21. Parallel Processing

6.004x Computation Structures
Part 3 – Computer Organization

Copyright © 2016 MIT EECS

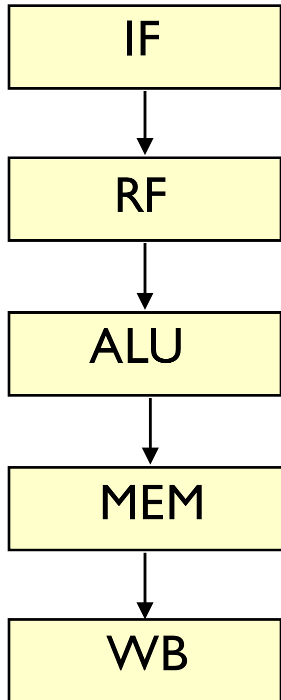
Instruction-level Parallelism

Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \; \underbrace{\frac{\text{Cycles}}{\text{Instruction}}}_{\text{CPI}} \; \underbrace{\frac{\text{Time}}{\text{Cycle}}}_{t_{CLK}}$$

- Pipelining lowers t_{CLK}. What about CPI?
- CPI = CPI_{ideal} + CPI_{stall}
  - CPI_{ideal}: cycles per instruction if no stall
- CPI_{stall} contributors
  - Data hazards
  - Control hazards: branches, exceptions
  - Memory latency: cache misses

5-Stage Pipelined Processors



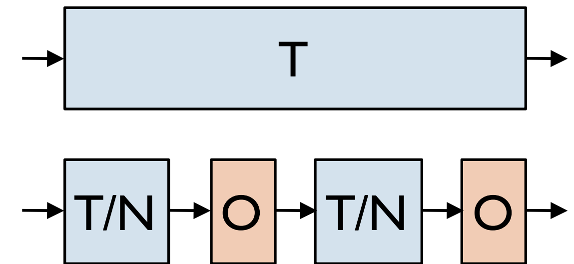
- Advantages
  - CPI_{ideal} is 1 (pipelining)
  - Simple, elegant
    - Still used in ARM & MIPS processors
- Room for improvement
  - Upper performance bound is CPI=1
  - High-latency instructions not handled well
    - 1 stage for accesses to large caches or multiplier
    - Long clock cycle time
  - Unnecessary stalls due to rigid pipeline
    - If one instruction stalls, anything behind it stalls

Improving 5-stage Pipeline Performance

- Lower t_{CLK}: deeper pipelines
  - Overlap more instructions

Limits to Pipeline Depth

- Each pipeline stage introduces some overhead (O)
  - Propagation delay of pipeline registers
  - Setup and hold times
  - Clock skew
  - Inequalities in work per stage
    - Cannot break up work into stages at arbitrary points



- If original t_{CLK} was T, with N stages t_{CLK} is T/N+O
  - If $N \rightarrow \infty$, speedup = T / (T/N+O) $\rightarrow$ T/O
    - Assuming that CPI stays constant
  - Eventually overhead dominates and deeper pipelines have diminishing returns

Improving 5-stage Pipeline Performance

- Lower t_{CLK}: deeper pipelines
  - Overlap more instructions
- Higher CPI_{ideal}: wider pipelines
  - Each pipeline stage processes multiple instructions
- Lower CPI_{stall}: out-of-order execution
  - Execute each instruction as soon as its source operands are available
- Balance conflicting goals
  - Deeper & wider pipelines $\Rightarrow$ more control hazards
  - Branch prediction

- It all works because of instruction-level parallelism (ILP)

Instruction Level Parallelism (ILP)

Sequential Code

```
loop:
    LD(n, r1)
    CMPLT(r31, r1, r2)
    BF(r2, done)
    LD(r, r2)
    LD(n,r1)
    MUL(r1, r2, r3)
    ST(r3, r)
    LD(n,r4)
    SUBC(r4, 1, r4)
    ST(r4, n)
    BR(loop)
done:
```

$$r = \prod_{i=1}^{n} i$$

"Safe" Parallel Code

```
loop:

    LD(n, r1)

    CMPLT(r31, r1, r2)

    BF(r2, done)
    ----------------------------------------
    LD(r, r2)  LD(n,r1)   LD(n,r4)

    MUL(r1, r2, r3)  SUBC(r4, 1, r4)

    ST(r3, r)  ST(r4, n)  BR(loop)

done:
```

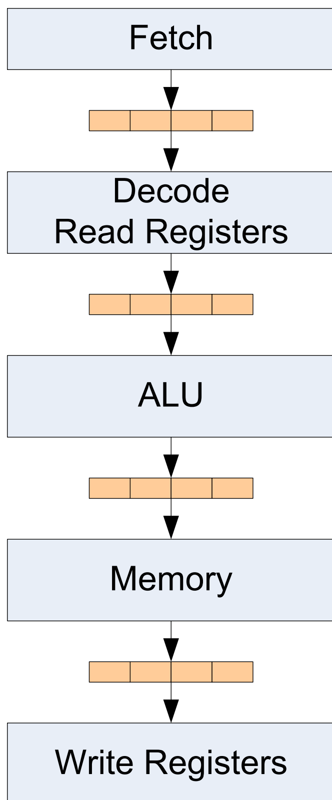
Read-after-write

Write-after-write

Write-after-read

These last two can be solved with renaming, i.e., giving each result a unique register name.

Wider or Superscalar Pipelines

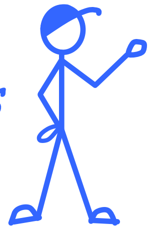


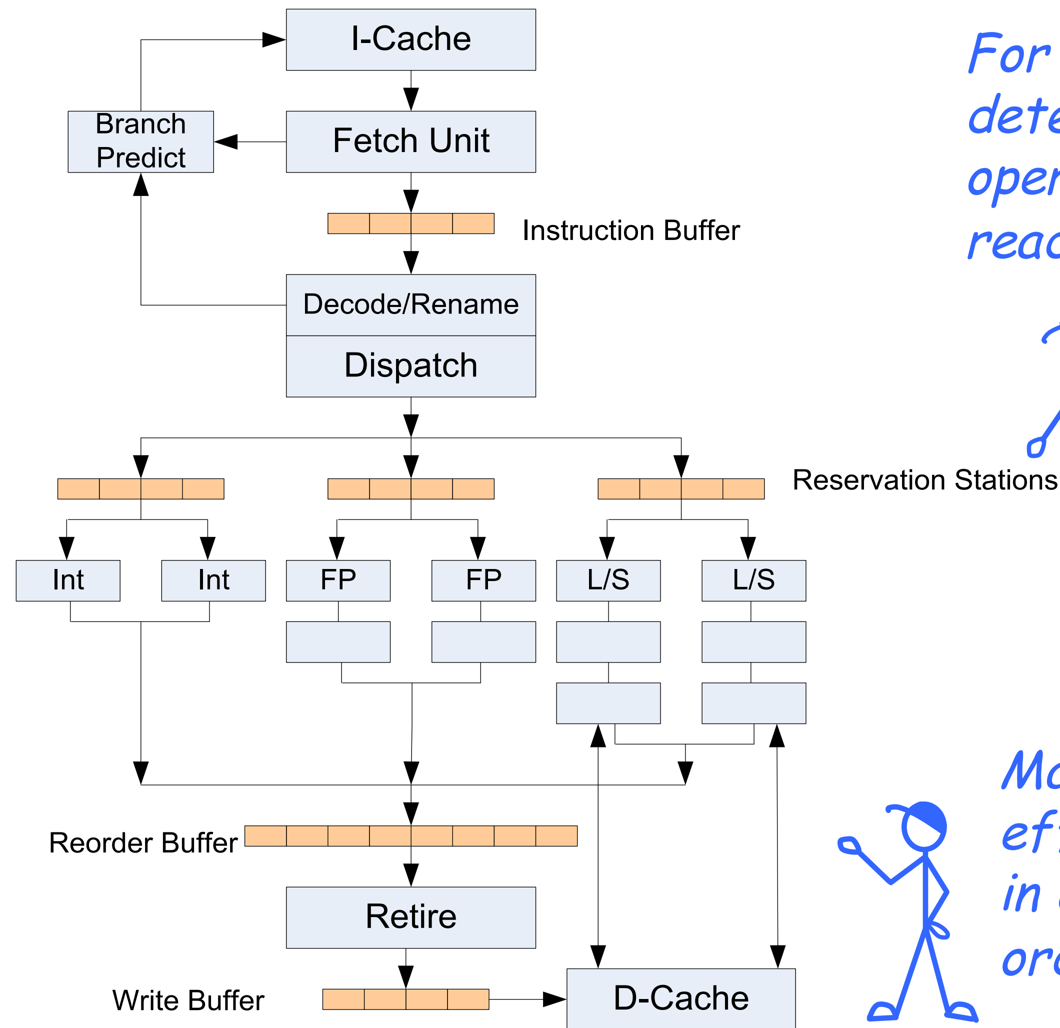
- Each stage operates on up to N instructions each clock cycle
  - Known as wide or superscalar pipelines
  - $CPI_{ideal} = 1/N$
- Options (from simpler to harder)
  - One integer and one floating-point instruction
  - Any N=2 instructions
  - Any N=4 instructions
  - Any N=? Instructions
    - What are the limits?

See http://people.ee.duke.edu/~sorin/ece252/lectures/3-superscalar.pdf

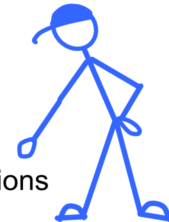
A Modern Out-of-Order Superscalar Processor

Needed to avoid high CPI_{STALL} on deep pipelines

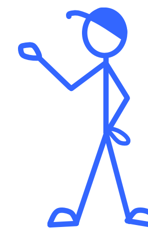




For OoO: determine when operands are ready for inst.



Make sure side-effects happen in correct order!



Limits To Single-Processor Performance

- Pipeline depth: getting close to pipelining limits
  - Clocking overheads, CPI degradation
- Branch prediction & memory latency limit the practical benefits of out-of-order execution
- Power grows superlinearly with higher frequency & more OoO logic
- Extreme design complexity

- Limited ILP → Must exploit DLP and TLP
  - Data-Level Parallelism: Vector extensions, GPUs
  - Thread-Level Parallelism: Multiple threads and cores

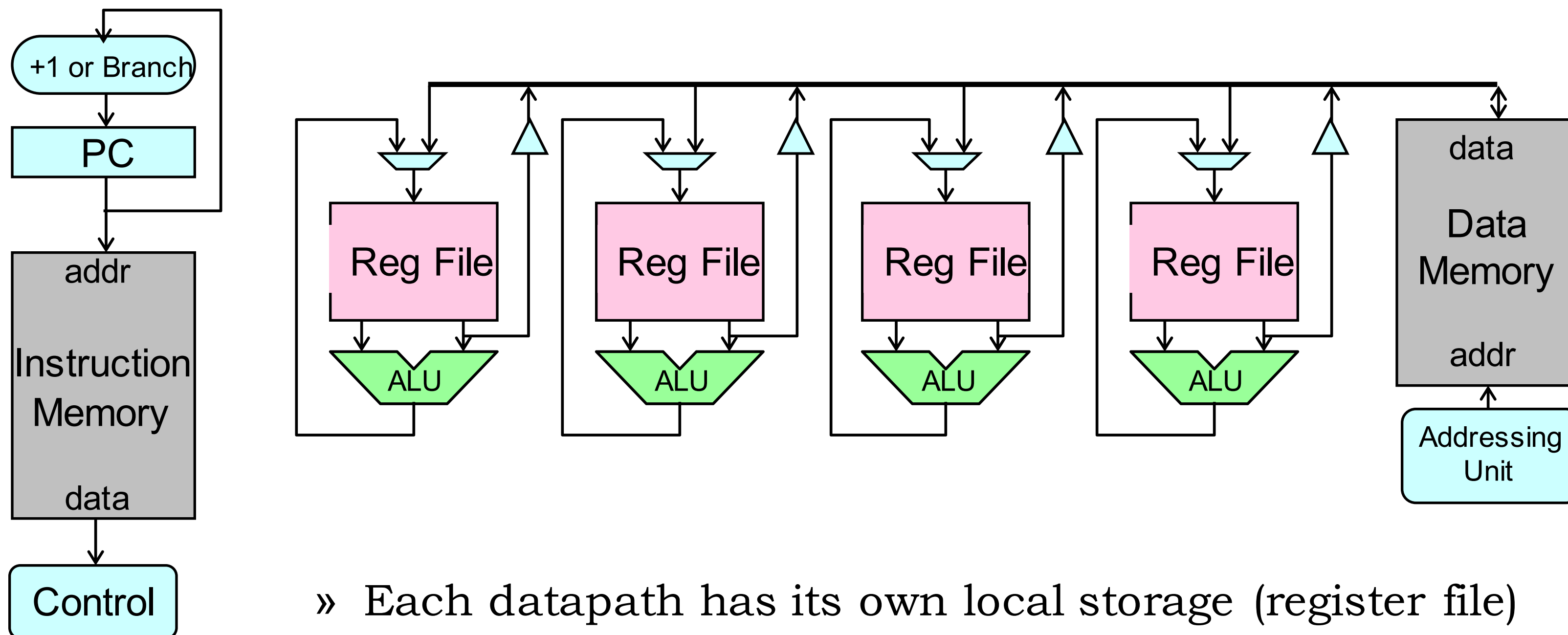
Data-level Parallelism

Data-Level Parallelism

- Same operation applied to multiple data elements

```
for (int i = 0; i < 16; i++) x[i] = a[i] + b[i];
```

- Exploit with vector processors or vector ISA extensions



» Each datapath has its own local storage (register file)

» All datapaths execute the same instruction

» Memory access with vector loads and stores + wide memory port

Vector Code Example

```
for (i = 0; i < 16; i++)  x[i] = a[i] + b[i];
```

Beta assembly

```
      CMOVE(16, R0)
loop: LD(R1, 0, R4)
      LD(R2, 0, R5)
      ADDC(R1, 4, R1)
      ADDC(R2, 4, R2)
      ADD(R4, R5, R6)
      ST(R6, 0, R3)
      ADDC(R3, 4, R3)
      SUBC(R0, 1, R1)
      BNE(R0, loop)
```

# of cycles = 1 + 10*15 + 9 = 160

Equivalent vector assembly

```
LD.V(R1, 0, V1)
LD.V(R2, 0, V2)
ADD.V(V1, V2, V3)
ST.V(V3, 0, R3)
```

# of cycles = 4

Data-dependent Vector Operations

```
for (i = 0; i < 16; i++)
   if (a[i] < b[i]) c[i] = c[i] + 3;
```

Equivalent vector assembly

```
LD.V(R1, 0, V1)  // load a[i]
LD.V(R2, 0, V2)  // load b[i]
LD.V(R3, 0, V3)  // load c[i]
CMPLT.V(V1, V2)  // set local predicate flags

// predicated instructions perform the
// indicated operation if the local predicate
// flag istrue or isfalse.
ADDC.V.iftrue(V3, 3, V3)
```

Vector Processing Implementations

- Advantages of vector ISAs:
  - Compact: 1 instruction defines N operations
  - Parallel: N operations are (data) parallel and independent
  - Expressive: Memory operations describe regular patterns

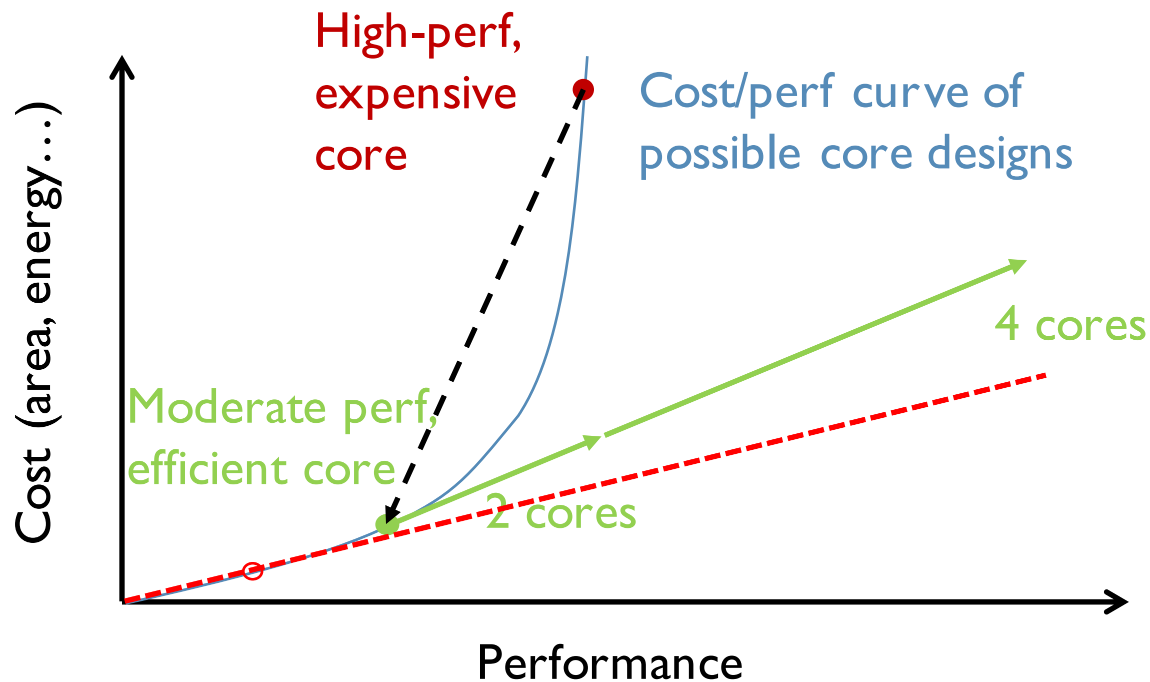
- Modern CPUs: Vector extensions & wider registers
  - SSE: 128-bit operands (4x32-bit or 2x64-bit)
  - AVX (2011): 256-bit operands (8x32-bit or 4x64-bit)
  - AVX-512 (upcoming): 512-bit operands
  - Explicit parallelism, extracted at compile time (vectorization)

- GPUs: Designed for data parallelism from the ground up
  - 32 to 64 32-bit floating-point elements
  - Implicit parallelism, scalar binary with multiple instances executed in lockstep (and regrouped dynamically)

Thread-level Parallelism

Multicore Processors

If applications have a lot of parallelism, using a larger number of simpler cores is more efficient!



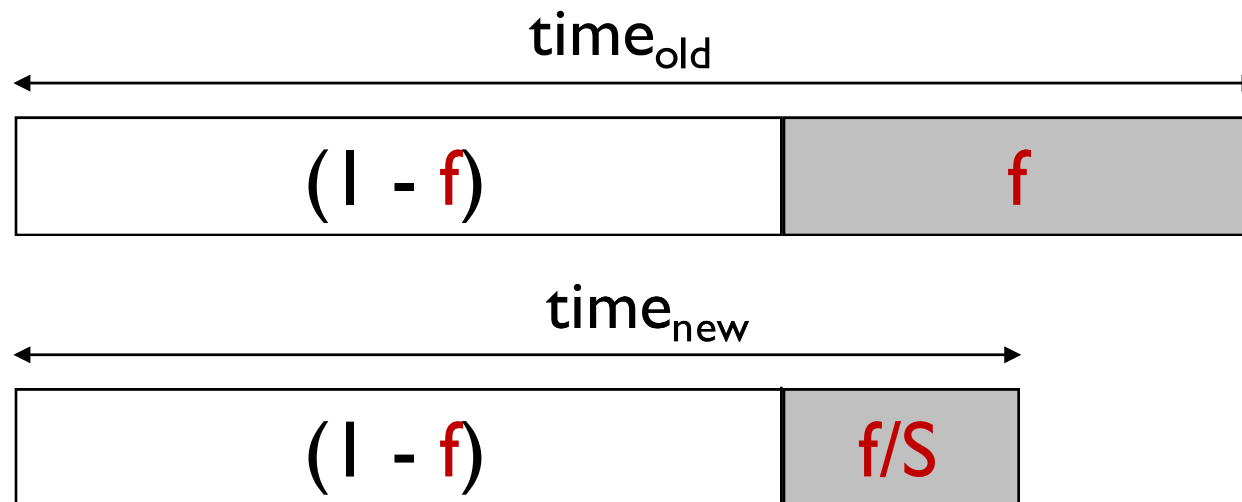
What is the optimal tradeoff between core cost and number of cores?

Amdahl's Law

- Speedup= $\text{time}_{\text{without enhancement}}$ / $\text{time}_{\text{with enhancement}}$
- Suppose an enhancement speeds up a fraction f of a task by a factor of S

$$\text{time}_{\text{new}} = \text{time}_{\text{old}} \cdot ((1-f) + f/S)$$

$$S_{\text{overall}} = \text{time}_{\text{old}} / \text{time}_{\text{new}} = 1 / ((1-f) + f/S)$$



Corollary: Make the common case fast

Amdahl's Law and Parallelism

What is the maximum speedup you can get by running on a multicore machine?

$$S_{overall} = 1 \,/\, \left((1-f) + f/S \right)$$

$$S_{overall} \xrightarrow[S\rightarrow\infty]{\lim} 1 \,/\, (1-f)$$

Say you write a program that can do 90% of the work in parallel, but the other 10% is sequential

$$f = 0.9,\ S{=}\infty \rightarrow S_{overall} = 10$$

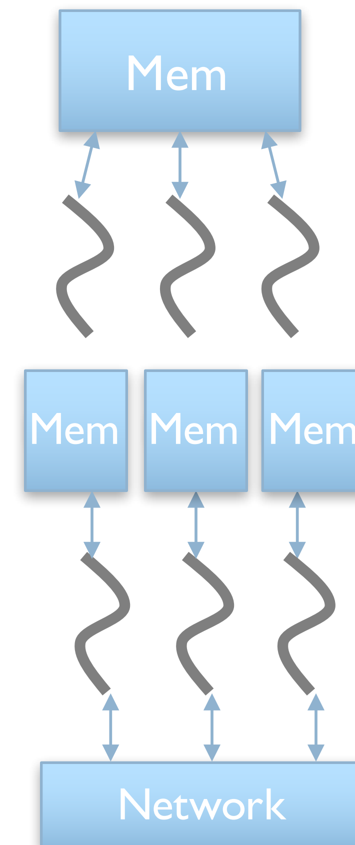
What f do you need to use a 1000-core machine well?

$$S_{overall} = 500 \rightarrow f = 0.998$$

Thread-Level Parallelism

- Divide computation among multiple threads of execution
  - Each thread executes a different instruction stream
  - More flexible than vector processing, but more expensive

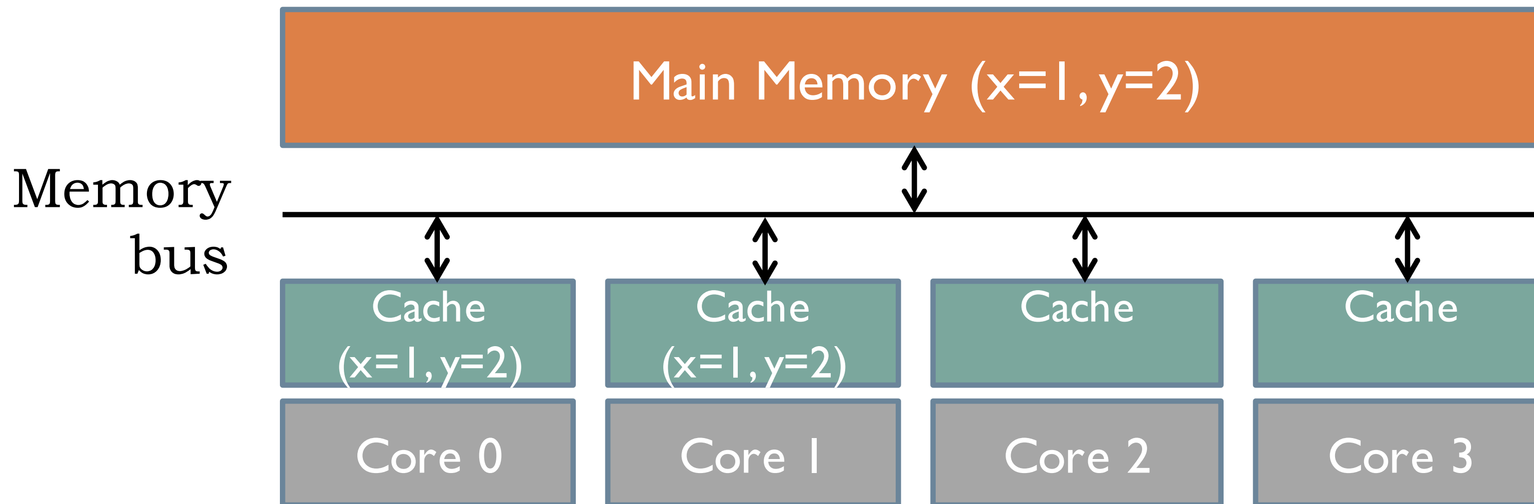
- Communication models:
  - Shared memory:
    - Single address space
    - Implicit communication by memory loads & stores
  - Message passing:
    - Separate address spaces
    - Explicit communication by sending and receiving messages



Shared Memory & Caches

Multicore Caches

- Multicores have multiple private caches for performance
- We want the semantics of a single shared memory



Consider the following trivial threads running on C_0 and C_1:

<u>Thread A (C_0)</u>

```
x = 3;
print(y);
```

<u>Thread B (C_1)</u>

```
y = 4;
print(x);
```

What Are the Possible Outcomes?

Thread A

```
x = 3;
print(y);
```

$\$_1$: x = ~~1~~ **3**
y = 2

Thread B

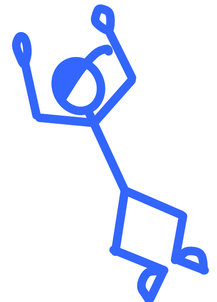
```
y = 4;
print(x);
```

$\$_2$: x = 1
y = ~~2~~ **4**

Plausible execution sequences:

SEQUENCE	A prints	B prints
x=3; print(y); y=4; print(x);	2	1
x=3; y=4; print(y); print(x);	2	1
x=3; y=4; print(x); print(y);	2	1
y=4; x=3; print(x); print(y);	2	1
y=4; x=3; print(y); print(x);	2	1
y=4; print(x); x=3; print(y);	2	1

Hey, we get the same answer every time… Let's go build it!



Uniprocessor Outcome

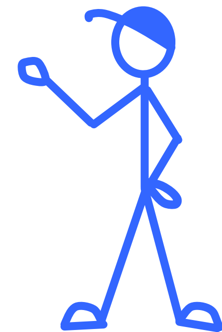
But, what are the possible outcomes if we ran Thread A and Thread B on a single timed-shared processor?

Thread A	Thread B
x = 3; print(y);	y = 4; print(x);

Plausible Uniprocessor execution sequences:

SEQUENCE	A prints	B prints
x=3; print(y); y=4; print(x);	2	3
x=3; y=4; print(y); print(x);	4	3
x=3; y=4; print(x); print(y);	4	3
y=4; x=3; print(x); print(y);	4	3
y=4; x=3; print(y); print(x);	4	3
y=4; print(x); x=3; print(y);	4	1

Notice that the outcome 2, 1 does not appear in this list!



Sequential Consistency

Semantic constraint:

Result of executing N parallel threads should correspond to *some* interleaved execution on a single processor.

Shared Memory

```
int x=1, y=2;
```

Thread A

```
x = 3;
print(y);
```

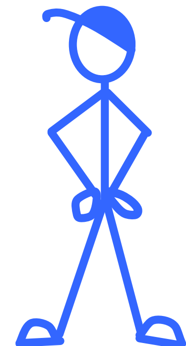
Thread B

```
y = 4;
print(x);
```

Possible printed values: 2, 3;   4, 3;   4, 1.
(each corresponds to at least one interleaved execution)

IMPOSSIBLE printed values: 2, 1
(corresponds to NO valid interleaved execution).

Weren't caches supposed to be invisible to programs?



Alternatives to Sequential Consistency?

ALTERNATIVE MEMORY SEMANTICS:

"WEAK" consistency

EASIER GOAL: Memory operations from each thread appear to be performed in order issued by that thread ;

Memory operations from different threads may overlap in arbitrary ways (not necessarily consistent with any interleaving).

ALTERNATIVE APPROACH:

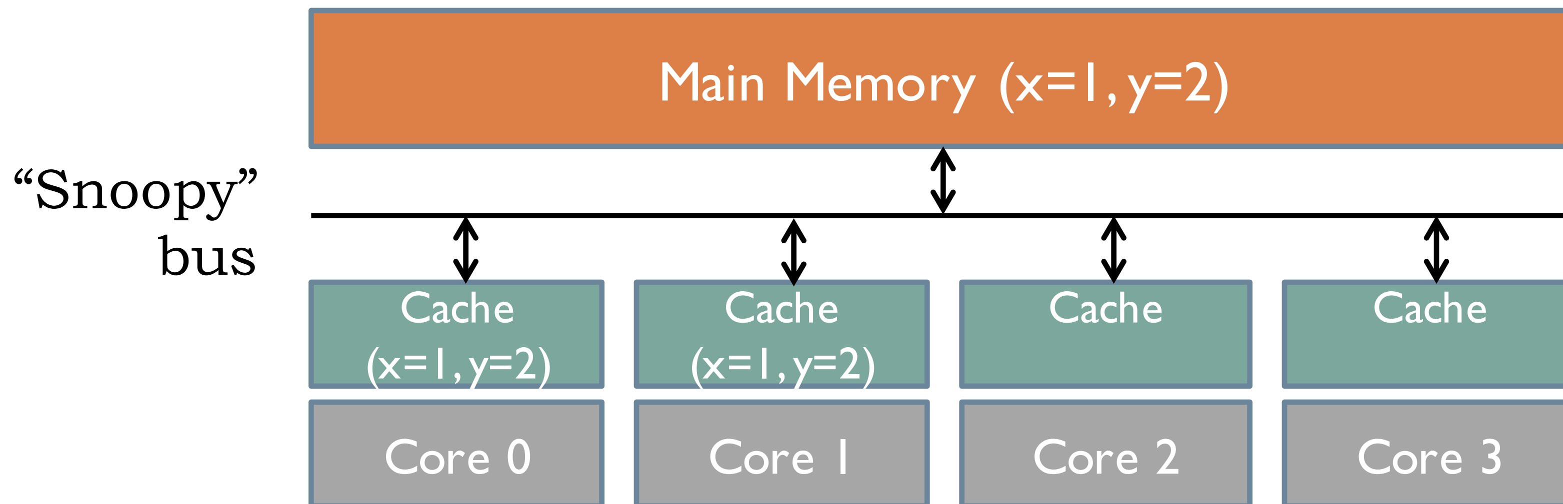
- Weak consistency, by default;
- MEMORY BARRIER instruction: stalls thread until all previous memory operations have completed.

See http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf for a very readable discussion of memory semantics in multicore systems.

Cache Coherence

Fix: “Snoopy” Cache Coherence Protocol

Idea: Have caches communicate over shared bus, letting other caches know when a shared cached value changes



Goal: minimize contention for snoopy bus by communicating only when necessary, i.e., when there’s a shared value.

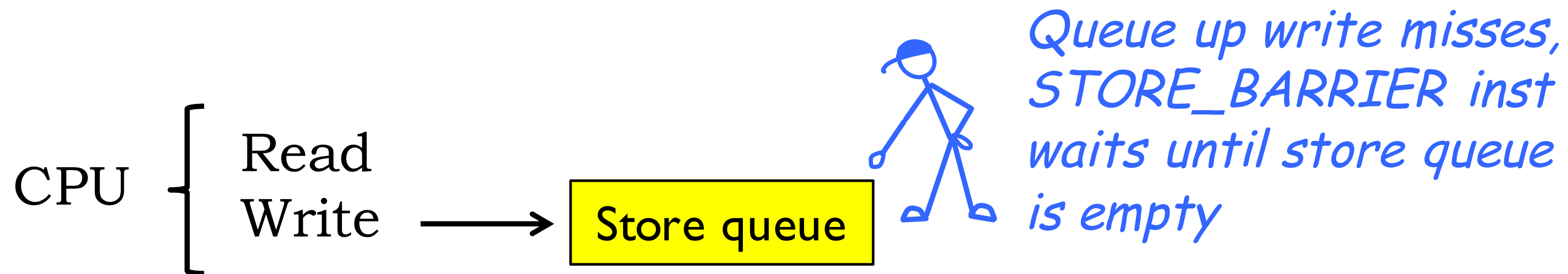
Example: MESI Cache Coherence Protocol

Cache line:

- ***Modified*** The cache line is present only in the current cache, and is dirty; it has been modified from the value in main memory. The cache is required to write the data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state.
- ***Exclusive*** The cache line is present only in the current cache, but is clean; it matches main memory. It may be changed to the Shared state at any time, in response to a bus read request. Alternatively, it may be changed to the Modified state when writing to it.
- ***Shared*** Indicates that this cache line may be stored in other caches of the machine and is clean; it matches the main memory. The line may be discarded (changed to the Invalid state) at any time. Writes to SHARED cache lines get special handling…
- ***Invalid*** Indicates that this cache line is invalid (unused).

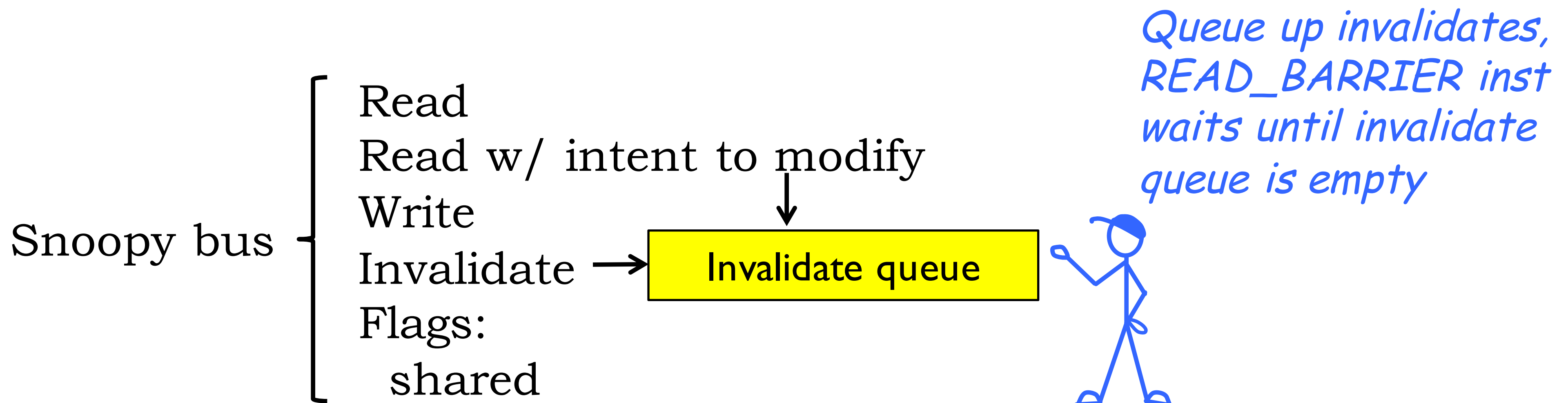
https://en.wikipedia.org/wiki/MESI_protocol

The Cache Has Two Customers!

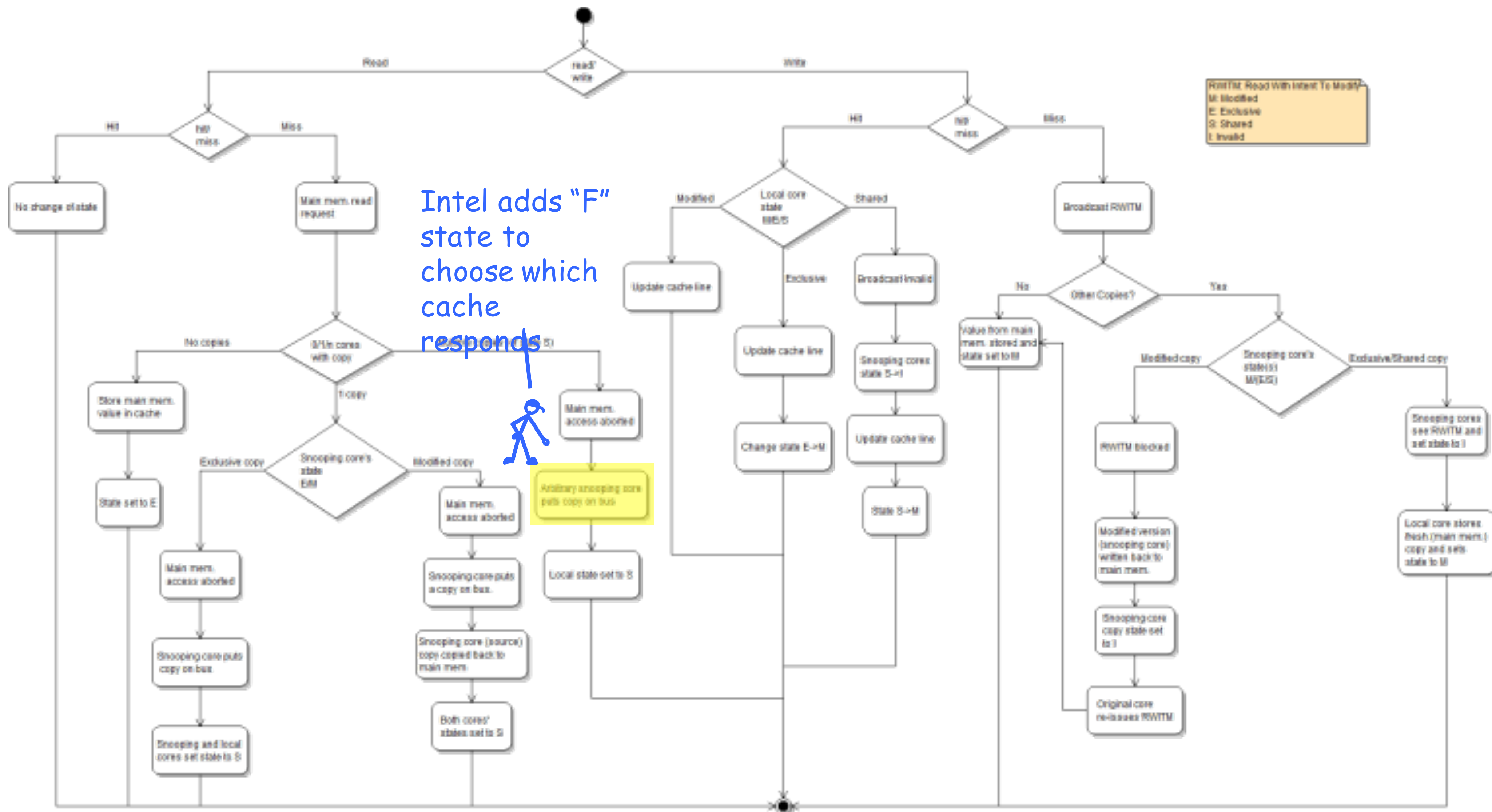


Cache line:

State	Tag	Data



MESI Activity Diagram



CC0: https://en.wikipedia.org/wiki/MESI_protocol#/media/File:MESI_protocol_activity_diagram.png

Cache Coherence in Action

Thread A	Thread B
❶ x = 3;	❷ y = 4;
❹ print(y);	❸ print(x);
$\$_0$: [S] x =1, [S] y = 2	$\$_1$: [S] x =1, [S] y = 2

1. x = 3 → $\$_0$ sends invalidate x, update cache

$\$_0$: [M] x =3, [S] y = 2	$\$_1$: [S] y = 2

2. y = 4 → $\$_1$ sends invalidate y, update cache

$\$_0$: [M] x =3	$\$_1$: [M] y = 4

3. print(x) → $\$_1$ read x, $\$_0$ responds with Shared flag, update mem

$\$_0$: [S] x =3	$\$_1$: [S] x = 3, [M] y = 4

4. print(y) → $\$_0$ read y, $\$_1$ responds with Shared flag, update mem

$\$_0$: [S] x =3, [S] y = 4	$\$_1$: [S] x = 3, [S] y = 4

Parallel Processing Summary

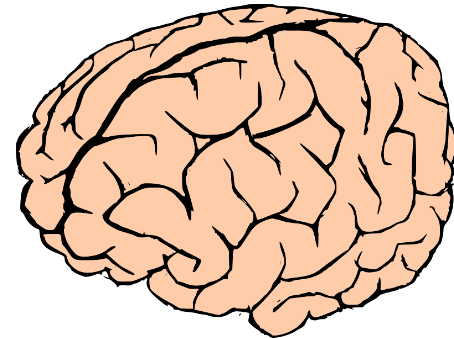
Prospects for future CPU architectures:

Pipelining - Well understood, but mined-out
Superscalar - At its practical limits
Vector/GPU - Useful for special applications

Prospects for future Computer System architectures:

Single-thread limits: forcing multicores, parallelism

Brains work well, with dismal clock rates ... parallelism?

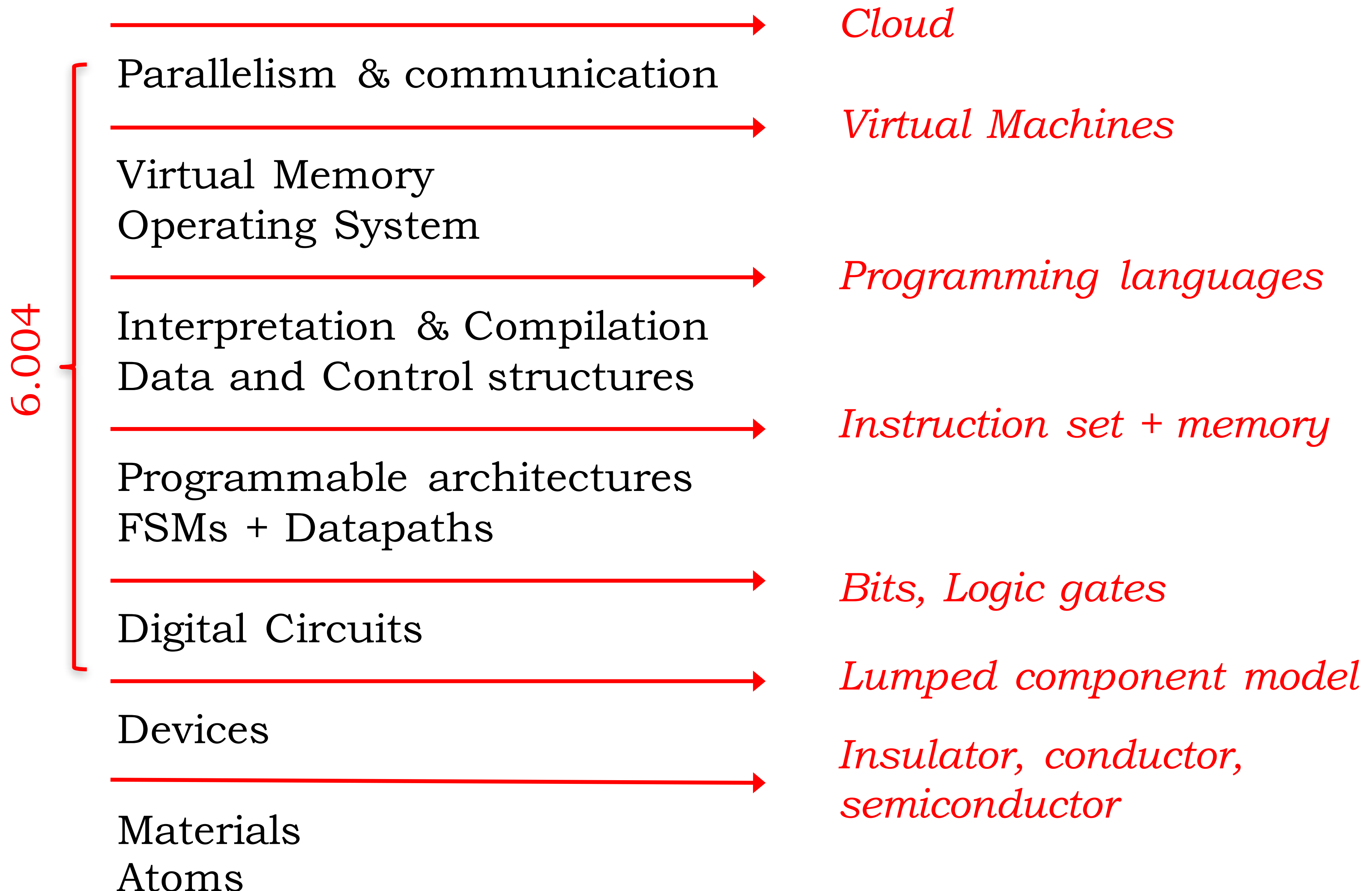


Needed: NEW models, NEW ideas, NEW approaches

FINAL ANSWER: It's up to YOUR generation!

6.004 Wrap-up

From Atoms to Amazon



The Power of Engineering Abstractions

Good abstractions allow us to reason about behavior while shielding us from the details of the implementation.

Corollary: implementation technologies can evolve while preserving the engineering investment at higher levels.

Leads to hierarchical design:

- Limited complexity at each level ⇒ shorten design time, easier to verify
- Reusable building blocks

Cloud

Virtual Machines

Programming languages

Instruction set + memory

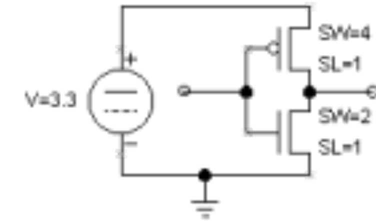
Bits, Logic gates

Lumped component model

Insulator, conductor, semiconductor

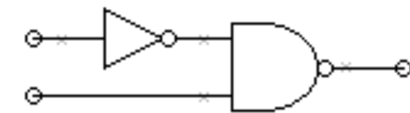
6.004: The Big Lesson

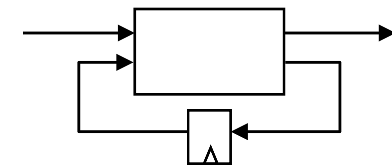
You've built, debugged, understood a complex computer from FETs to OS… what have you learned?



Engineering Abstractions:

- Understanding of their technical underpinnings
- Respect for their value
- Techniques for using them



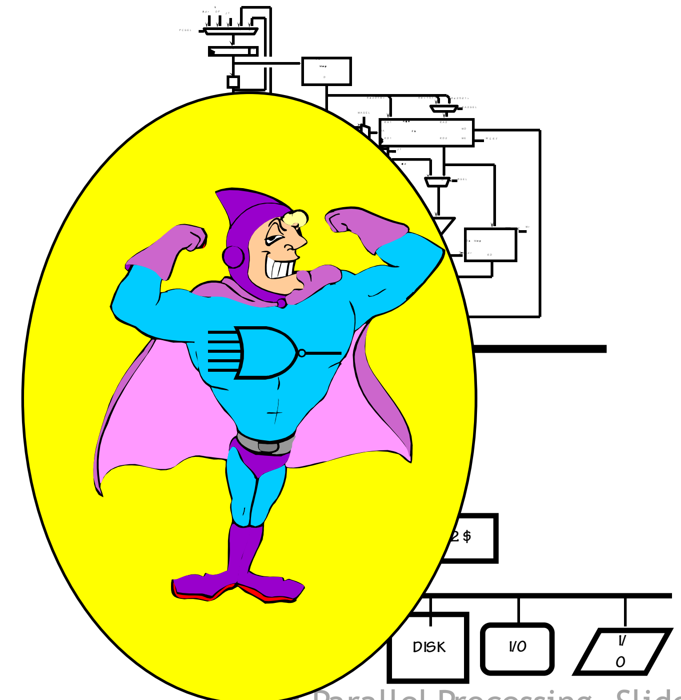


But, most importantly:

The self assurance to discard them, in favor of new abstractions!

Good engineers *use* abstractions;

GREAT engineers *create* them!



Things to look forward to…

6.004 is only an appetizer!

Processors
- Superscalars
- Deep pipelines
- Multicores

Languages & Models
- Python/Java/Ruby/…
- Objects/Streams/Aspects
- Networking

Tools
- Design Languages
- FPGA prototyping
- Timing Analyzers

Algorithms
- Arithmetic
- Signal Processing
- Language implementation

Systems Software
- Storage
- Virtual Machines
- Networking

Thinking Outside the Box

Will computers always look and operate the way computers do today?

Some things to question:

- Well-defined system "state"
- Silicon-based logic
- Logic at all
- Programming

Von Neumann Architectures

Synchronous Clocked Systems

Si

MOSFET transistors

Boolean Logic

THE END!

Computing is slow…
The future is in your hands.
Start innovating!
-- 6.004 Staff

The only problem
with Haiku is that you just
get started and then
-- Roger McGough