On Power and Multi-Processors

Finishing up power issues and how those issues have led us to multi-core processors.

Introduce multi-processor systems.

Capacitive Power dissipation

Capacitance:
Function of wire length, transistor size

Supply Voltage:
Has been dropping with successive fab generations

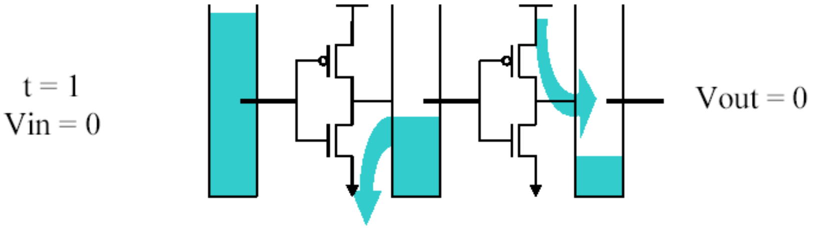
$$\text{Power} \sim \frac{1}{2} CV^2Af$$

Activity factor:
How often, on average, do wires switch?

Clock frequency:
Increasing…

With more voltage you can get a higher frequency

- Back to our water analogy:
  - The higher voltage is a higher water tower so a higher water pressure.
  - The “buckets” fill up faster
  - The circuit is faster.



- This is roughly a linear relationship over a fairly small dynamic range.

And so…

- Power ~ ½ CV^2Af.
  - We can scale frequency with voltage.
  - We can claim that power ~proportional to f^3.
- Performance ~f.
  - Doubling frequency doesn't double performance (memory latency) but it's close enough for our "~"
- Say we have a processor that uses 100W and can do 1 billion operations per second (1GOP).
  - Increasing performance to 1.1 GOPs with voltage/frequency scaling will need $(1.1)^3$ *100W= 133W

Can we do better?

- Can’t we get speedup in other ways than frequency scaling?
  - Of course. Bigger caches, increasing superscalar, etc.
  - But most of these have a pretty high performance/power ratio also.
    - Consider being more superscalar
      - The cost for each “level” is more than the previous (why?)
      - The benefit of each “level” is less than the previous (why?)
    - Caches are similar.
      - Doubling a cache size *roughly* halves its miss rate.
        - » That’s a doubling in cache power for fewer and fewer misses removed (and higher and higher latency!)

Sample problem

- Say I have a 200W budget and a single-core processor that can do 10 GOPs.
  - What performance would I expect to get out of two cores using the same power budget?
  - Four cores?
  - Eight cores?
- Does this really scale so nicely?
  - Of course not, but it's another dimension to extract performance from
    - Thread-level!

So…

- Multiprocessors seem like a good place to improve performance with a more reasonable power cost.

Why multi-processors?

- **Why multi-processors?**
  - Multi-processors have been around for a long time.
    - Originally used to get best performance for certain highly-parallel tasks.

  - We now use them to get solid performance per unit of energy.
    - Basic theme: it’s much less energy to do two things slowly than one thing twice as fast.

- **So that’s it?**
  - Not so much.
    - We need to make it possible/reasonable/easy to use.
    - Nothing comes for free.
      - If we take a task and break it up so it runs on a number of processors, there is going to be a price.

Thread-Level Parallelism

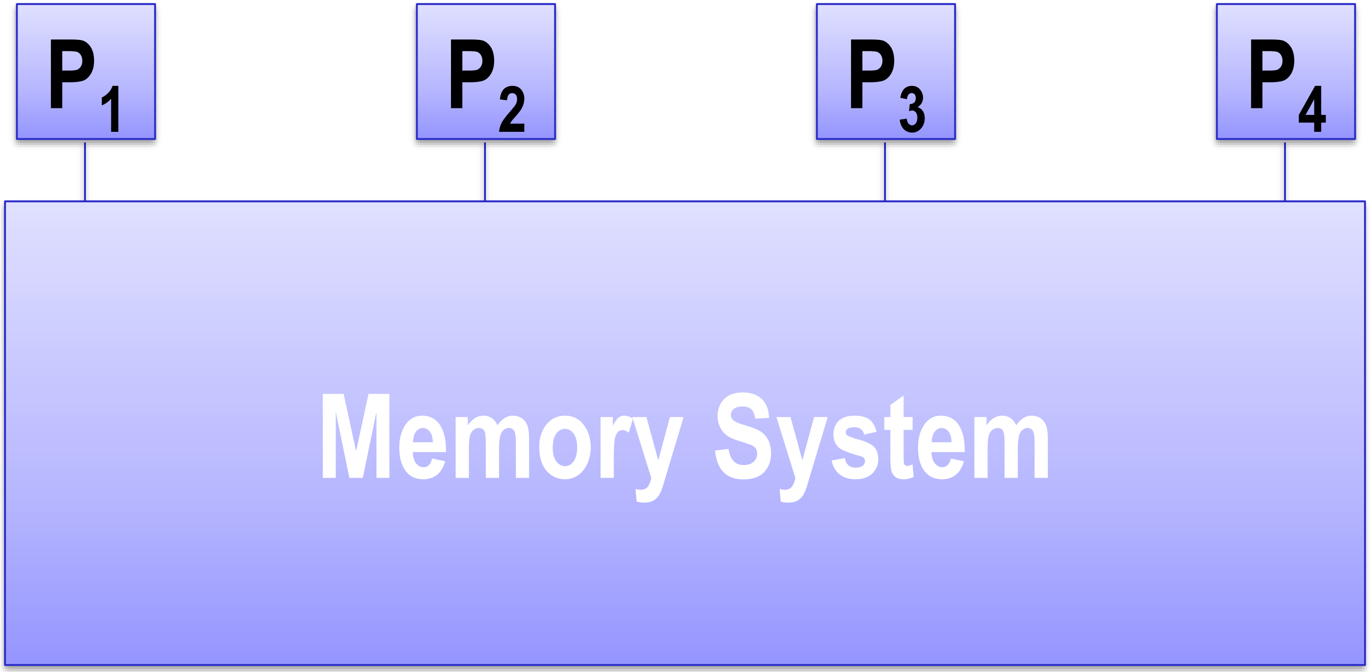
```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id,amt;
if (accts[id].bal >= amt)
{
   accts[id].bal -= amt;
   spew_cash();
}
```

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
6: ... ... ...
```

- **Thread-level parallelism (TLP)**
  - Collection of asynchronous tasks: not started and stopped together
  - Data shared loosely, dynamically
- Example: database/web server (each query is a thread)
  - `accts` is **shared**, can't register allocate even if it were scalar
  - `id` and `amt` are private variables, register allocated to `r1`, `r2`

Shared-Memory Multiprocessors

- **Shared memory**
  - Multiple execution contexts sharing a single address space
    - Multiple programs (MIMD)
    - Or more frequently: multiple copies of one program (SPMD)
  - Implicit (automatic) communication via loads and stores



What's the other option?

- **Basically the only other option is "message passing"**
  - We communicate via explicit messages.
  - So instead of just changing a variable, we'd need to call a function to pass a specific message.
- **Message passing systems are easy to build and pretty efficient.**
  - But harder to code.
- **Shared memory programming is basically the same as multi-threaded programming on one processors**
  - And (many) programmers already know how to do that.

So Why Shared Memory?

Pluses

- For applications looks like multitasking uniprocessor
- For OS only evolutionary extensions required
- Easy to do communication without OS being involved
- Software can worry about correctness first then performance

Minuses

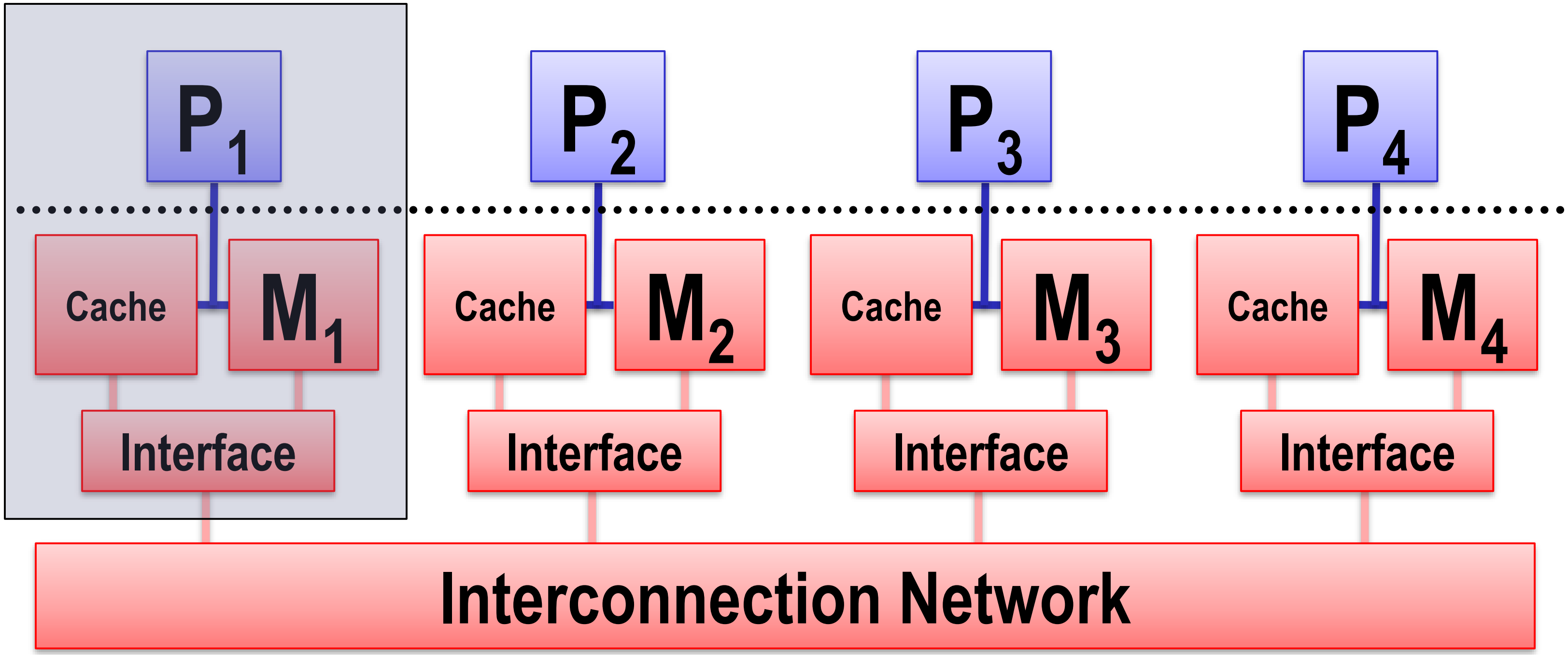
- Proper synchronization is complex
- Communication is implicit so harder to optimize
- Hardware designers must implement

Result

- Traditionally bus-based Symmetric Multiprocessors (SMPs), and now the CMPs are the most success parallel machines ever
- And the first with multi-billion-dollar markets

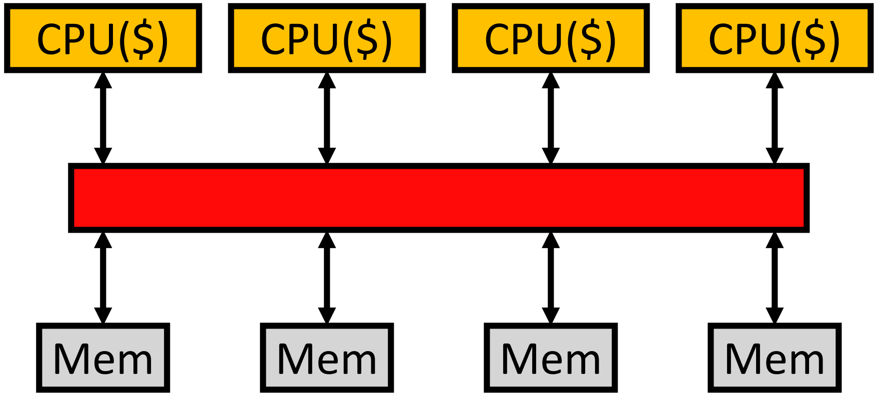
Shared-Memory Multiprocessors

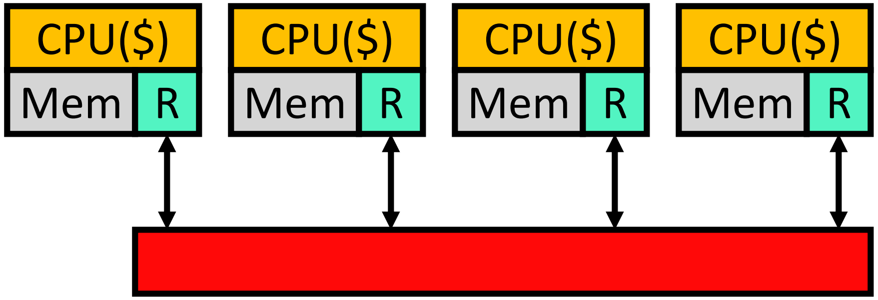
- There are lots of ways to connect processors together



Paired vs. Separate Processor/Memory?

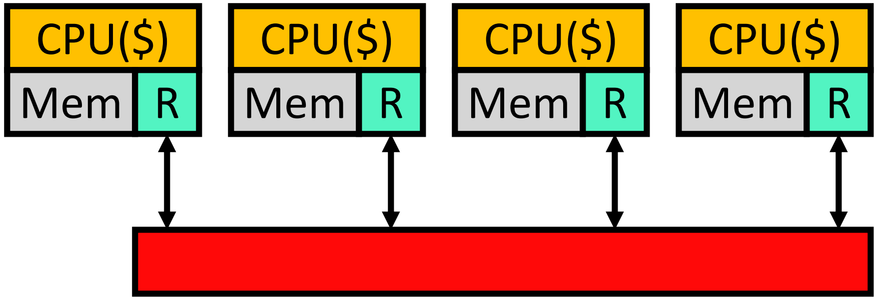
- **Separate processor/memory**
  - **Uniform memory access (UMA):** equal latency to all memory
  - \+ Simple software, doesn't matter where you put data
  - – Lower peak performance
  - Bus-based UMAs common: **symmetric multi-processors (SMP)**
- **Paired processor/memory**
  - **Non-uniform memory access (NUMA)**: faster to local memory
  - – More complex software: where you put data matters
  - \+ Higher peak performance: assuming proper data placement

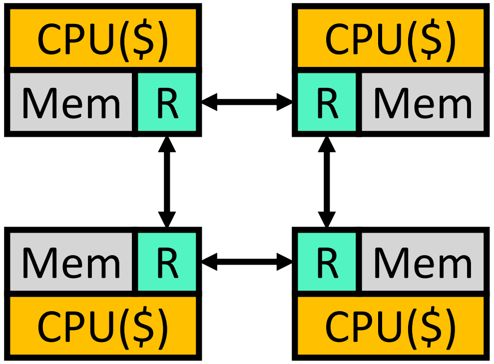




Shared vs. Point-to-Point Networks

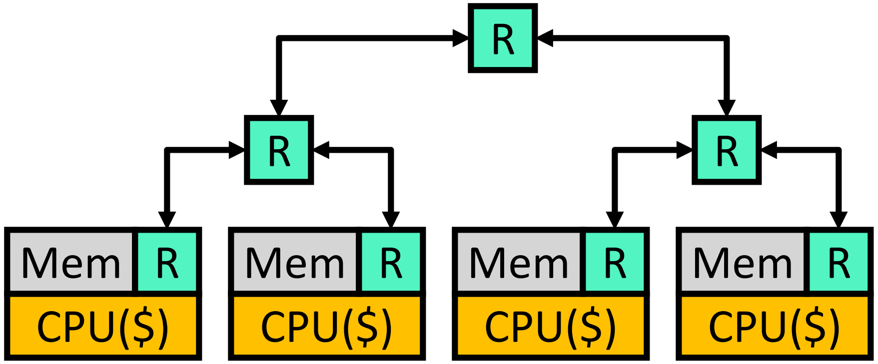
- **Shared network**: e.g., bus (left)
  - + Low latency
  - – Low bandwidth: doesn’t scale beyond ~16 processors
  - + Shared property simplifies cache coherence protocols (later)
- **Point-to-point network**: e.g., mesh or ring (right)
  - – Longer latency: may need multiple “hops” to communicate
  - + Higher bandwidth: scales to 1000s of processors
  - – Cache coherence protocols are complex

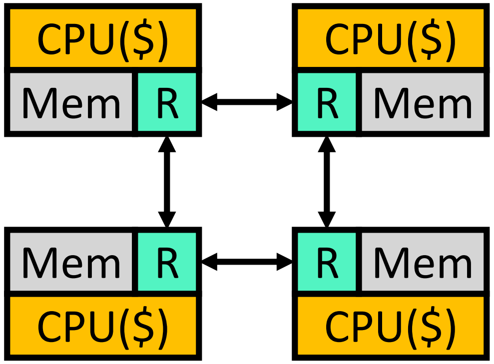




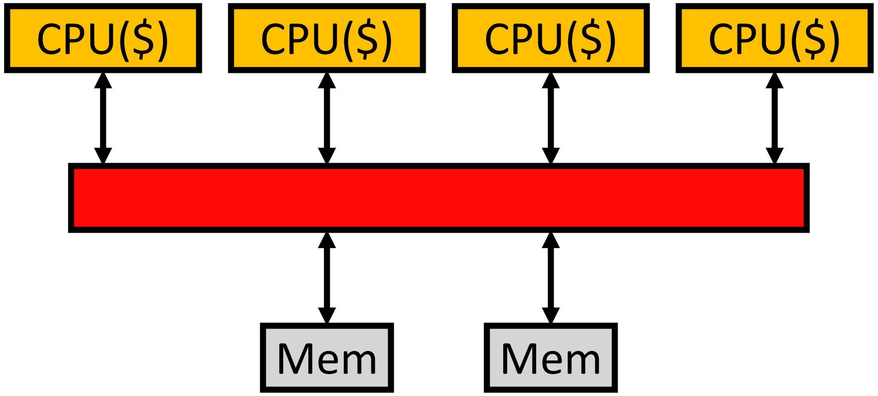
Organizing Point-To-Point Networks

- **Network topology**: organization of network
  - Tradeoff performance (connectivity, latency, bandwidth) ↔ cost
- Router chips
  - Networks that require separate router chips are **indirect**
  - Networks that use processor/memory/router packages are **direct**
    - Fewer components, “Glueless MP”
- Point-to-point network examples
  - Indirect tree (left)
  - Direct mesh or ring (right)



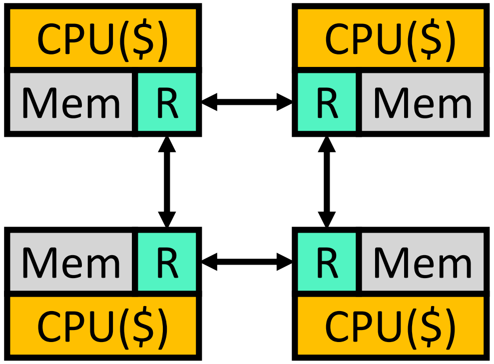


Implementation #1: Snooping Bus MP



- Two basic implementations
- Bus-based systems
  - Typically small: 2–8 (maybe 16) processors
  - Typically processors split from memories (UMA)
    - Sometimes **multiple processors on single chip (CMP)**
    - **Symmetric multiprocessors (SMPs)**
    - Common, I use one everyday

Implementation #2: Scalable MP



- General point-to-point network-based systems
  - Typically processor/memory/router blocks (NUMA)
    - **Glueless MP**: no need for additional “glue” chips
  - Can be arbitrarily large: 1000’s of processors
    - **Massively parallel processors (MPPs)**
  - In reality only government (DoD) has MPPs…
    - Companies have much smaller systems: 32–64 processors
    - **Scalable multi-processors**

Issues for Shared Memory Systems

- Two in particular
  - Cache coherence
  - Memory consistency model

- Closely related to each other

An Example Execution

```
Processor 0                 Processor 1
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash           0: addi r1,accts,r3
                            1: ld 0(r3),r4
                            2: blt r4,r2,6
                            3: sub r4,r2,r4
                            4: st r4,0(r3)
                            5: call spew_cash
```

CPU0	CPU1	Mem

- Two \$100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track `accts[241].bal` (address is in `r3`)

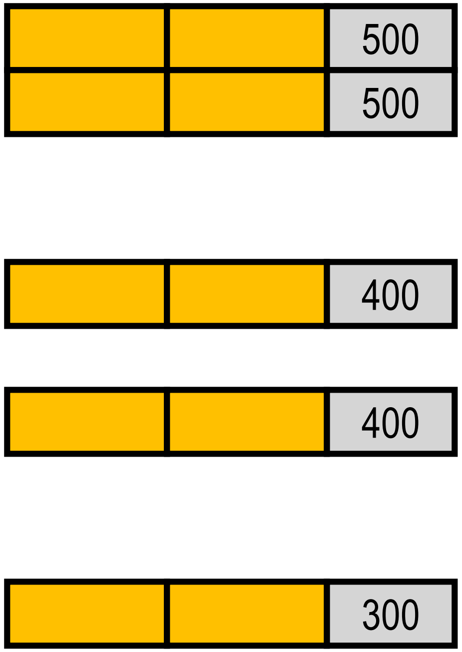
No-Cache, No-Problem

Processor 0

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

Processor 1

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```



- Scenario I: processors have no caches
  - No problem

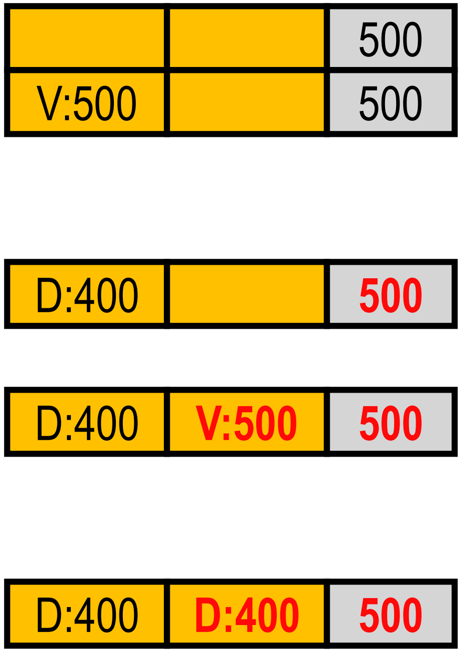
Cache Incoherence

Processor 0

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

Processor 1

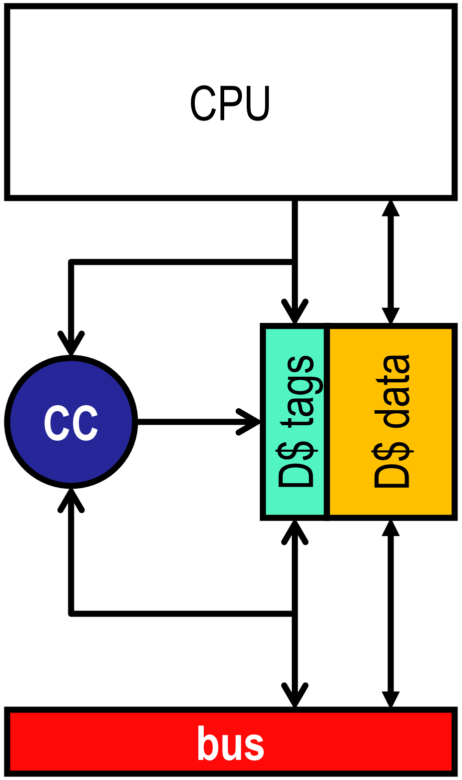
```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```



		500
V:500		500
D:400		500
D:400	V:500	500
D:400	D:400	500

- Scenario II: processors have write-back caches
  - Potentially 3 copies of **`accts[241].bal`**: memory, p0$, p1$
  - Can get incoherent (inconsistent)

Hardware Cache Coherence



- **Coherence controller**:
  - Examines bus traffic (addresses and data)
  - Executes **coherence protocol**
    - What to do with local copy when you see different things happening on bus

Snooping Cache-Coherence Protocols

Bus provides serialization point

Each cache controller “snoops” all bus transactions

- take action to ensure coherence
  - invalidate
  - update
  - supply value
- depends on state of the block and the protocol

Snooping Design Choices

Controller updates state of blocks in response to processor and snoop events and generates bus xactions

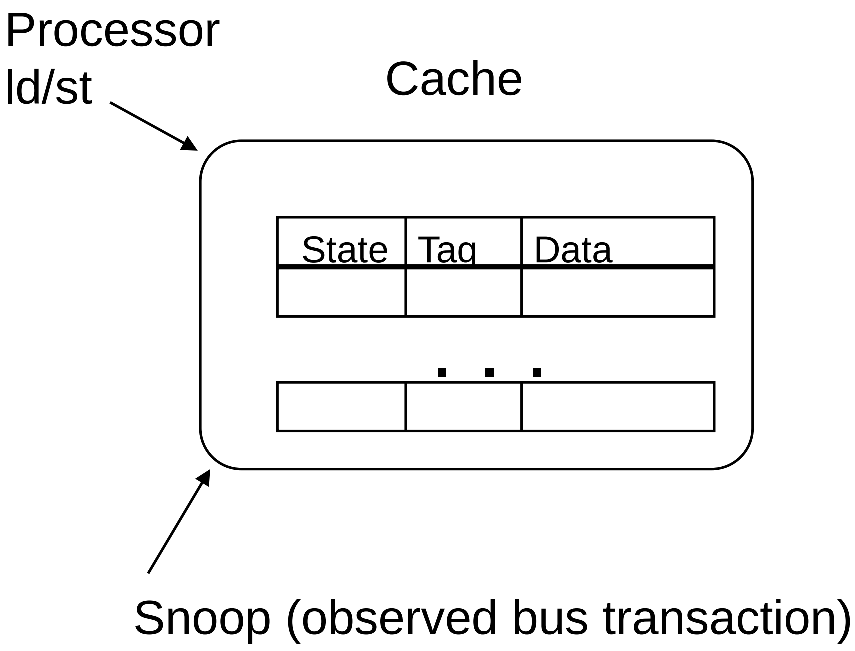
Often have duplicate cache tags

Snoopy protocol

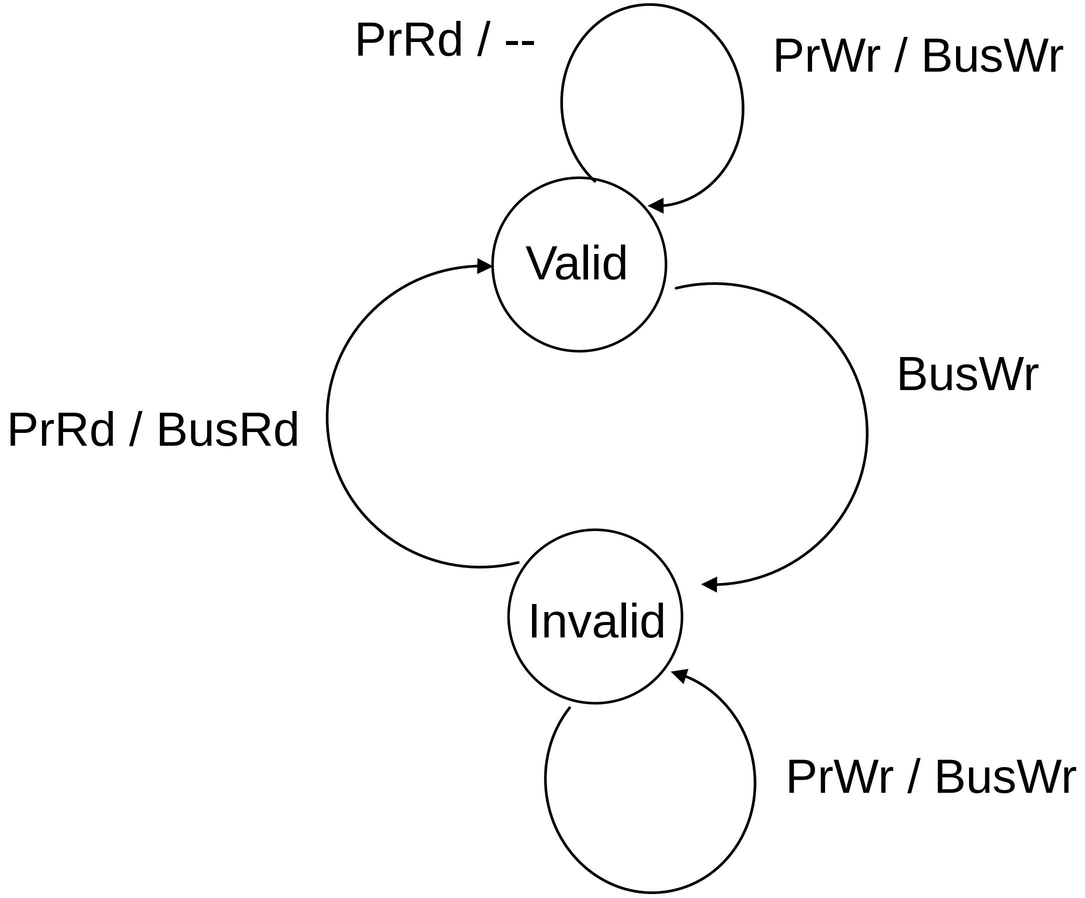
- set of states
- state-transition diagram
- actions

Basic Choices

- write-through vs. write-back
- invalidate vs. update



The Simple Invalidate Snooping Protocol

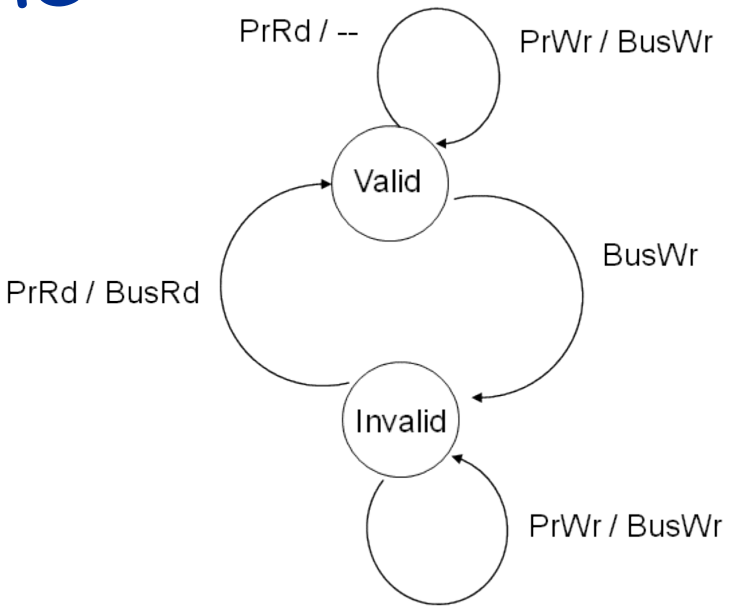


Write-through, no-write-allocate cache

Actions: PrRd, PrWr, BusRd, BusWr

Example time

Processor 1		**Processor 2**		**Bus**
Processor Transaction	Cache State	Processor Transaction	Cache State	
Read A				
Read A				
		Read A		
Write A				
		Read A		
		Write A		
Write A				



Actions:
- PrRd, PrWr,
- BusRd, BusWr

More Generally: MOESI

[Sweazey & Smith ISCA86]

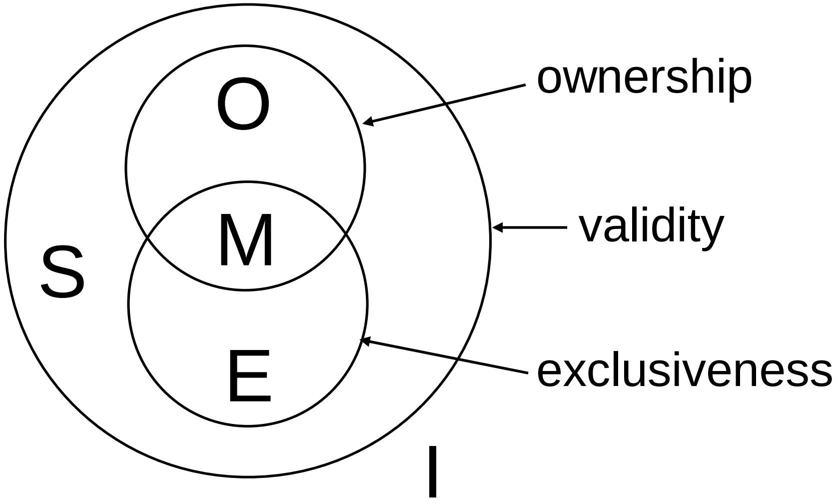
M - Modified (dirty)

O - Owned (dirty but shared)   WHY?

E - Exclusive (clean unshared) only copy, not dirty

S - Shared

I - Invalid



Variants

- MSI
- MESI
- MOSI
- MOESI

MESI example

Actions:

- PrRd, PrWr,
- BRL – Bus Read Line (BusRd)
- BWL – Bus Write Line (BusWr)
- BRIL – Bus Read and Invalidate
- BIL – Bus Invalidate Line

Processor 1		**Processor 2**		**Bus**
Processor Transaction	Cache State	Processor Transaction	Cache State	
Read A				
Read A				
		Read A		
Write A				
		Read A		
		Write A		
Write A				

- M - Modified (dirty)
- E - Exclusive (clean unshared) only copy, not dirty
- S - Shared
- I - Invalid