

Multi-Core & Parallel Processing

I've gotta spend at least
10 hours studying for
the Comp 411 final!

I'm going to study with 9
friends... we'll be done in
an hour.



Chapter 9 - available on disk

TIPS Anyone?

I guess that means
that there are 10^{12}
microphones in a
Megaphone?



$$\text{MIPS} = \frac{\text{Clock Frequency (in MHz)}}{\text{Clocks per Instruction}}$$

Mega – 10^6 Giga – 10^9 Tera – 10^{12} Peta – 10^{15}

Light travels about 1 ft / 10^{-9} secs in free space.

A Tera-Hertz uniprocessor could have no clock-to-clock path longer than 300 microns...

We already know of problems that require greater than a TIP
(Simulations of weather, weapons, brains)

Driving Down the Denominator

Techniques for increasing parallelism:

Pipelining – reasonable for a small number of stages (5-10), after that bypassing and stalls become unmanageable.

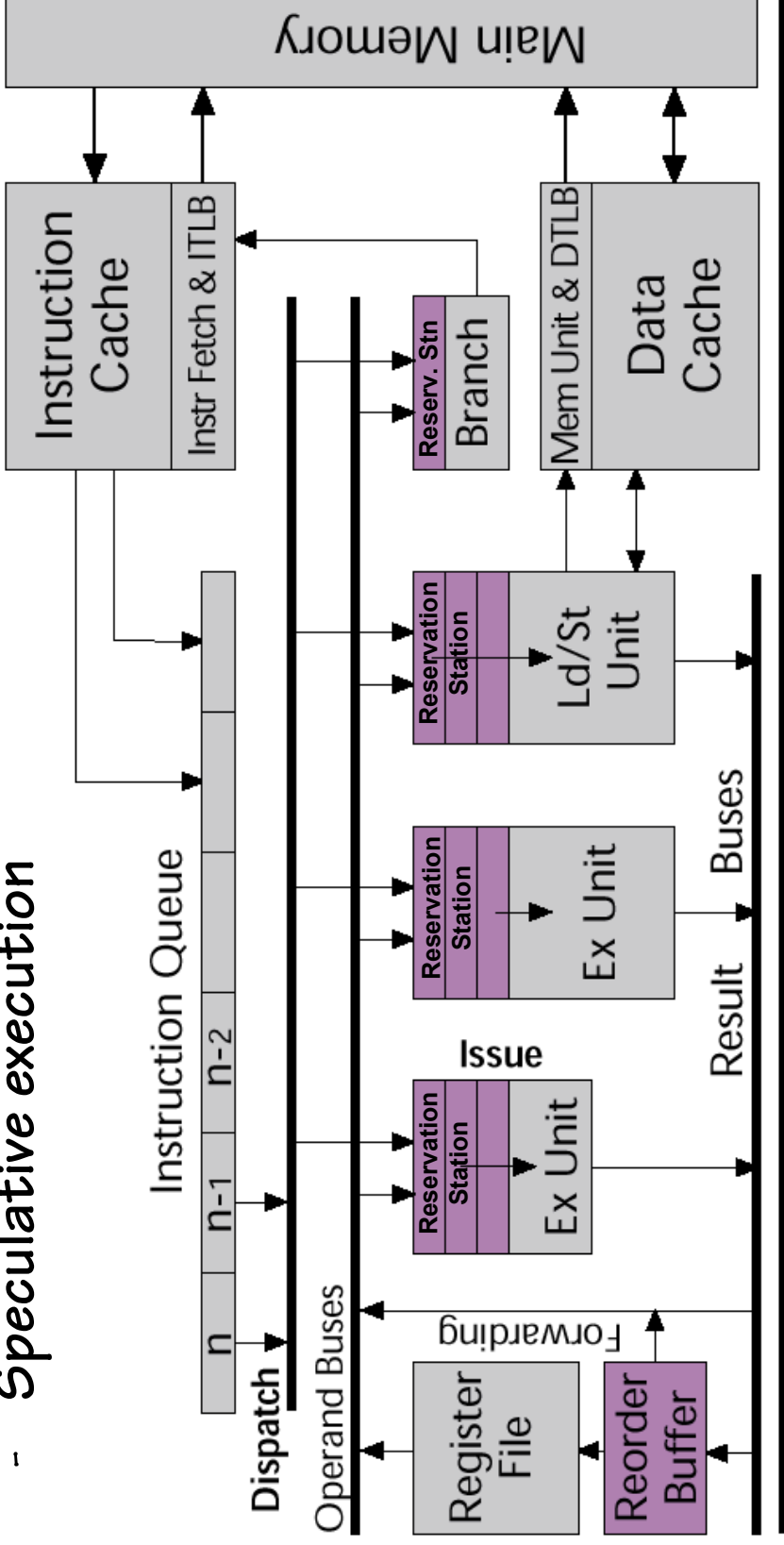
Superscalar – replicate data paths and design control logic to discover parallelism in traditional programs.

Explicit parallelism – must learn how to write programs that run on multiple CPUs.

Superscalar Parallelism

- Multiple Functional Units (ALUs, Addr units, etc)
- Multiple instruction dispatch
- Dynamic Pipeline Scheduling
- Speculative execution

Popular Now— but is the end near?



Explicit Parallelism

Control	Communication	Processing Elements
Unified	Shared Memory	Homogeneous
Distributed	Message Passing	Heterogeneous

Decoding the Parallel Processor Alphabet Soup:

SIMD - Single-Instruction-Multiple-Data

Unified control, Homogeneous processing elements

VLIW - Very-Long-Instruction-Word

Unified control, Heterogeneous processing elements

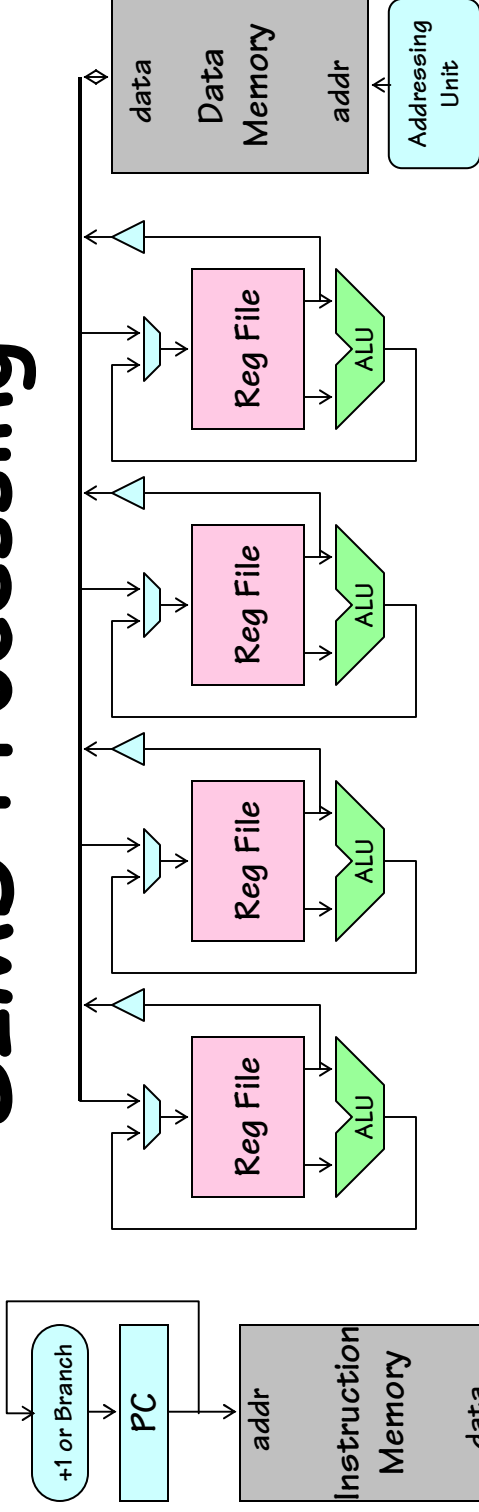
MIMD - Multiple-Instruction-Multiple-Data

Distributed control

SMP – Symmetric Multi-Processor

Distributed control, Shared memory, Homogenous PEs

SIMD Processing



Each datapath has its own local data (Register File)

All data paths execute the same instruction

Conditional branching is difficult...

(What if only one CPU has $R1 == \$0$?)

Conditional operations are common in SIMD machines

if (flag1) $Rc = Ra <op> Rb$

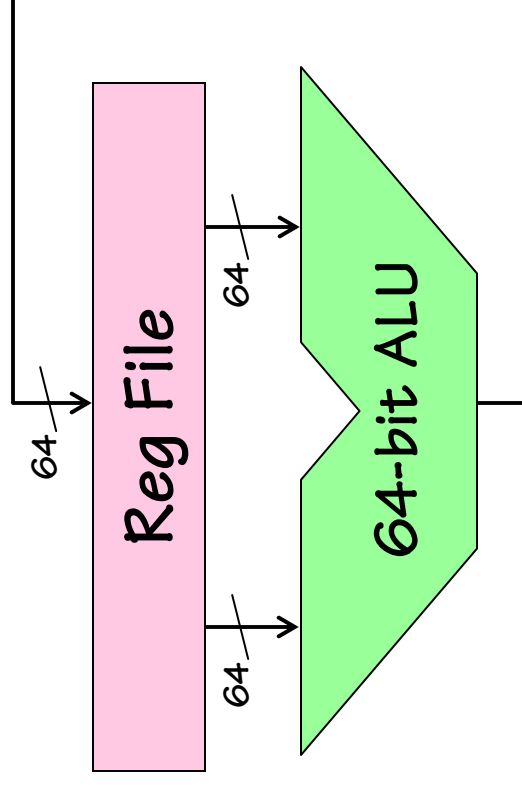
Global ANDing or ORing of flag registers are used for high-level control

This sort of construct is also becoming popular on modern uniprocessors



SIMD Coprocessing Units

“Intel MMX”



SIMD data path added to a traditional CPU core

Register-only operands

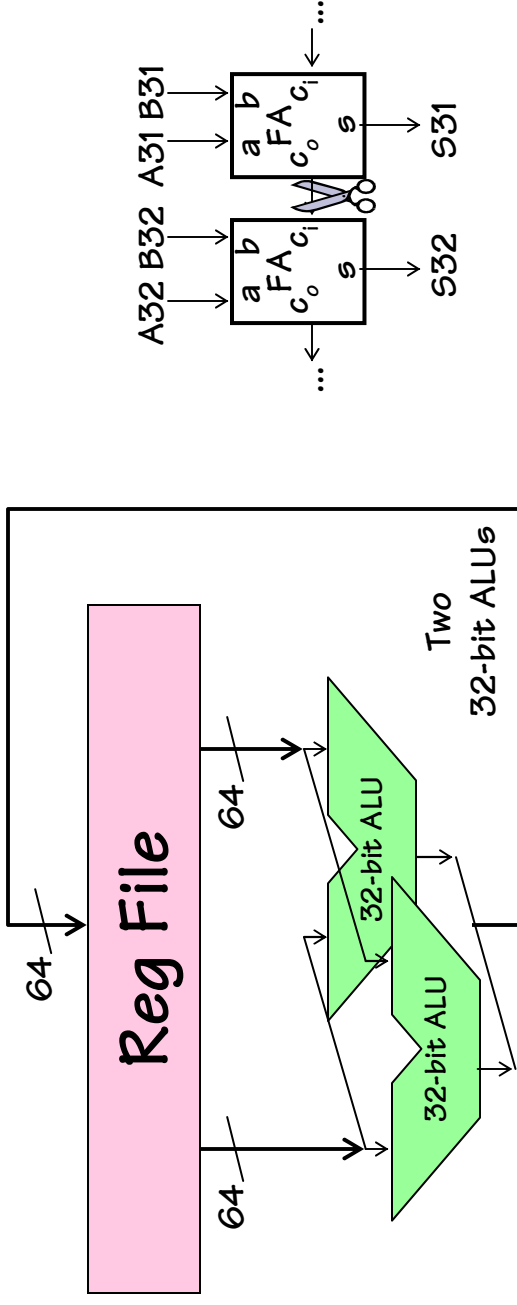
Core CPU handles memory traffic

Partitionable Datapaths for variable-sized

“PACKED OPERANDS”

SIMD Coprocessing Units

“Intel MMX”



SIMD data path added to a traditional CPU core

Register-only operands

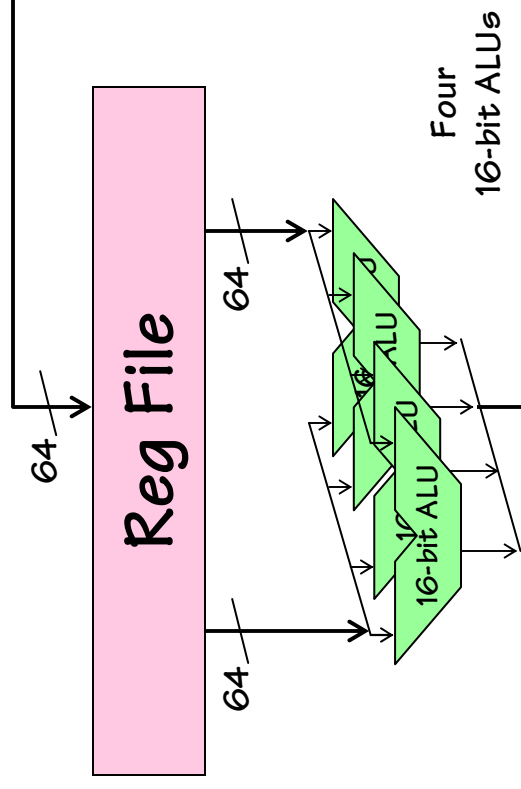
Core CPU handles memory traffic

Partitionable Datapaths for variable-sized

“PACKED OPERANDS”

SIMD Coprocessing Units

“Intel MMX”



Nice data size for:
Graphics,
Signal Processing,
Multimedia Apps,
etc.

SIMD data path added to a traditional CPU core

Register-only operands

Core CPU manages memory traffic

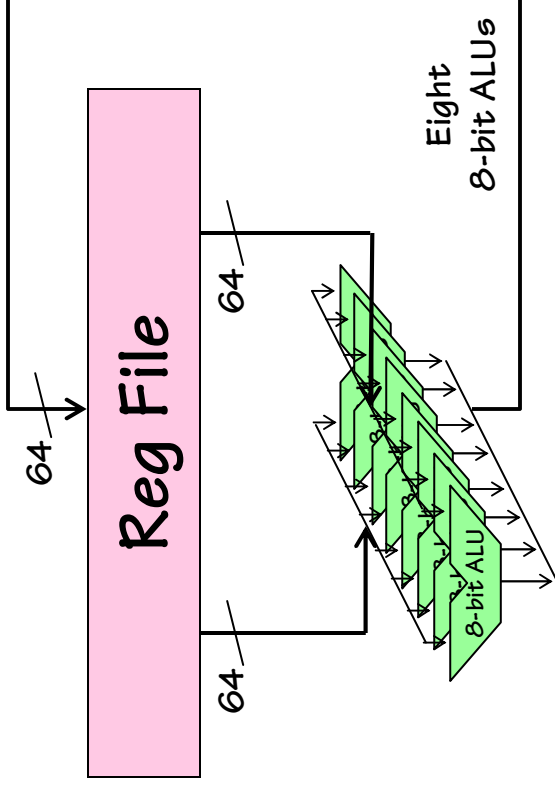
Partitionable Datapaths for variable-sized

“PACKED OPERANDS”

SIMD Coprocessing Units

“Intel MMX”

MMX instructions:
PADDB - add bytes
PADDD - add 16-bit words
PADDD - add 32-bit words
(unsigned & w/saturation)
PSUB{B,W,D} - subtract
PMULTLW - multiply low
PMULTHW - multiply high
PMADDW - multiply & add
PACK -
UNPACK -
PAND -
POR -



SIMD data path added to a traditional CPU core

Register-only operands

Core CPU manages memory traffic

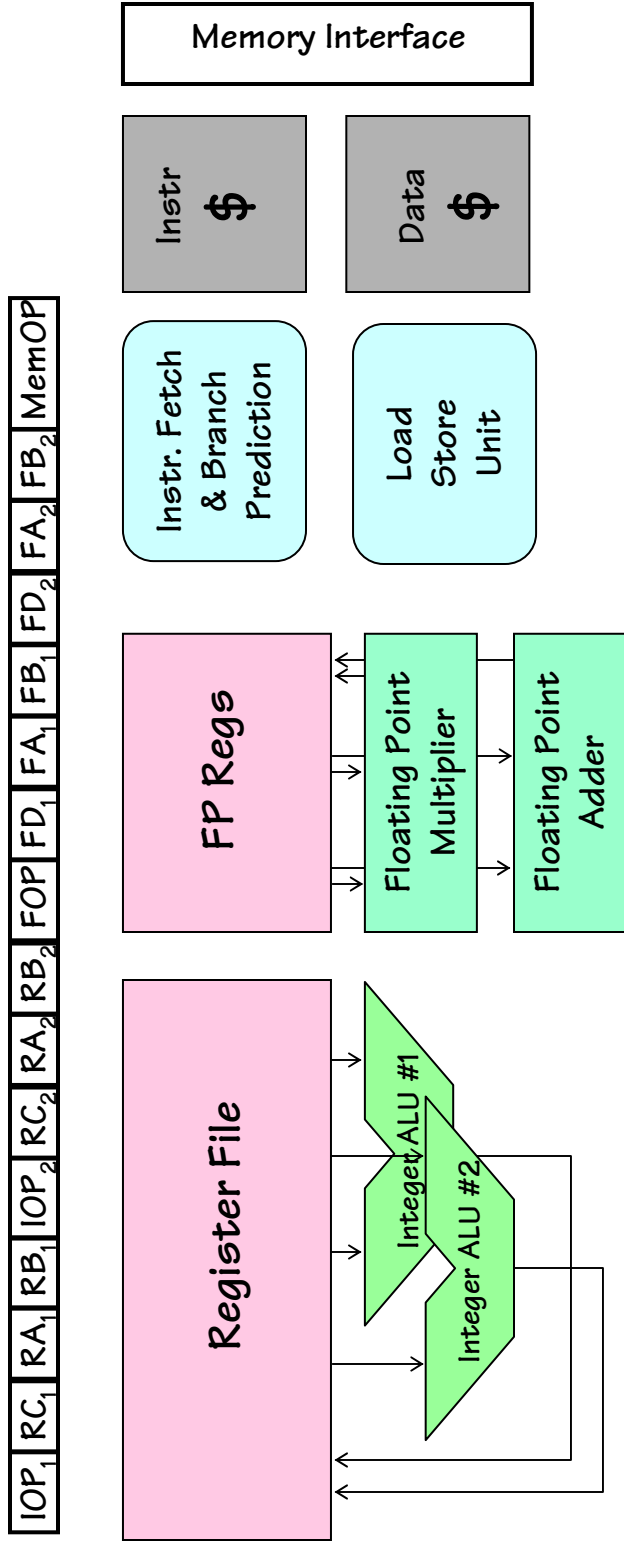
Partitionable Datapaths for variable-sized

“PACKED OPERANDS”

VLIW Variant of SIMD Parallelism

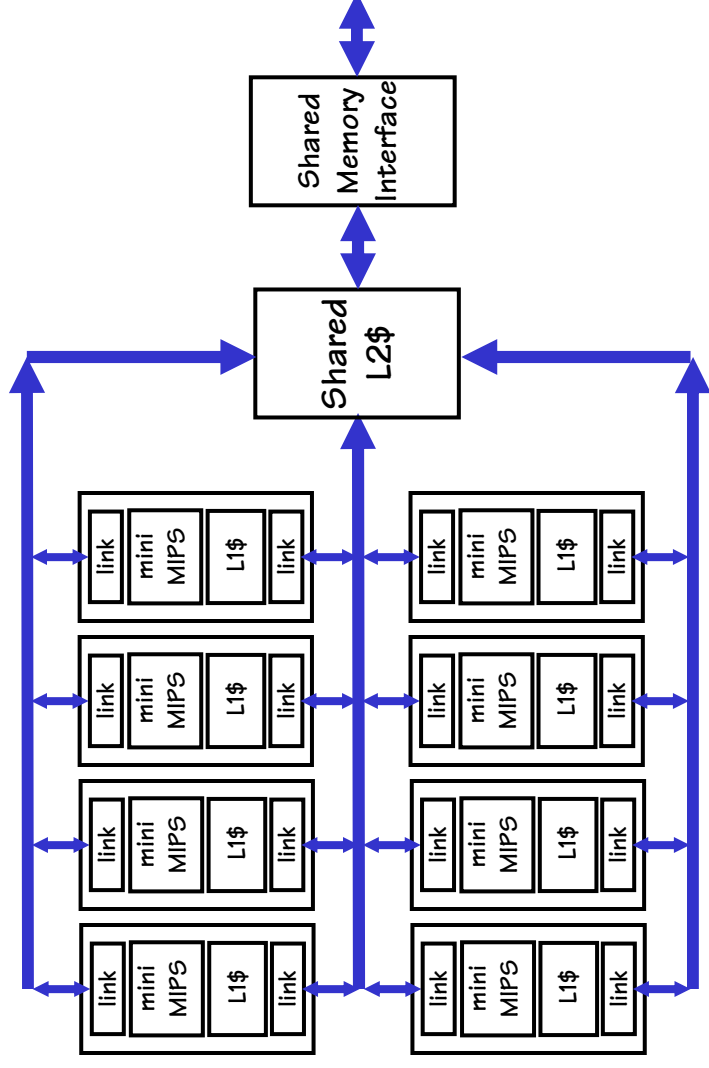
A single-WIDE instruction controls multiple heterogeneous datapaths.

Exposes parallelism to compiler (S/W vs. H/W)



Multi-core Architecture

- Reaction to Superscalar Approach
 - Diminishing returns for H/W to find instruction-level parallelism
 - Superscalar H/W more and more complex
 - Give up and let the S/W folks figure it out
- Simple repeated CPU design (like of 5-stage Minimips)
- H/W focuses on communication
 - Crossbars (switches to share multiple buses)
 - Meshes (point-to-point store-and-forward communication)
 - Shared Caches and memory interfaces (further taxing a known bottleneck)
- S/W focuses on partitioning of data & algorithms



MIMD Processing - Shared memory

All processors share a common main memory

Leverages existing CPU designs

Easy to migrate “Processes” to “Processors”

Share data and program

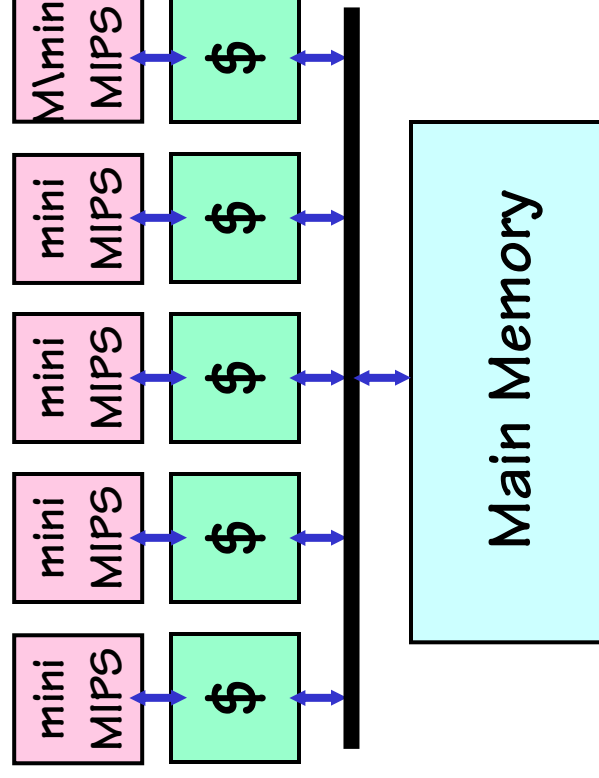
Communicate through shared memory

Upgradeable

Problems:

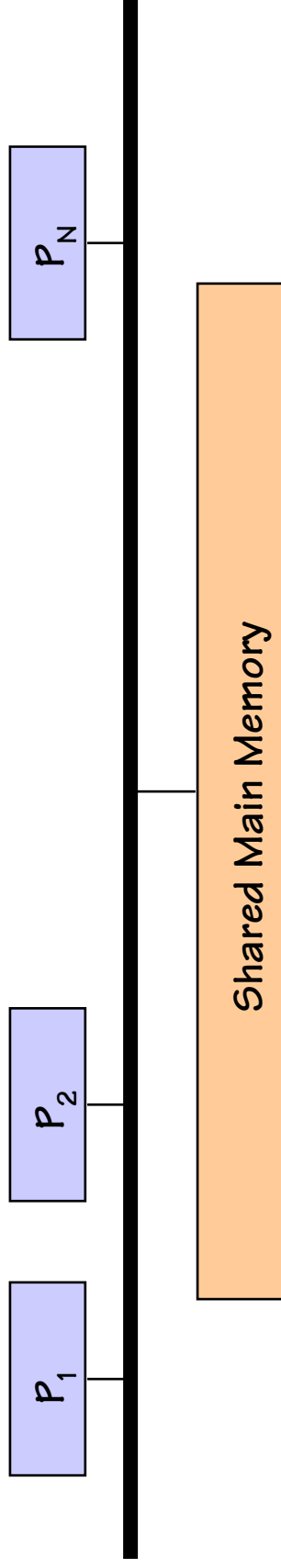
Scalability

Synchronization



Symmetric Multiprocessor Fantasies

If one processor is good, N processors are GREAT:



IDEA:

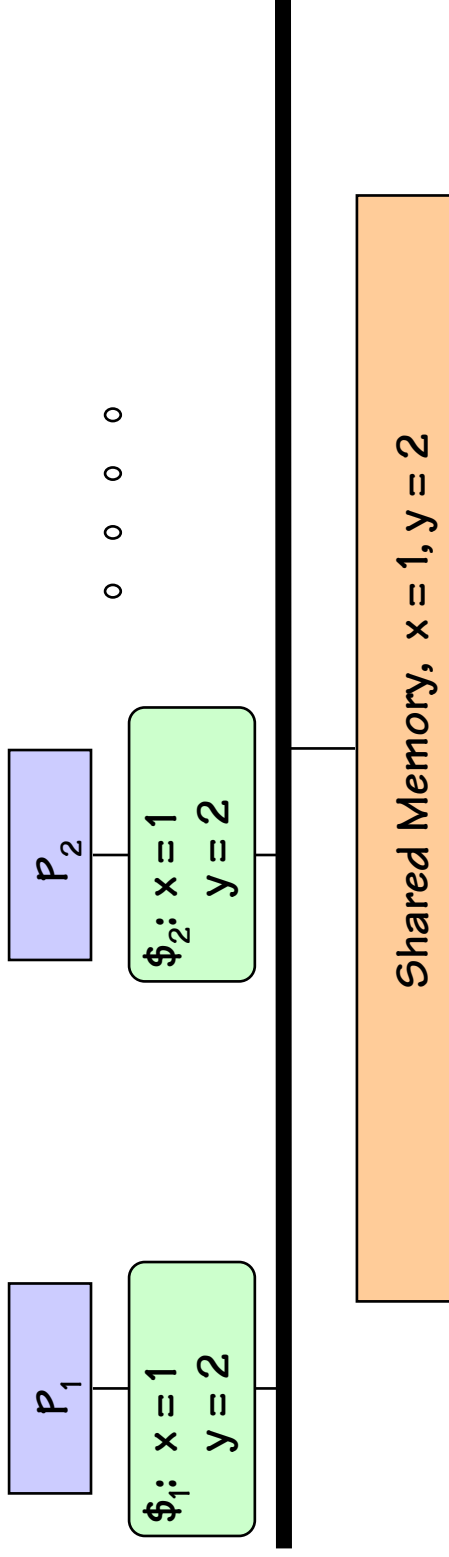
- Run N processes, each on its OWN processor!
- Processors compete for bus mastership, memory access
- Bus **SERIALIZES** memory operations (via arbitration for mastership)

PROBLEM:

The Bus quickly becomes the **BOTTLENECK**

Multiprocessor with Caches

But, we've seen this problem before. The solution, add CACHES.



Consider the following trivial processes running on P₁ and P₂:

Program A

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

Program B

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

What are the Possible Outcomes?

Process A

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

\$₁: x = 1
y = 2

Process B

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

\$₂: x = 1
y = 2

Plausible execution sequences:

SEQUENCE

```
x=3; print(y); y=4; print(x);  
x=3; y=4; print(y); print(x);  
x=3; y=4; print(x); print(y);  
y=4; x=3; print(x); print(y);  
y=4; x=3; print(y); print(x);  
y=4; print(x); x=3; print(y);
```

A prints B prints

```
2   1  
2   1  
2   1  
2   1  
2   1  
2   1
```

Uniprocessor Outcome

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

Process A

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

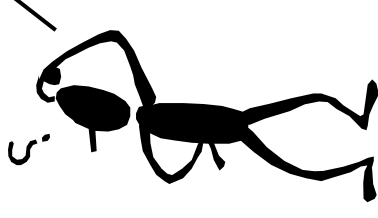
Process B

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

Plausible Uniprocessor execution sequences:

<u>SEQUENCE</u>	<u>A prints</u>	<u>B prints</u>
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 programs should correspond to some interleaved execution on a single processor.

Shared Memory

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

Process A

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

Process B

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

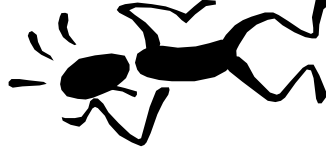
Possible printed values: 2, 3; 4, 3; 4, 1.

(each corresponds to at least one interleaved execution)

IMPOSSIBLE printed values: 2, 1

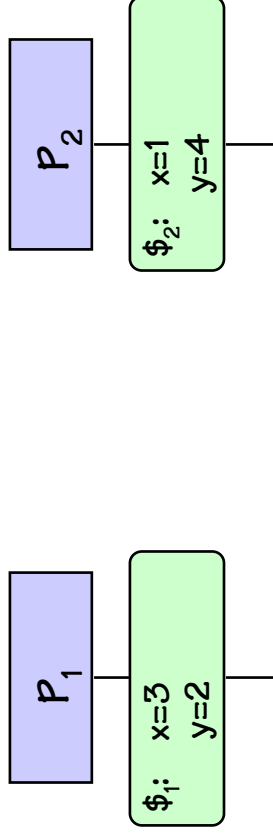
(corresponds to NO valid interleaved execution).

Weren't
caches
supposed to
be invisible
to
programs?



Cache Incoherence

PROBLEM: "stale" values in cache ...



The problem is not that memory has stale values, but that other caches may!



Process A

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

Process B

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

Q: How does B know that A has changed the value of x?

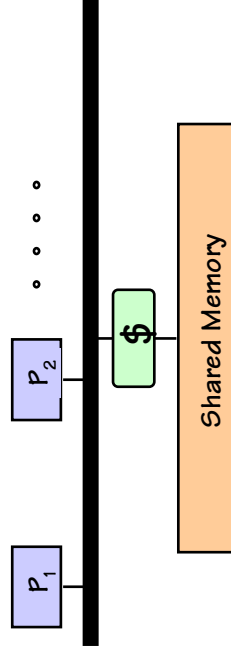
Cache Coherence Solutions

Problem: A writes data into shared memory; B still sees “stale” cached value.

Solutions:

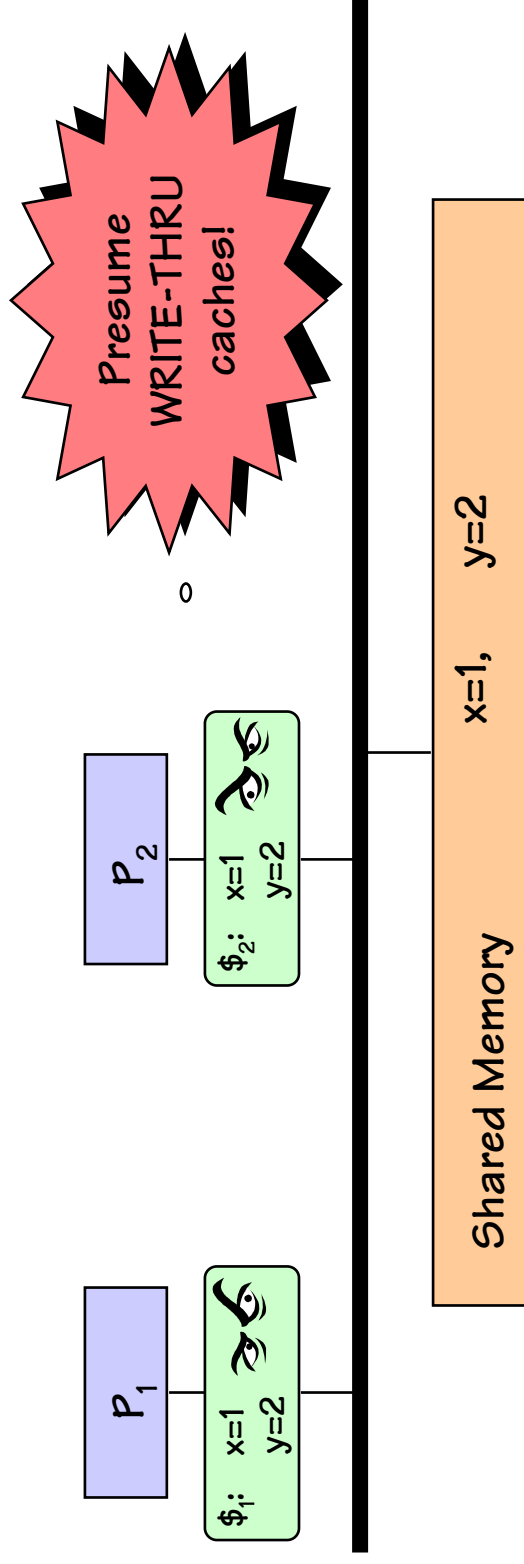
1. Don't cache shared Read/Write pages.
COST: Longer access time to shared memory.
2. Attach cache to shared memory, not to processors...
... share the cache as well as the memory!

COSTS: 1. Bus Contention
2. Locality



3. Make caches talk to each other, maintain a consistent story.

"Snoopy" Caches

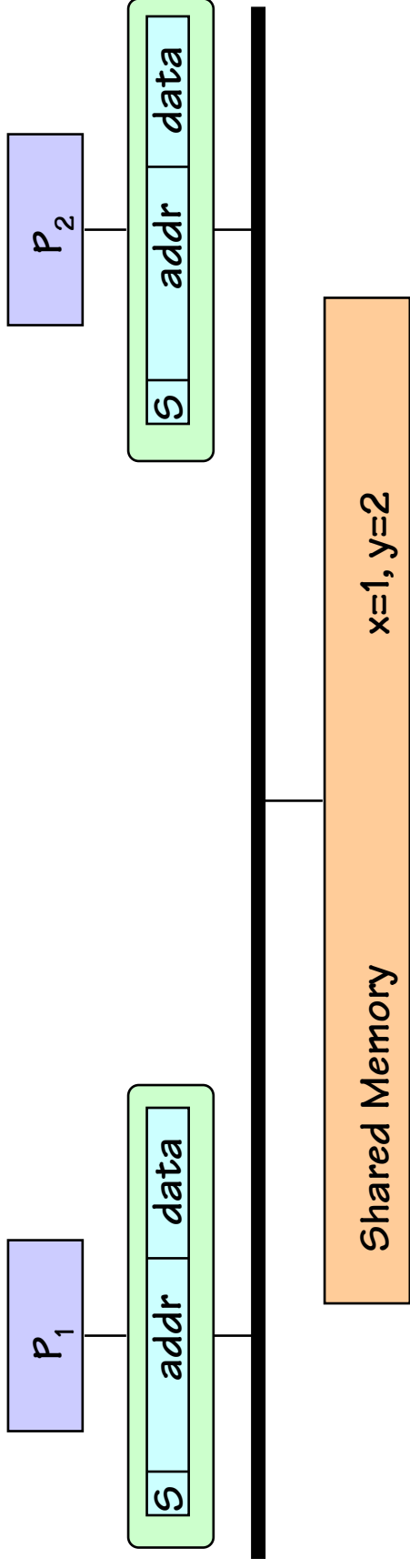


IDEA:

- P_1 writes 3 into x ; write-thru cache causes bus transaction.
- P_2 , snooping, sees transaction on bus. INVALIDATES or UPDATES its cached x value.

MUST WE use a write-thru strategy?

Coherency w/ Write Back



IDEA:

- Various caches can have
 - Multiple SHARED read-only copies; OR
 - One UNSHARED exclusive-access read-write copy.
- Keep STATE of each cache line in extra bits of tag
- Add bus protocols -- “messages” -- to allow caches to maintain globally consistent state

Coherent Cache States

Two-bit STATE in cache line encodes one of M, E, S, I states (“MESI” cache):

INVALID: cache line unused.

SHARED ACCESS: read-only, valid, not dirty. Shared with other read-only copies elsewhere. Must invalidate other copies before writing.

EXCLUSIVE: exclusive copy, not dirty. On write becomes modified.

MODIFIED: exclusive access; read-write, valid, dirty. Must be written back to memory eventually; meanwhile, can be written or read by local processor.

Current state	Read Hit	Read Miss, Snoop Hit	Read Miss, Snoop Miss	Write Hit	Write Miss	Snoop for Read	Snoop for Write
Modified	Modified	Invalid (Wr-Back)	Invalid (Wr-Back)	Modified	Invalid (Wr-Back)	Shared (Push)	Invalid (Push)
Exclusive	Exclusive	Invalid	Invalid	Modified	Invalid	Shared	Invalid
Shared	Shared	Invalid	Invalid	Modified (Invalidate)	Invalid	Shared	Invalid
Invalid	X	Shared (Fill)	Exclusive (Fill)	X	Modified (Fill-Inv)	X	X

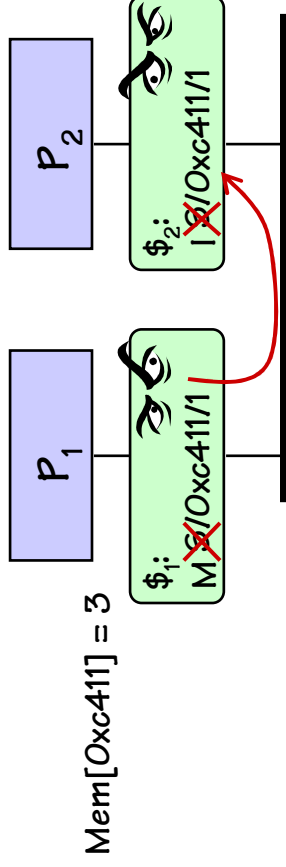


(FREE!!: Can redefine VALID and DIRTY bits)

MESI Examples

Local WRITE request hits cache line in **Shared** state:

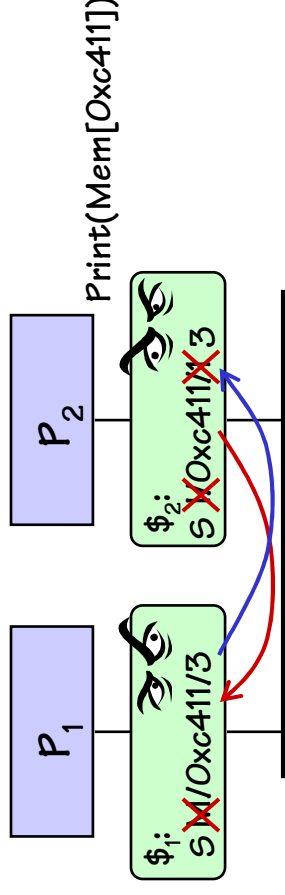
- Send INVALIDATE message forcing other caches to I states
- Change to **Modified** state, proceed with write.



External Snoop READ hits cache line in **Modified**

state:

- Write back cache line
- Change to **Shared** state



Sequential Inconsistency

Process A

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

Process B

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

Shared Memory

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

Plausible sequence of events:

- A writes 3 into x, sends INVALIDATE message.
- B writes 4 into y, sends INVALIDATE message.
- A reads 2 from y, prints it...
- B reads 1 from y, prints it...
- A, B each receive respective INVALIDATE messages.

FIX: Wait for INVALIDATE messages to be acknowledged before proceeding with a subsequent reads.

COST: Loss of performance (writes stall reads)...
must provide for fast invalidates

Who Needs Sequential Consistency, Anyway?

ALTERNATIVE MEMORY SEMANTICS:

“WEAK” consistency

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

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

COMMON APPROACH:

- Weak consistency, by default;
- **MEMORY BARRIER** instructions: stalls processor until all previous memory operations have completed.

"Dusty Deck" Problem

How do we make our old sequential programs run on parallel machines? After all, what's easier, designing new H/W or rewriting all our S/W?

Programs have inertia. Familiar languages, S/W engineering practice reinforce "Sequential Programming Semantics"

By treating PROCESSES or THREADS as a programming constructs... and by assigning each process to a separate processor... we can take advantage of some parallelism.

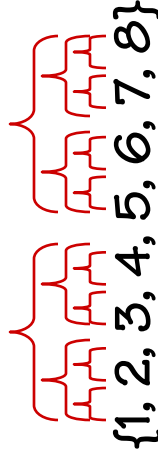


Programming the Beast



Comp 411 (circa 2025):

```
int factorial(int n) {  
    return facthelp(1, n);  
}  
  
parallel int facthelp(int from, int to) {  
    int mid;  
    if (from >= to) return from;  
    mid = (from + to)/2;  
    return (facthelp(from, mid))*facthelp(mid+1, to);  
}
```



Calls facthelp() $2n - 1$ times
(nodes in a binary tree with n leaves).
Runs in $O(\log_2(N))$ time
(on N processors)

Comp 411 (circa 2007):

```
int factorial(int n) {  
    if (n > 0)  
        return n*fact(n-1);  
    else  
        return 1;  
}
```

Calls factorial() only n times
Runs in $O(N)$ time

Parallel Processing Summary

Prospects for future CPU architectures:

- Pipelining - Well understood, but mined-out
- Superscalar - Nearing its practical limits
- SIMD - Limited use for special applications
- VLIW - Returns controls to S/W. The future?



Prospects for future Computer System architectures:

- SMP - Limited scalability. Harder than it appears.
- MIMD/message-passing - It's been the future for over 20 years now. How to program?

WHAT ABOUT THE FUTURE?

New BOUNDARIES, New PROBLEMS

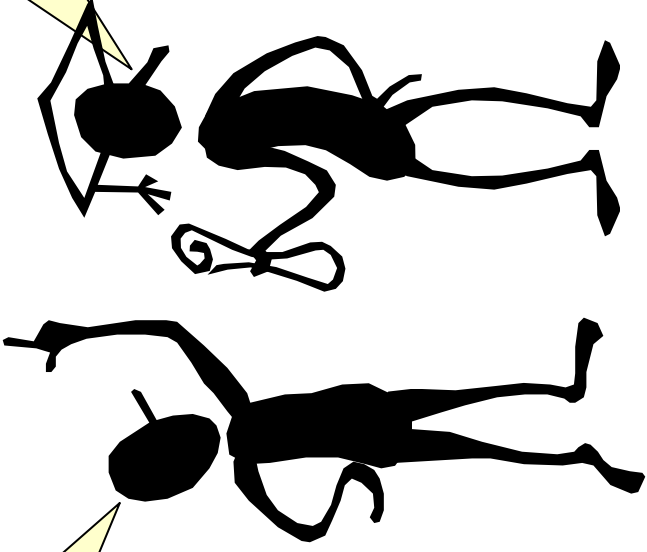
Computer Organization: What's Ahead?

Prediction is very difficult, especially about the future.

-Neils Bohr

The best way to predict the future is to invent it.

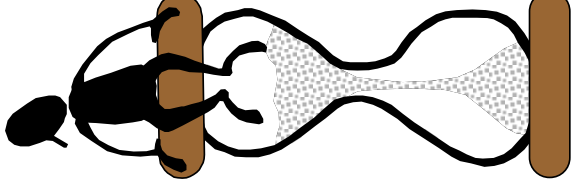
-Alan Kay



Final Exam – 12/10 from 12:00-3:00 in SNO14

Various Futures

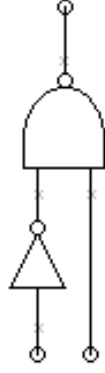
1. YOUR future...
 - Where you've been
 - What's next
2. Future of digital systems
 - Contemporary Architectures
 - Contemporary “hangups”
 - Thinking outside the box ... some historic examples
3. Comp 411's **big** lessons



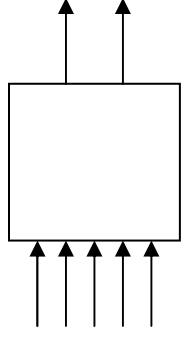
You've Mastered a Lot...



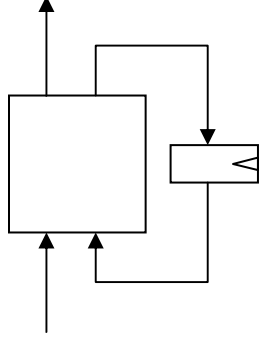
FETs & voltages



Logic gates



Combinational
logic circuits



Sequential logic

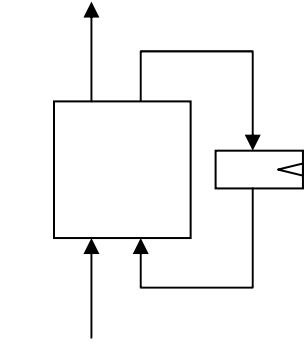


- Combinational contract:
- ◆ discrete-valued inputs
 - ◆ complete in/out spec.
 - ◆ static discipline

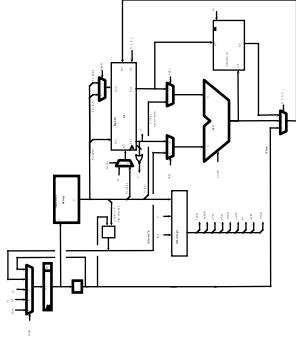
- Acyclic connections
- Summary specification
- Design:
- ◆ sum-of-products
 - ◆ simplification
 - ◆ muxes, ROMs, PLAs

- Storage & state
- Dynamic discipline
- Finite-state machines
- Metastability
- Throughput & latency
- Pipelining

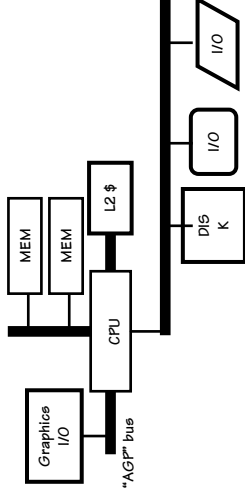
... a WHOLE Lot ...



Sequential logic



CPU Architecture



Computer Systems



Computing Theory
 Instruction Set Architectures
 CPU implementation
 Pipelined CPU
 Software conventions
 Memory architectures

Interconnect
 Virtual machines
 Interprocess communication
 Operating Systems
 Parallel Processing

The Goals of Comp 411

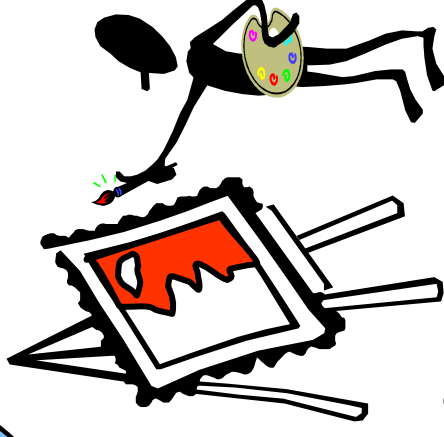
1) TO DEMYSTIFY COMPUTERS

... they shouldn't seem
mysterious anymore



2) THE POWER OF ABSTRACTION

... techniques for understanding,
harnessing, and constructing
systems will millions of components



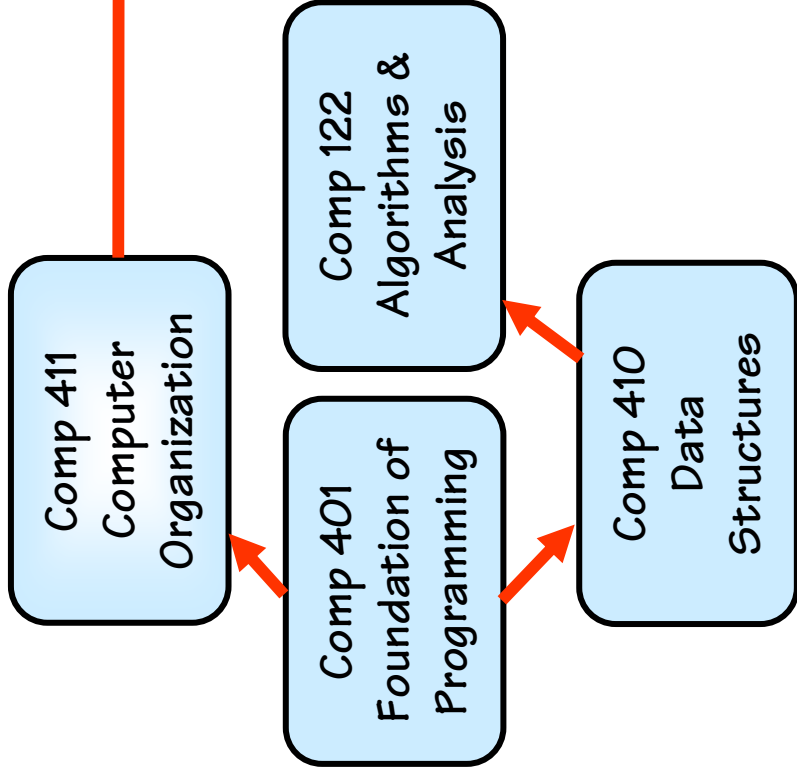
3) LEARN THE STRUCTURES OF COMPUTATION

... what's inside a computer and how they work

What's next?

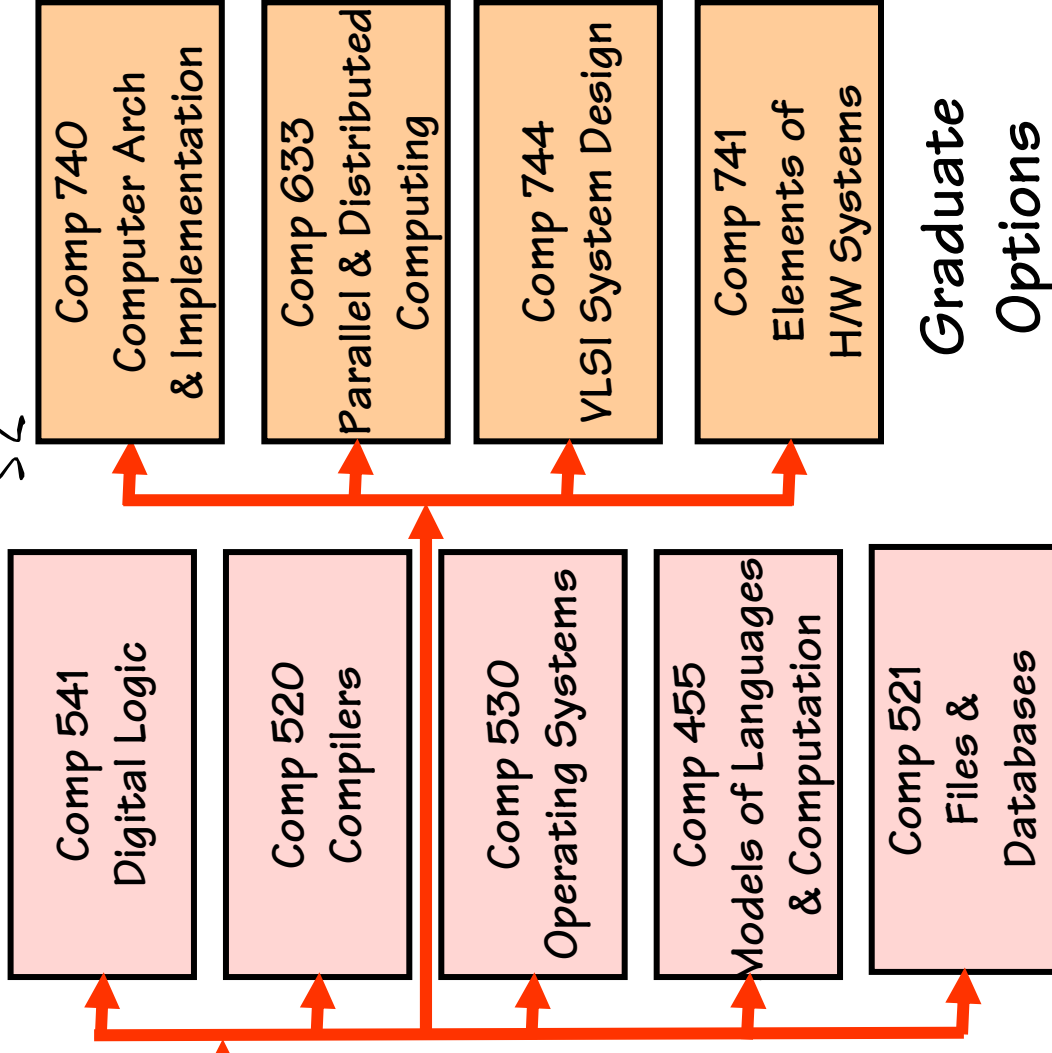
Some options...

Comp 411 was necessarily broad



... but not very deep

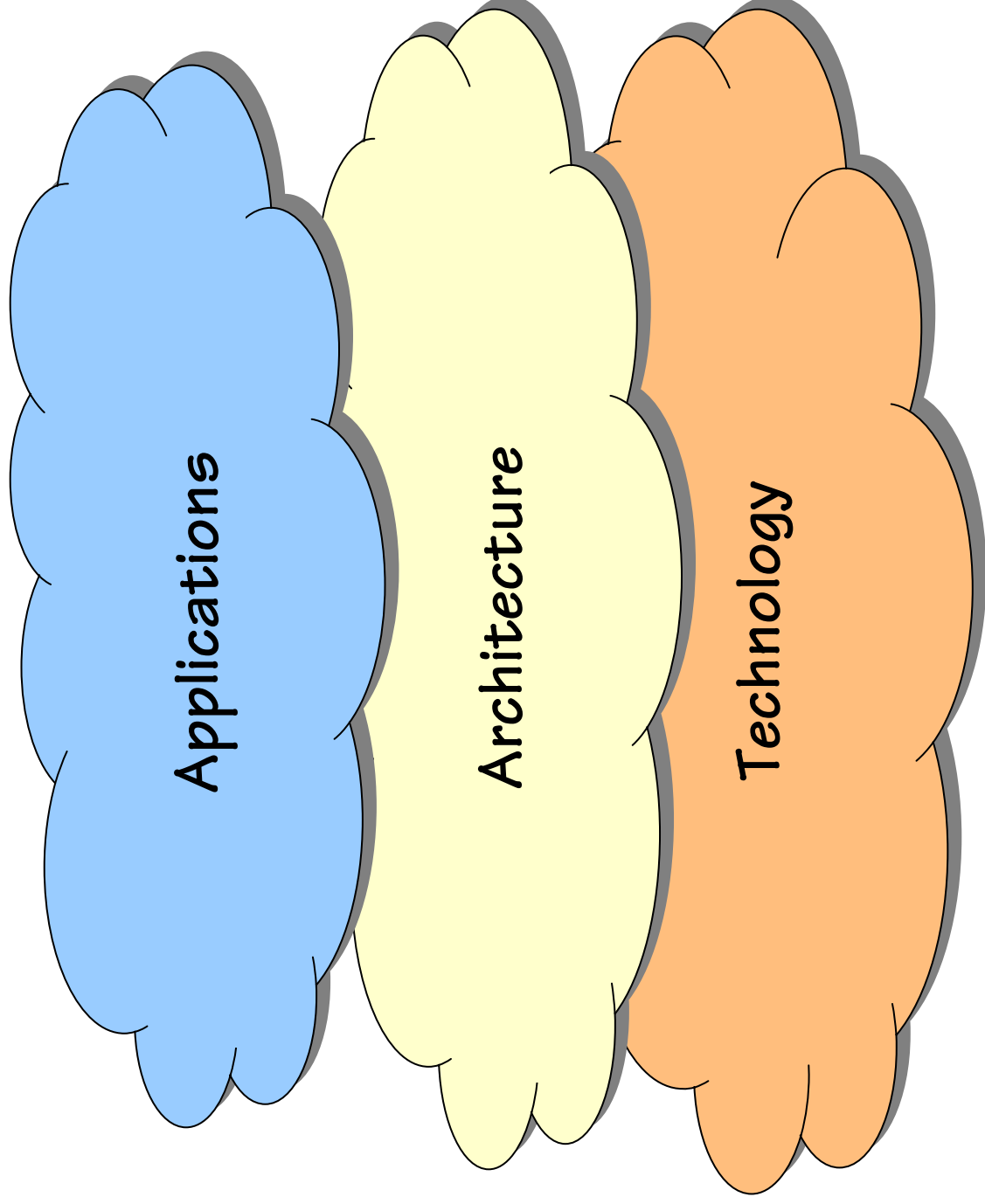
ARGH!— I can never remember all the new course numbers!



Graduate
Options

Undergrad Options

System Design



2007

MS Office, E-commerce, WWW, Digital
Cameras, MP3, DVD, Games, Wireless, GUIs,
...

Von Neumann Architecture
Procedures, Objects, Processes
(hidden: pipelining, superscalar, SIMD, ...)

CMOS: 200 million transistors/chip
10x transistors every 5 years
1% performance/week!

2025?

Natural language/speech, vision,
synthesized video, weather,
simulation, ...

Von Neumann Architecture???
1024-way multicore?
Neural Nets?

CMOS:
20 billion transistors
20 GHz clock

Computer
Science is the
fastest
changing
field in the
history of
mankind!

THE END!

Computers are tools
that attempt to realize
their programmers' dreams.

The only problem
with Haiku is that you just
get started and then...

