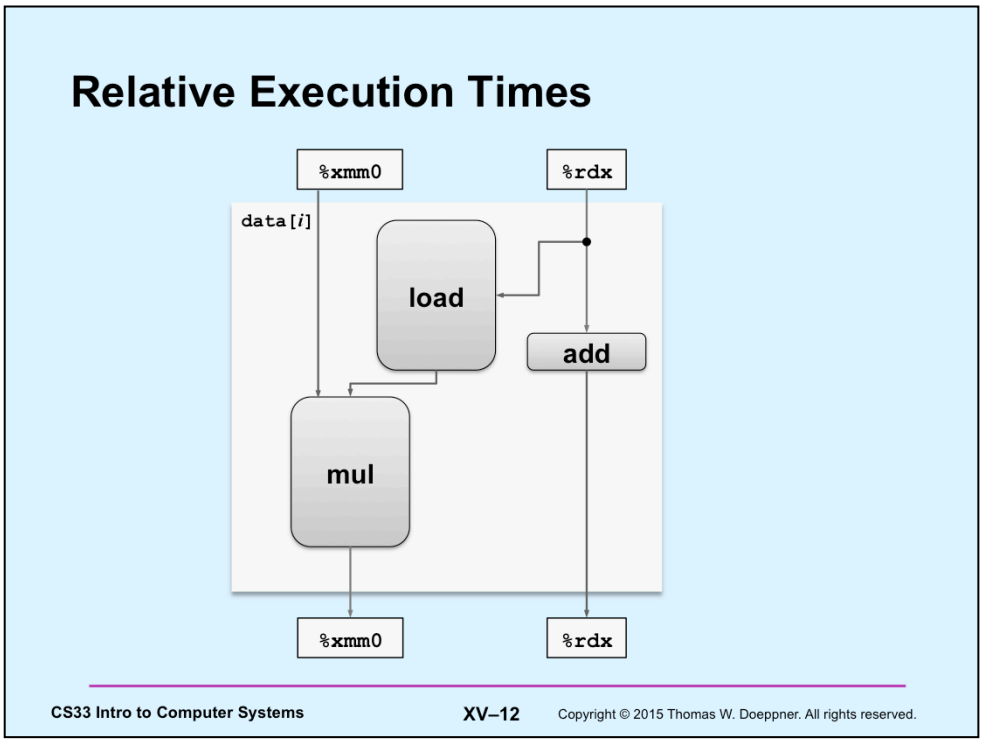
Supplied by CMU.



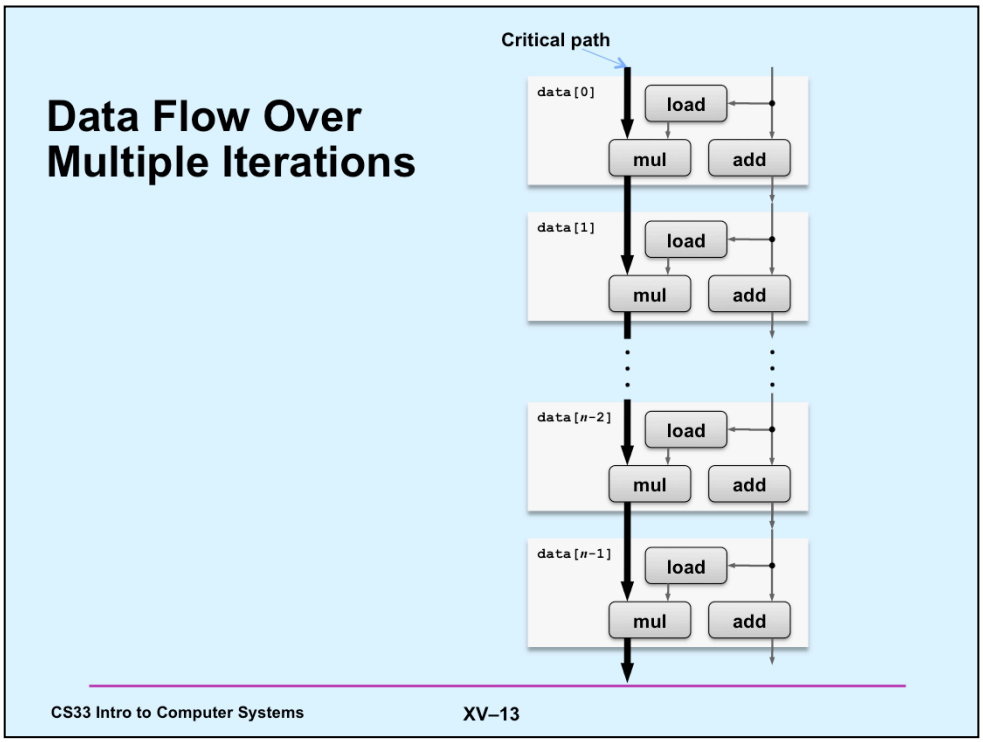
This is Figure 5.13 of Bryant and O'Hallaron. It shows the code for the single-precision floating-point version of our example.



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

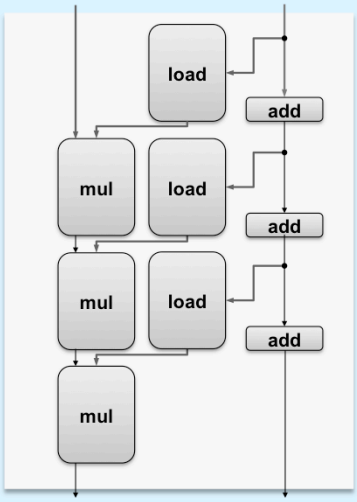


Here we modify the graph of the previous slide to show the relative times required of *mul*, *load*, and *add*.



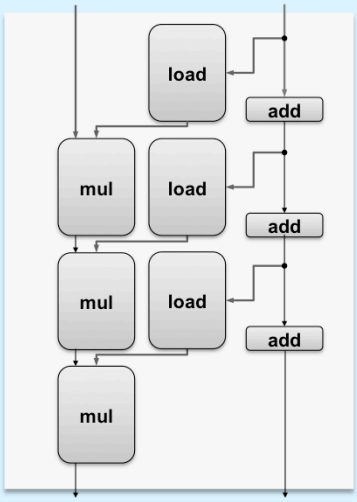
This is Figure 5.15 of Bryant and O'Hallaron.

Pipelined Data-Flow Over Multiple Iterations

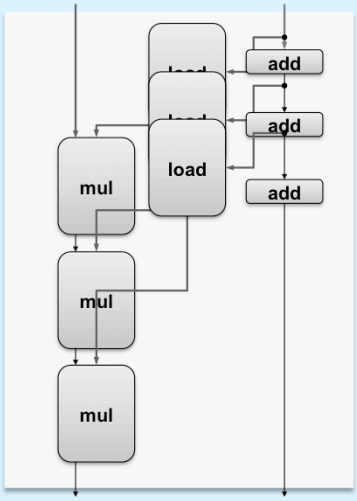


Without pipelining, the data flow would appear as shown in the slide.

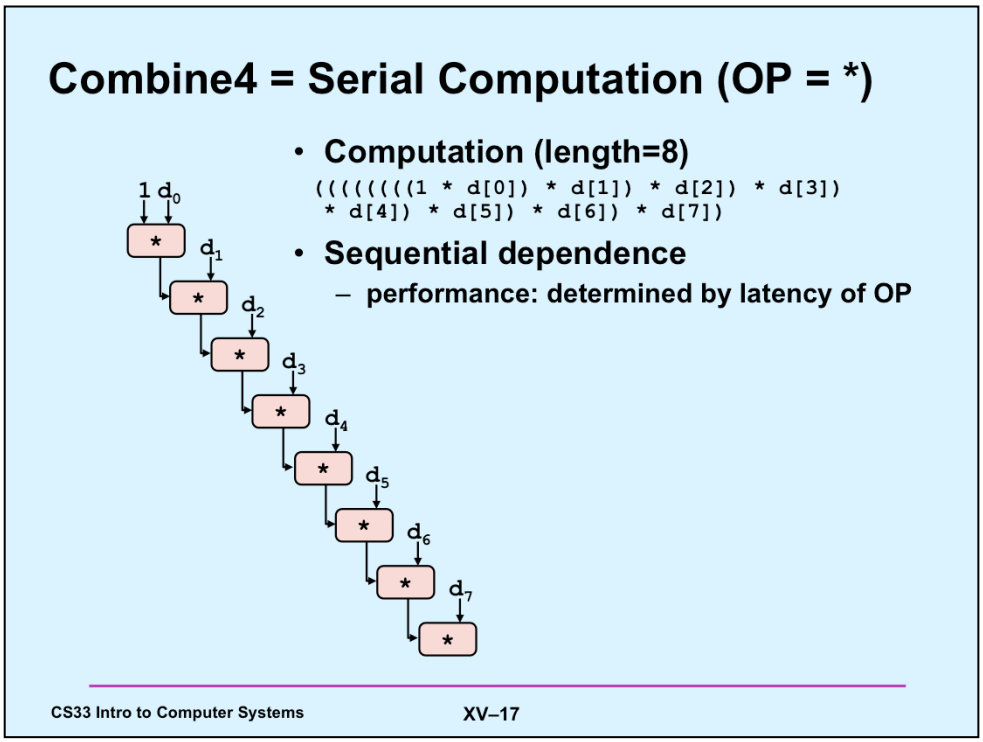
Pipelined Data-Flow Over Multiple Iterations



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.

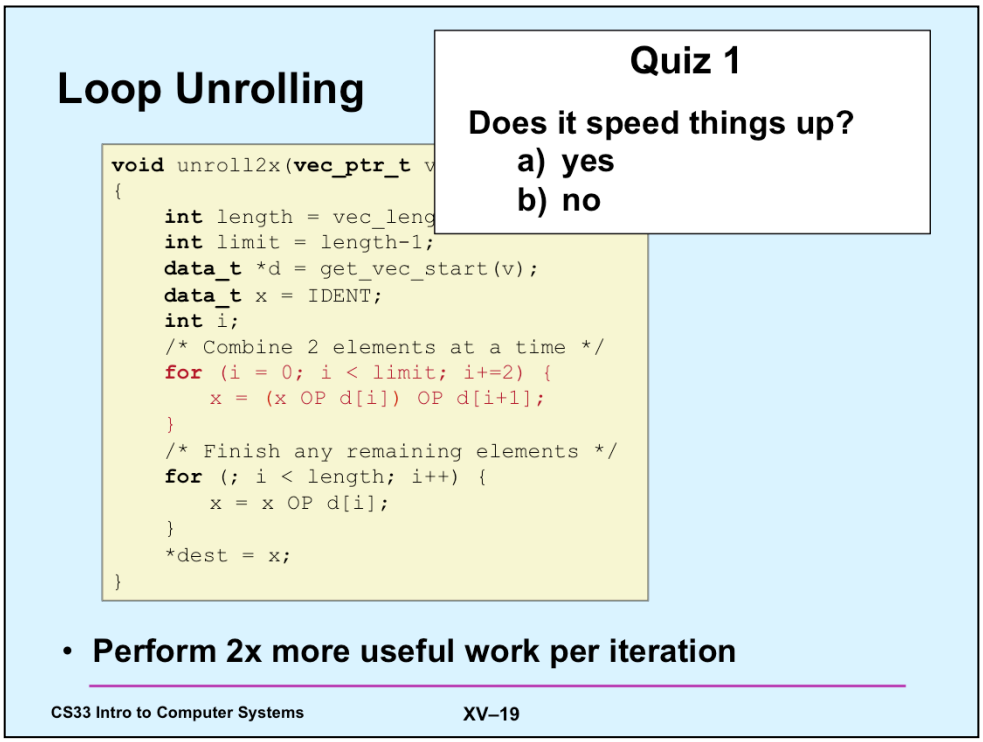
Since the multiplies form the critical path, here we focus only on them.

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	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.

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?***

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

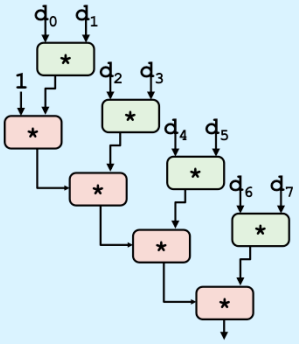
```
x = x OP (d[i] OP d[i+1]);
```

  - 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 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	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

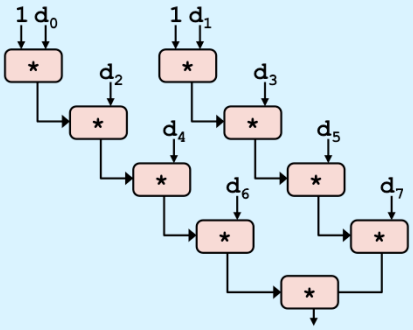
- 2x speedup (over unroll2x) for int *, FP +, FP *
  - breaks sequential dependency in a "cleaner," more obvious way

```
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.

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

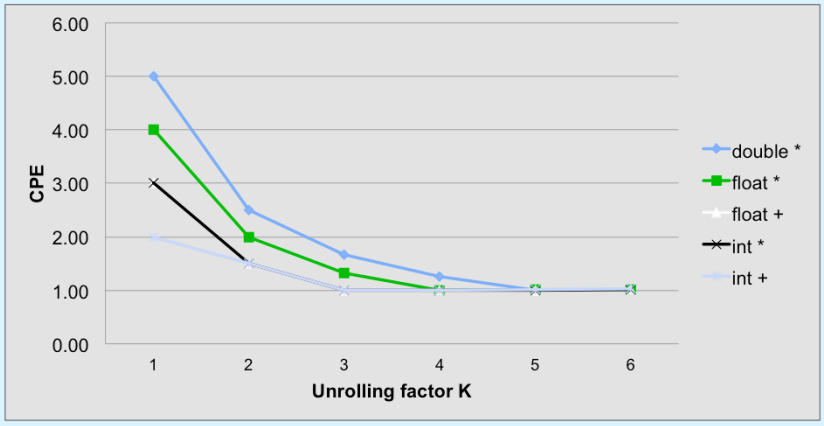
b) false

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

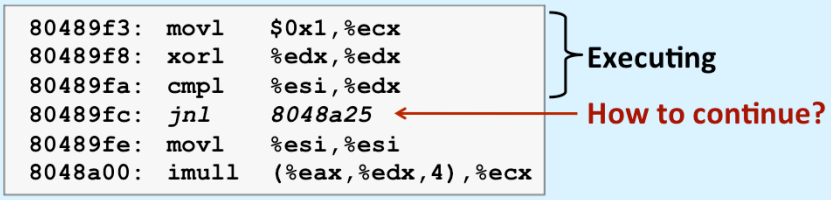
- **Make use of SSE Instructions**
  - parallel operations on multiple data elements

Supplied by CMU.

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

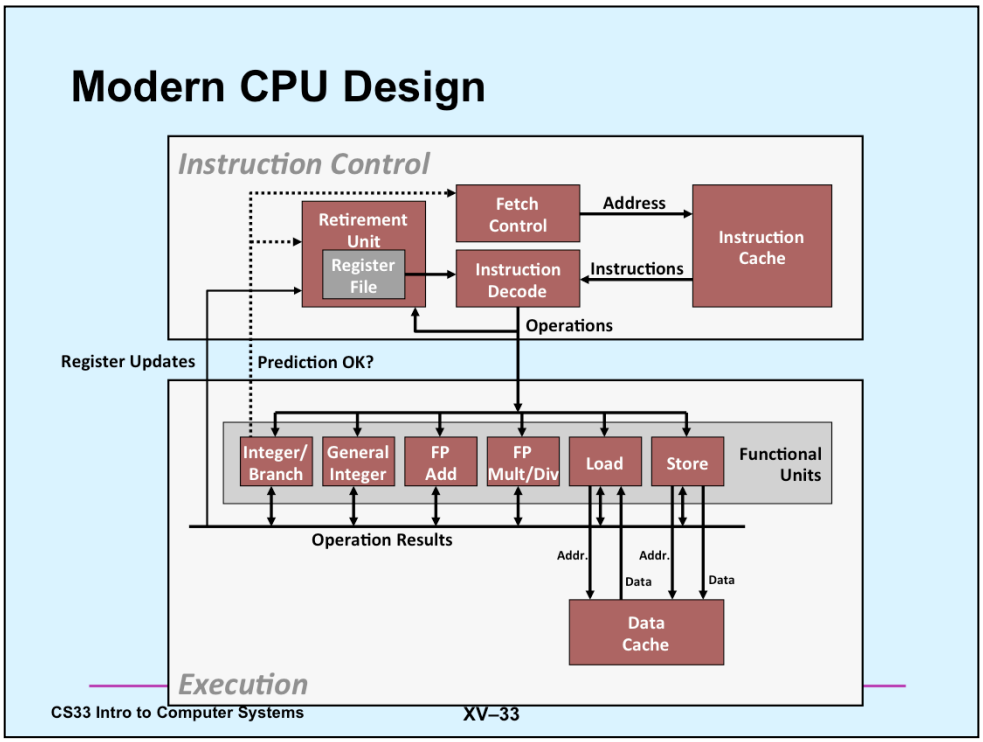
What About Branches?

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

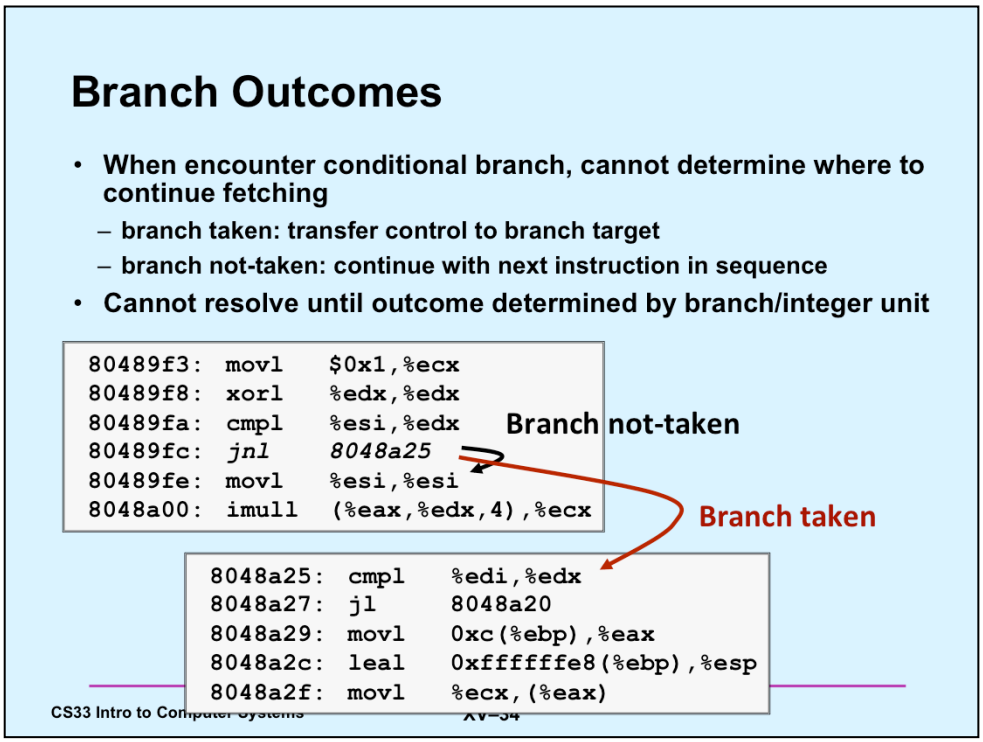


  - when it encounters conditional branch, cannot reliably determine where to continue fetching

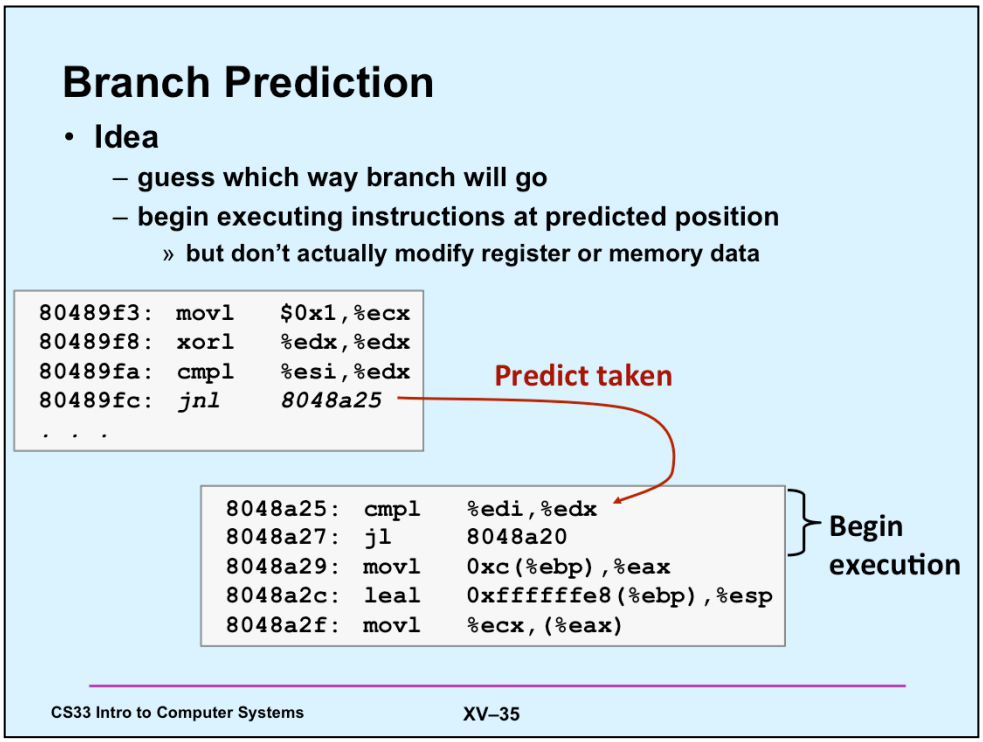
Supplied by CMU.



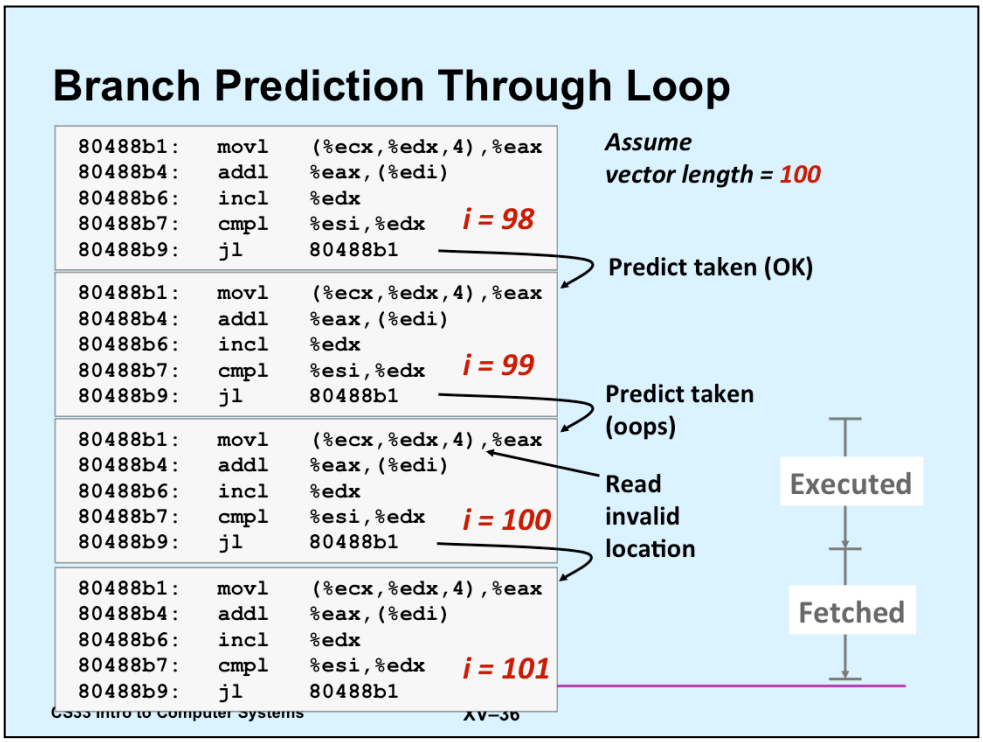
Supplied by CMU.



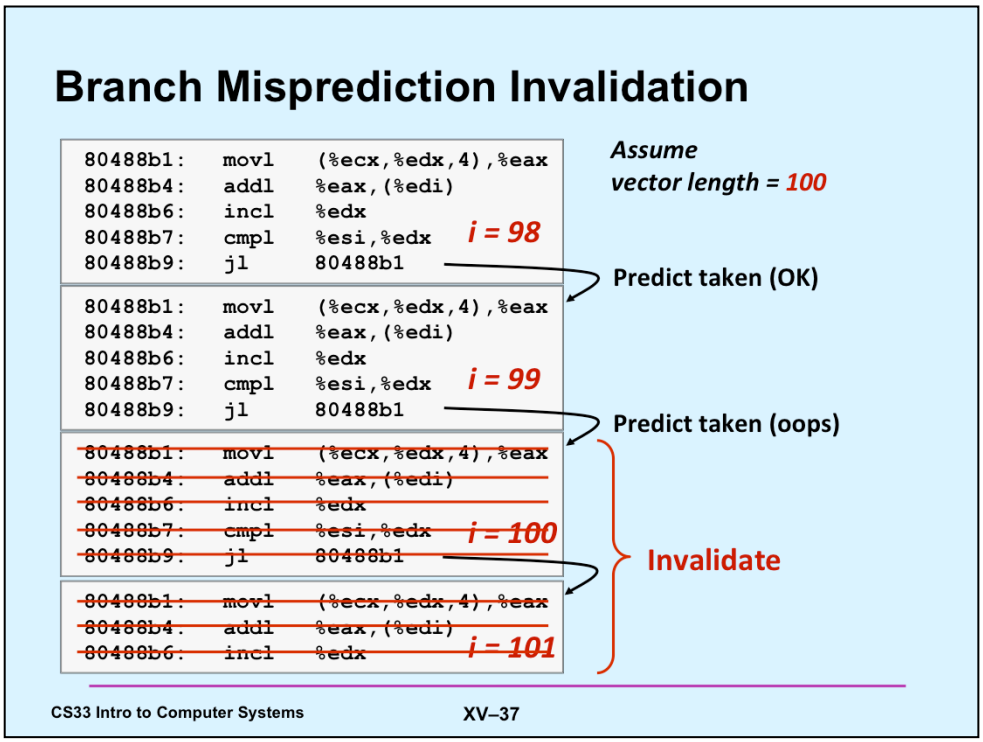
Supplied by CMU.



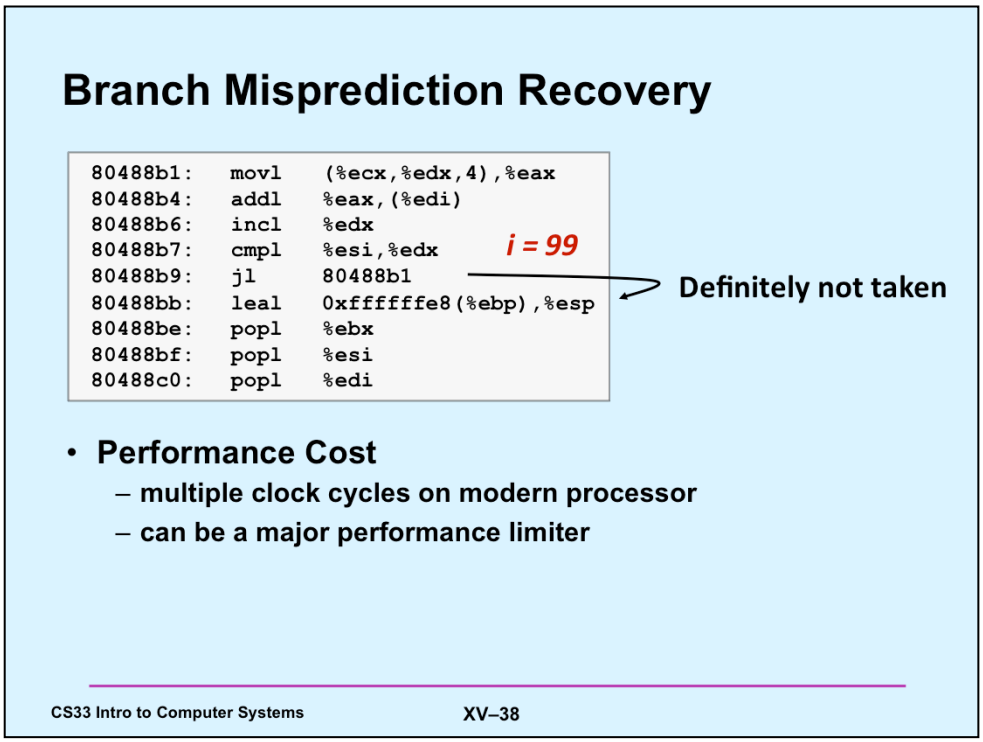
Supplied by CMU.



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. 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.

Latency of Loads

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

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

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

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

- **4 CPE**

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
int main() {
  volatile int dest[100];
  int iter;
  for (iter=0; iter<ITERS; iter++) {
    long i;
    for (i=0; i<100; i++)
      dest[i] = 0;
  }
}
```


This is adapted from Figure 5.32 of the textbook. Due to dependencies on *i*, the stores (into *dest*) cannot be fully pipelined.

Clearing an Array ... Unwound

```
#define ITERS 100000000
int main() {
  volatile int dest[100];
  int iter;
  for (iter=0; iter<ITERS; iter++) {
    long i;
    for (i=0; i<97; i+=4) {
      dest[i] = 0;
      dest[i+1] = 0;
      dest[i+2] = 0;
      dest[i+3] = 0;
    }
  }
}
```

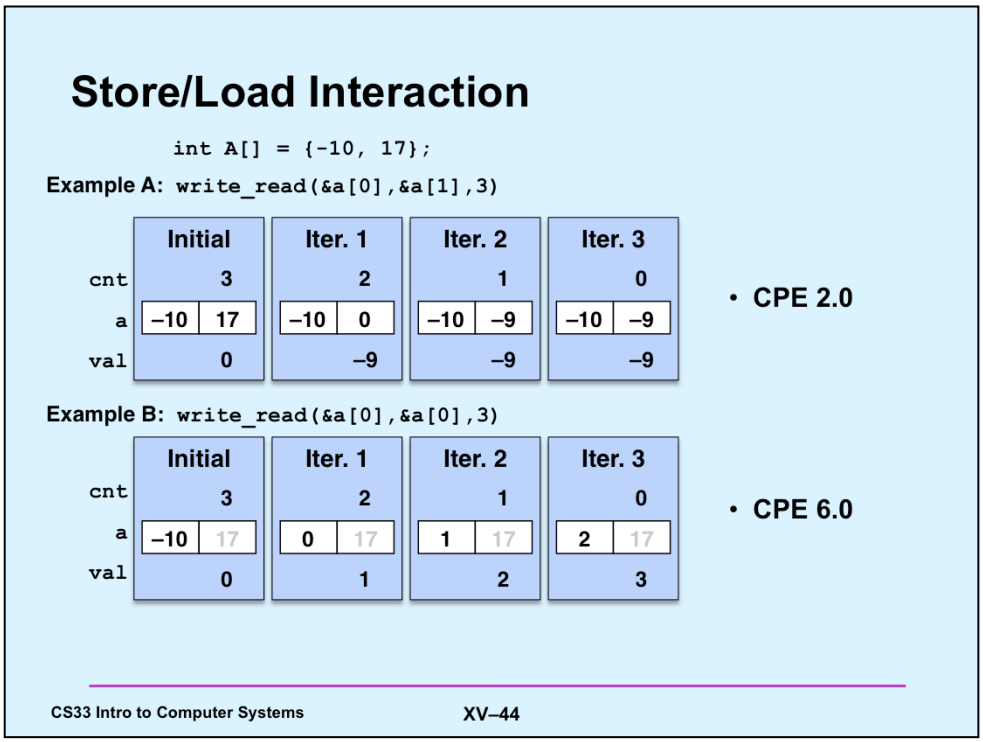

This is adapted from Figure 5.32 of the textbook. By unrolling the loop four times, the stores can be pipelined and thus a store (which takes four cycles to complete) can be started every cycle.

Store/Load Interaction

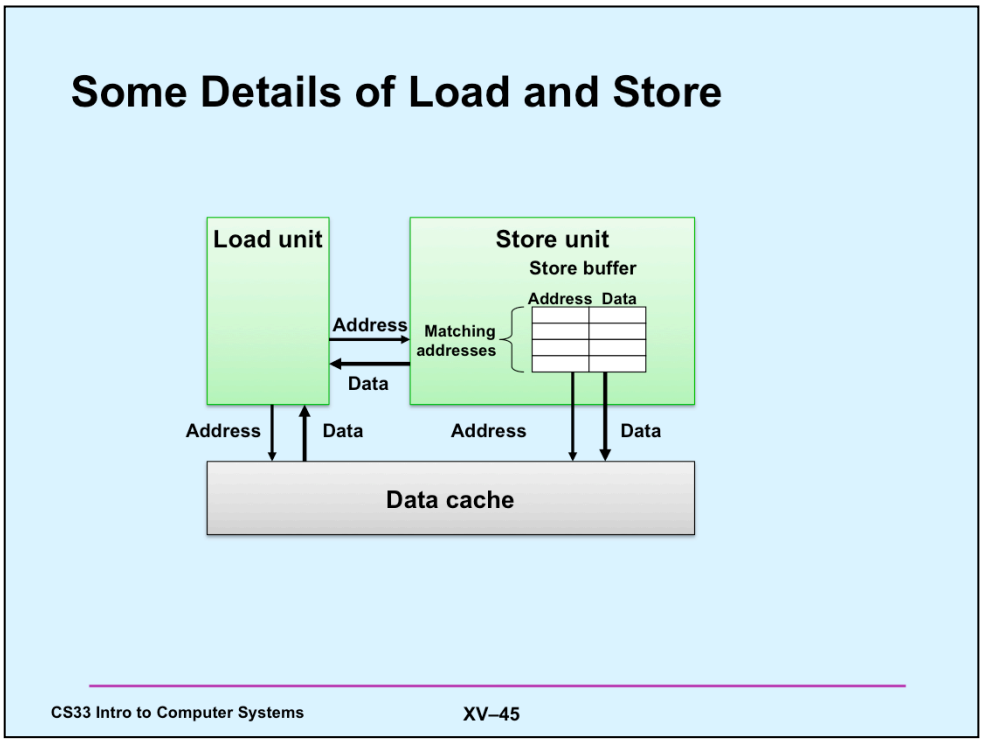
```
void write_read(int *src, int *dest, int n) {
  int cnt = n;
  int val = 0;

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

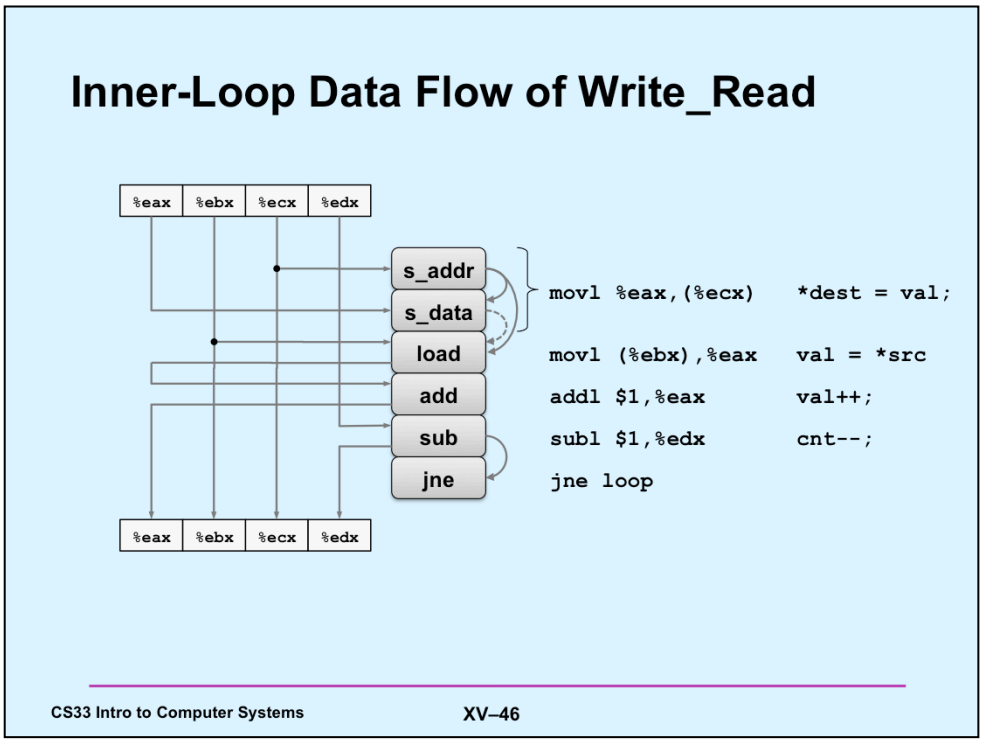

This code is from the textbook.



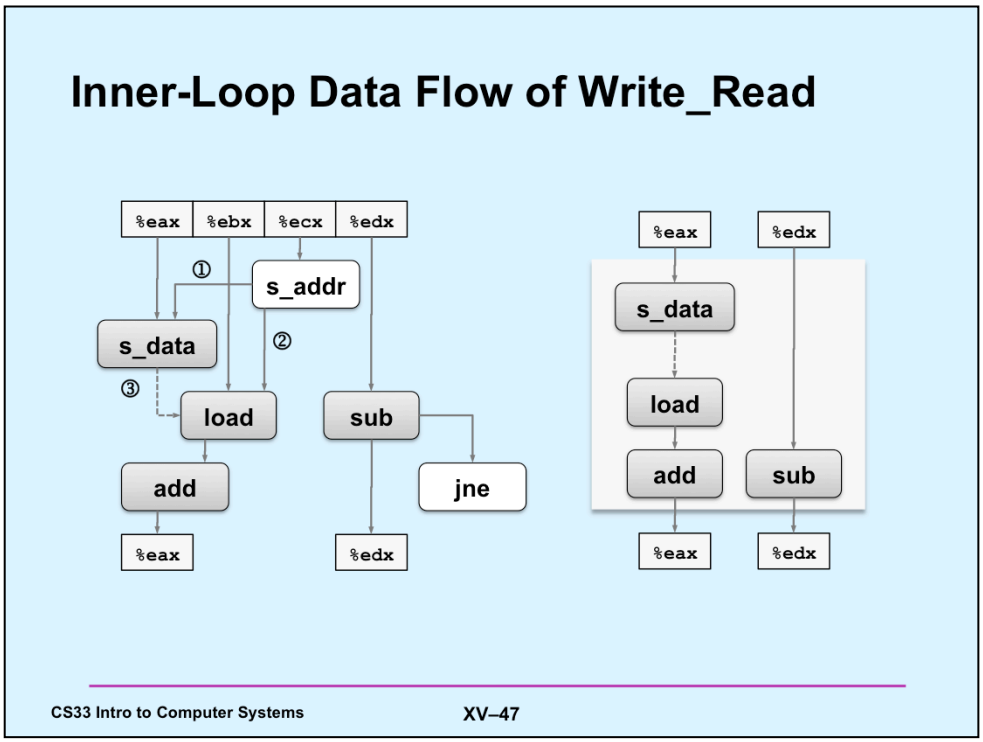
This is Figure 5.33 of the textbook. Performance depends upon whether *src* and *dest* are the same location.



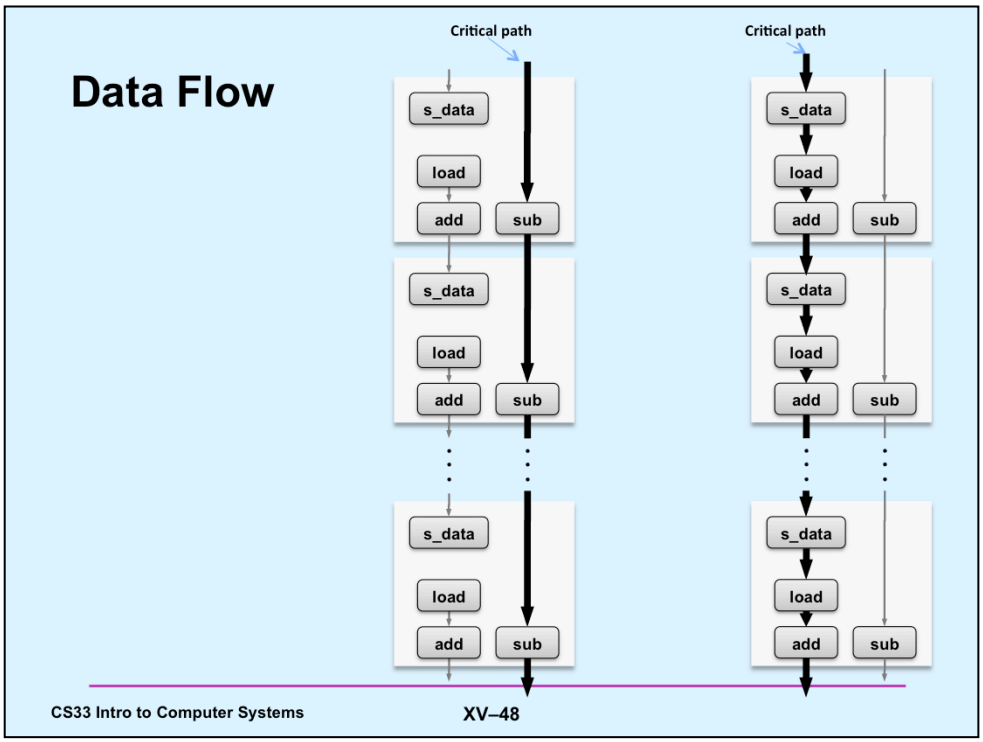
This is Figure 5.34 of the textbook.



This is Figure 5.35 of the textbook.



This is Figure 5.36 of the textbook.



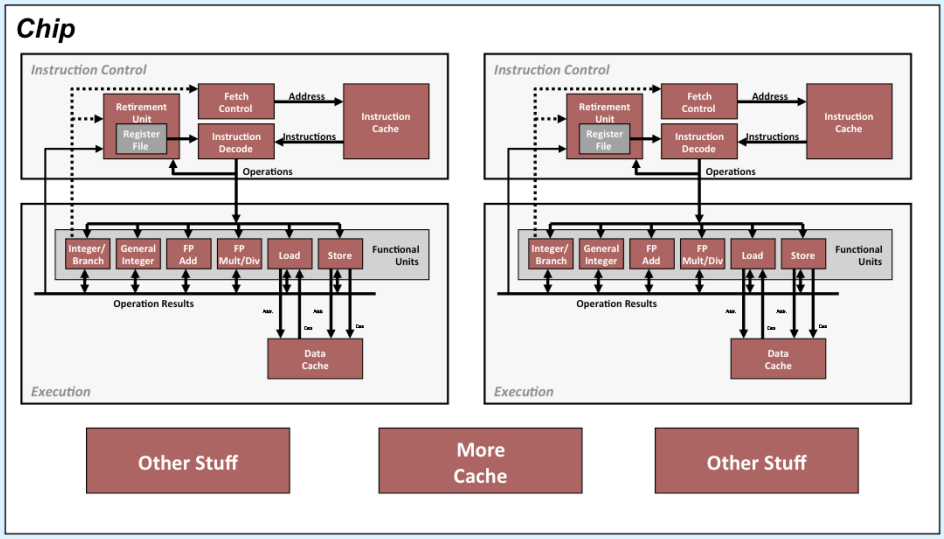
This is adapted from Figure 5.37 of the textbook.

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.

Multiple Cores



Hyper Threading