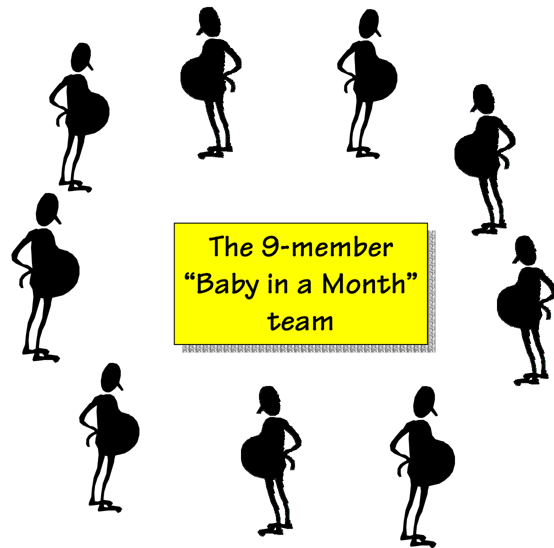
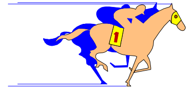
Parallel Processing



The Home Stretch



FEEDBACK: if you haven’t already, please take a few minutes to **fill out the on-line anonymous course survey**!

Tomorrow’s section: QUIZ 5

Tuesday 12/10: LAST LECTURE! Some futures… (finis)

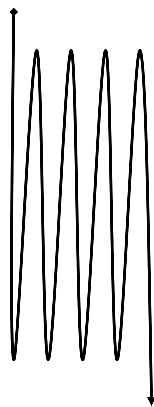
Wednesday 12/11:

- Lab #7 due
- Immense Satisfaction/Rejoicing/Relief/Celebration/Wild Partying.

Taking a step back

Static Code

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

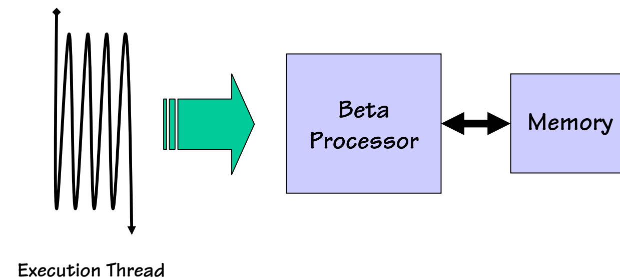


Dynamic Execution Path

aka “Thread of Execution”

Path Length = number of instructions along path

We have been building machines to execute threads (quickly)



$$\text{Time} = \frac{\text{Path Length} \times \text{Clocks-per-Instruction}}{\text{Clocks-per-second}}$$

How to Decrease Time

Reduce Path Length

- Better compilation
- Complex Instructions that "do more"

Reduce Clocks per Instruction (CPI)

- Simple, pipeline-able instructions
- Reducing pipeline stalls

Increase Clock Rate

- Faster transistors and better circuits
- Deeper pipelines
- Simpler logic and smaller footprints

Can we make CPI < 1 ?

...Implies we can complete
more than one instruction each clock cycle!

Two Places to Find Parallelism

Instruction Level (ILP) – Fetch and issue groups of independent instructions within a thread of execution

Thread Level (TLP) – Simultaneously execute multiple execution streams

Instruction-Level Parallelism

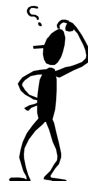
Sequential Code

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

"Safe" Parallel Code

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

What if I tried to do multiple iterations at once?



This is okay, but smarter coding does better in this example!

A many-dimensioned space!

Do we deduce parallelism at compile-time or execution-time or do we explicitly program for it?

Do we exploit special, more regular forms or do we try to be as general-purpose as possible?

How would we encode parallelism we find? A new ISA?

Should we be even more aggressive and speculate on parallelism?

Or really aggressive and relax sequential semantics?

...

Let's Look at…

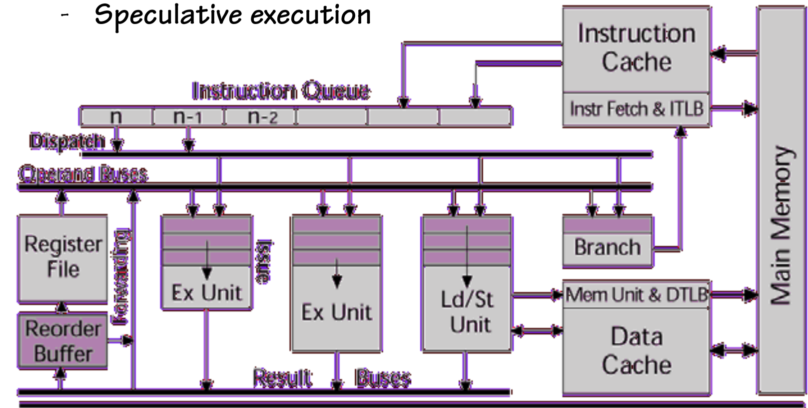
Superscalar – replicate data paths and design control logic to discover ILP in existing binaries.

SIMD/VLIW – new ISAs that explicitly designate ILP.

SMP – Harness multiple processors as first step towards TLP

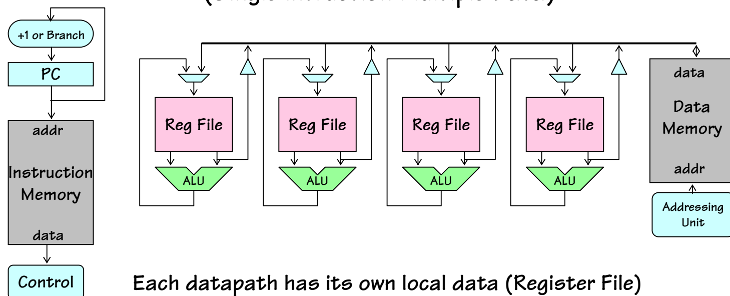
Superscalar Parallelism

- Popular now, but the limits are near (8-issue)
- Multiple instruction dispatch
- Speculative execution



SIMD Processing

(Single Intruction Multiple Data)



This sort of construct is also becoming popular on modern uniprocessors

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?)

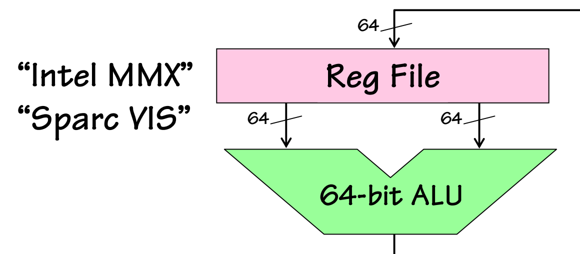
Conditional operations are common in SIMD machines
if (flag1) Rc = Ra <op> Rb

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

Model: "hide" parallelism in primitives (eg, vector operations)

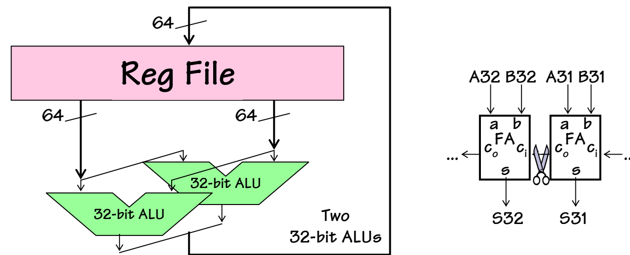
SIMD Coprocessing Units

"Intel MMX"
"Sparc VIS"



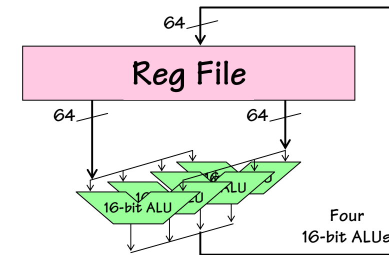
SIMD data path added to a traditional CPU core
- Register-only operands
- Core CPU handles memory traffic
- Partitionable Datapaths for variable-sized "PACKED OPERANDS"

SIMD Coprocessing Units



SIMD data path added to a traditional CPU core
- Register-only operands
- Core CPU handles memory traffic
- Partitionable Datapaths for variable-sized
  "PACKED OPERANDS"

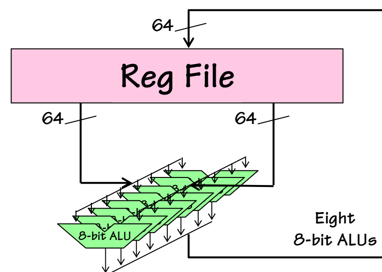
SIMD Coprocessing Units



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



MMX instructions:
- PADDB - add bytes
- PADDW - 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 -

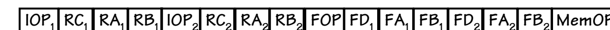
SIMD data path added to a traditional CPU core
- Register-only operands
- Core CPU manages memory traffic
- Partitionable Datapaths for variable-sized
  "PACKED OPERANDS"

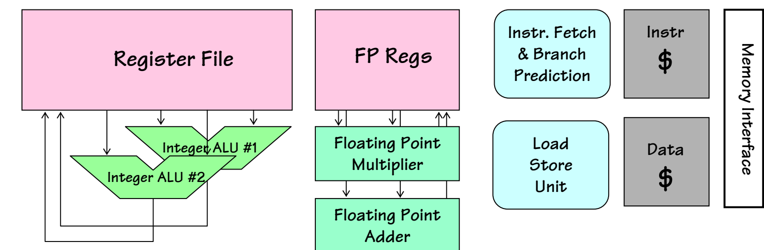
VLIW Variant of SIMD Parallelism

(Very Long Instruction Word)

A single-WIDE instruction controls multiple heterogeneous datapaths.

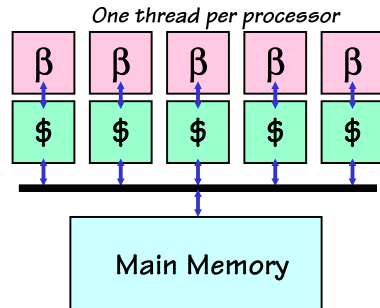
Exposes parallelism to compiler (S/W vs. H/W)





Hey! Let's Just Paste Down Lots of Processors and Exploit TLP

- All processors share a common main memory
- Leverages existing CPU designs
- Easy to map "Processes (threads)" to "Processors"
- Share data and program
- Communicate through shared memory
- Upgradeable
- Problems:
  - Scalability
  - Synchronization



An SMP – Symmetric Multi-Processor

Programming the Beast

6.004 (circa 2002):

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

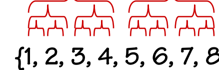
Calls factorial() only n times

Runs in O(N) time

6.004 (circa 2020)?

```
int factorial(int n) {
   return facthelp(1, n);
}

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

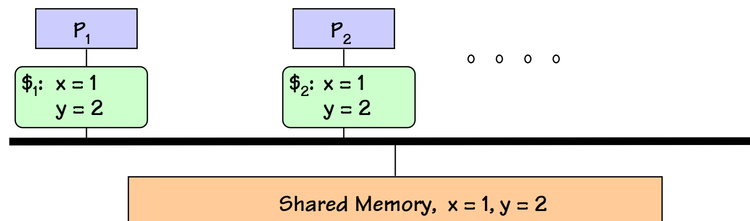


{1, 2, 3, 4, 5, 6, 7, 8}

Calls facthelp() 2n – 1 times
(nodes in a binary tree with n leaves).

Runs in $O(\log_2(N))$ time
(on N processors)

Hmmm….does it even work?



Consider the following trivial processes running on P_1 and P_2:

Process A

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

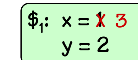
Process B

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


What are the Possible Outcomes?

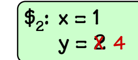
Process A

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



Process B

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



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

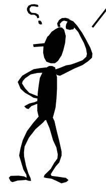
But, what are the possible outcomes if we ran Process A and Process B on a single timed-shared processor?

Process A	Process B
`x = 3;`	`y = 4;`
`print(y);`	`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 programs should correspond to *some* interleaved execution on a single processor.

Shared Memory

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

Process A	Process B
`x = 3;`	`y = 4;`
`print(y);`	`print(x);`

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

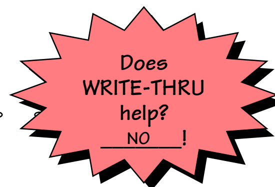
IMPOSSIBLE printed values: 2, 1
(corresponds to NO valid interleaved execution).

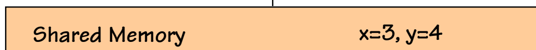
Weren't caches supposed to be invisible to programs?

Cache Incoherence

PROBLEM: "stale" values in cache ...

P_2
$_2: x=1
y=4



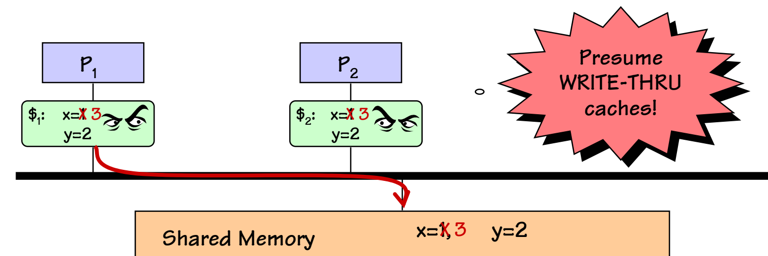


The problem is not that memory has stale values, but that other caches may!

Process A	Process B
`x = 3;`	`y = 4;`
`print(y);`	`print(x);`

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

"Snoopy" Caches

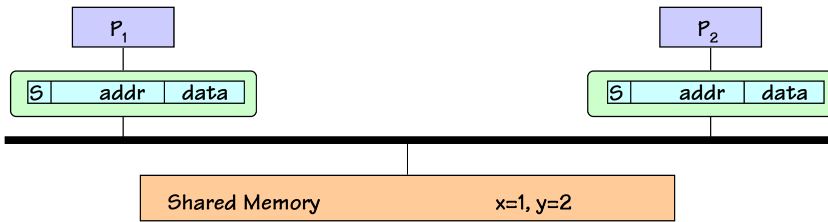


IDEA:

- 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

Snoopy Cache Design

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.

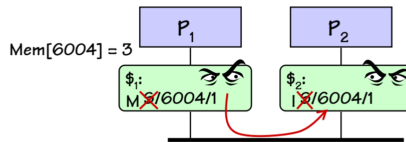
Current state	Read Hit	Read Miss, Snoop Hit	Read Miss, Snoop Miss	Write Hit	Write Miss	Snoop for Read	Snoop for Write
Modified	Modified	Invalid (Wr-Back)	Invalid (Wr-Back)	Modified	Invalid (Wr-Back)	Shared (Push)	Invalid (Push)
Exclusive	Exclusive	Invalid	Invalid	Modified	Invalid	Shared	Invalid
Shared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

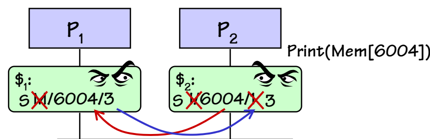
Local WRITE request hits cache line in Shared state:

- 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



Write Acknowledgement

UNIPROCESSORS can post writes or "write behind" --

-- continue with subsequent instructions after having initiated a WRITE transaction to memory (eg, on a cache miss).

HENCE WRITES appear FAST.

Can we take the same approach with multiprocessors?

Consider our example (again)

Process A

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

Shared Memory

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

Process B

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

SEQUENTIAL CONSISTENCY allows (2,3), (4,3), (4,1) printed; but not (2,1).

Sequential Inconsistency

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

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

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

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 read.

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).

DEC ALPHA APPROACH:

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

Semaphores in Parallel Processors

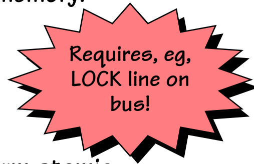
Can semaphores be implemented using ONLY atomic reads and writes?

ANSWER: Only if you're **Dijkstra**

Contemporary solution:

HARDWARE support for atomic *sequences* of memory transactions.

- Explicit LOCK, UNLOCK controls on access to memory:
```
lock bus
read semaphore
branch …
write modified semaphore
unlock bus
```



- Single "Test and Set" instructions which perform atomic READ/MODIFY/WRITE using bus-locking mechanism.
- (ALPHA Variant): "unlock" instruction returns 0 if sequence was interrupted, else returns 1. Program can repeat sequence until its not interrupted.

What a mess!

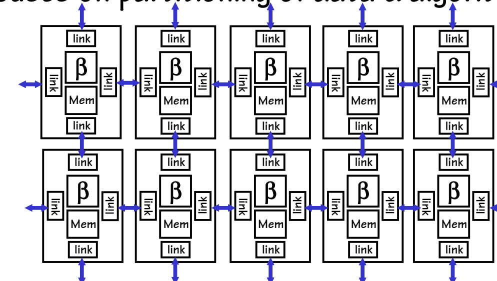
Let's Just connect a bunch of uniprocessors over a network!

So-called "Horizontially scaled" (aka MIMD)

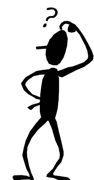
Can Leverage existing CPU designs / development tools

H/W focuses on communication (2-D Mesh, N-Cube)

S/W focuses on partitioning of data & algorithms



Hum.... I thought the point of all these layers of abstraction and virtuality was so the S/W could be designed independent of the H/W



Parallel Processing Summary

Prospects for future CPU architectures:

Pipelining - Well understood, but mined-out
Superscalar - At 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 (≈ 100). Hard!

MIMD/message-passing - It's been the future for over 20 years now. How to program?

NEXT TIME: Some futures --

DISRUPTIONS and FUNDAMENTAL LIMITS