CS 33

Intro to Computer Architecture

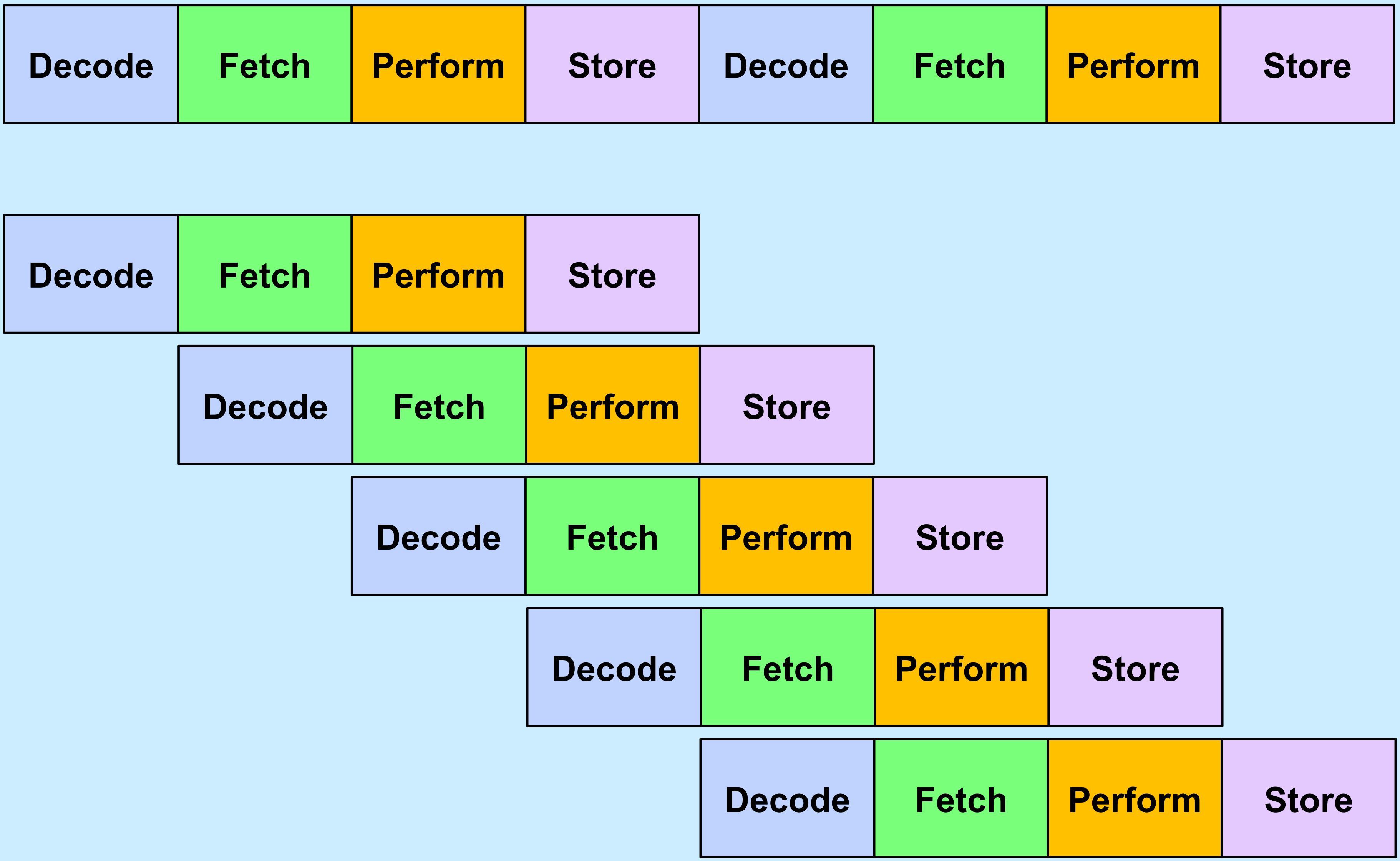
Simplistic View of Processor

```
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

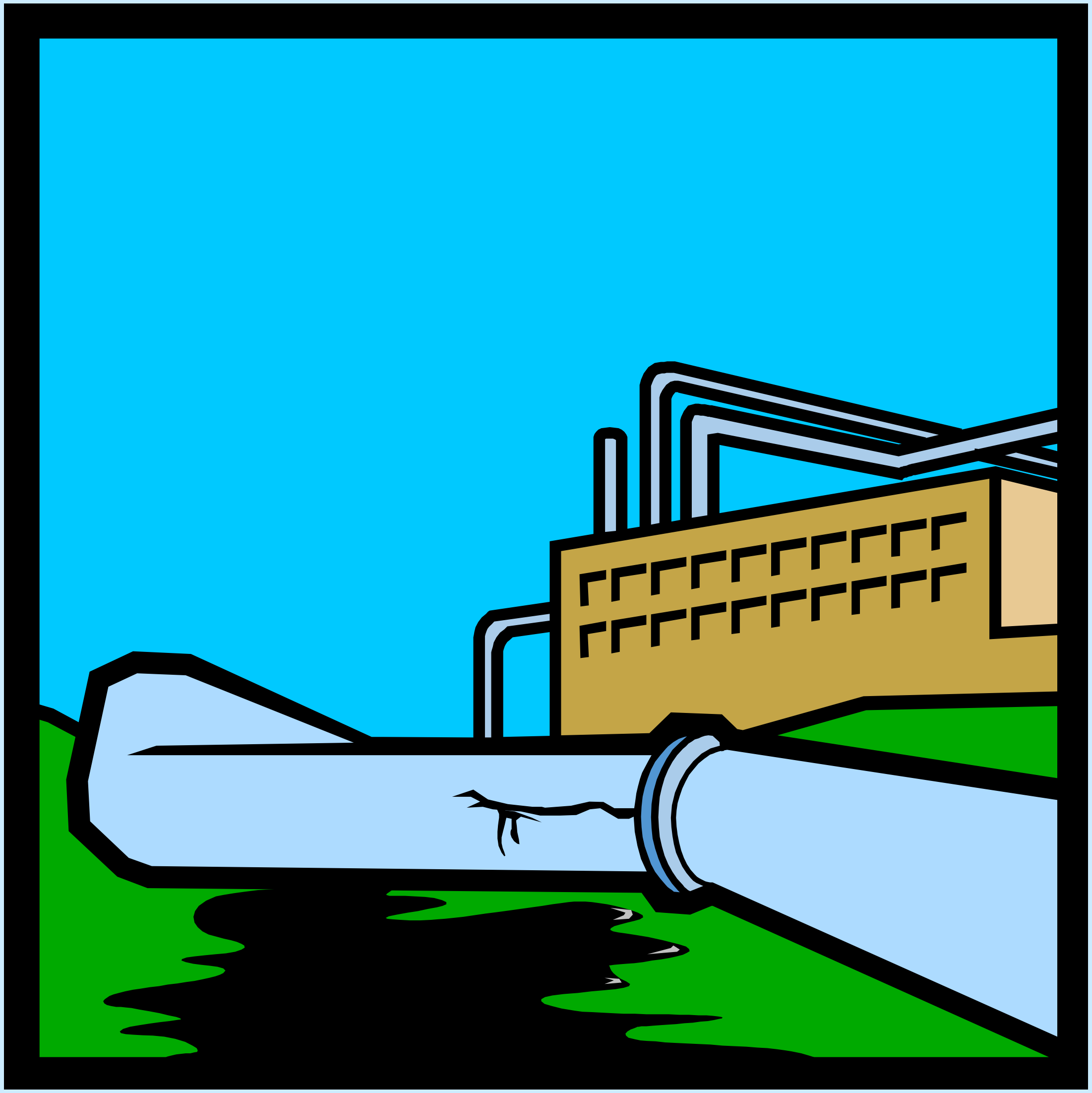


Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store

Analysis

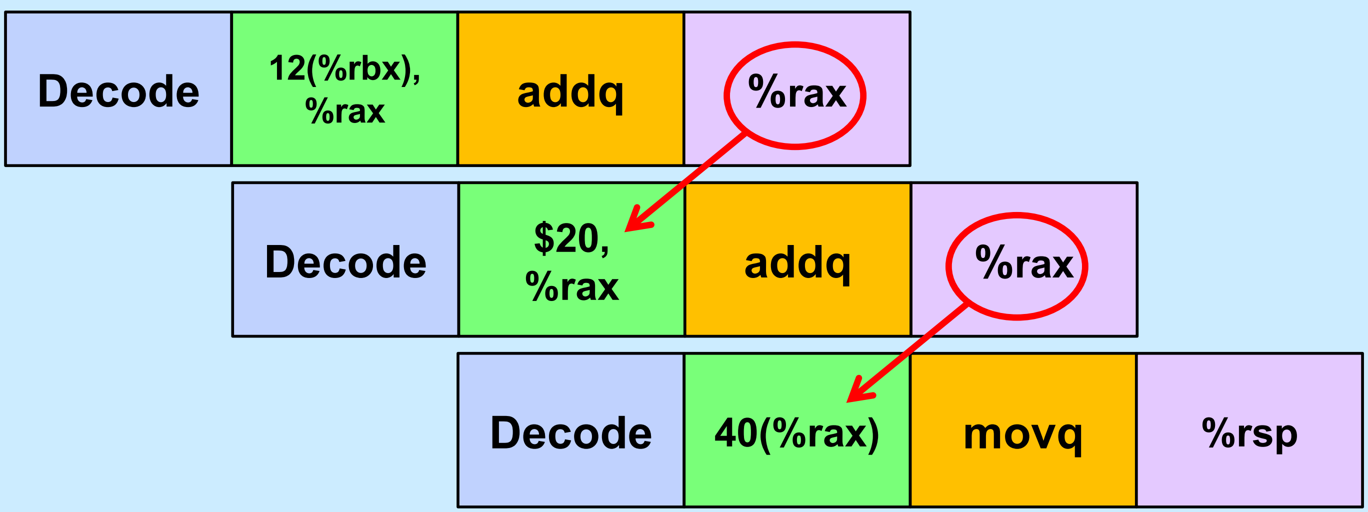
- **Not pipelined**
  - each instruction takes, say, 3.2 nanoseconds
    - » 3.2 ns latency
  - 312.5 million instructions/second (MIPS)
- **Pipelined**
  - each instruction still takes 3.2 ns
    - » latency still 3.2 ns
  - an instruction completes every .8 ns
    - » 1.25 billion instructions/second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

Hazards ...

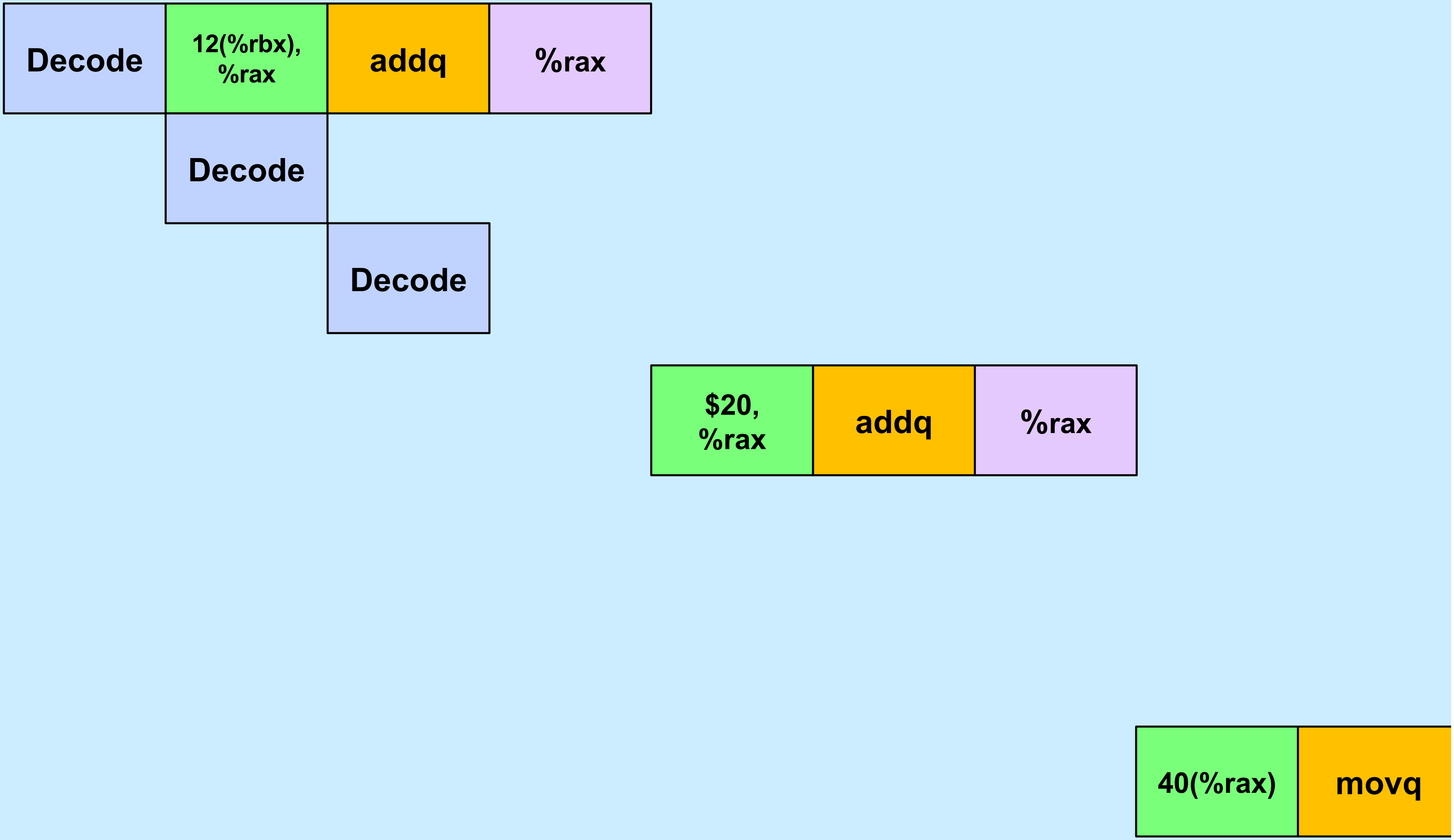


Data Hazards

```
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

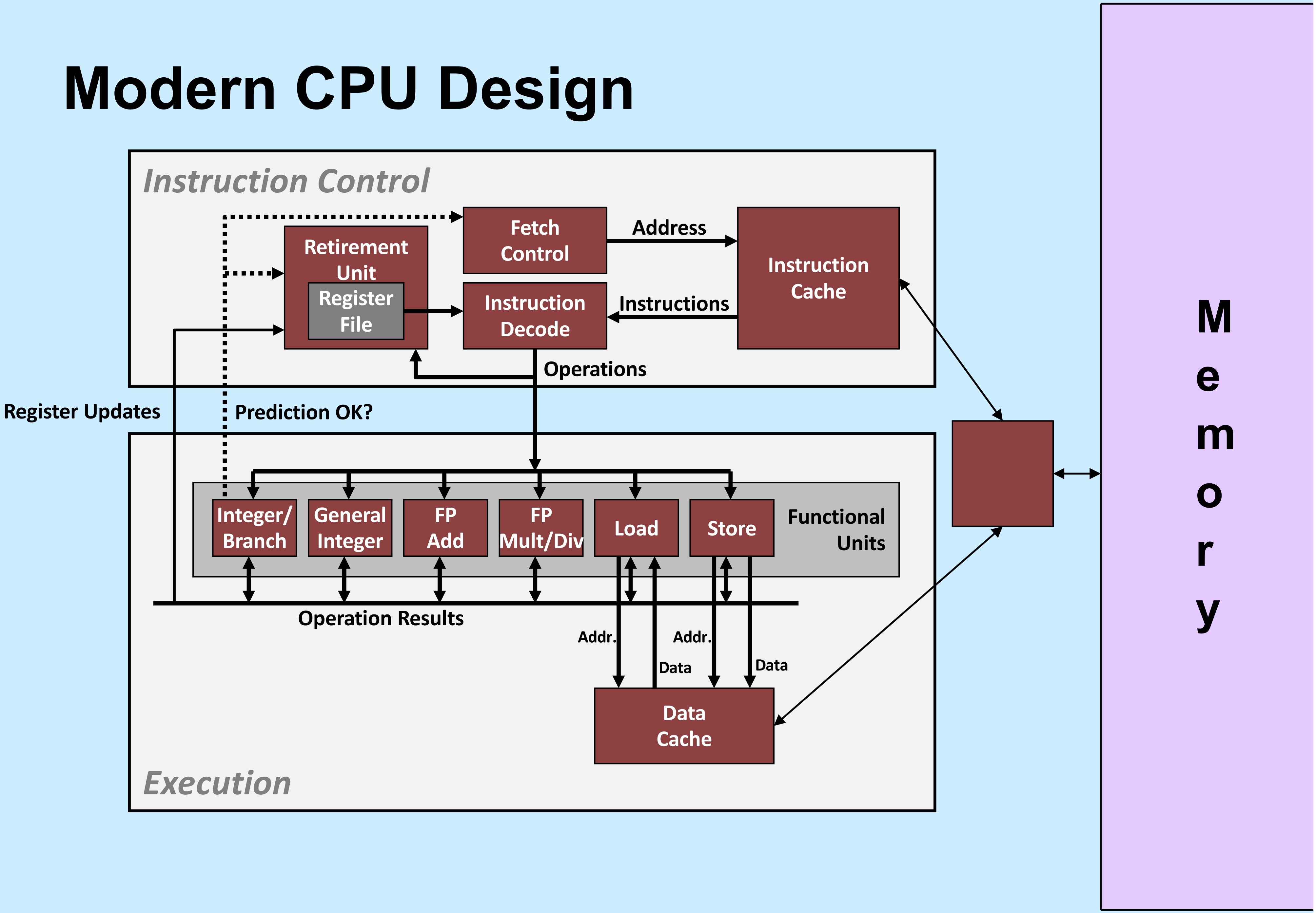
Control Hazards

```
   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 CPU Design



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

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

Haswell CPU Performance Bounds

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

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];
}
```

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
```

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
```

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;
```

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
```

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
```


Results: Long

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

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

Results: Double

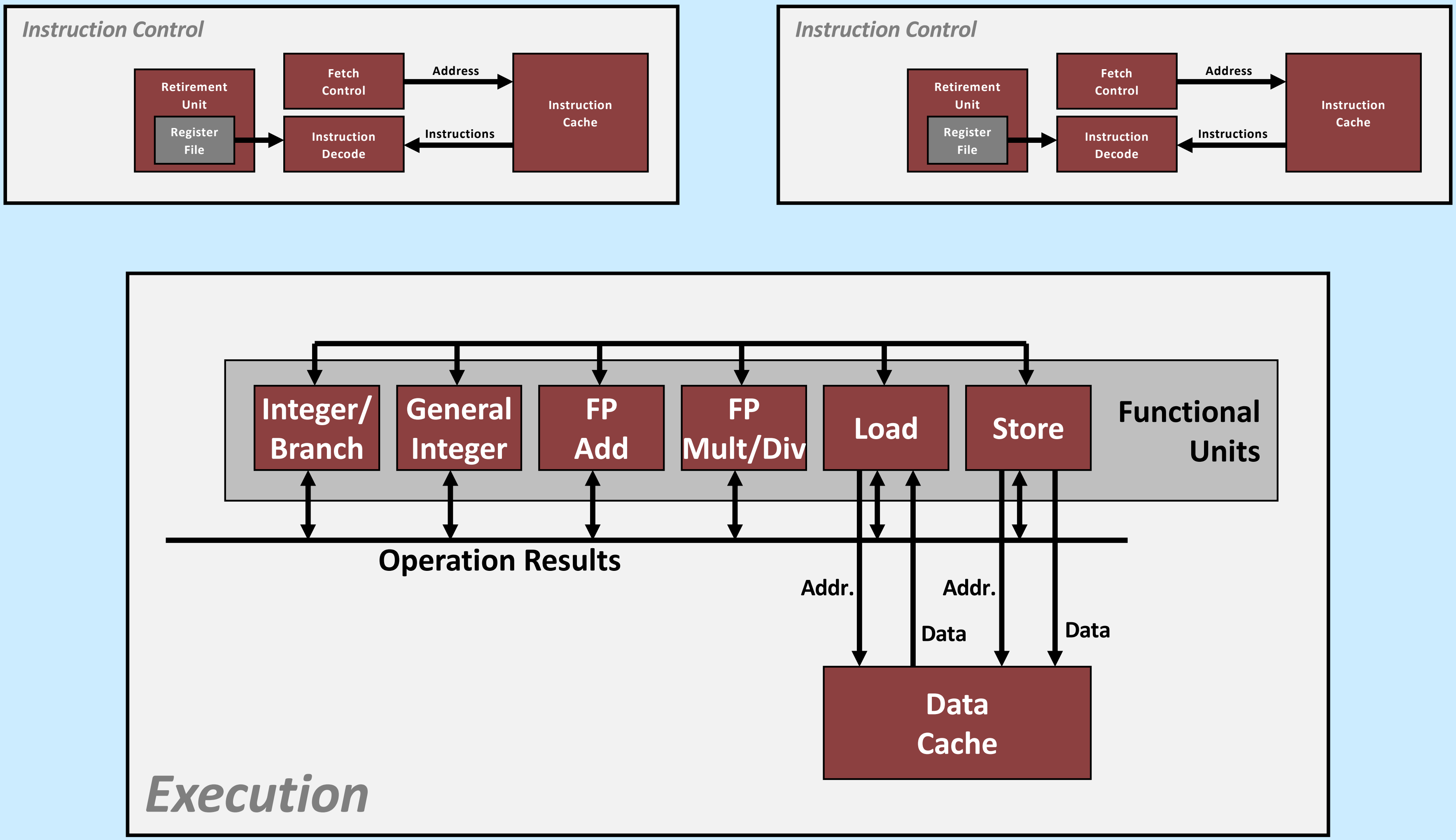
```
$ ./arraySumDouble
sum = 144115187606093856.000000
CPU time = 462777 microseconds

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

Quiz 2

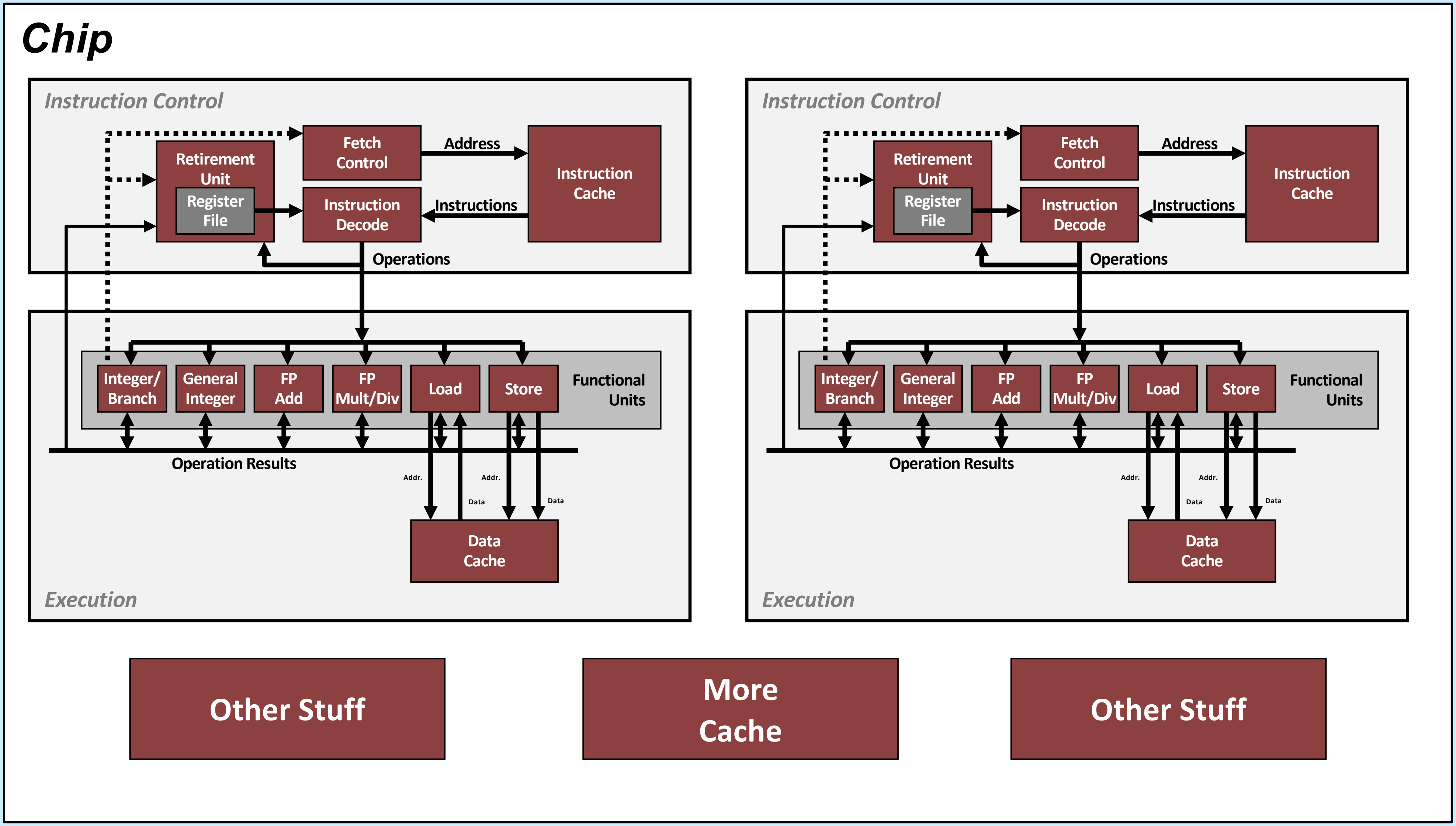
The sums given in the previous slide are different for the two cases. Why is this?

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



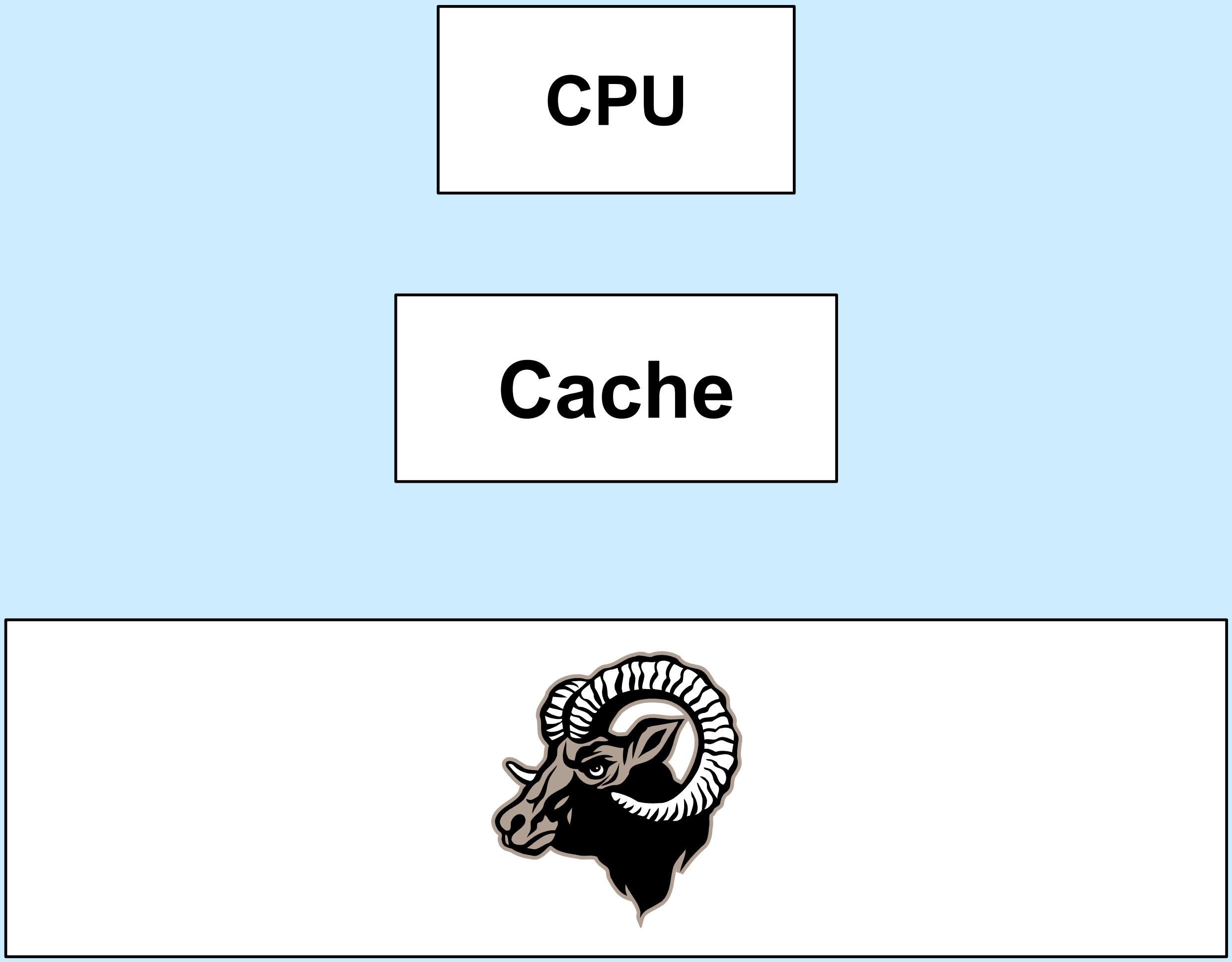
Multiple Cores



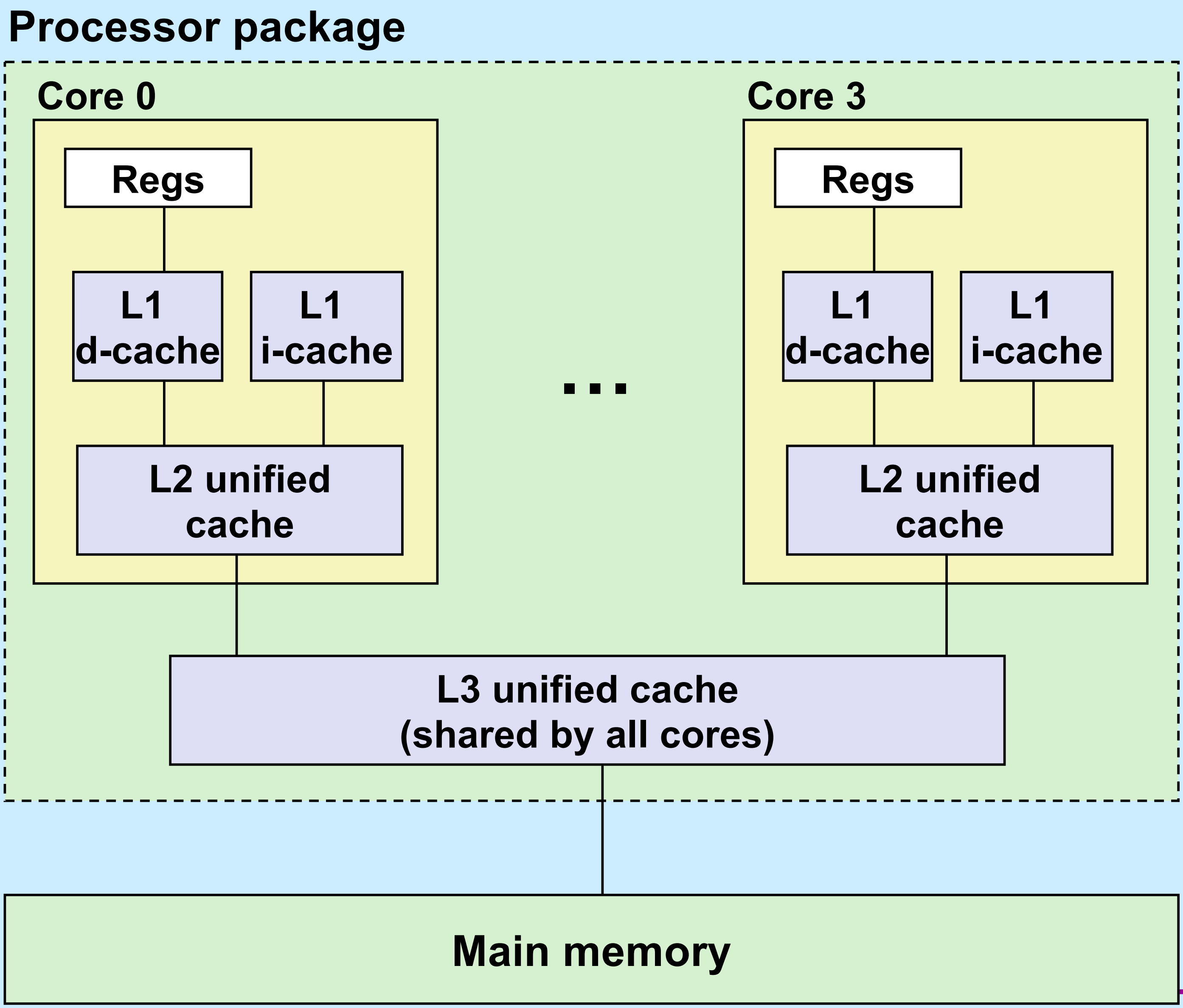
A Mismatch

- 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



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

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

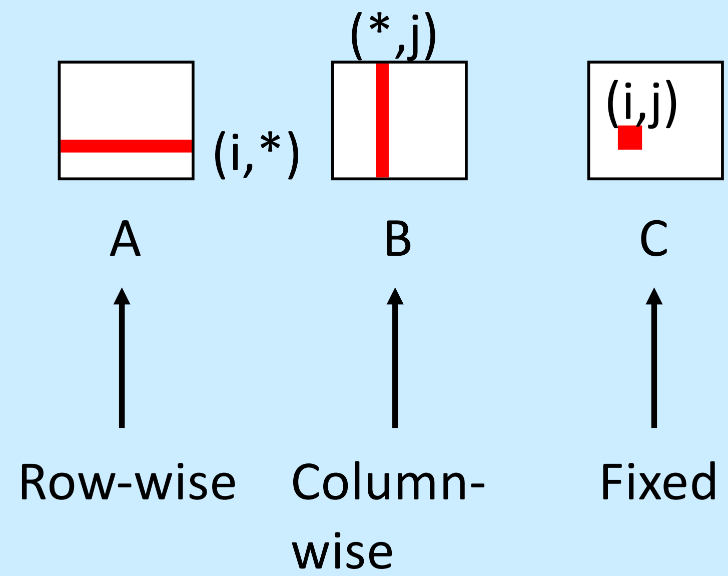
Layout of C Matrices in Memory

- **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**

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;
  }
}
```

Inner loop:

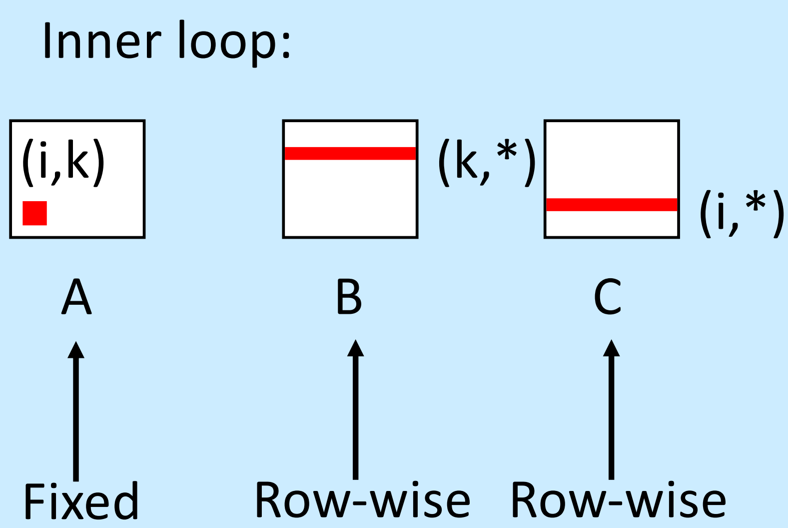


Misses per inner loop iteration:

A	B	C
0.125	1.0	0.0

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];
  }
}
```

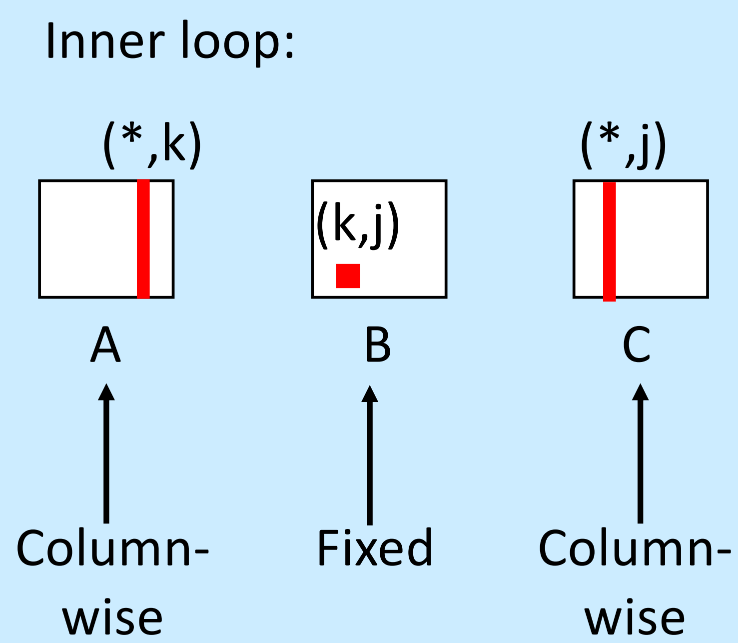


Misses per inner loop iteration:

A	B	C
0.0	0.125	0.125

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;
  }
}
```



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

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**

In Real Life ...

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