# Modern Processor Design

What does a processor look like today?

- Multiple instructions executed per cycle
- Many ALUs
- Large number of registers
- Deep pipelines
- Branch prediction
- Register renaming
- Out-of-order execution
- Non-blocking loads/stores
- Prefetch
- Multiprocessor support

CSL

# Modern Processor Design

Typical modern out-of-order processor:

- fetch a block of instructions from icache
- predict next PC
- decode block and rename registers
- insert instructions into issue window
- check pending bits; issue as many instructions as possible
- when instruction complete, update registers
- keep track of order of instructions for exception handling
- instructions *retire* in program order

# Modern Processor Design

- loads on cache miss wait until data returns from memory
- any prediction error: cancel invalid instructions

Branch misprediction penalty can be high: 10-15 cycles!

Trade-off between sophisticated algorithms and efficient VLSI implementation...

- R10000: out-of-order, 195-225MHz
- Alpha 21164: in-order, 300-600MHz

# What is Parallel Computer Architecture?

A parallel computer is a collection of processing elements that cooperate to solve large problems fast

Some broad issues:

- resource allocation:
  - how large a collection?
  - how powerful are the elements?
  - how much memory?
- data access, communication and synchronization
  - how do the elements cooperate and communicate?
  - how are data transmitted between processors?
  - what are the abstractions and primitives for cooperation?
- performance and scalability
  - how does it all translate into performance?
  - going after throughput or speedup?
  - how does it scale?


CSL

# Increasing Demand ...

You can only hope to contain it!

Application demands: the insatiable need for computing cycles

- *scientific computing*: CFD, biology, chemistry, physics, ...
- *general-purpose computing*: video, graphics, CAD, TP...quake!

Technology trends

- number of transistors on chip growing rapidly
- clock rates expected to go up only slowly

Architecture trends

- instruction-level parallelism valuable but limited
- coarser-level parallelism, as in MPs, the most viable approach

Economics

<u>Current trends</u>:

- today's microprocessors have multiprocessor support
- servers and workstations becoming MP: Sun, SGI, Intel...
- tomorrow's microprocessors *are* multiprocessors

# Measuring Performance

Goal of applications in using parallel machines: Speedup

$$Speedup\ (p\ processors) = \frac{Performance\ (p\ processors)}{Performance\ (1\ processor)}$$

For a fixed problem size (input data set), Performance = 1/Time

$$Speedup_{\ fixed\ problem}\ (p\ processors) = \frac{Time\ (1\ processor)}{Time\ (p\ processors)}$$


# Applications

Transition to parallel computing has occurred for scientific and engineering computing

In rapid progress in commercial computing

- database and transactions as well as financial
- usually smaller-scale, but large-scale systems also used

Desktop also uses multithreaded programs, which are a lot like parallel programs

- SMT can help here

Demand for improving throughput on sequential workloads

- greatest use of small-scale multiprocessors
- web servers dominate market

Solid application demand exists and will increase


# Economics

Commodity microprocessors not only fast but *cheap*

- development cost is tens of millions of dollars (5-100 typical)
- but, many more are sold compared to supercomputers
- crucial to take advantage of the investment, and use the commodity building block

Multiprocessors being pushed by software vendors (e.g. database) as well as hardware vendors

Standardization by Intel makes small, bus-based SMPs commodity

Desktop: few smaller processors versus one larger one?

- multiprocessor on a chip (CMP)


CSL

# UMA vs NUMA

Bus-based SMPs are UMA machines

- Uniform Memory Access
- all processors see the same memory latency to all locations

As we will see, scalable machines are NUMA machines

- Non-Uniform Memory Access (keep local memory fast)
- can preserve UMA in scalable machines, but all memory becomes uniformly slow

Cache-coherent scalable machines are ccNUMA machines

- they are typically distributed shared memory (DSM) machines

What is cache coherence?

# Caches and Cache Coherence

Caches play key performance role

- reduce average data access time
- reduce bandwidth demands placed on shared interconnect

But private processor caches create a problem

- copies of a variable can be present in multiple caches
- a write by one processor may not become visible to others
  - they'll keep accessing stale value in their caches
- *cache coherence* problem
- need to take actions to ensure visibility, preserve programmer intuition


CSL

# Coherent Memory Systems

Reading a location should return latest value written

Easy in uniprocessors

- except for I/O: coherence between I/O devices and processors
- but infrequent so software solutions work
  - uncacheable memory, uncacheable operations, flush pages, pass I/O data through caches

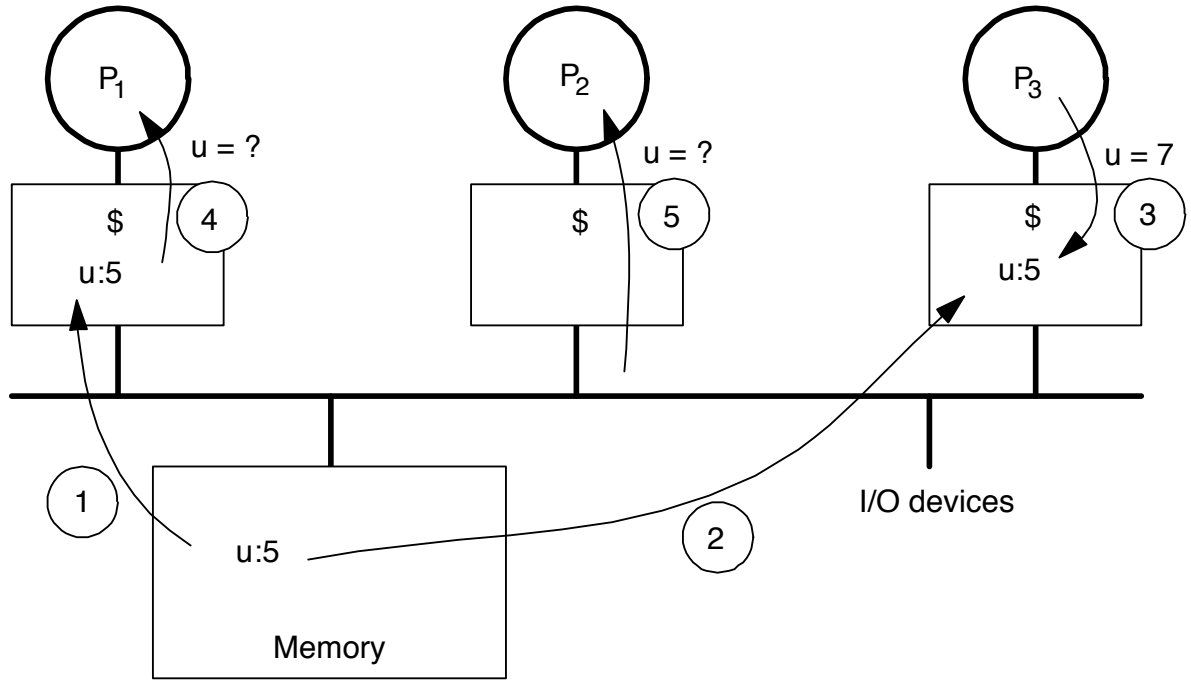
Want same to hold when processes run on different processors

- e.g. as if the processes were interleaved on a uniprocessor

But coherence problem much more critical in multiprocessors

- pervasive
- performance-critical
- must be treated as a basic hardware design issue


# Cache Coherence Problem




- Processors see different values for u after event 3
- With writeback caches, value written back to memory depends on which cache flushes or writes back value first
  - processes accessing main memory may see very stale value
- Unacceptable to programs, and frequent!

# Solution = Snoop Bus

Built on top of two fundamentals of uniprocessor systems

- bus transactions
- state transition diagram in cache

Uniprocessor bus transaction:

- three phases: arbitration, command/address, data transfer
- all devices observe addresses, one is responsible

Uniprocessor cache states:

- effectively, every block is a finite state machine
- write-through, write no-allocate has two states: valid, invalid
- writeback caches have one more state: modified (“dirty”)

Multiprocessors extend cache states to implement coherence


# Snoopy Cache Coherence

*Basic Idea*

Transactions on bus are visible to all processors

Processors or their representatives can snoop (monitor) bus and take action on relevant events (e.g. change state)

*Implementing a Protocol*

Cache controller now receives inputs from both sides:

- requests from processor, bus requests/responses from snooper

In either case, takes zero or more actions

- updates state, responds with data, generates new bus xactions

Protocol is distributed algorithm: cooperating state machines

- set of states, state transition diagram, actions

Granularity of coherence is typically cache block

- like that of allocation in cache and transfer to/from cache

# Validation Based Protocols

Exclusive means can modify without notifying anyone else

- i.e. without bus transaction
- must first get block in exclusive state before writing into it
- even if already valid, need transaction, so called a write miss

Store to non-dirty data generates a *read-exclusive* transaction

- tells others about impending write, obtains exclusive ownership
  - makes the write visible, i.e. write is performed
  - may be actually observed (by a read miss) only later
  - write hit made visible when block updated in writer's cache
- only one RdX can succeed at a time per block: serialized by bus

Rd and RdX bus transactions drive coherence actions

- writeback transactions also, but not caused by memory operation and quite incidental to coherence protocol
  - note: replaced block that is not in modified state can be dropped


# Basic Protocol

States

- Invalid (I)
- Shared (S): one or more
- Dirty or Modified (M): one only

Processor events:

- PrRd (read)
- PrWr (write)

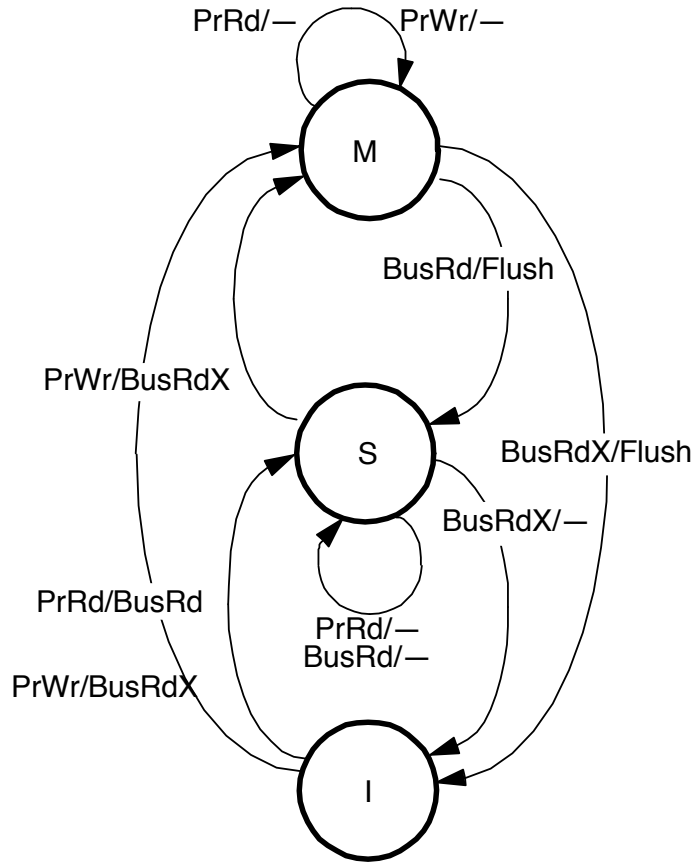
Bus transactions

- BusRd: asks for copy with no intent to modify
- BusRdX: asks for copy with intent to modify
- BusWB: updates memory

Actions

- update state, perform bus transaction, flush value onto bus

# State Diagram




- Write to shared block:
  - Already have latest data; can use upgrade (BusUpgr) instead of BusRdX
- In M state, why go to S on BusRd? Not I?

# Scalability

Big problem with bus-based MPs

- shared bus becomes a bandwidth bottleneck as P increases
- all-to-all communication (broadcast!)
- bus now has communication *and* coherence traffic

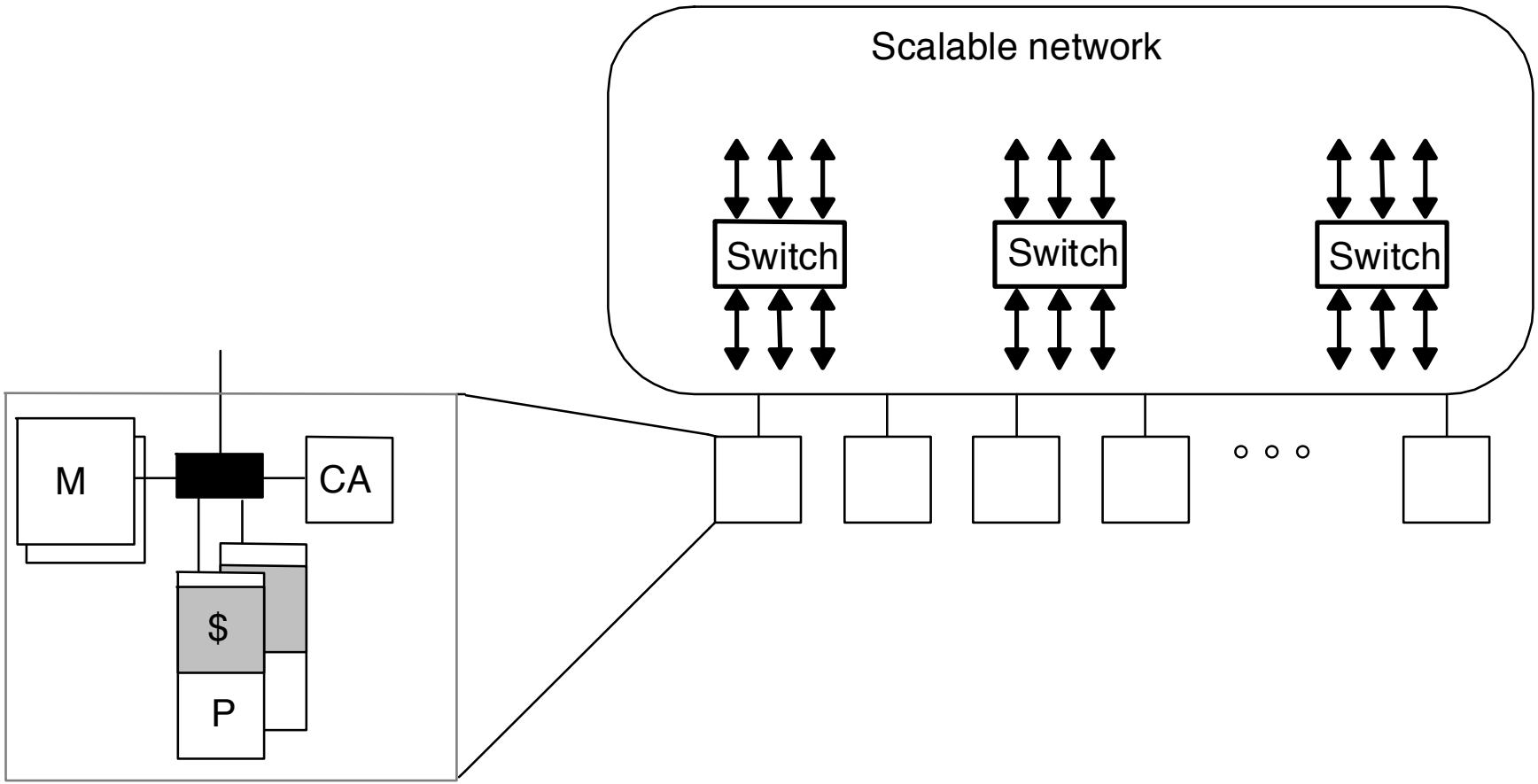
Need a different solution above modest processor count

Use switched network topologies to connect nodes

- point-to-point non-broadcast medium
- much improved bandwidth
- NUMA!
- cache coherence?

CSL

# Distributed Shared Memory (DSM)



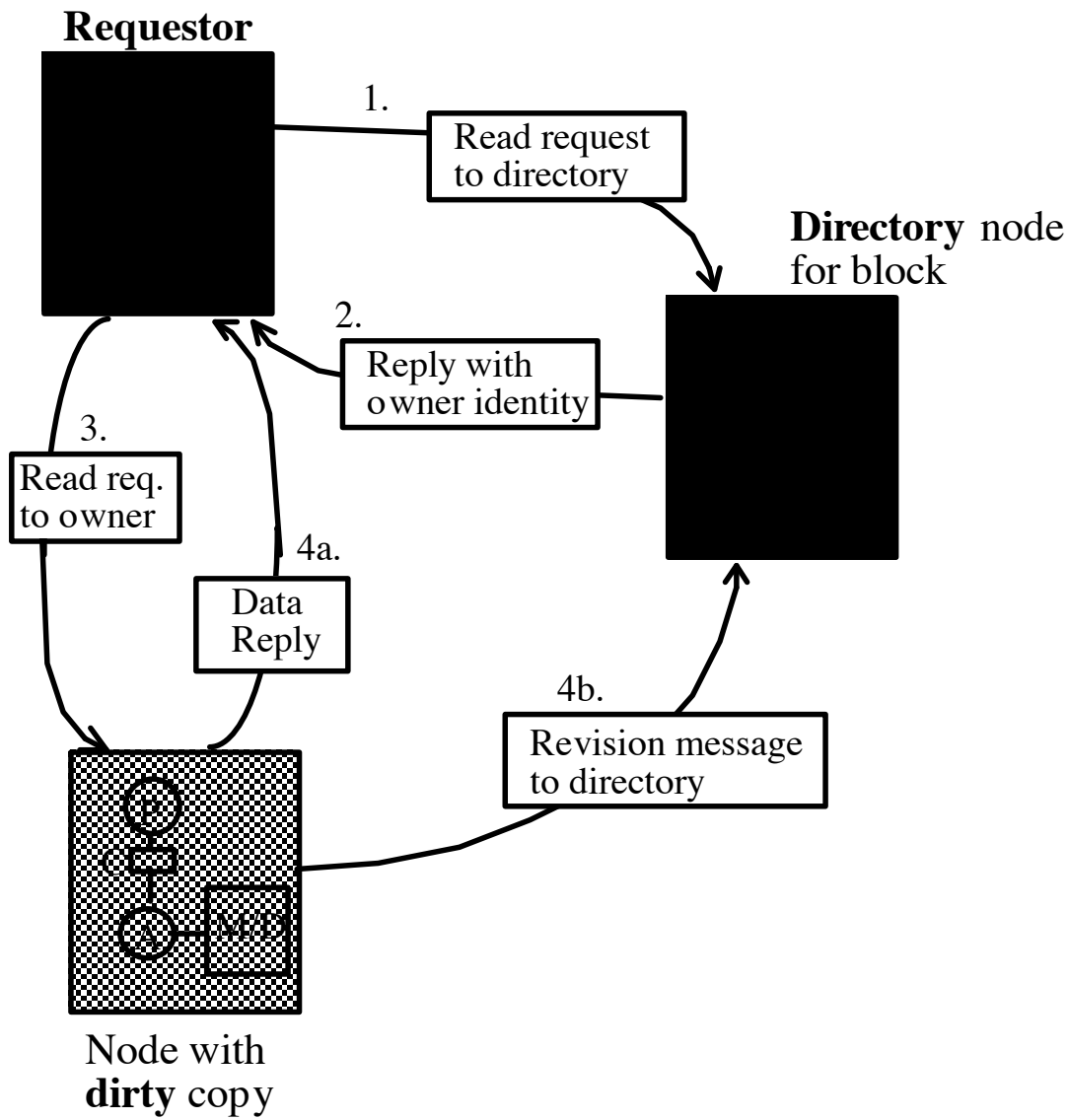

CSL

# Directories

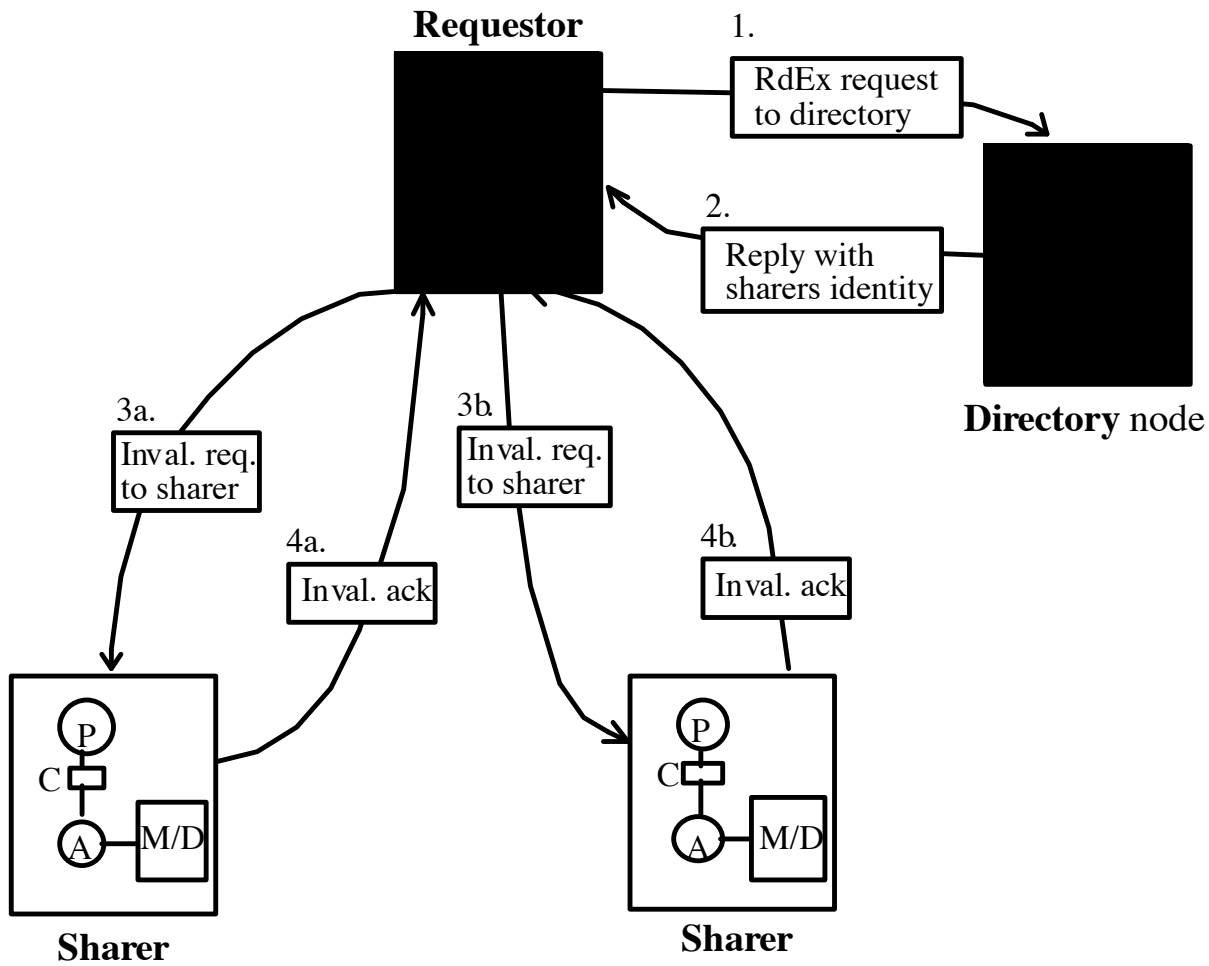
Every memory block has associated directory information

- keeps track of copies of cached blocks and their states
- on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
- in scalable networks, comm. with directory and copies is through *network transactions*


# Example: Read Miss - Dirty Block



# Example: Write Miss - Shared Block

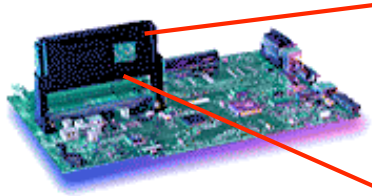


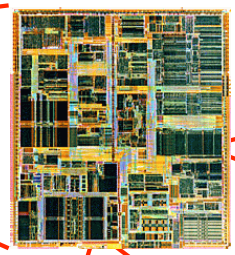

CSL

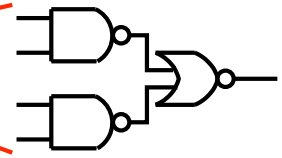
# What 314 Was About

System Architecture

Processor Design







Logic Design

```
jal _getnext
ori $a0,$0,0
lw $t0,8($v0)
lw $t0,12($t0)
beq $t0,0,0x401834
li $t1,4
beq $t0,$t1,0x4018a0
```

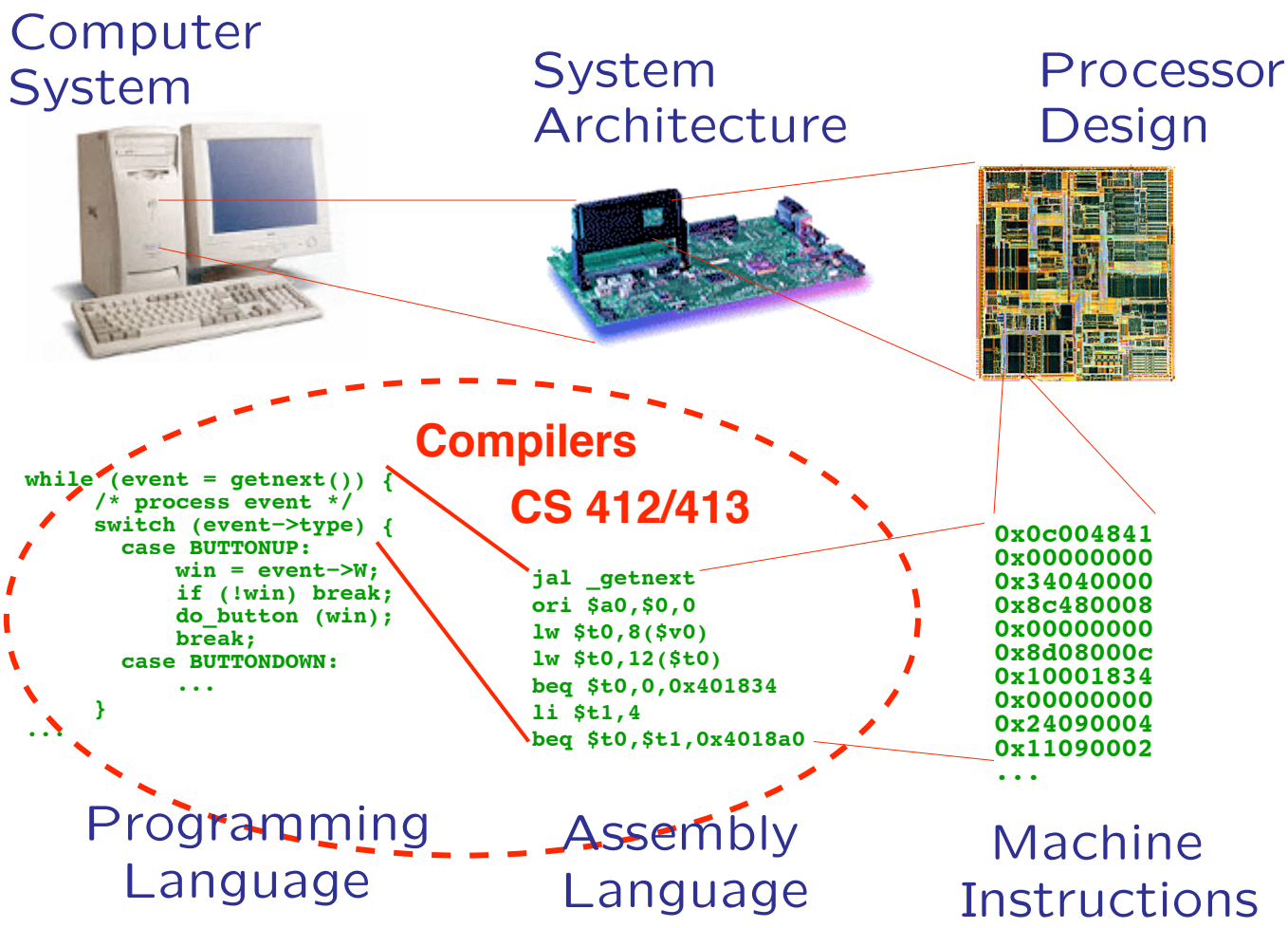
```
0x0c004841
0x00000000
0x34040000
0x8c480008
0x00000000
0x8d08000c
0x10001834
0x00000000
0x24090004
0x11090002
...
```

Assembly Language

Machine Instructions

CSL

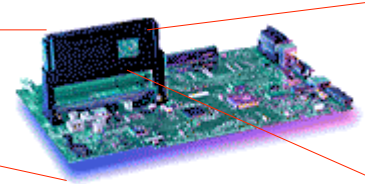
# Where Does 314 Lead?



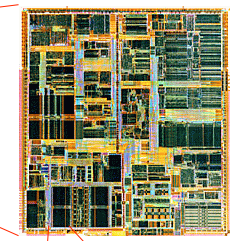
# Where Does 314 Lead?

Computer System

System Architecture



Processor Design



**Operating Systems**

**CS 414/415**

```
while (event = getnext()) {
     /* process event */
     switch (event->type) {
       case BUTTONUP:
            win = event->W;
            if (!win) break;
            do_button (win);
            break;
       case BUTTONDOWN:
            ...
     }
...
```

Programming Language

```
jal _getnext
ori $a0,$0,0
lw $t0,8($v0)
lw $t0,12($t0)
beq $t0,0,0x401834
li $t1,4
beq $t0,$t1,0x4018a0
```

Assembly Language

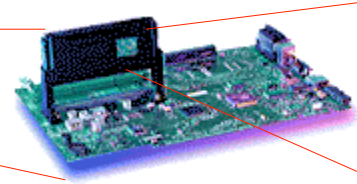
```
0x0c004841
0x00000000
0x34040000
0x8c480008
0x00000000
0x8d08000c
0x10001834
0x00000000
0x24090004
0x11090002
...
```

Machine Instructions

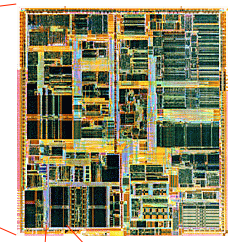
# Where Does 314 Lead?

Computer System

System Architecture



Processor Design



**ECE 475/575**
**Computer Architecture**

```
while (event = getnext()) {
     /* process event */
     switch (event->type) {
       case BUTTONUP:
           win = event->W;
           if (!win) break;
           do_button (win);
           break;
       case BUTTONDOWN:
           ...
     }
...
```

Programming Language

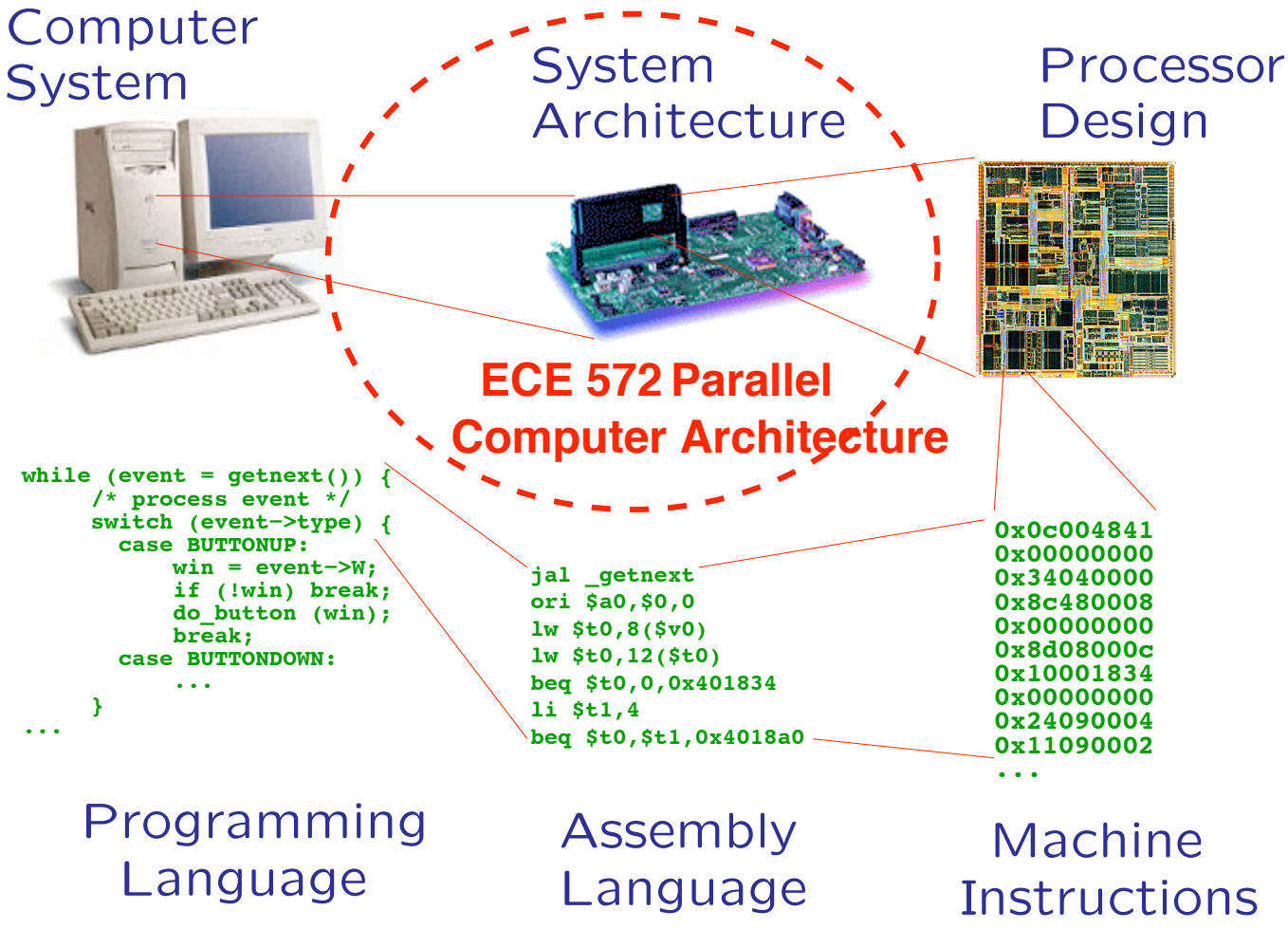
```
jal _getnext
ori $a0,$0,0
lw $t0,8($v0)
lw $t0,12($t0)
beq $t0,0,0x401834
li $t1,4
beq $t0,$t1,0x4018a0
```

Assembly Language

```
0x0c004841
0x00000000
0x34040000
0x8c480008
0x00000000
0x8d08000c
0x10001834
0x00000000
0x24090004
0x11090002
...
```

Machine Instructions

# Where Does 314 Lead?

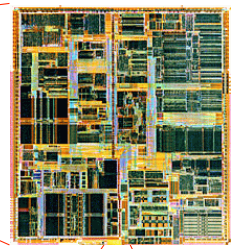


# Where Does 314 Lead?

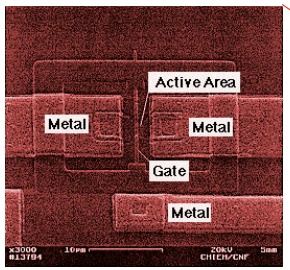
Computer System

System Architecture

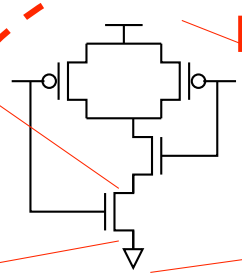
Processor Design



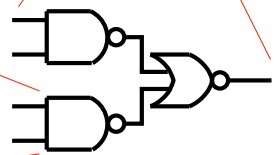
ECE 474/574
Digital VLSI Design




Device Fabrication



Circuit/VLSI Design



Logic Design