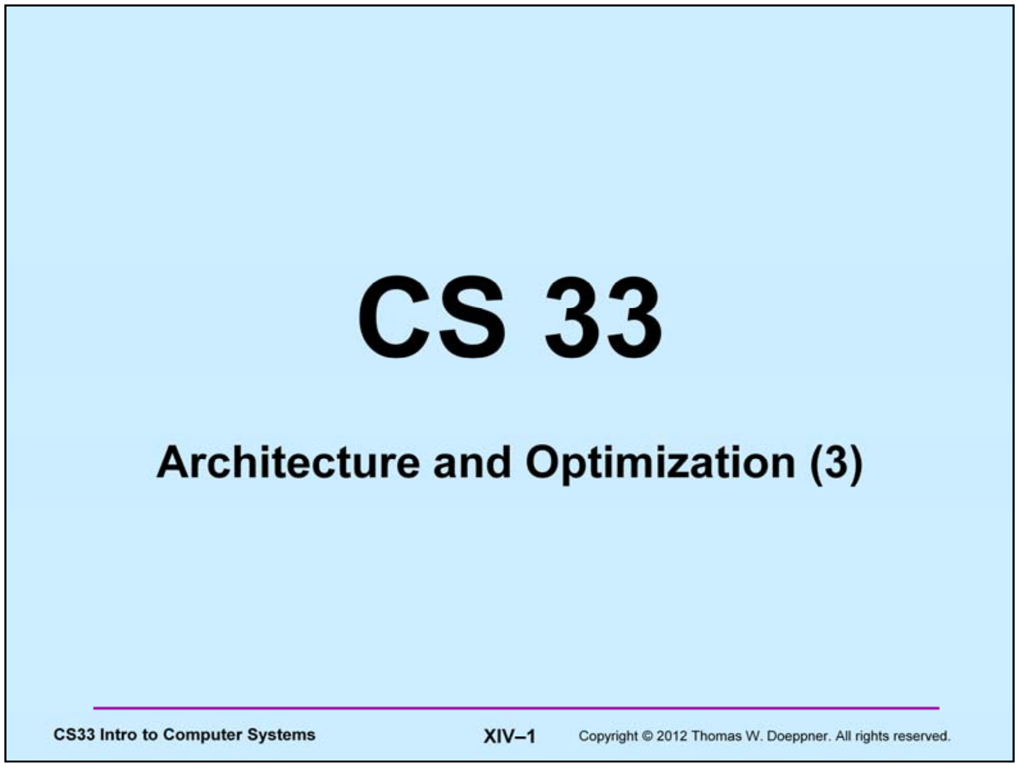
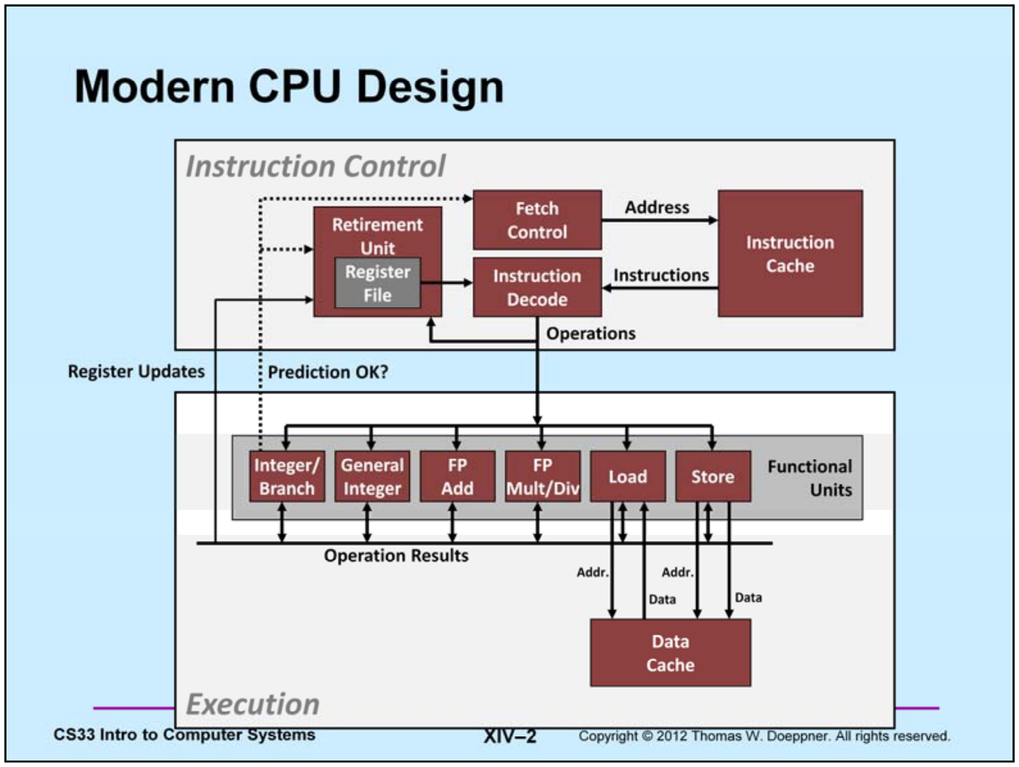


CS 33

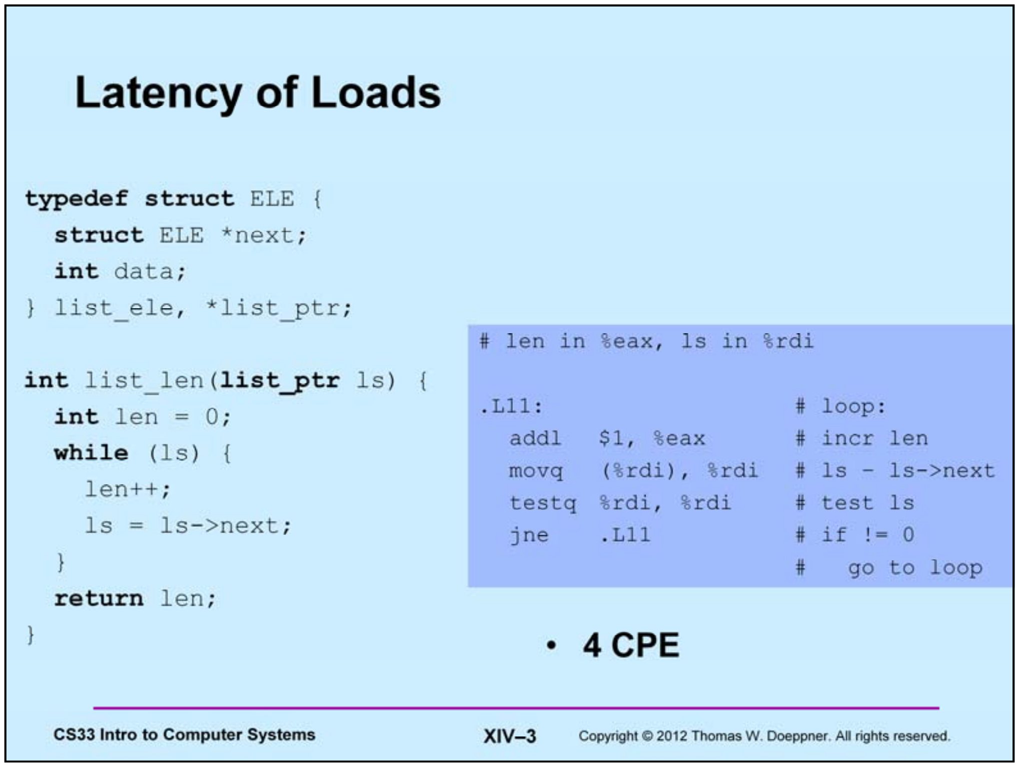
Architecture and Optimization (3)

Some 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.



Modern CPU Design

Supplied by CMU.

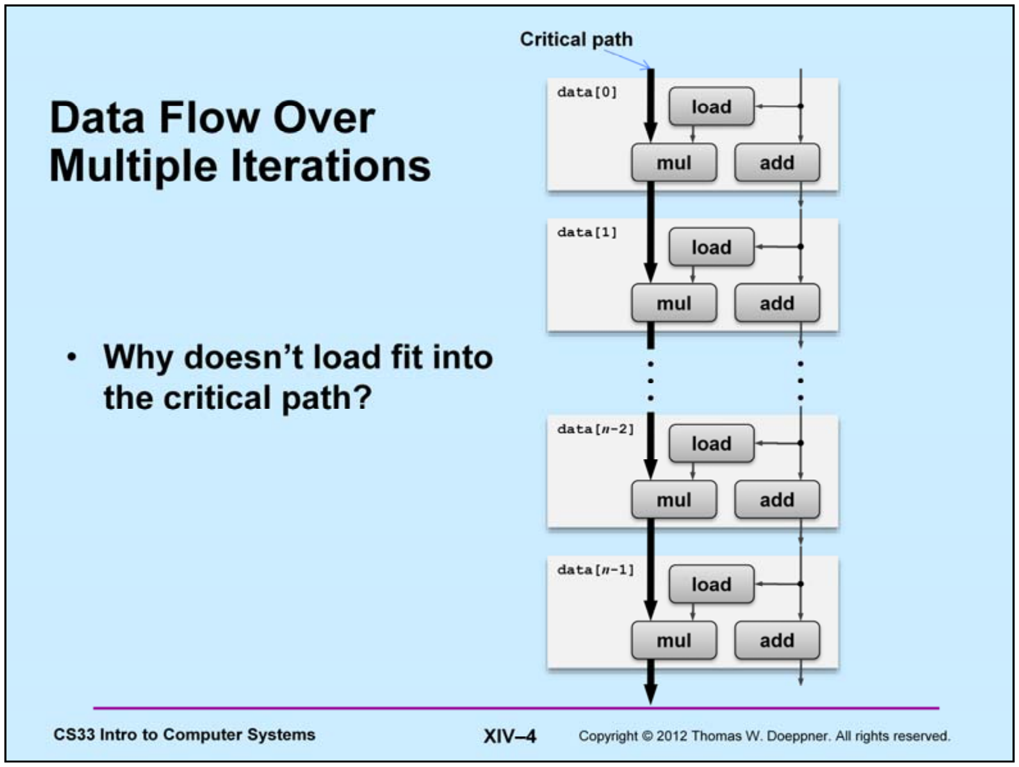

Latency of Loads

```
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
```

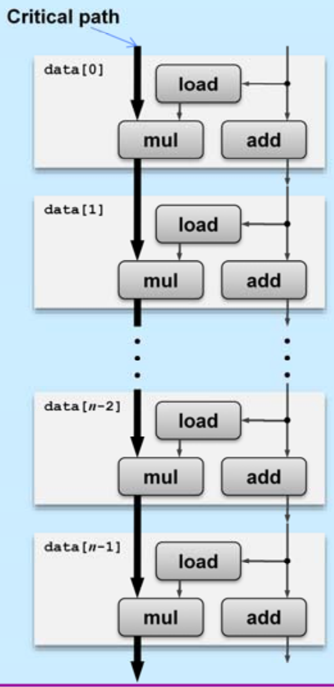
- **4 CPE**

This example is from the textbook (Figure 5.31). The point is that loads (fetching data from memory) have a latency of 4 cycles.



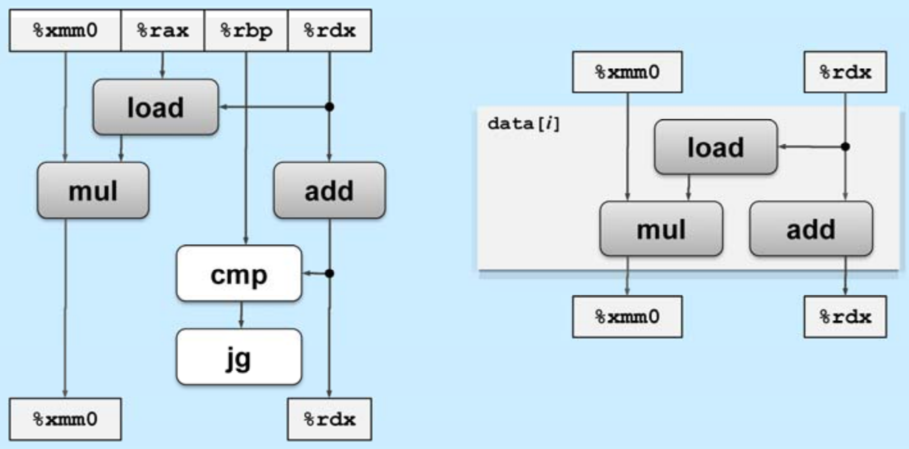
Data Flow Over Multiple Iterations

- **Why doesn't load fit into the critical path?**

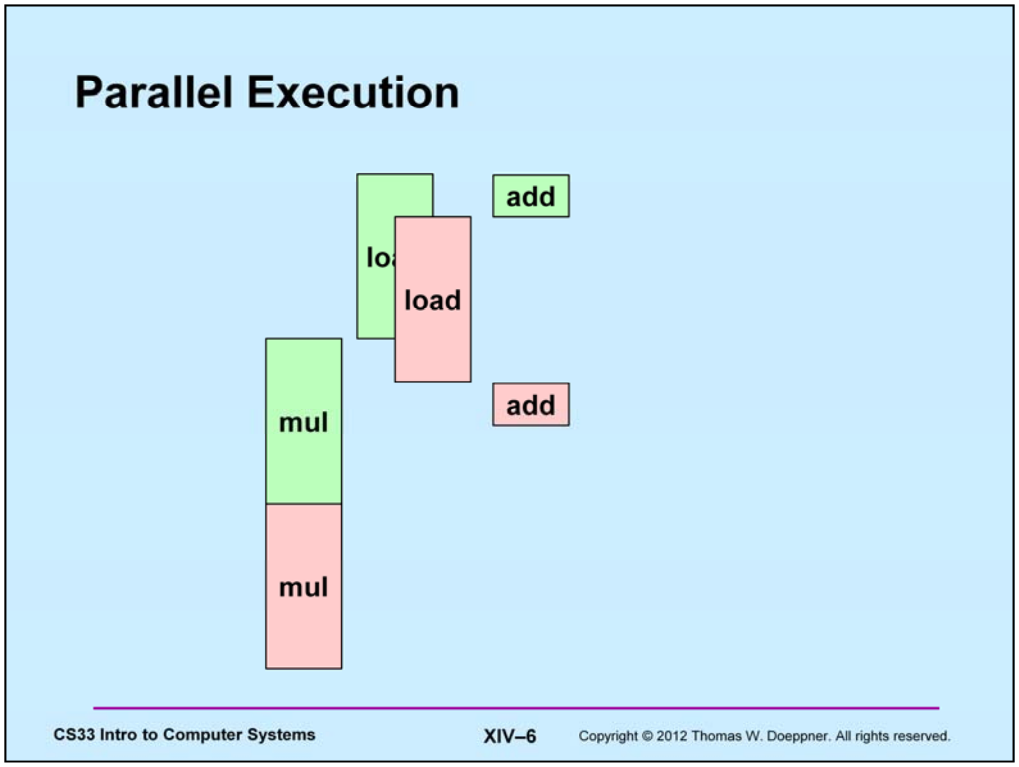


This is Figure 5.15 of Bryant and O'Hallaron.

Data-Flow Graphs of Inner Loop



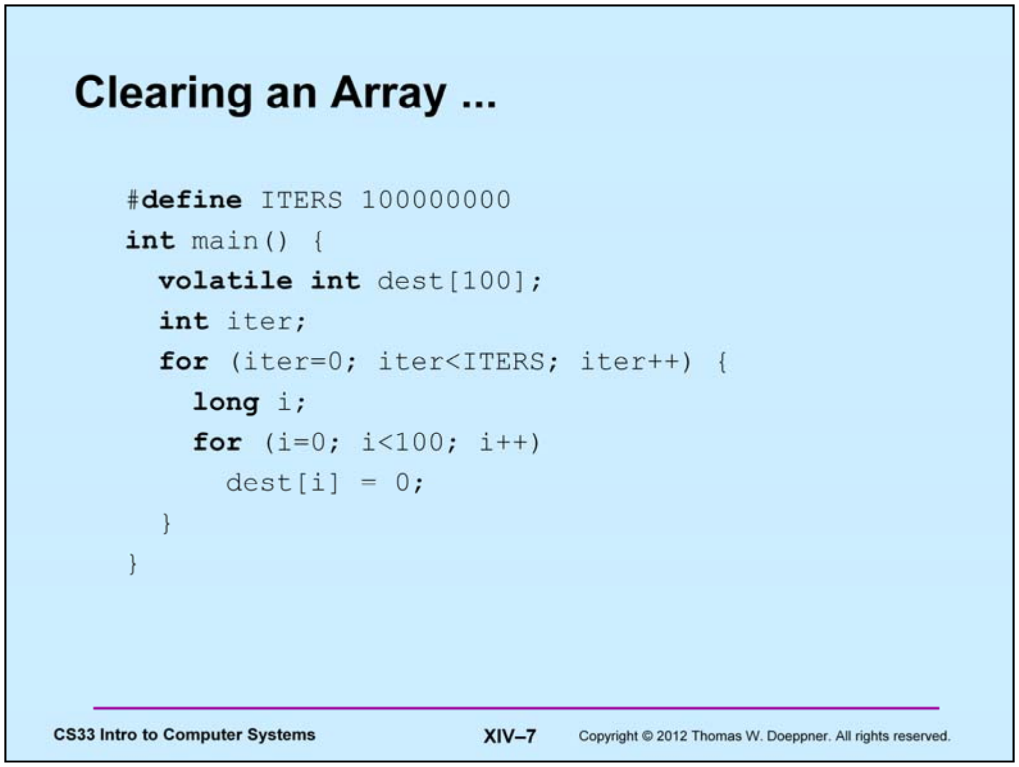
These are Figures 5.14 a and b of Bryant and O’Hallaron.



Parallel Execution

Here we look in detail at the timing requirements of the example of the previous slides. In the current slide, the length of a box indicates how much latency the operation requires: *mul* and *load* require 4 cycles, *add* requires 1. The first operations that can execute are the *load* and the *add*. Once the *load* completes, the *mul* operation may start, since it's waiting on the value loaded. However, note that the next instance of load may start as soon as add has updated %rdx. Since loads are pipelined, it thus may start execution one cycle after the first *load* started. The next instance of *add* may start as soon as all users of %rdx (in this case, the *load* and *cmp* instructions) are done with %rdx. Thus it certainly may execute anytime after the second *load* completes.

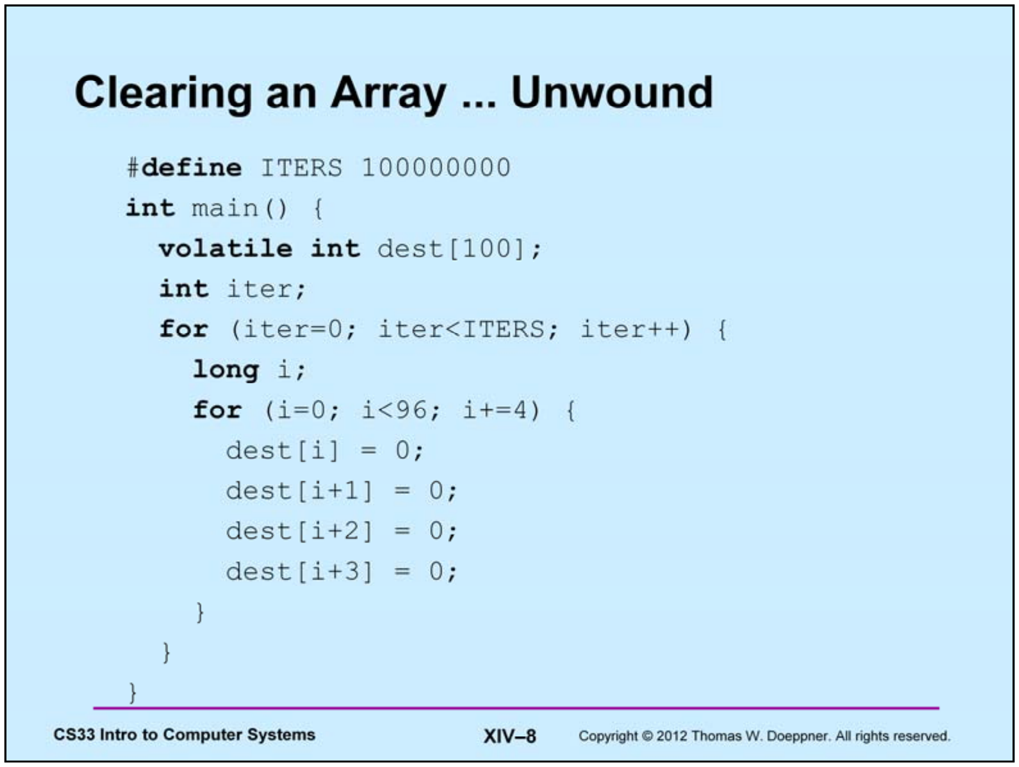
The second *mul* may not start until after the first *mul* has produced its result (in %xmm0) and after *load* has loaded the multiplier. But since the second *load* produced the multiplier for the second *mul* shortly after the first *mul* began, the second *mul* must wait only until the first *mul* completes.



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;
  }
}
```

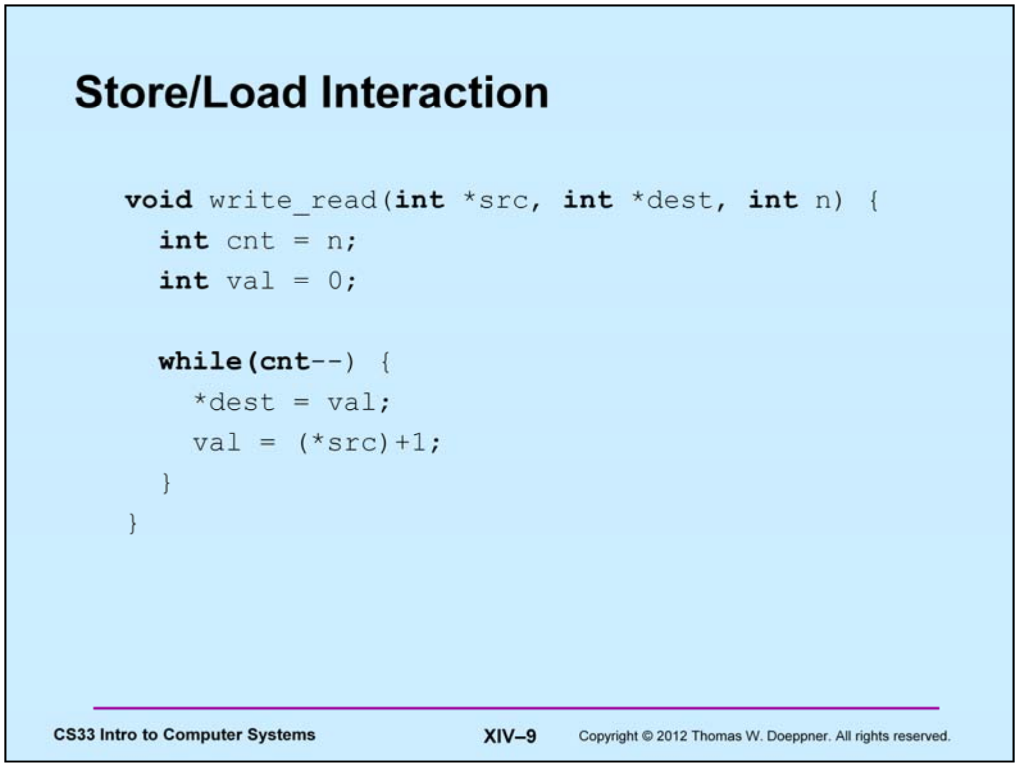
This is adapted from Figure 5.32 from the textbook.



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<96; i+=4) {
      dest[i] = 0;
      dest[i+1] = 0;
      dest[i+2] = 0;
      dest[i+3] = 0;
    }
  }
}
```

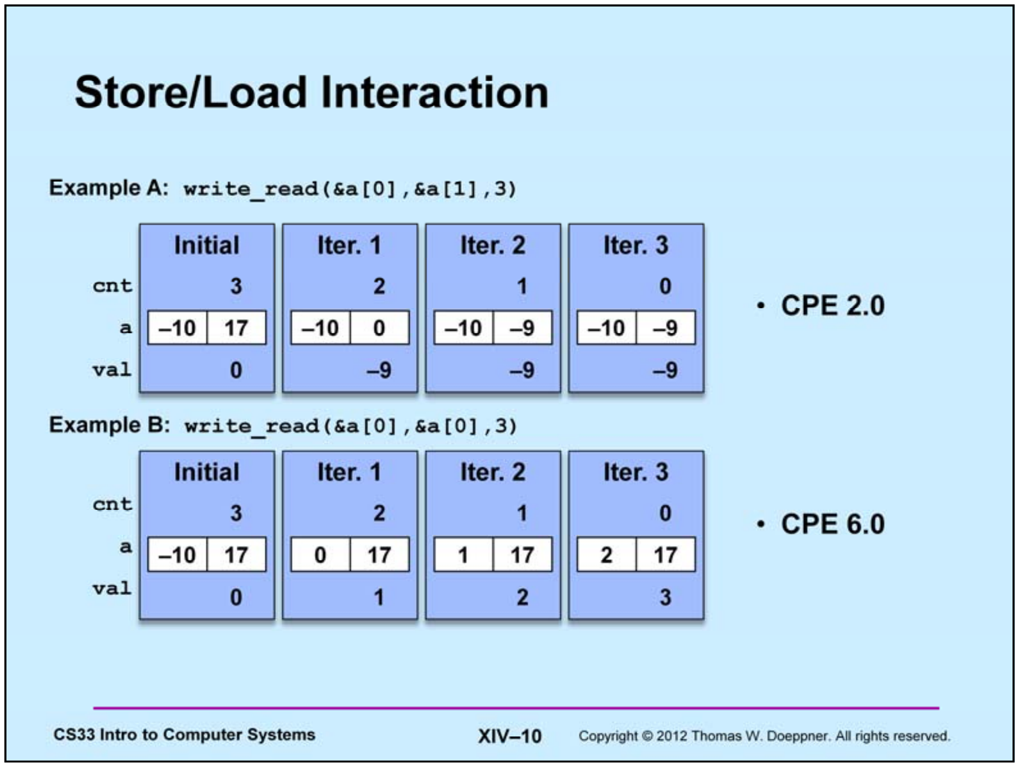
This is adapted from Figure 5.32 from the textbook.



Store/Load Interaction

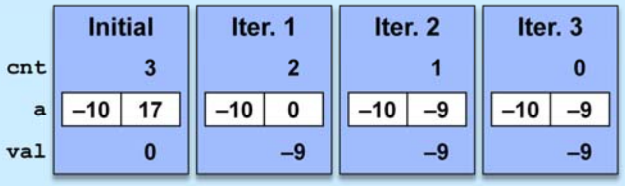
```
void write_read(int *src, int *dest, int n) {
  int cnt = n;
  int val = 0;

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

This code is from the textbook.



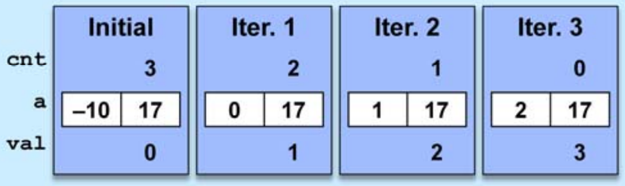
Store/Load Interaction

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



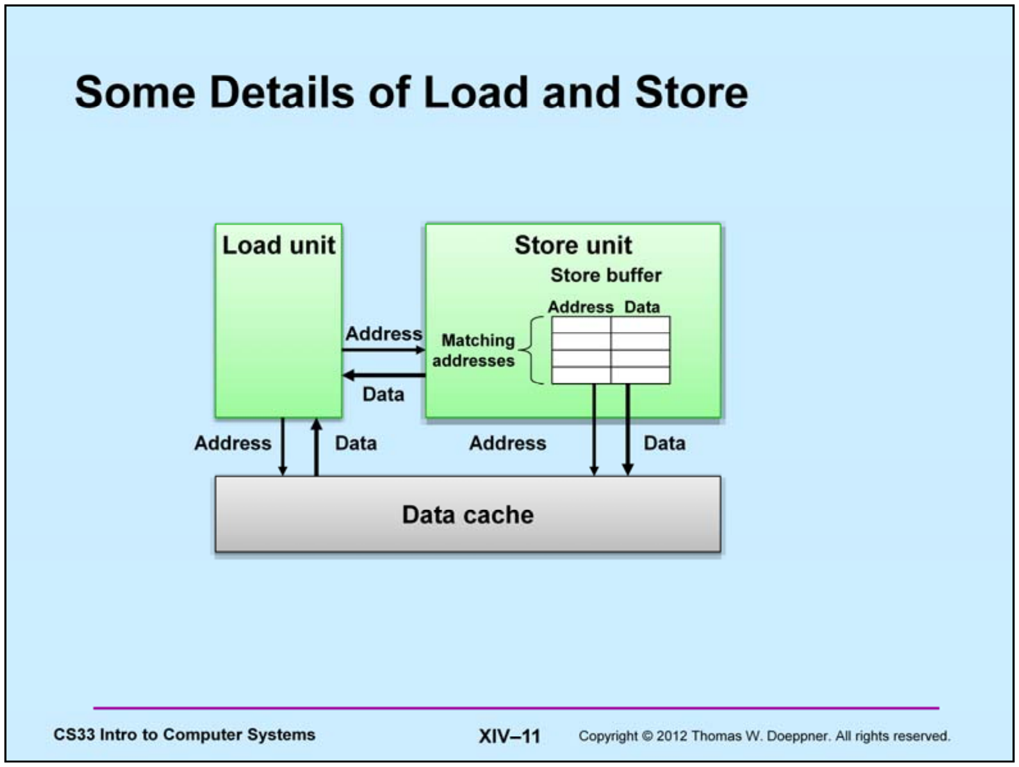
- CPE 2.0

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

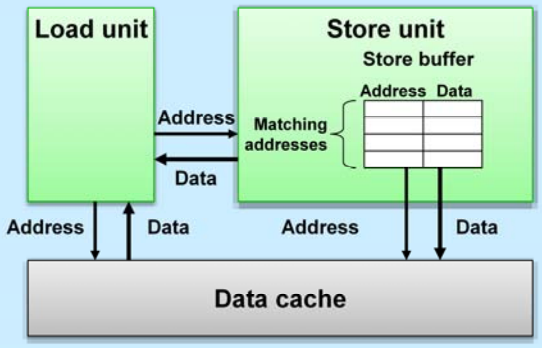


- CPE 6.0

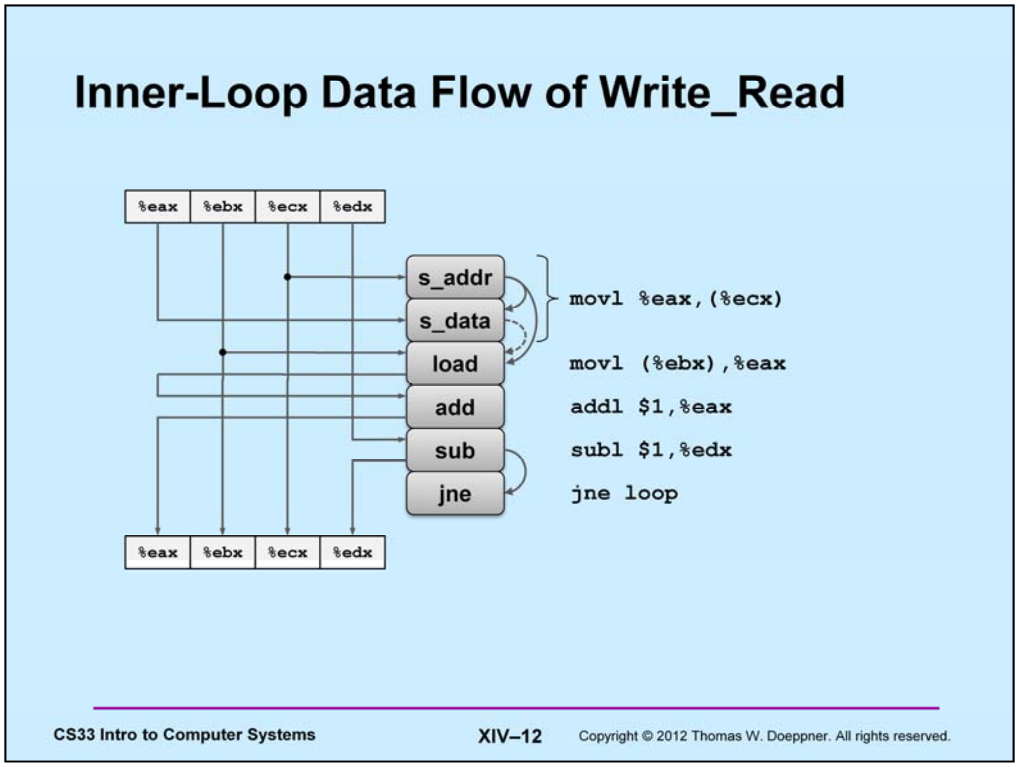
This is Figure 5.33 from the textbook.



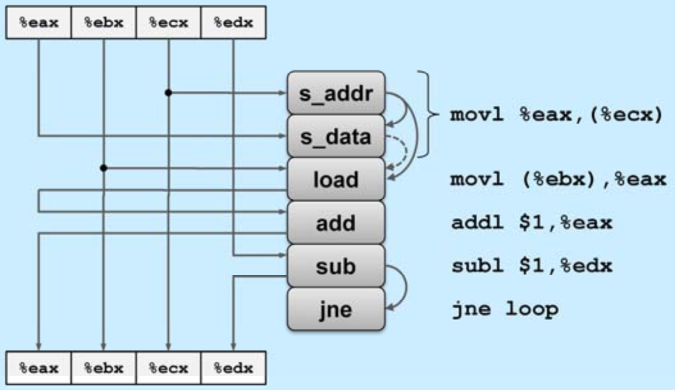
Some Details of Load and Store



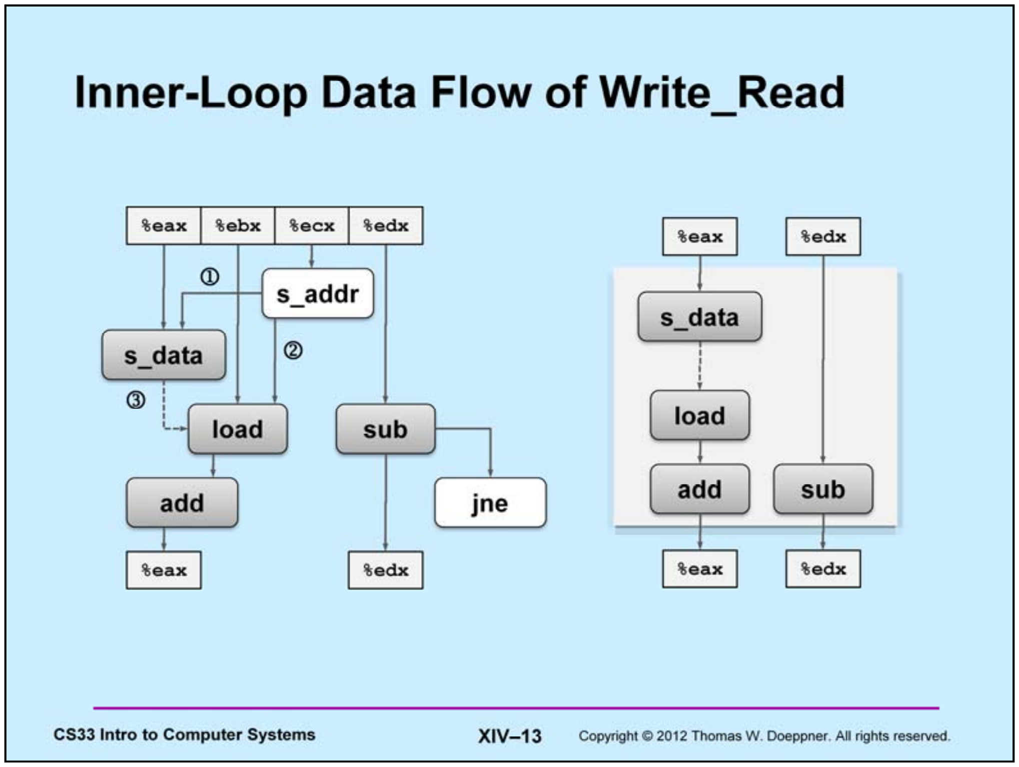
This is Figure 5.34 from the textbook.



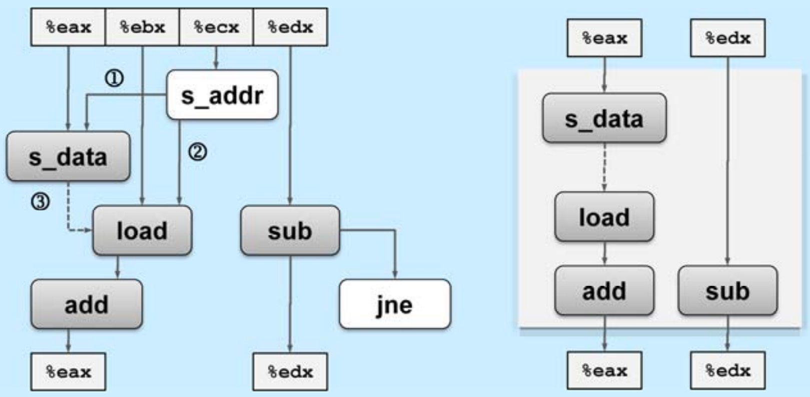
Inner-Loop Data Flow of Write_Read



This is Figure 5.35 from the textbook.

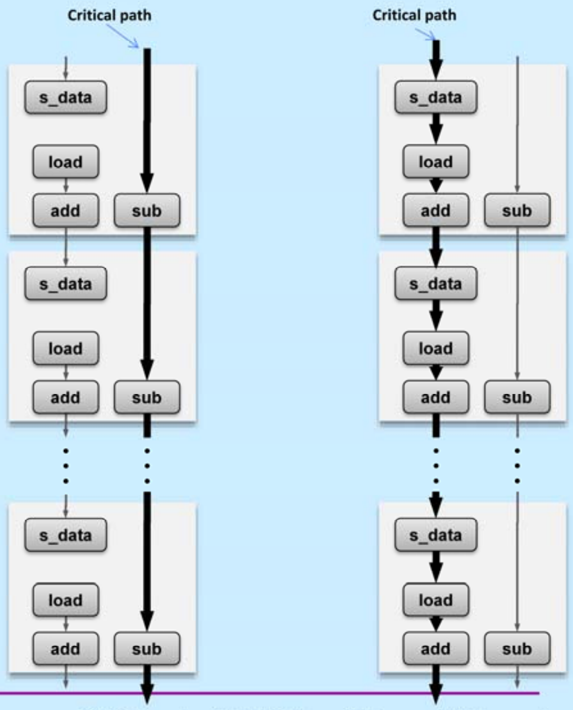


Inner-Loop Data Flow of Write_Read



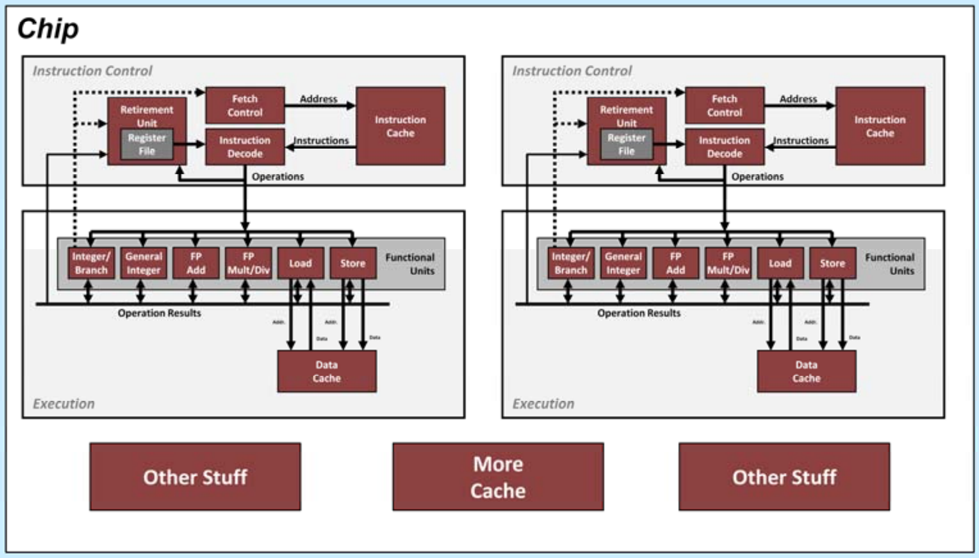
This is Figure 5.36 from the textbook.

Data Flow

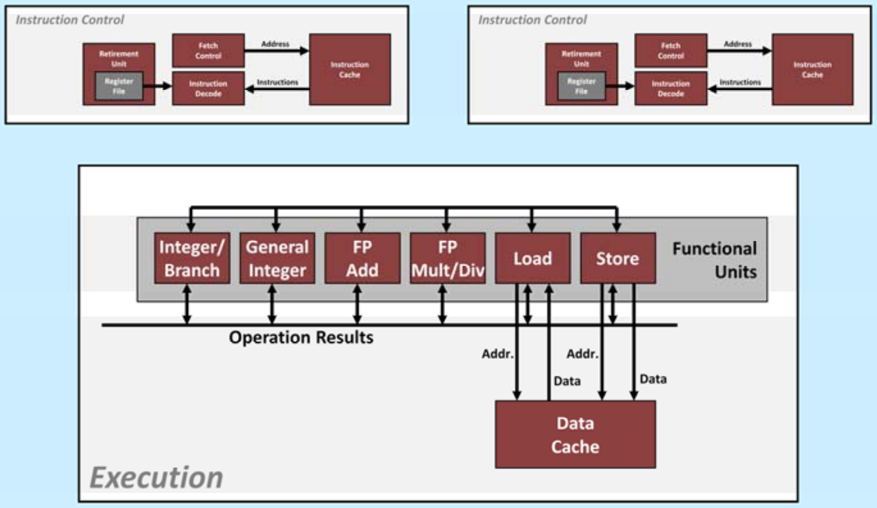


This is adapted from Figure 5.37 from the textbook.

Multiple Cores



Hyper Threading


Instruction Control
Retirement Unit
Register File
Fetch Control
Address
Instruction Cache
Instruction Decode
Instructions
Instruction Control
Retirement Unit
Register File
Fetch Control
Address
Instruction Cache
Instruction Decode
Instructions
Integer/ Branch
General Integer
FP Add
FP Mult/Div
Load
Store
Functional Units
Operation Results
Addr.
Addr.
Data
Data
Data Cache
Execution