Parallel Processing

The Home Stretch

TODAY 5/7: Lab 8 (LAST!) due

Friday 5/8 section: LAST QUIZ (#5)!

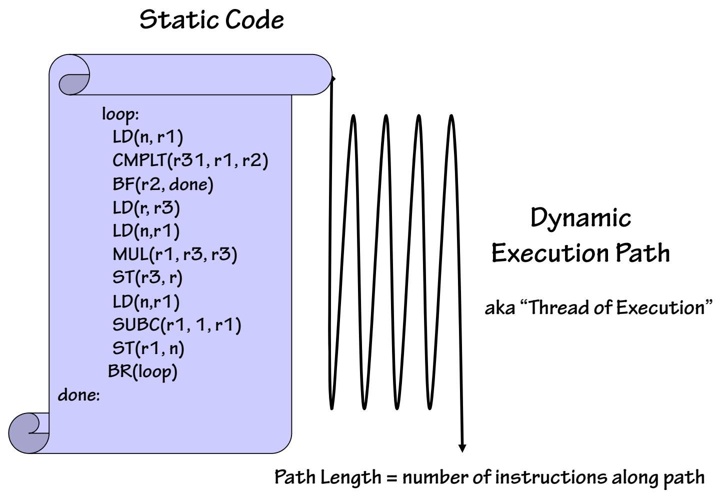
Tu 5/12: Wrapup (LAST!) Lecture!

Wednesday 5/13:

NO SECTION MEETINGS!

- Optional DESIGN PROJECT due
- ALL (late) Assignments due
- Immense Satisfaction/Rejoicing/Relief/Celebration/Wild Partying.

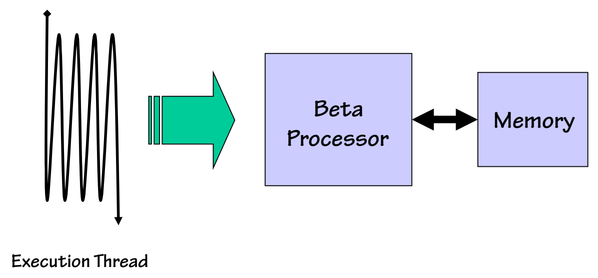
Taking a step back



```
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:
```

Path Length = number of instructions along path

We have been building machines to execute one thread (quickly)



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

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

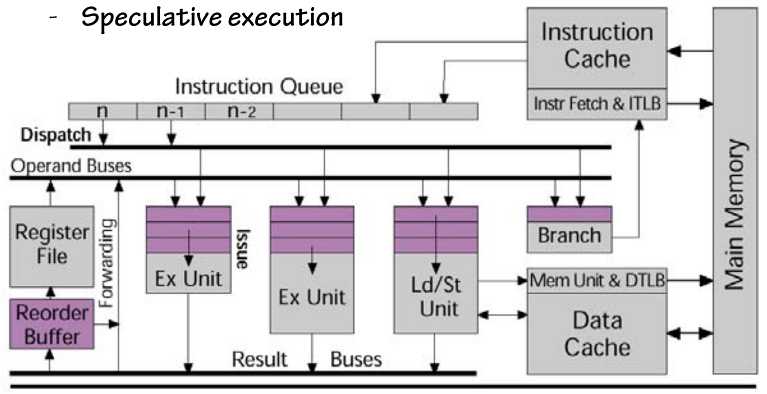
```
loop:
  LD(n, r1)
  CMPLT(r31, r1, r2)
  BF(r2, done)
  LD(r, r3)  LD(n,r1)  LD(n,r4)
  MUL(r1, r3, r3) SUBC(r4, 1, r4)
  ST(r3, r) ST(r4, n) BR(loop)
done:
```

What if I tried to do multiple iterations at once?

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

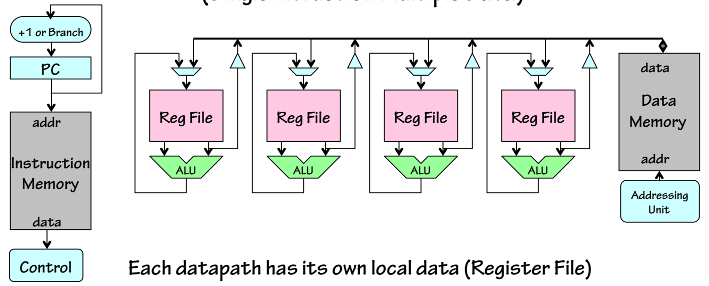
Superscalar Parallelism

- 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

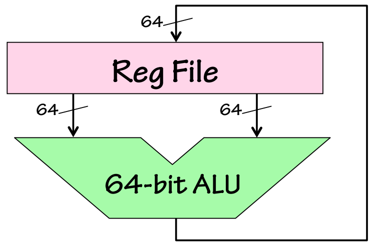
Conditional branching is difficult...
(What if only one CPU has R1 = 0?)

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)

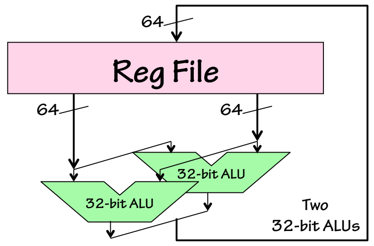
SIMD Coprocessing Units

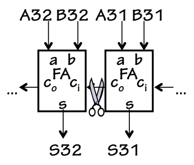


“Intel MMX, SSE”
“Sparc VIS”

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

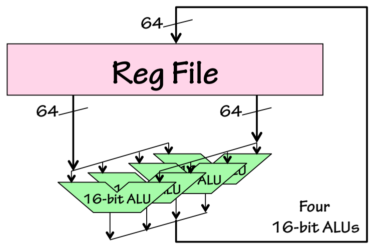
SIMD Coprocessing Units





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

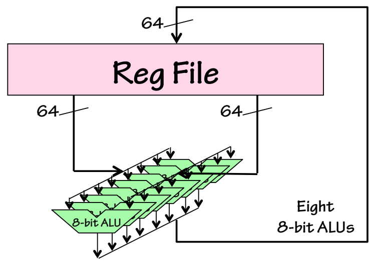
SIMD Coprocessing Units



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 -

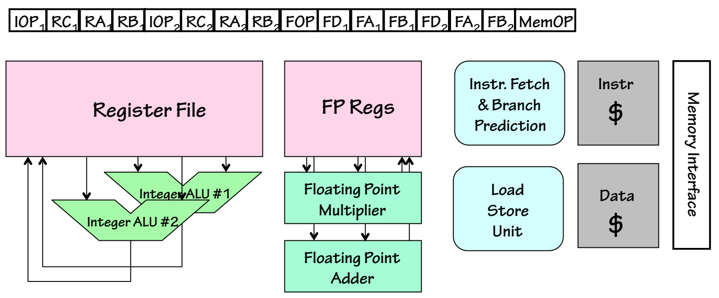
SIMD data path added to a traditional CPU core
- Register-only operands
- Core CPU manages memory traffic
- Partitionable Datapaths for variable-sized “PACKED OPERANDS”

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)


IOP_1 | RC_1 | RA_1 | RB_1 | IOP_2 | RC_2 | RA_2 | RB_2 | FOP | FD_1 | FA_1 | FB_1 | FD_2 | FA_2 | FB_2 | MemOP
Register File
Integer ALU #1
Integer ALU #2
FP Regs
Floating Point Multiplier
Floating Point Adder
Instr. Fetch & Branch Prediction
Load Store Unit
Instr $
Data $
Memory Interface

Multiple Instruction Streams: MIMD

Exploiting Thread Level Parallelism

All processors share a common main memory

Leverages existing CPU designs

Easy to map "Processes (threads)" to "Processors"

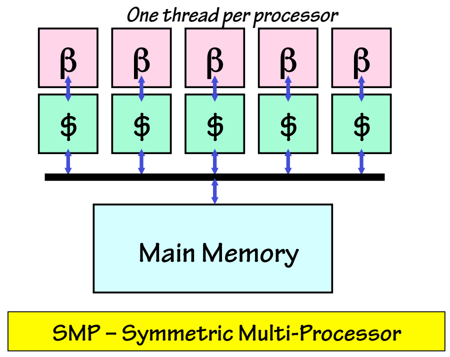
Share data and program

Communicate through shared memory

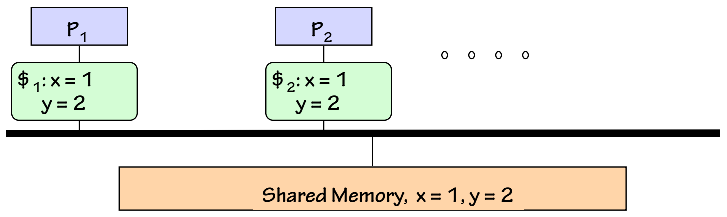
Upgradeable

Problems:

- Scalability
- Synchronization



Hmmm….does it even work?



Consider the following trivial processes running on P_1 and P_2:

Process A

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

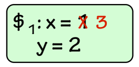
Process B

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

What are the Possible Outcomes?

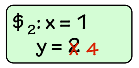
Process A

```
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

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

Process B

```
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 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);
```

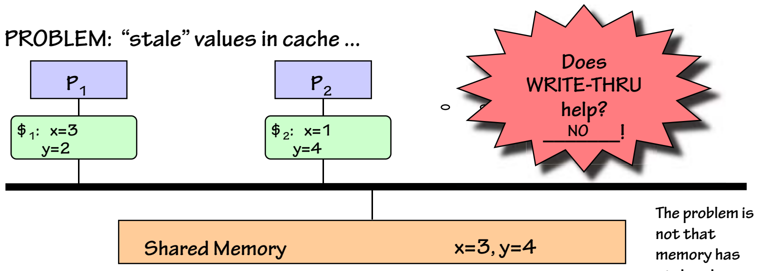
Weren't caches supposed to be invisible to programs?

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).

Cache Incoherence

PROBLEM: "stale" values in cache ...



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

Process A

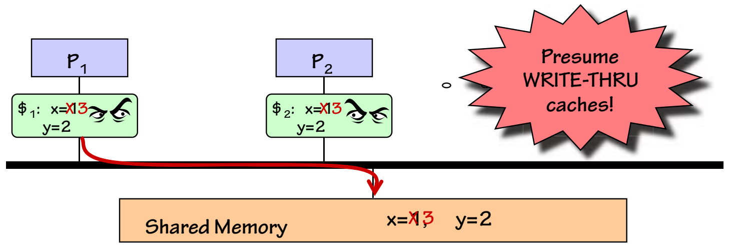
```
x = 3;
print(y);
```

Process B

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

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

"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.

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

4-state FSM for each cache line!

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

Who needs Sequential Consistency, anyway?

ALTERNATIVE MEMORY SEMANTICS:

"WEAK" consistency

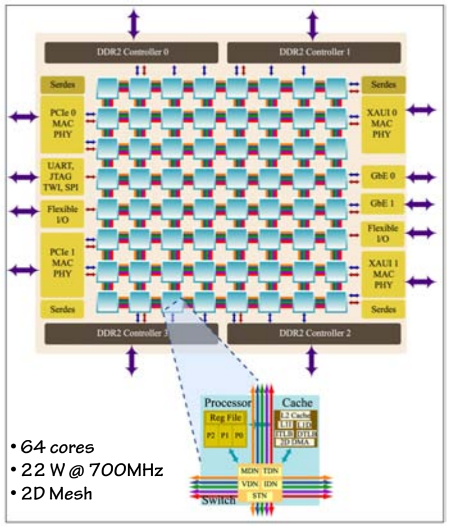
EASIER GOAL: Memory operations from each processor appear to be performed in order issued by that processor;

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

ALTERNATIVE APPROACH:

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

MIMD Multicore Arrays



- 64 cores
- 22 W @ 700MHz
- 2D Mesh

- Can Leverage existing CPU designs / development tools
- H/W focuses on communication 2-D Mesh / cache hierarchy/ …)
- S/W focuses on partitioning, extracting parallelism
- "speculative execution" hacks

- 16 cores/32 thr
- 250W @ 2.3GHz
- Transactional Mem
- Thread Speculation
- "scout" threads

http://www.tilera.com/

Figure by MITOpenCourseWare.

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… but inflexible

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: Its up to YOUR generation!