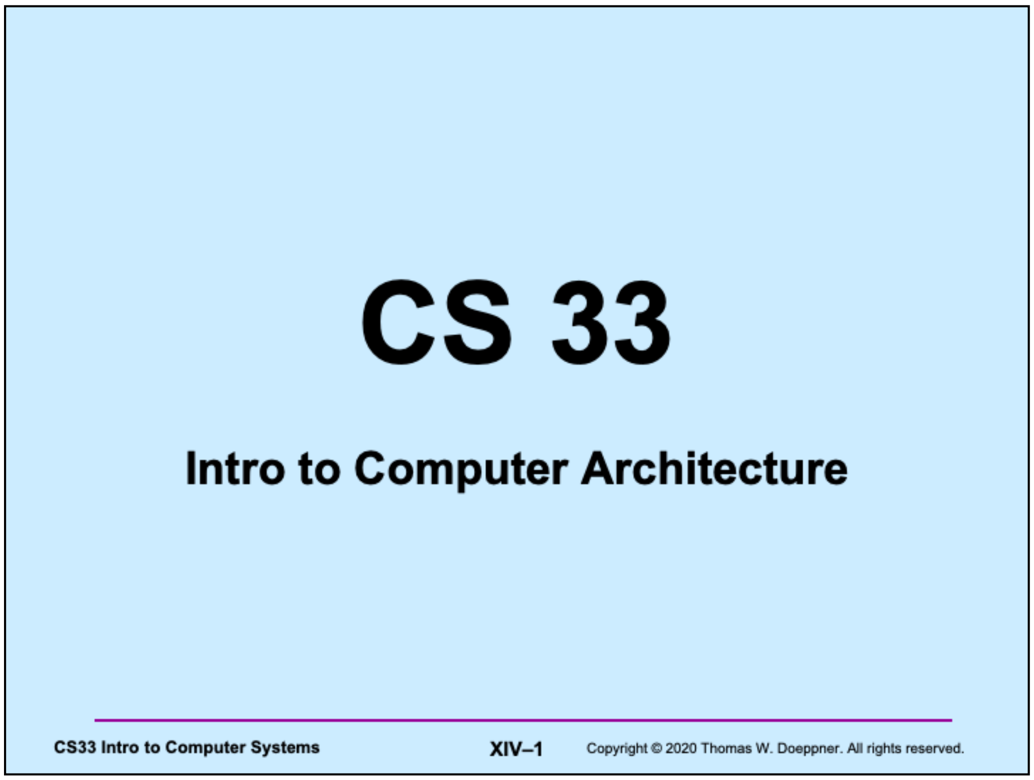


CS 33

Intro to Computer Architecture

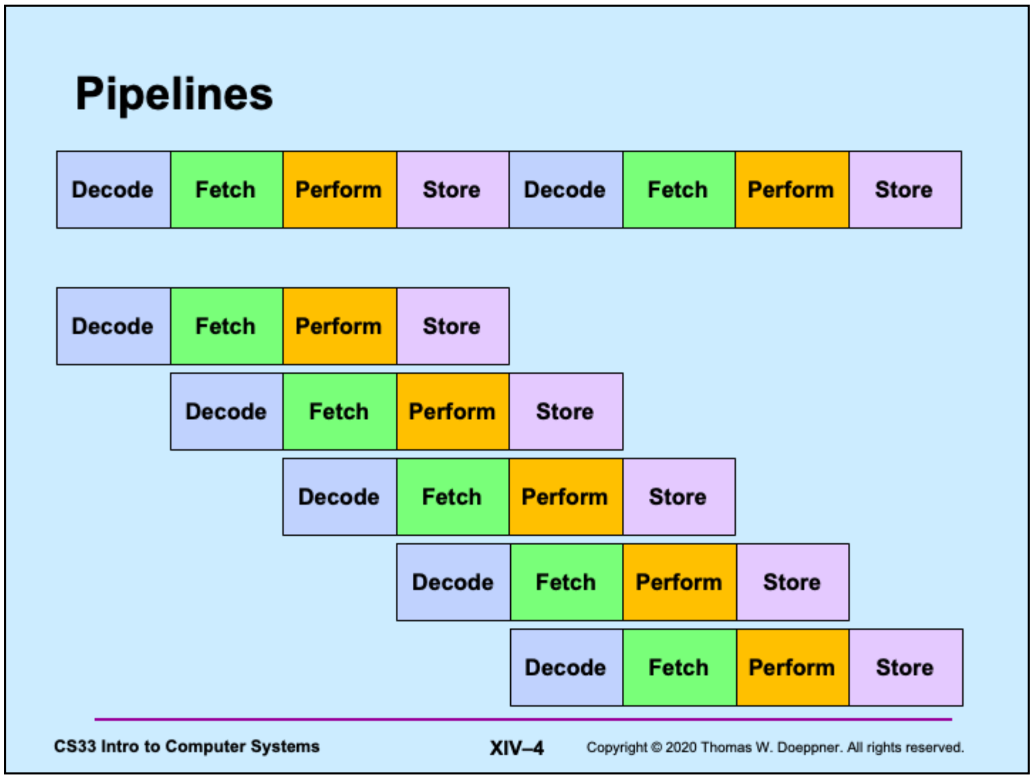
A few of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in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

Simplistic View of Processor

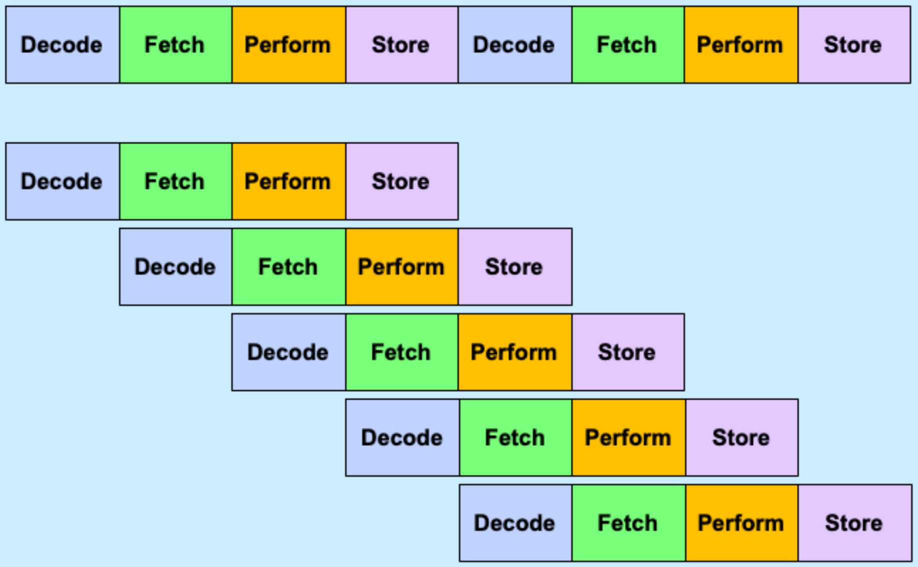
```
while (true) {
  instruction = mem[rip];
  execute(instruction);
}
```

Some Details ...

```
void execute(instruction_t instruction) {
  decode(instruction, &opcode, &operands);
  fetch(operands, &in_operands);
  perform(opcode, in_operands, &out_operands);
  store(out_operands);
}
```



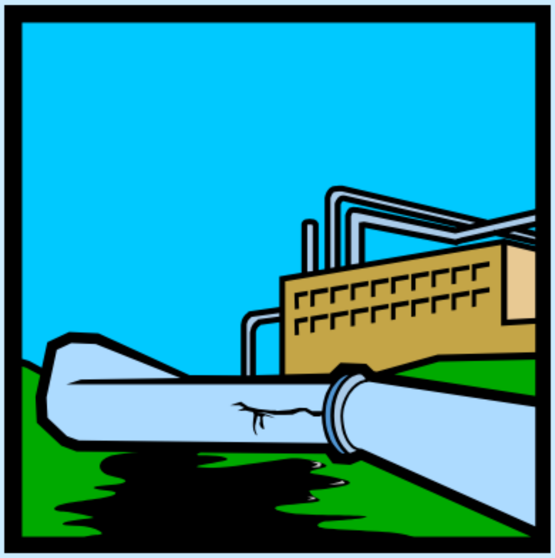
Pipelines



Analysis

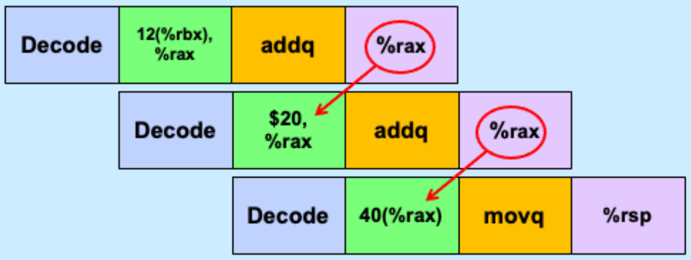
- Not pipelined
  - each instruction takes, say, 3.2 nanoseconds
    - » 3.2 ns latency
  - 312.5 million instructions/second (MIPS)
- Pipelined
  - each instruction still takes 3.2 ns
    - » latency still 3.2 ns
  - an instruction completes every .8 ns
    - » 1.25 billion instructions/second (GIPS) throughput

Hazards ...

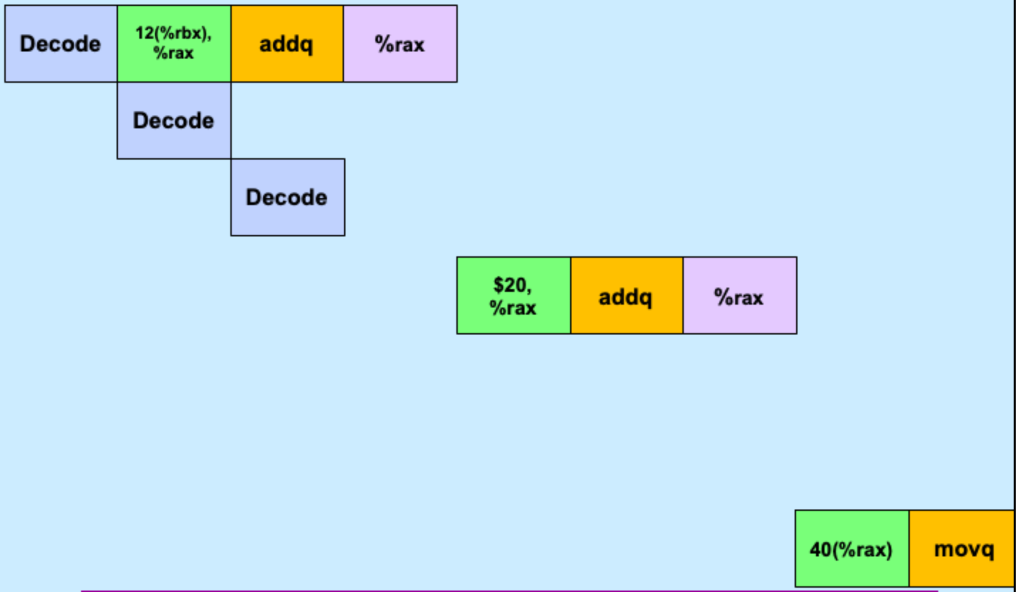


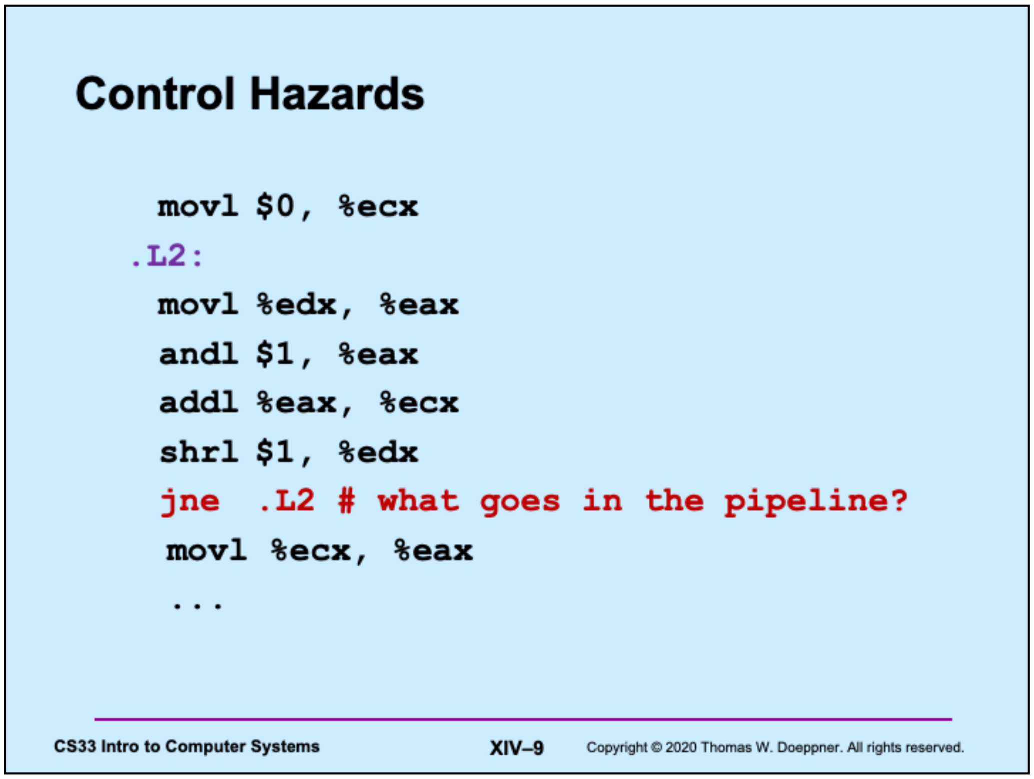
Data Hazards

```
addq 12(%rbx), %rax
addq $20, %rax
movq 40(%rax), %rsp
```



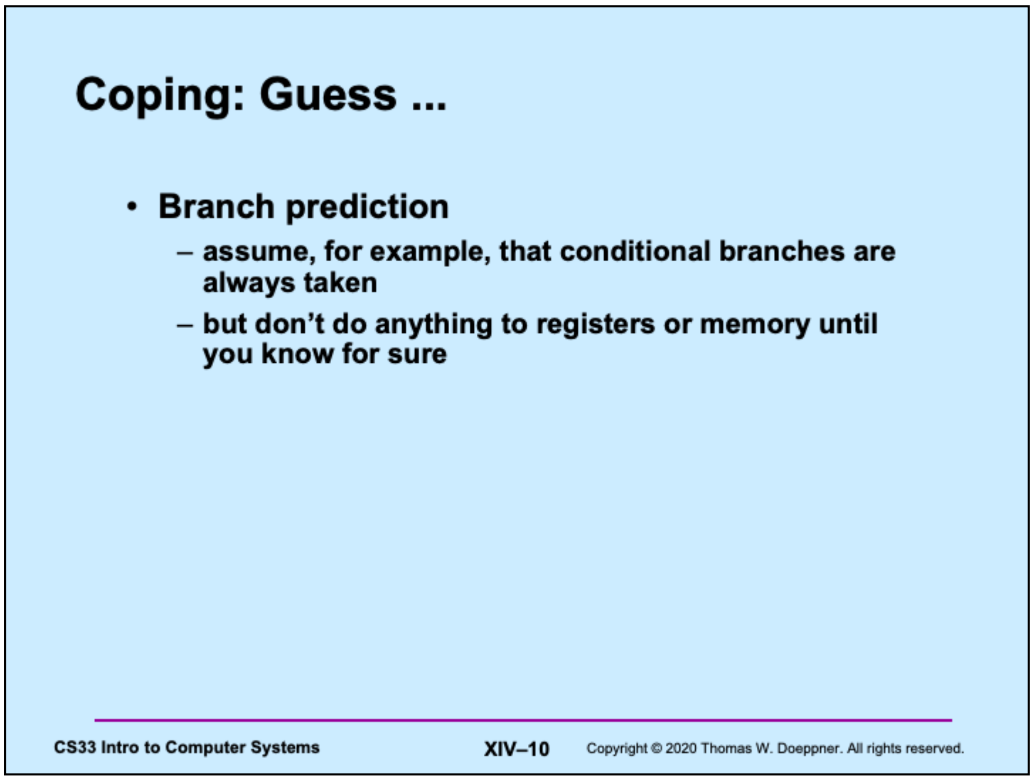
Coping


Decode
12(%rbx), %rax
addq
%rax
Decode
Decode
$20, %rax
addq
%rax
40(%rax)
movq



Control Hazards

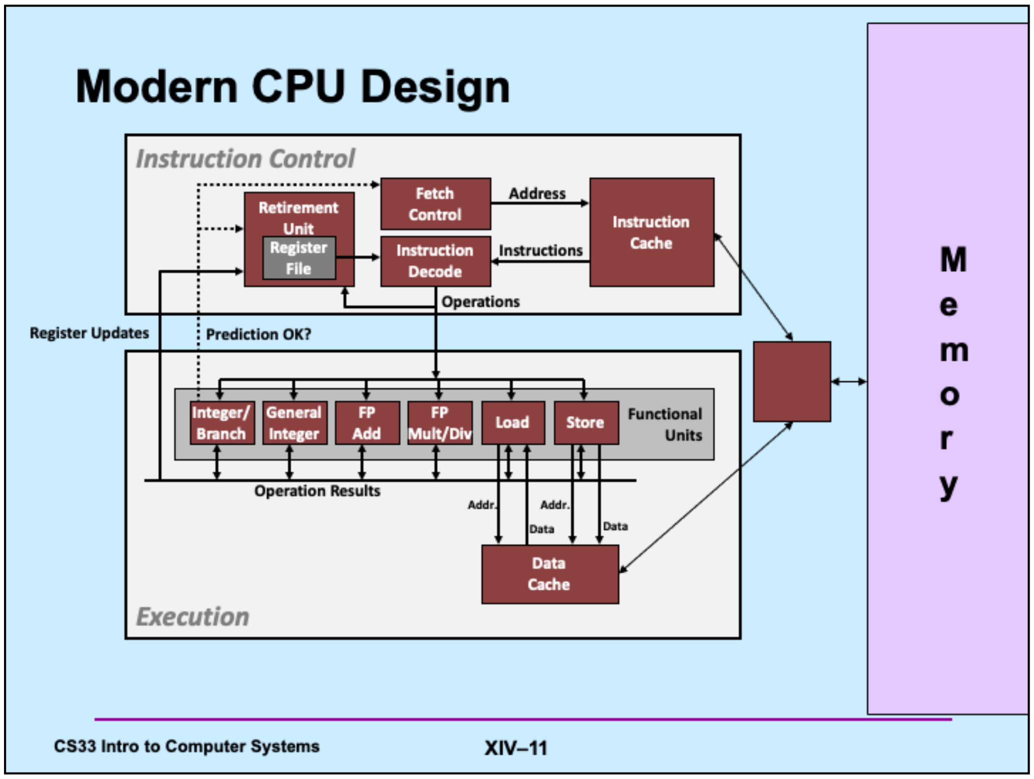
```
     movl $0, %ecx
   .L2:
     movl %edx, %eax
     andl $1, %eax
     addl %eax, %ecx
     shrl $1, %edx
     jne  .L2 # what goes in the pipeline?
      movl %ecx, %eax
      ...
```



Coping: Guess ...

- Branch prediction
  - assume, for example, that conditional branches are always taken
  - but don't do anything to registers or memory until you know for sure

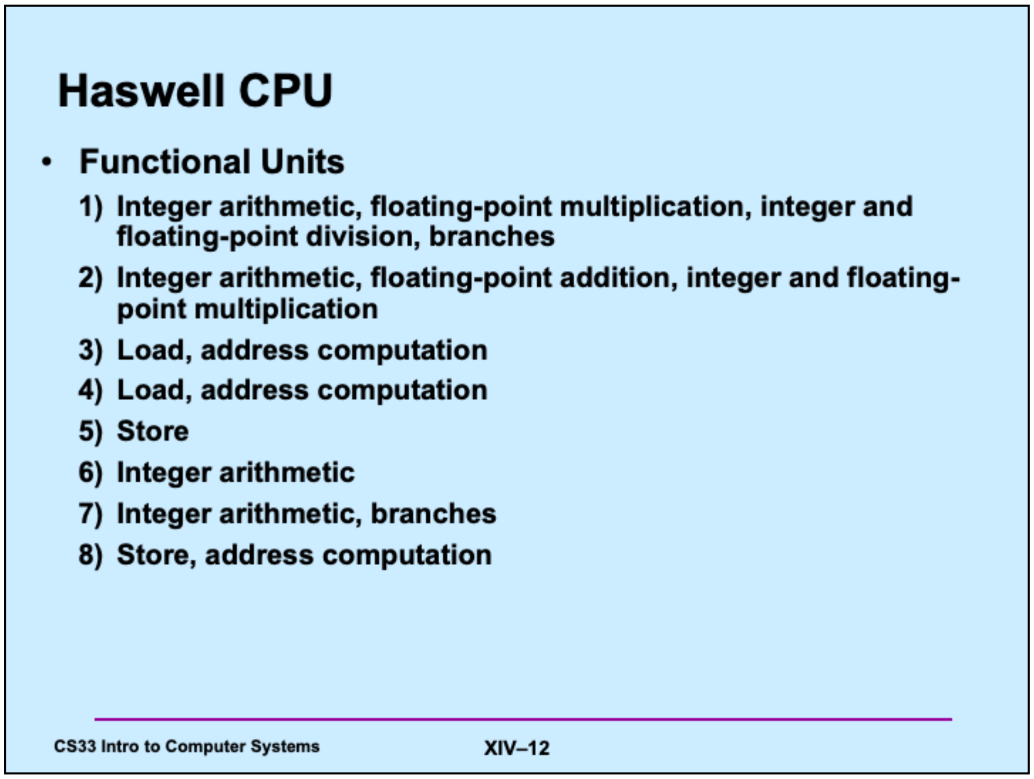
Modern processors have sophisticated algorithms for doing "branch prediction".



Modern CPU Design

Adapted from slide supplied by CMU.

Note that the functional units operate independently of one another. Thus, for example, the floating-point add unit can be working on one instruction, which the general integer unit can be working on another. Thus there are additional possibilities for parallel execution of instructions.

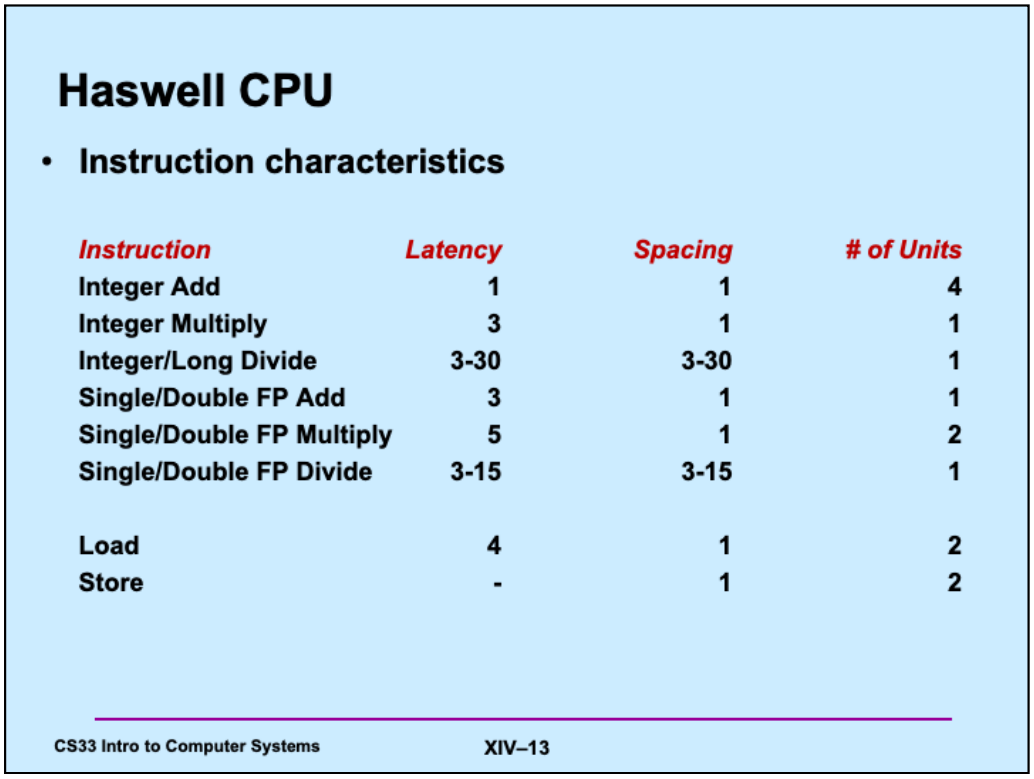


Haswell CPU

- **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

Supplied by CMU.

"Haswell" is Intel's code name for recent versions of its Core I7 and Core I5 processor design. Most of the computers in Brown CS employ Core I5 processors



Haswell CPU

- **Instruction characteristics**

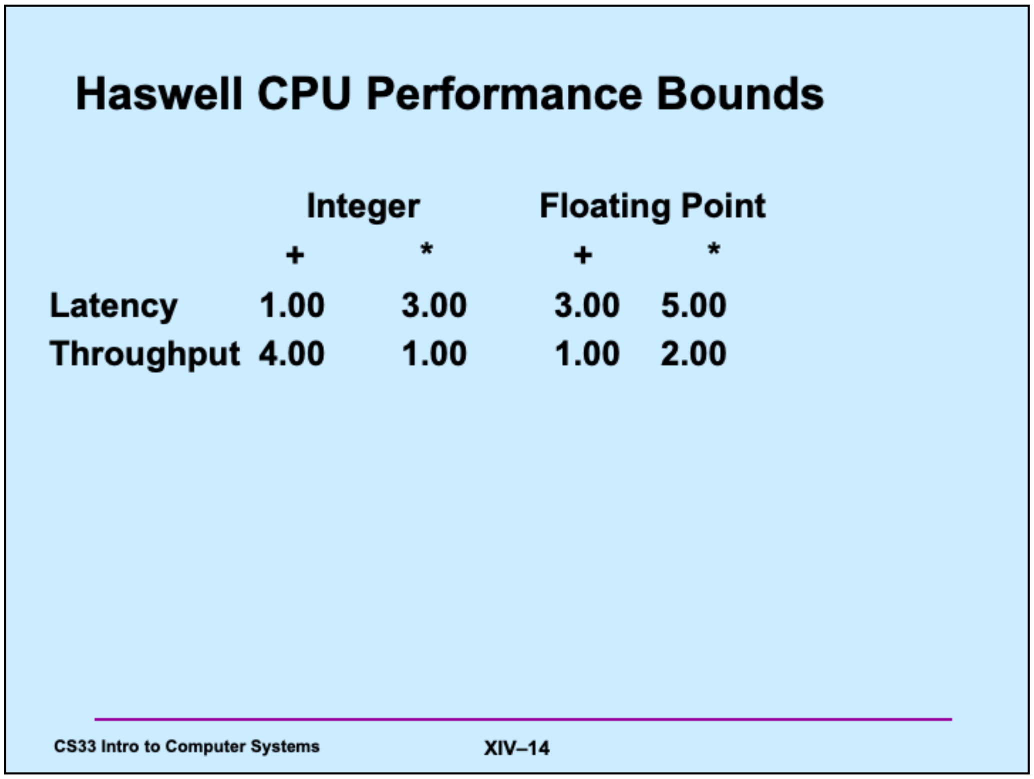
Instruction	*Latency*	*Spacing*	*# of Units*
Integer Add	1	1	4
Integer Multiply	3	1	1
Integer/Long Divide	3-30	3-30	1
Single/Double FP Add	3	1	1
Single/Double FP Multiply	5	1	2
Single/Double FP Divide	3-15	3-15	1
Load	4	1	2
Store	-	1	2

Adapted from a slide supplied by CMU.

These figures are for those cases in which the operands are either in registers or are immediate. For the other cases, additional time is required to load operands from memory or store them to memory.

"Spacing" is the number of clock cycles that must occur from the start of execution of one instruction to the start of execution to the next, i.e., how frequently can a functional unit start new instructions. The reciprocal of this value is the throughput: the number of instructions (typically a fraction) that can be completed per cycle. Note that for some types of instructions there is more than one instructional unit that can handle them, thus the maximum total throughput is the per-unit throughput times the number of units.

The figures for load and store assume the data is coming from/going to the data cache. Much more time is required if the source or destination is RAM.



Haswell CPU Performance Bounds

	Integer +	Integer *	Floating Point +	Floating Point *
Latency	1.00	3.00	3.00	5.00
Throughput	4.00	1.00	1.00	2.00

Derived from a slide provided by CMU.

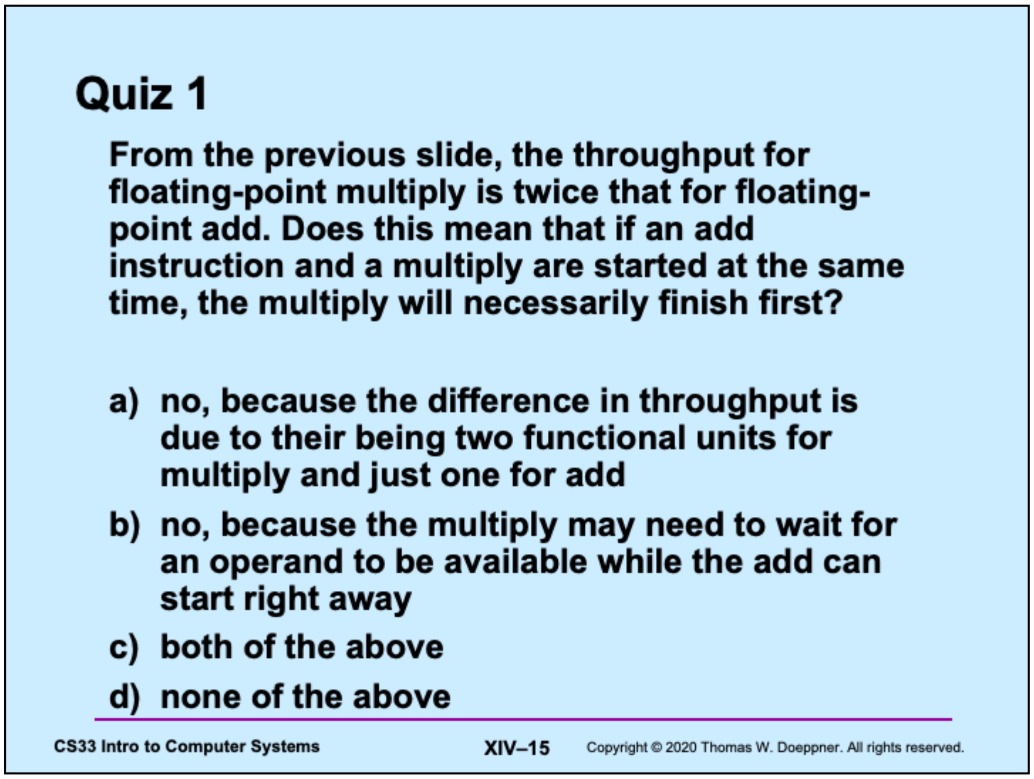
We assume that the source and destination are either immediate (source only) or registers. Thus any bottlenecks due to memory access do not arise.

Note that the units for latency are clock cycles (it takes that number of cycles for an operation to be executed from start to finish), while the units for throughput are instructions/cycle – how many instructions can complete per clock cycle.

Each functional unit doing an integer add requires one clock cycle of latency. However, since there are four such functional units, all four can be kept busy with integer add instructions and thus the aggregate throughput can be as good as one integer add instruction completing, on average, every .25 clock cycles, for a throughput of 4 instructions/cycle.

Each integer multiply requires three clock cycles. But since a new multiply instruction can be started every clock cycle (i.e., they can be pipelined), the aggregate throughput can be as good as one integer multiply completing every clock cycle.

Each floating point multiply requires five clock cycles, but they can be pipelined with one starting every clock cycle. Since there are two functional units that can perform floating point multiply, the aggregate throughput can be as good as one completing every .5 clock cycles, for a throughput of 2 instructions/cycle.



Quiz 1

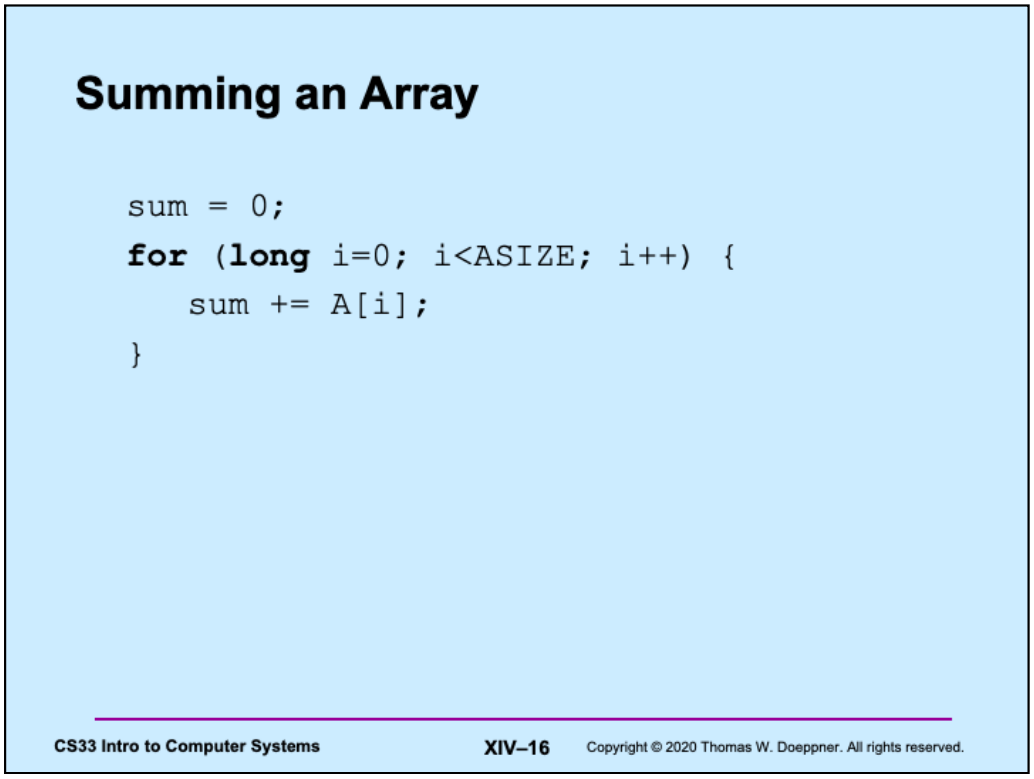
From the previous slide, the throughput for floating-point multiply is twice that for floating-point add. Does this mean that if an add instruction and a multiply are started at the same time, the multiply will necessarily finish first?

a) no, because the difference in throughput is due to their being two functional units for multiply and just one for add

b) no, because the multiply may need to wait for an operand to be available while the add can start right away

c) both of the above

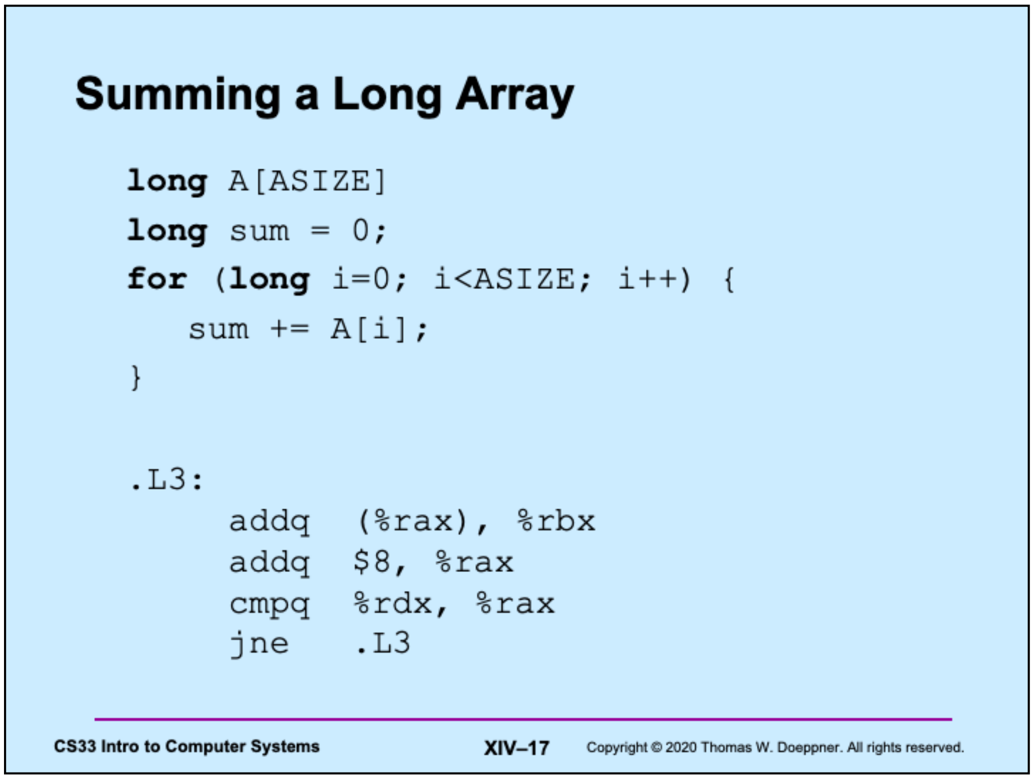
d) none of the above



Summing an Array

```
sum = 0;
for (long i=0; i<ASIZE; i++) {
    sum += A[i];
}
```

The code for summing the elements of an array is straightforward. However, because the add operation in each iteration of the loop requires the result of the add operation of the previous iteration, the processor is inhibited from pipelining the adds (a data hazard).

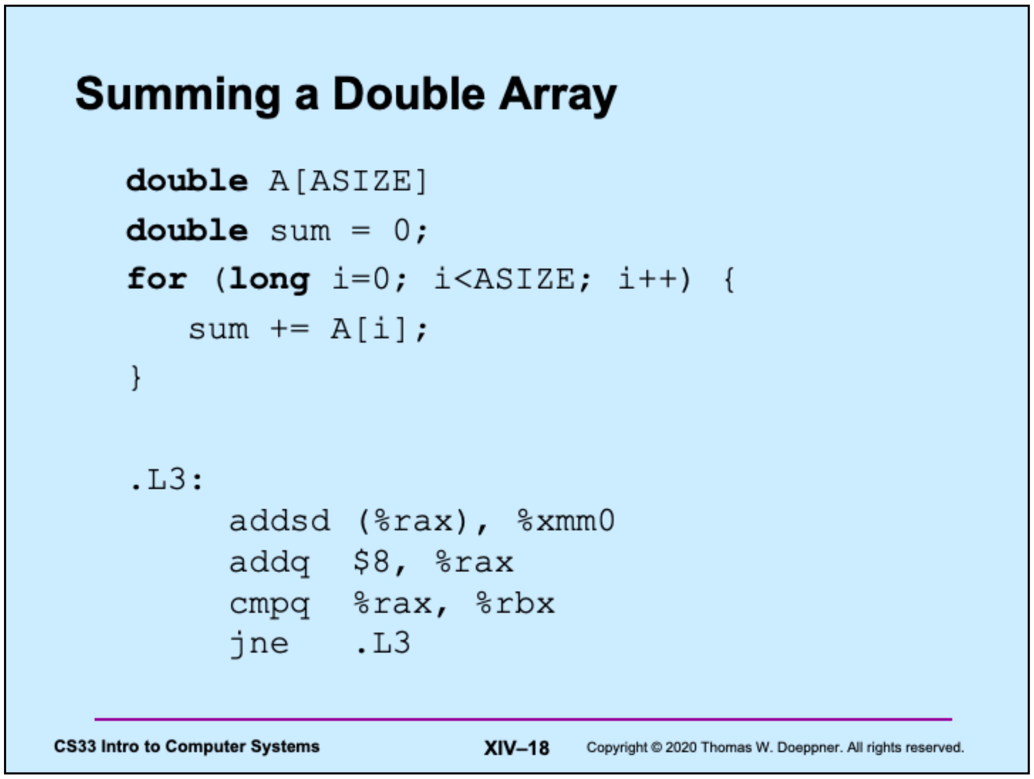


Summing a Long Array

```
long A[ASIZE]
long sum = 0;
for (long i=0; i<ASIZE; i++) {
   sum += A[i];
}

.L3:
     addq  (%rax), %rbx
     addq  $8, %rax
     cmpq  %rdx, %rax
     jne   .L3
```

Here we see the code compiled by gcc (with the –O1 flag) for the case in which we're summing an array of longs. There are two add operations, but we have (at least) two functional units that can do adds. There is a conditional jump, but the processor will guess that the jump is taken. Thus the loop can be performed at speeds up to one clock cycle/iteration.

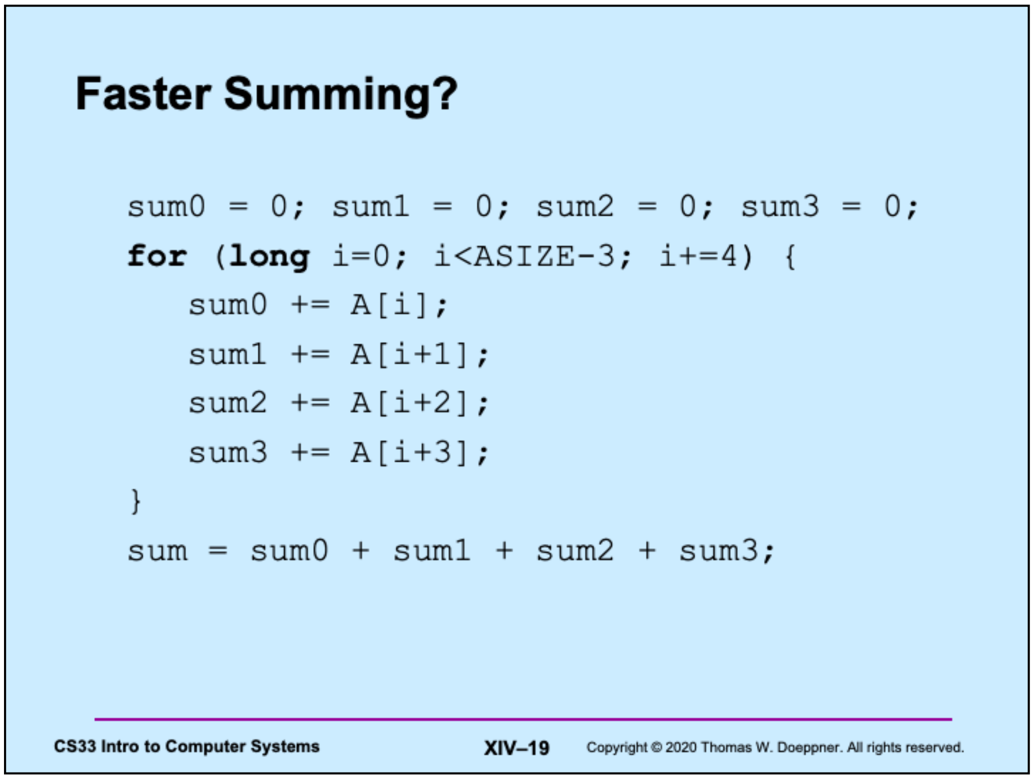


Summing a Double Array

```
double A[ASIZE]
double sum = 0;
for (long i=0; i<ASIZE; i++) {
   sum += A[i];
}
```

```
.L3:
     addsd (%rax), %xmm0
     addq  $8, %rax
     cmpq  %rax, %rbx
     jne   .L3
```

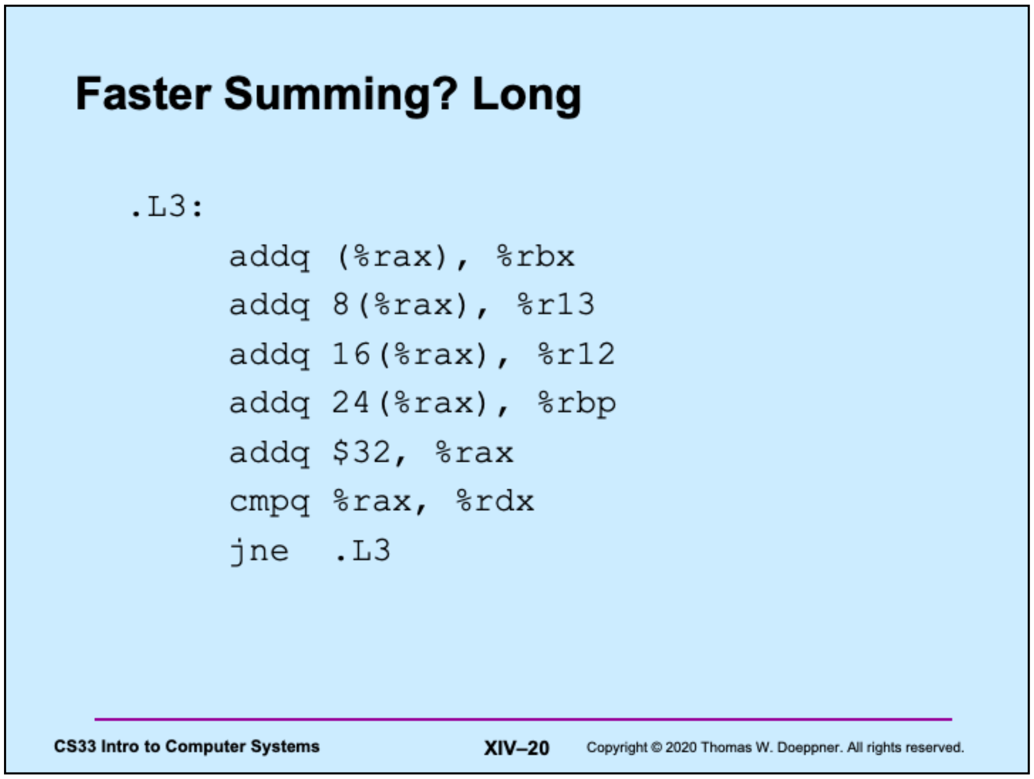
Here's the code produced by gcc for the case in which we have an array of doubles. (It was compiled with the –O2 flag – for some reason gcc doesn't produce particularly good code for this example when given the –O1 flag.) %xmm0 is one of sixteen floating-point registers. There are two add instructions: the first is "add scalar double precision" (there's a vector add as well, which we won't get to – using it, particularly in C programs, is beyond the scope of this course). The addq (computing the next value of i) completes in one clock cycle, but the addsd requires 3. Let's assume the processor guesses that the jump will be taken. Even though the value in %rax will be available at the start of the next iteration of the loop, the value in %xmm0 won't be available until three clock cycles have completed. Thus at least three clock cycles are required for each iteration of the loop.



Faster Summing?

```
sum0 = 0; sum1 = 0; sum2 = 0; sum3 = 0;
for (long i=0; i<ASIZE-3; i+=4) {
   sum0 += A[i];
   sum1 += A[i+1];
   sum2 += A[i+2];
   sum3 += A[i+3];
}
sum = sum0 + sum1 + sum2 + sum3;
```

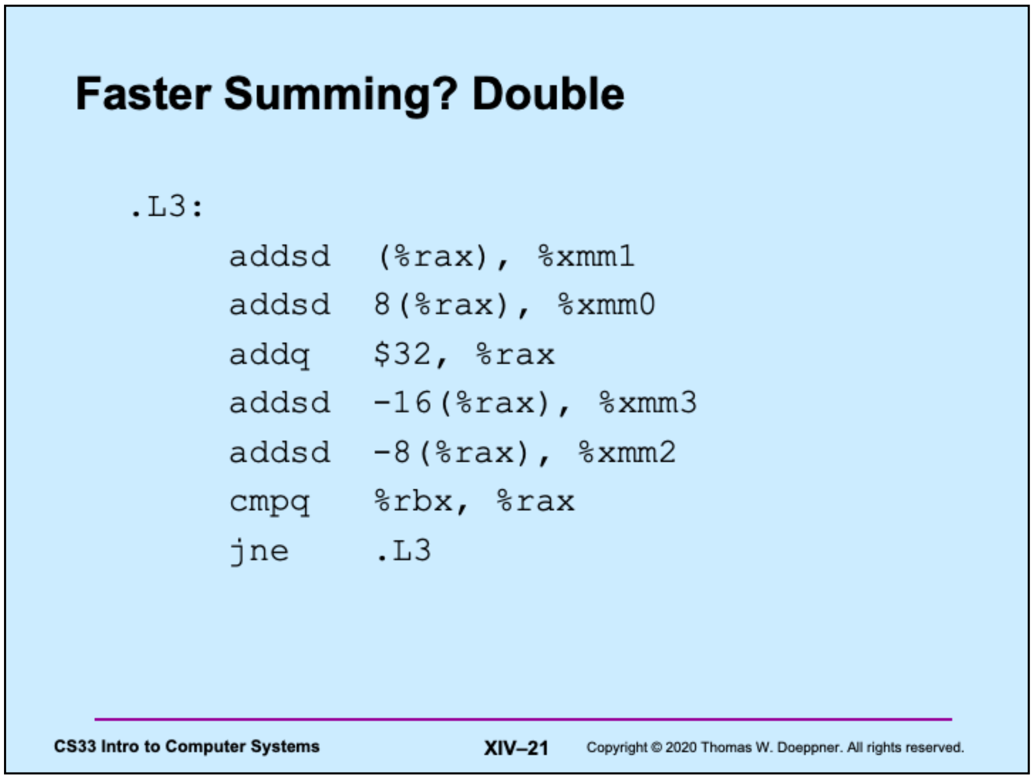
An alternative approach to coding the loop is shown here, where we've "unrolled" the loop and have separate temporary variables holding portions of the sum. The advantage of doing it this way is that the computation of each of the different sums does not depend on the results of the others. Thus all four sums of each iteration may be pipelined so that a sum completes each clock cycle. Note that the fact that we have four functional units that can do addition doesn't help here: if we used all four in each iteration, we're still completing each iteration in one clock cycle. The next iteration can't start until the previous has finished, since the results of each iteration are needed before the next can begin.



Faster Summing? Long

```
.L3:
     addq (%rax), %rbx
     addq 8(%rax), %r13
     addq 16(%rax), %r12
     addq 24(%rax), %rbp
     addq $32, %rax
     cmpq %rax, %rdx
     jne  .L3
```

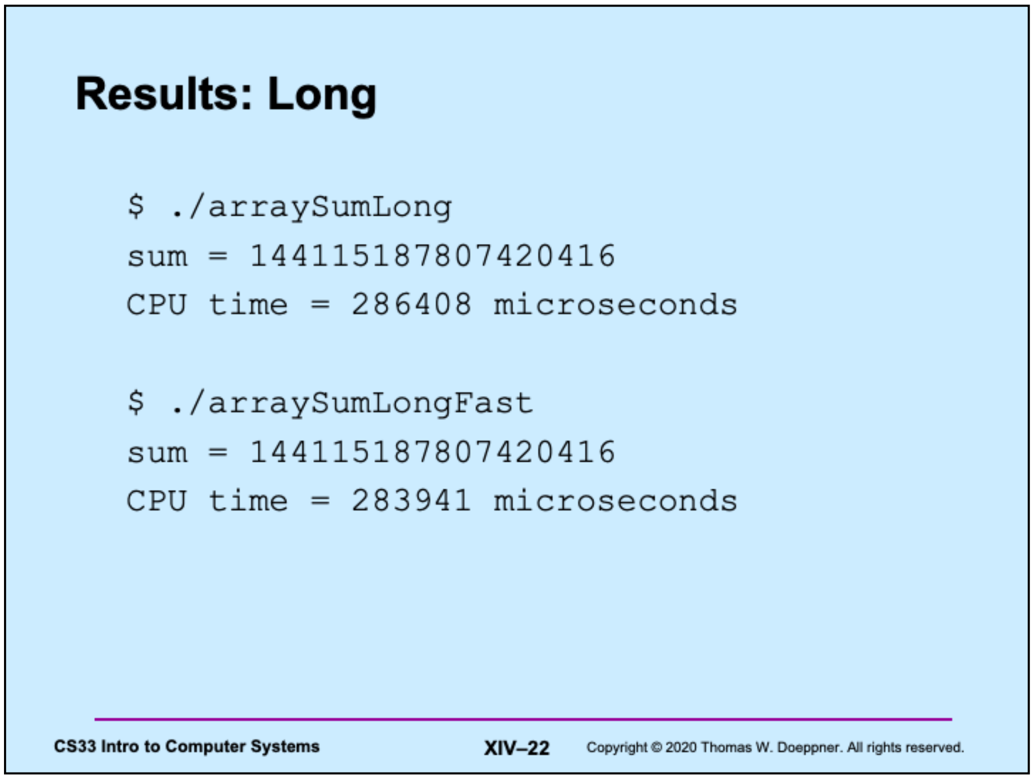
Here's the code produced by gcc –O1 for the case of longs. It uses four different registers to hold the temporary results.



Faster Summing? Double

```
.L3:
        addsd   (%rax), %xmm1
        addsd   8(%rax), %xmm0
        addq    $32, %rax
        addsd   -16(%rax), %xmm3
        addsd   -8(%rax), %xmm2
        cmpq    %rbx, %rax
        jne     .L3
```

Here's the code produced by gcc –O1 for the case of doubles. It uses four different registers to hold the temporary results, which, in this case, allows a single floating-point-add functional unit to pipeline the adds. It's not immediately clear why gcc chose to increment %rax in the middle of each loop iteration rather than at the end.

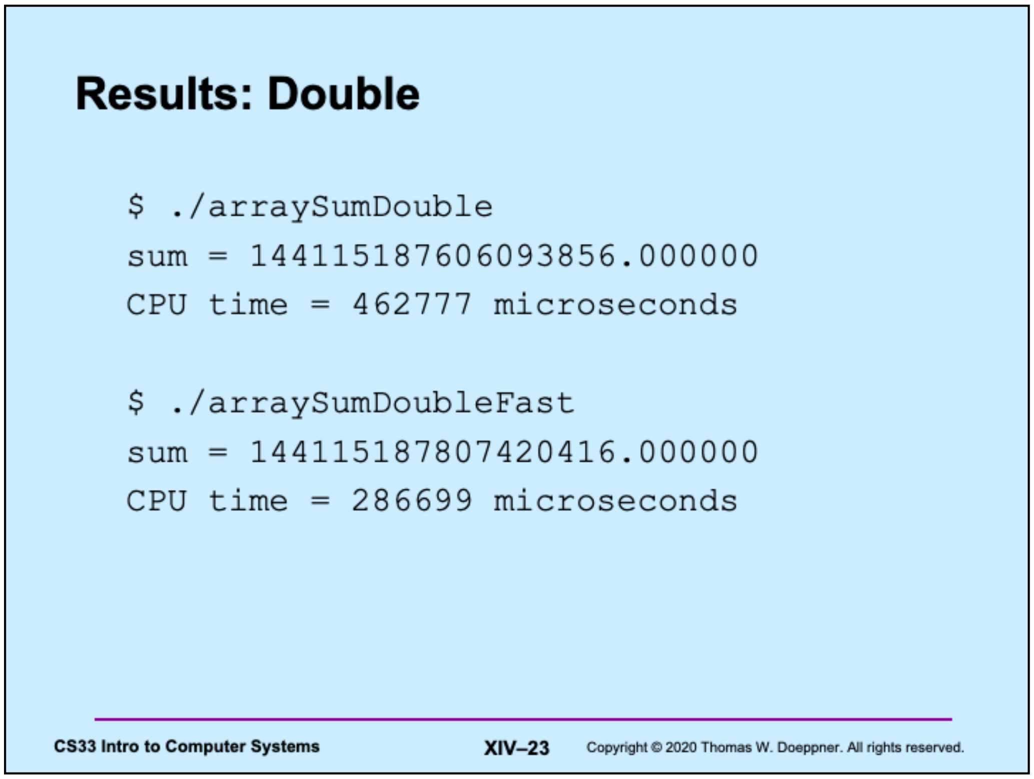


Results: Long

```
$ ./arraySumLong
sum = 144115187807420416
CPU time = 286408 microseconds

$ ./arraySumLongFast
sum = 144115187807420416
CPU time = 283941 microseconds
```

As expected, there's not much speedup for the case of summing the elements of an array of longs.


Results: Double

```
$ ./arraySumDouble
sum = 144115187606093856.000000
CPU time = 462777 microseconds

$ ./arraySumDoubleFast
sum = 144115187807420416.000000
CPU time = 286699 microseconds
```

However, we get a significant speedup for the case in which we sum the elements of an array of doubles.

Quiz 2

The sums given in the previous slide are different for the two cases. Why is this?

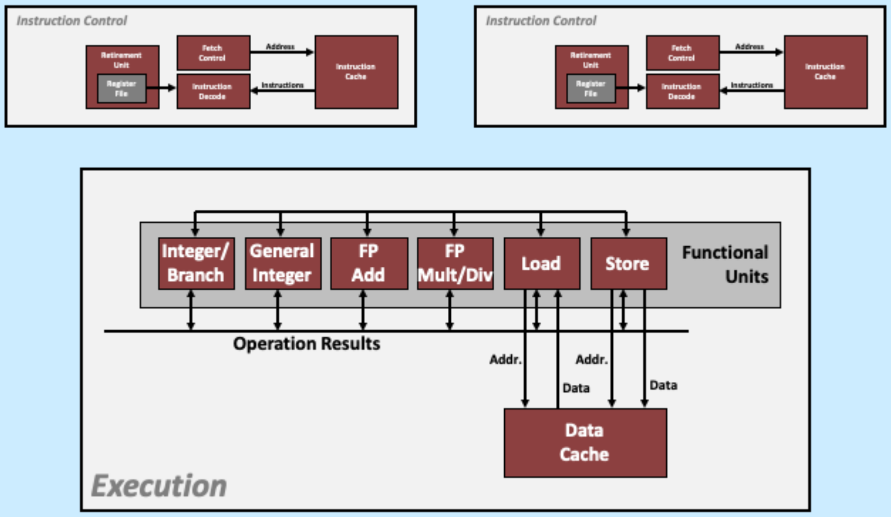
a) Floating-point arithmetic is not associative

b) A problem was introduced by incrementing %rax in the middle of each iteration rather than at the end

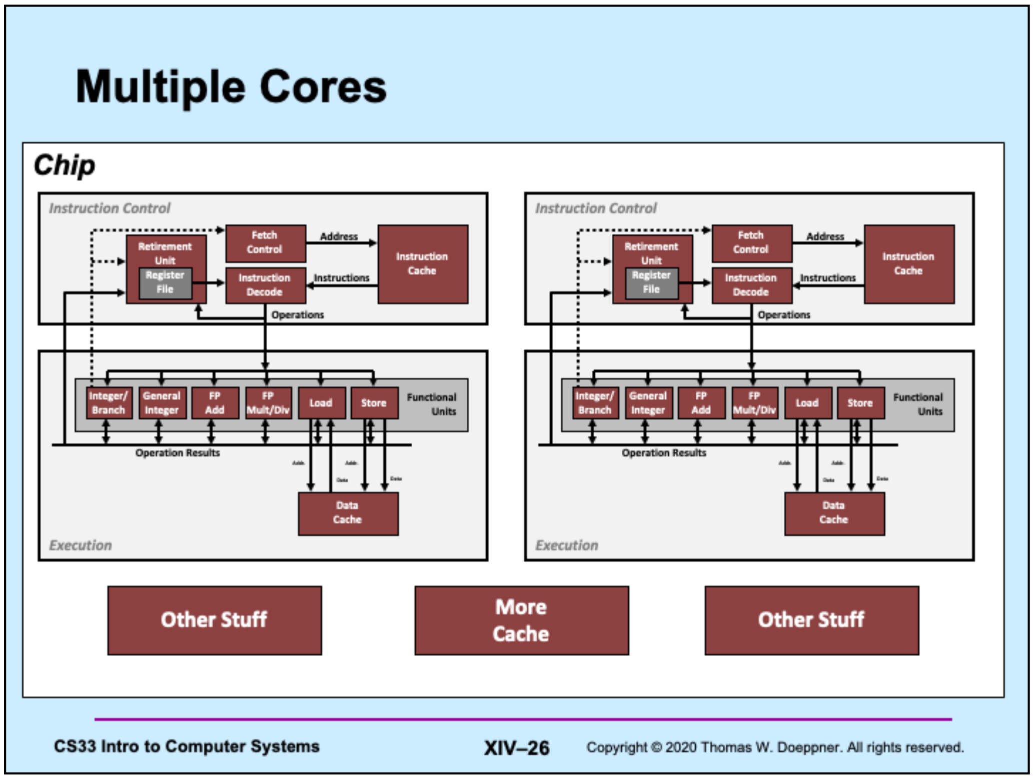
c) There's a bug in the C code for the "fast version" that results in too many iterations of the loop

d) Something else

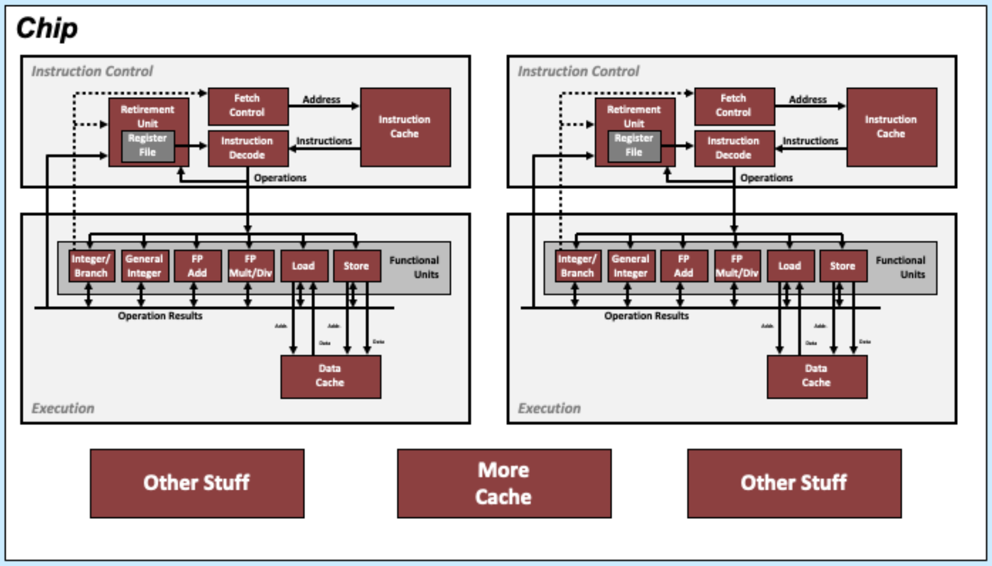
Hyper Threading



One way of improving the utilization of the functional units of a processor is hyperthreading. The processor supports multiple instruction streams ("hyper threads"), each with its own instruction control. But all the instruction streams share the one set of functional units.



Multiple Cores

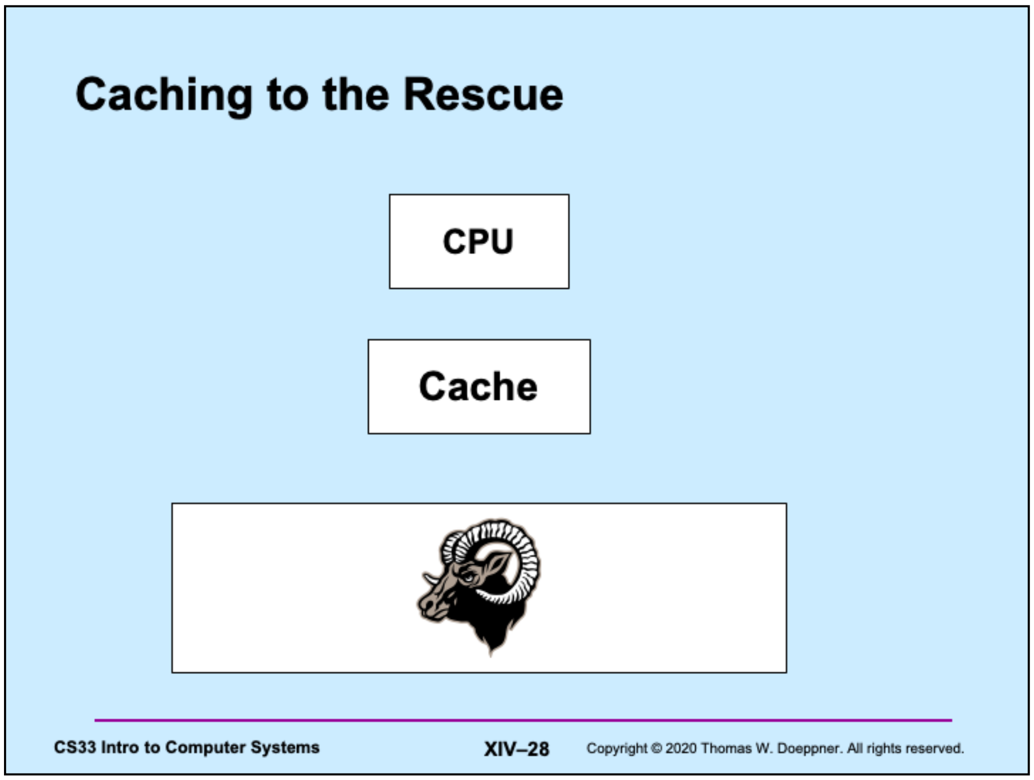


Going a step further, one can pack multiple complete processors onto one chip. Each processor is known as a core and can execute instructions independently of the other cores (each has its private set of functional units). In addition to each core having its own instruction and data cache, there are caches shared with the other cores on the chip. We discuss this in more detail in a subsequent lecture.

In many of today's processor chips, hyperthreading is combined with multiple cores. Thus, for example, a chip might have four cores each with four hyperthreads. Thus the chip might handle 16 instruction streams.

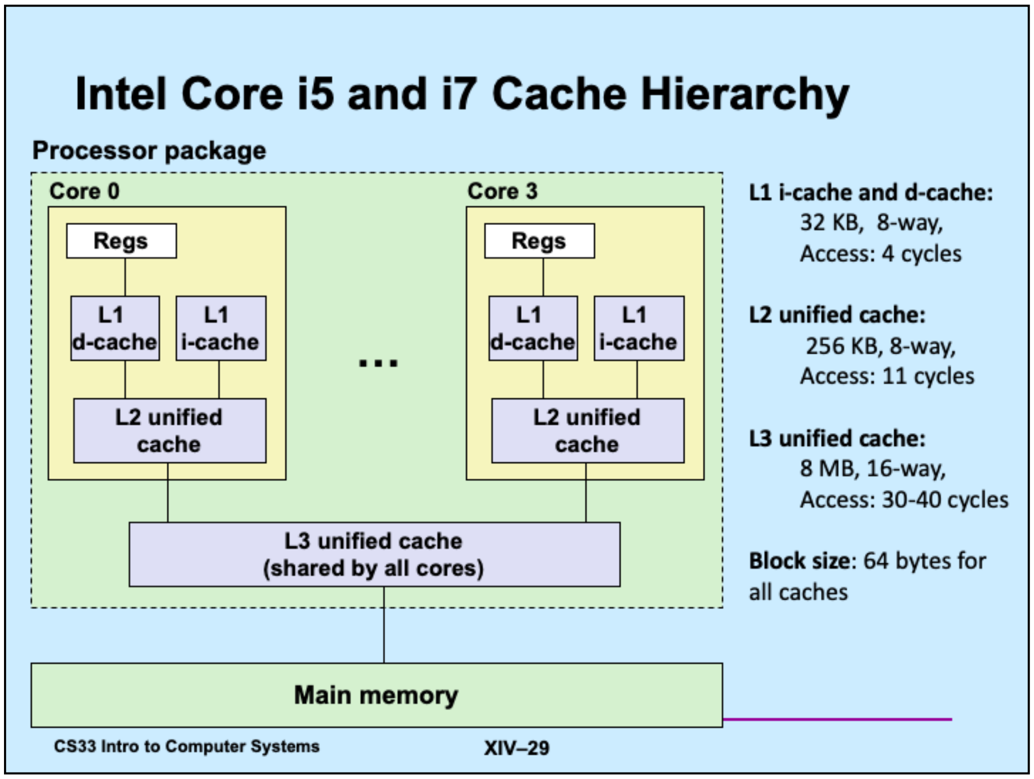
A Mismatch

- A processor clock cycle is ~0.3 nsecs
  - SunLab machines (Intel Core i5-4690) run at 3.5 GHz
- Basic operations take 1 – 10 clock cycles
  - .3 – 3 nsecs
- Accessing memory takes 70-100 nsecs
- How is this made to work?



Caching to the Rescue

Sitting between the processor and RAM are one or more caches. (They actually are on the chip along with the processor.) Recently accessed items by the processor reside in the cache, where they are much more quickly accessed than directly from memory. The processor does a certain amount of pre-fetching to get things from RAM before they are needed. This involves a certain amount of guesswork, but works reasonably well, given well behaved programs.



Intel Core i5 and i7 Cache Hierarchy

L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches

Supplied by CMU.

The L3 cache is known as the *last-level cache* (LLC) in the Intel documentation.

One concern is whether what's contained in, say, the L1 cache is also contained in the L2 cache. if so, caching is said to be *inclusive*. If what's contained in the L1 cache is definitely not contained in the L2 cache, caching is said to be *exclusive*. An advantage of exclusive caches is that the total cache capacity is the sum of the sizes of each of the levels, whereas for inclusive caches, the total capacity is just that of the largest. An advantage of inclusive caches is that what's been brought into the cache hierarchy by one core is available to the other core.

AMD processors tend to have exclusive caches; Intel processors tend to have inclusive caches.

We will explain what is meant by 8-way and 16-way caches in a subsequent lecture, if we have time. For the purposes of this lecture, the distinction is not important.

Accessing Memory

- Program references memory (load)
  - if not in cache (*cache miss*), data is requested from RAM
    - » fetched in units of 64 bytes
      - aligned to 64-byte boundaries (low-order 6 bits of address are zeroes)
    - » if memory accessed sequentially, data is pre-fetched
    - » data stored in cache (in 64-byte *cache lines*)
      - stays there until space must be re-used (least recently used is kicked out first)
  - if in cache (*cache hit*) no access to RAM needed
- Program modifies memory (store)
  - data modified in cache
  - eventually written to RAM in 64-byte units

Quiz 3

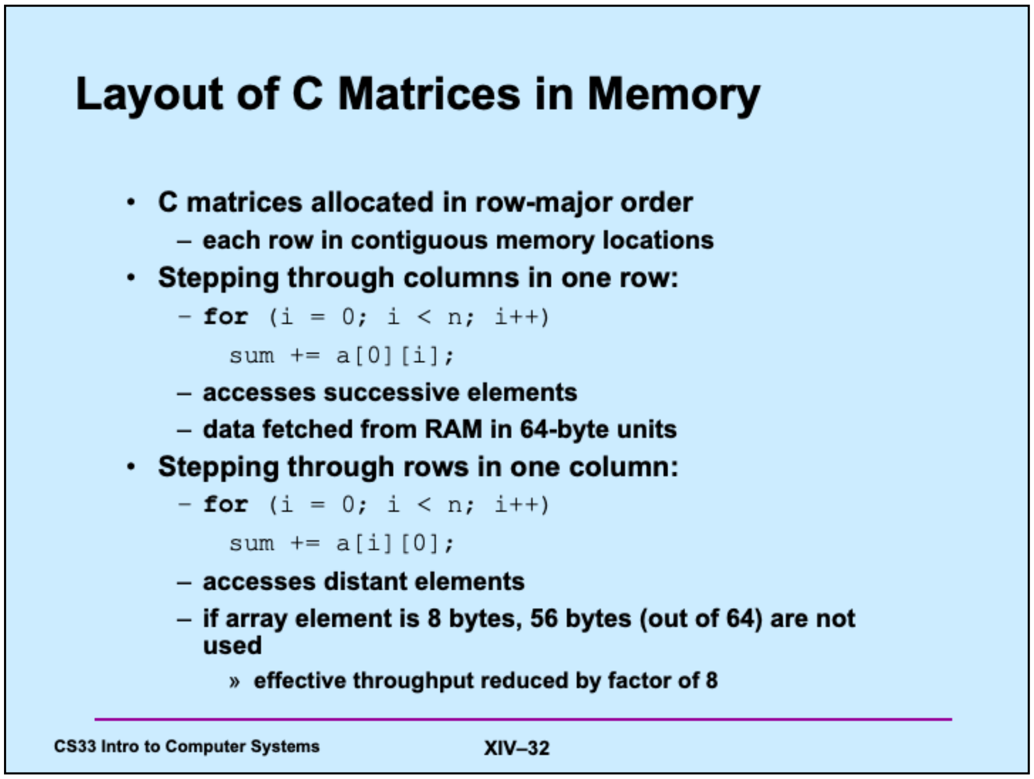
The previous slide said that 64 bytes of memory from contiguous locations are transferred at a time. Suppose we have memory that transfers 128 contiguous bytes in the same amount of time. If we have a program that reads memory one byte at a time from random (but valid) memory locations, how much faster will it run with the new memory system than with the old?

a) half as fast

b) roughly the same speed

c) twice as fast

d) four times as fast



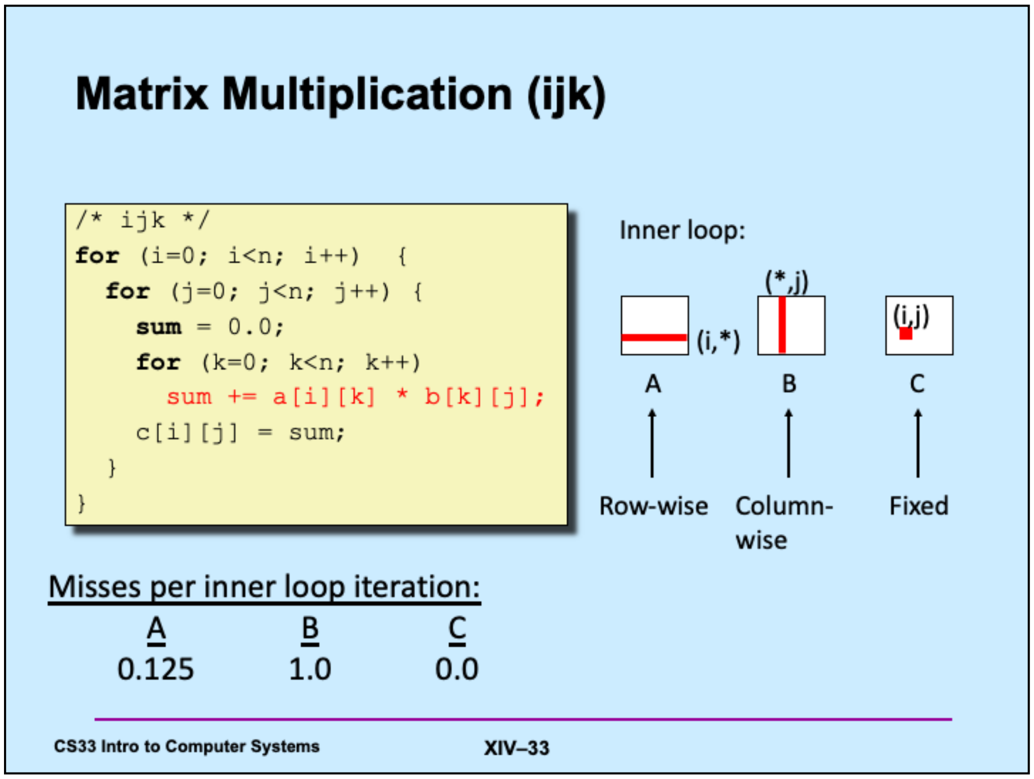
Layout of C Matrices in Memory

- C matrices allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:

  -
    ```
    for (i = 0; i < n; i++)
      sum += a[0][i];
    ```
  - accesses successive elements
  - data fetched from RAM in 64-byte units
- Stepping through rows in one column:

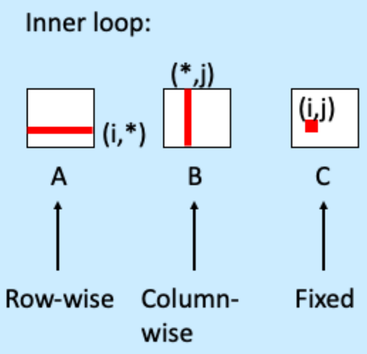
  -
    ```
    for (i = 0; i < n; i++)
      sum += a[i][0];
    ```
  - accesses distant elements
  - if array element is 8 bytes, 56 bytes (out of 64) are not used
    - » effective throughput reduced by factor of 8

Adapted from a slide supplied by CMU.



Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

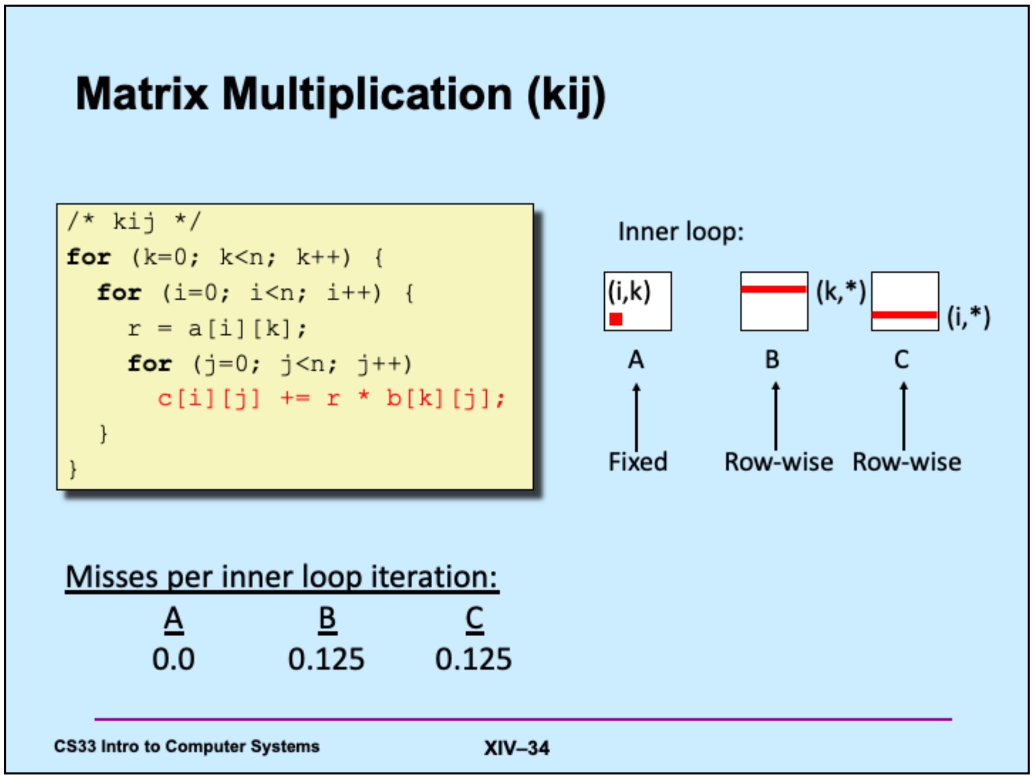


Misses per inner loop iteration:

A	B	C
0.125	1.0	0.0

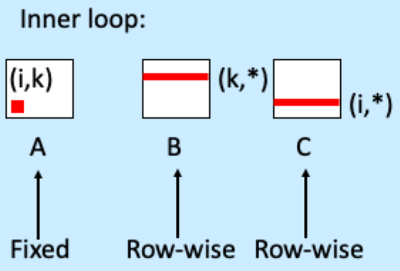
Supplied by CMU.

Assume we are multiplying arrays of doubles, thus each element is eight bytes long, and thus a cache line holds eight matrix elements.



Matrix Multiplication (kij)

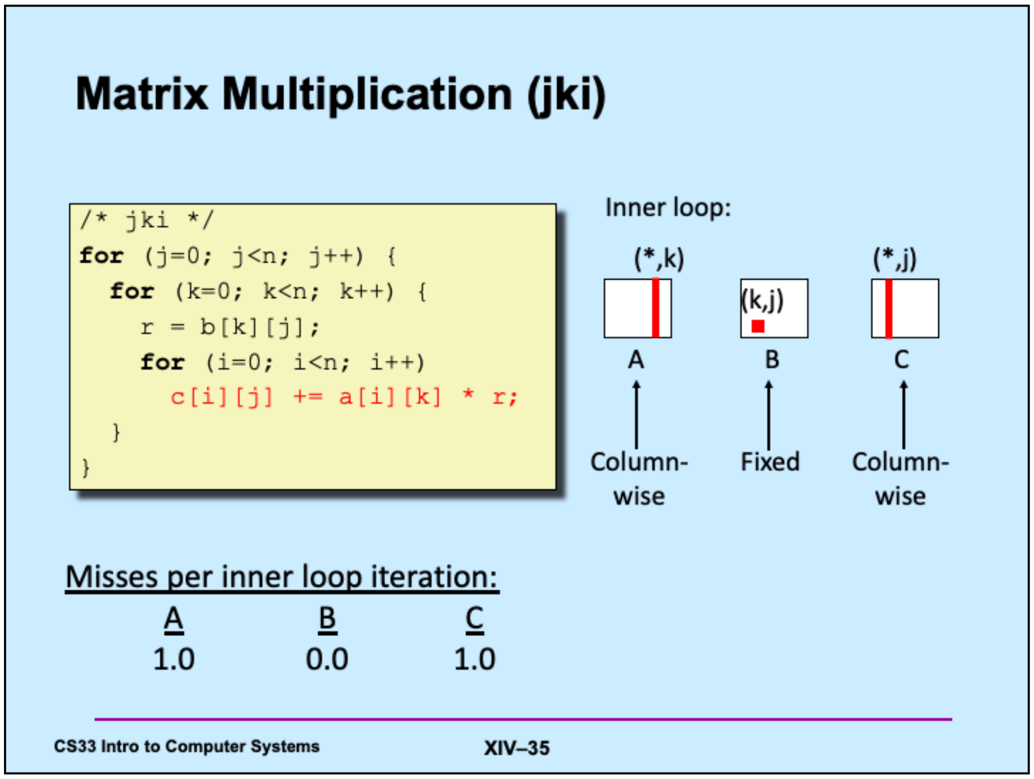
```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```



Misses per inner loop iteration:

A	B	C
0.0	0.125	0.125

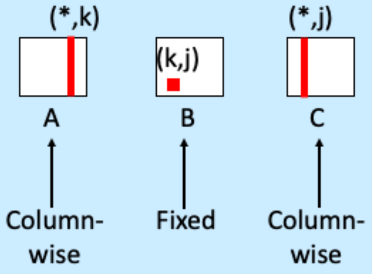
Supplied by CMU.



Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

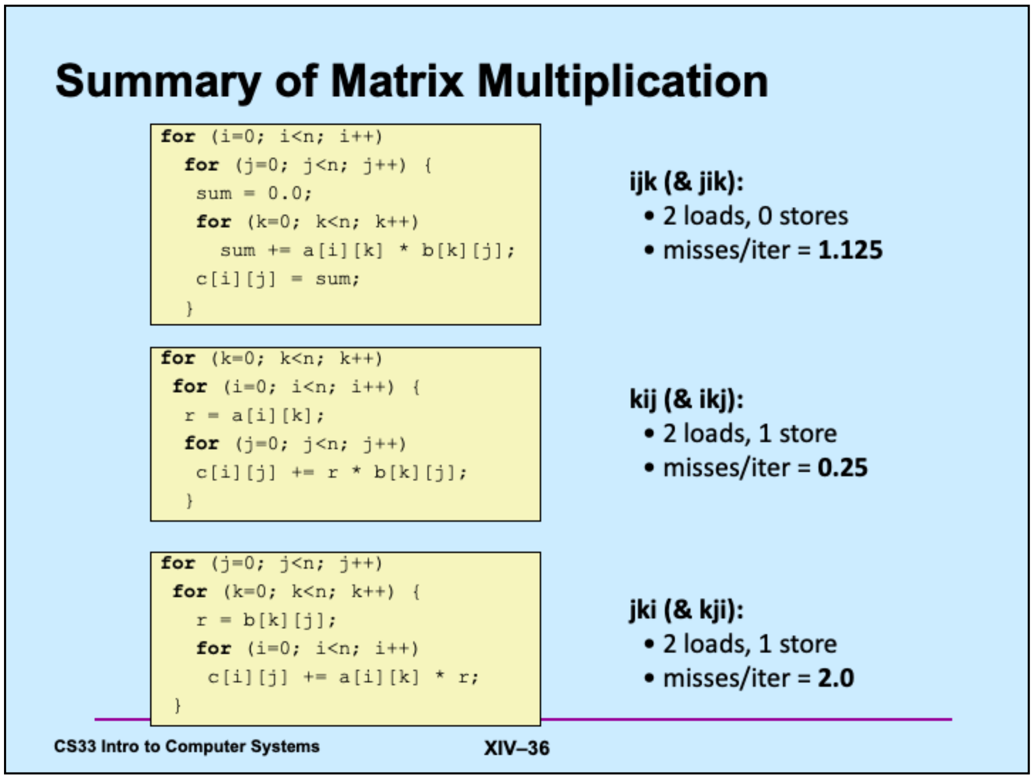
Inner loop:



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Supplied by CMU.



Summary of Matrix Multiplication

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
  }
```

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.125**

```
for (k=0; k<n; k++)
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
  }
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.25**

```
for (j=0; j<n; j++)
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

Supplied by CMU.

In Real Life ...

- Multiply two 1024x1024 matrices of doubles on sunlab machines
  - ijk
    - 4.185 seconds
  - kij
    - 0.798 seconds
  - jki
    - 11.488 seconds