

Cache Coherence Protocol

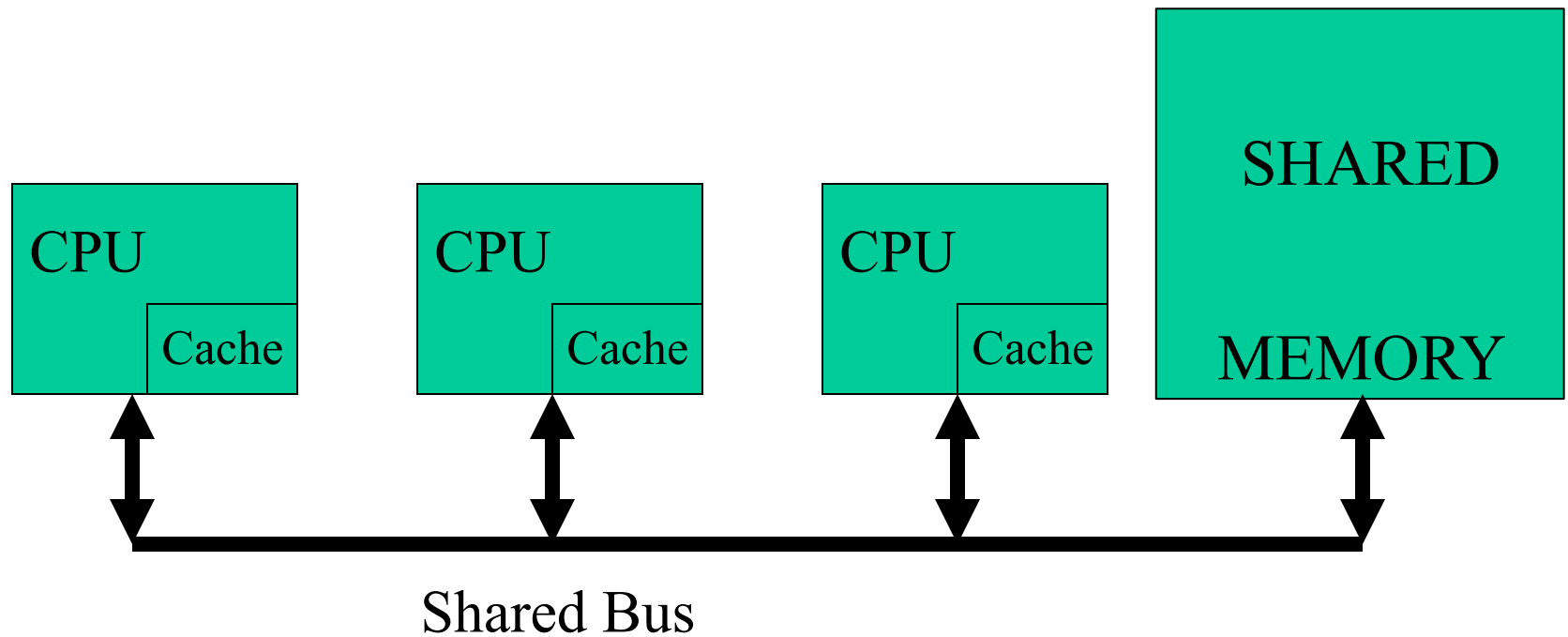
**for a multi-core processor with
Shared Cache Memory**

SHUBHAM PANDIT
SAI SHUBHAM

19110157
19110132

Bus-based Shared Memory Organization

Basic picture is simple :-



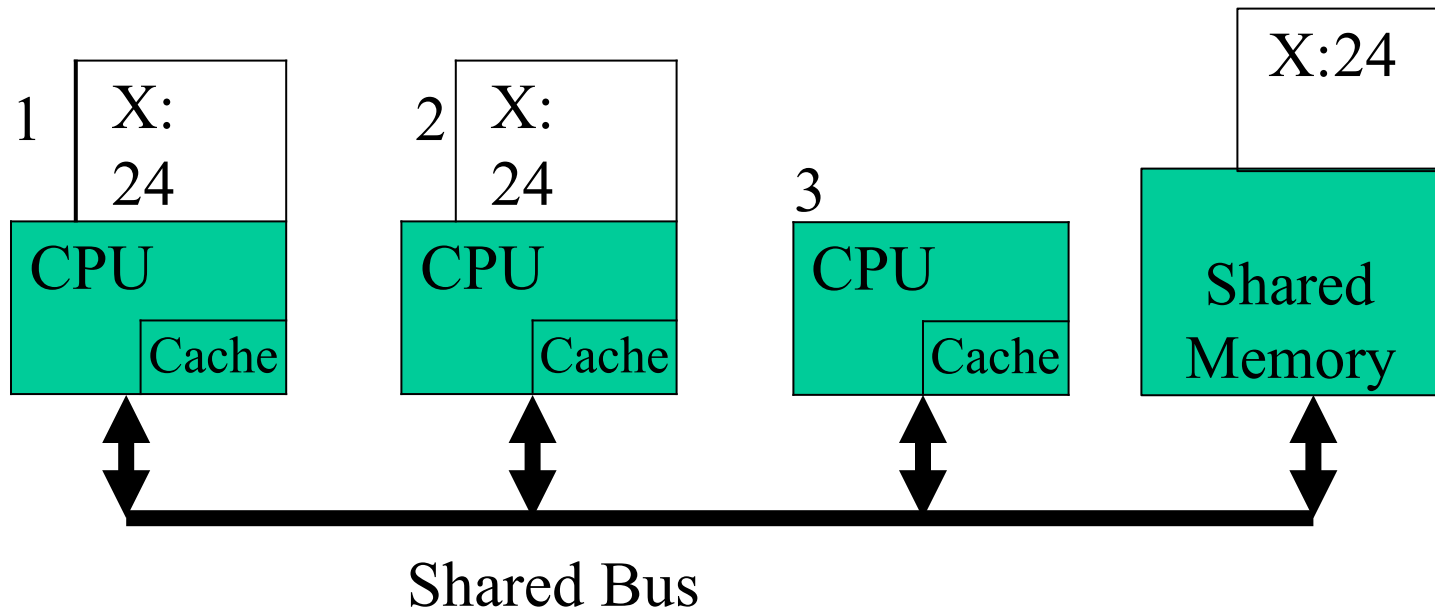
Organization

- Bus is usually simple physical connection (wires)
- Bus bandwidth limits no. of CPUs
- For now, assume that each CPU has only a single level of cache

Problem of Memory Coherence

- Assume just single level caches and main memory
- Processor writes to location in its cache
- Other caches may hold shared copies - these will be out of date
- Updating main memory alone is not enough

Example

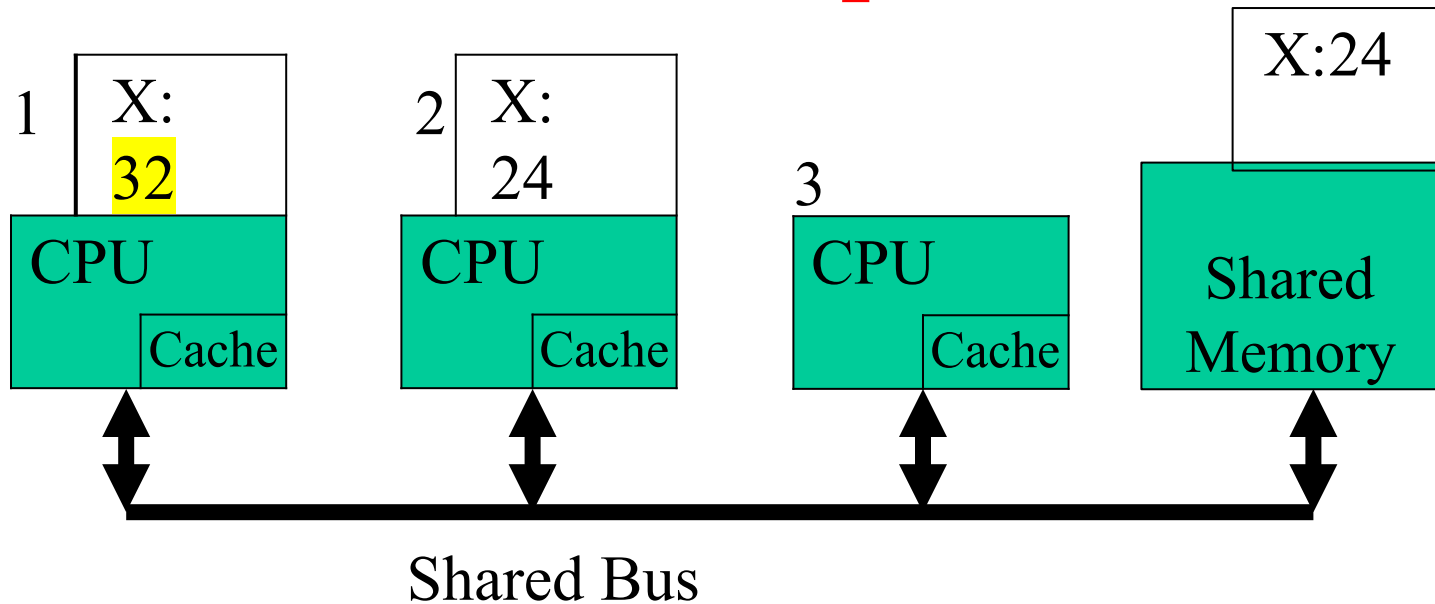


Processor 1 reads X: obtains 24 from memory and caches it
Processor 2 reads X: obtains 24 from memory and caches it
Processor 1 writes 32 to X: its locally cached copy is updated
Processor 3 reads X: what value should it get?

Memory and processor 2 think it is 24
Processor 1 thinks it is 32

Notice that having write-through caches is not good enough

Example



Processor 1 reads X: obtains 24 from memory and caches it
Processor 2 reads X: obtains 24 from memory and caches it
Processor 1 writes 32 to X: its locally cached copy is updated
Processor 3 reads X: what value should it get?

Memory and processor 2 think it is 24
Processor 1 thinks it is 32

Notice that having write-through caches is not good enough

Bus Snooping

- Each CPU (cache system) ‘snoops’ (i.e. watches continually) for write activity concerned with data addresses which it has cached.
- This assumes a bus structure which is ‘global’, i.e all communication can be seen by all.

Snooping Protocols

- Write Invalidate

- CPU wanting to write to an address, grabs a bus cycle and sends a ‘write invalidate’ message
- All snooping caches invalidate their copy of appropriate cache line
- CPU writes to its cached copy (assume for now that it also writes through to memory)
- Any shared read in other CPUs will now miss in cache and re-fetch new data.

MESI Protocol (1)

- A practical multiprocessor invalidate protocol which attempts to minimize bus usage.
- Allows usage of a ‘write back’ scheme - i.e. main memory not updated until ‘dirty’ cache line is displaced
- Extension of usual cache tags, i.e. invalid tag and ‘dirty’ tag in normal write back cache.

MESI Protocol (2)

Any cache line can be in one of 4 states (2 bits)

- **Modified** - cache line has been modified, is different from main memory - is the only cached copy. (multiprocessor 'dirty')
- **Exclusive** - cache line is the same as main memory and is the only cached copy
- **Shared** - Same as main memory but copies may exist in other caches.
- **Invalid** - Line data is not valid (as in simple cache)

MESI Protocol (3)

- Cache line changes state as a function of memory access events.
- Event may be either
 - Due to local processor activity (i.e. cache access)
 - Due to bus activity - as a result of snooping
- Cache line has its own state affected only if address matches

MESI Protocol (4)

- Operation can be described informally by looking at action in local processor
 - Read Hit
 - Read Miss
 - Write Hit
 - Write Miss
- More formally by state transition diagram

MESI Local Read Hit

- Line must be in one of MES
- This must be correct local value (if M it must have been modified locally)
- Simply return value
- No state change

MESI Local Read Miss (1)

- No other copy in caches
 - Processor makes bus request to memory
 - Value read to local cache, marked E
- One cache has E copy
 - Processor makes bus request to read
 - Memory provides the data
 - Local processor caches value
 - Both lines set to S

MESI Local Read Miss (2)

- Several caches have S copy
 - Processor makes bus request to read
 - Memory provides the data on the bus
 - Local processor caches value
 - Local copy set to S
 - Other copies remain S

MESI Local Read Miss (3)

- One cache has M copy
 - Processor makes bus request to read
 - Snooping cache puts copy value on the bus
 - Memory access is abandoned
 - Local processor caches value
 - Local copy tagged S
 - **Source (M) value copied back to memory**
 - Source value M \rightarrow S

MESI Local Write Hit (1)

Line must be one of MES

- M
 - line is exclusive and already 'dirty'
 - Update local cache value
 - no state change
- E
 - Update local cache value
 - State E -> M

MESI Local Write Hit (2)

- S
 - Processor broadcasts an invalidate on bus
 - Snooping processors with S copy change S->I
 - Local cache value is updated
 - Local state change S->M

MESI Local Write Miss (1)

Detailed action depends on copies in other processors

- No other copies
 - Value read from memory to local cache
 - Value updated
 - Local copy state set to M

MESI Local Write Miss (2)

- Other copies, either one in state E or more in state S
 - Value read from memory to local cache - bus transaction marked RWITM (read with intent to modify)
 - Snooping processors see this and set their copy state to I
 - Local copy updated & state set to M

MESI Local Write Miss (3)

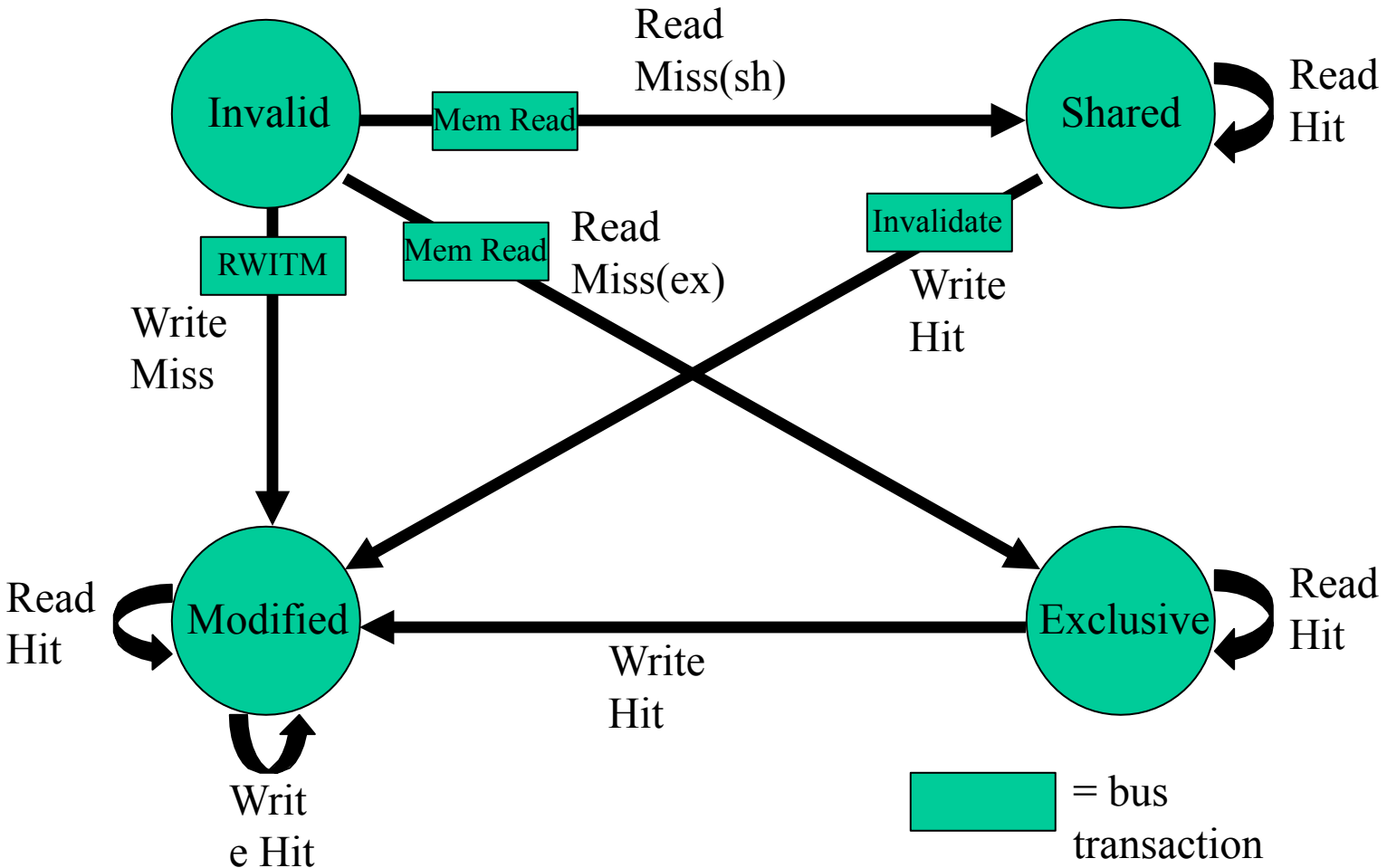
Another copy in state M

- Processor issues bus transaction marked RWITM
- Snooping processor sees this
 - Blocks RWITM request
 - Takes control of bus
 - Puts data on the bus and writes back its copy to memory (Write Back Method)
 - Sets its copy state to I
- Local Processor caches the data, followed by modifying (marked as M)

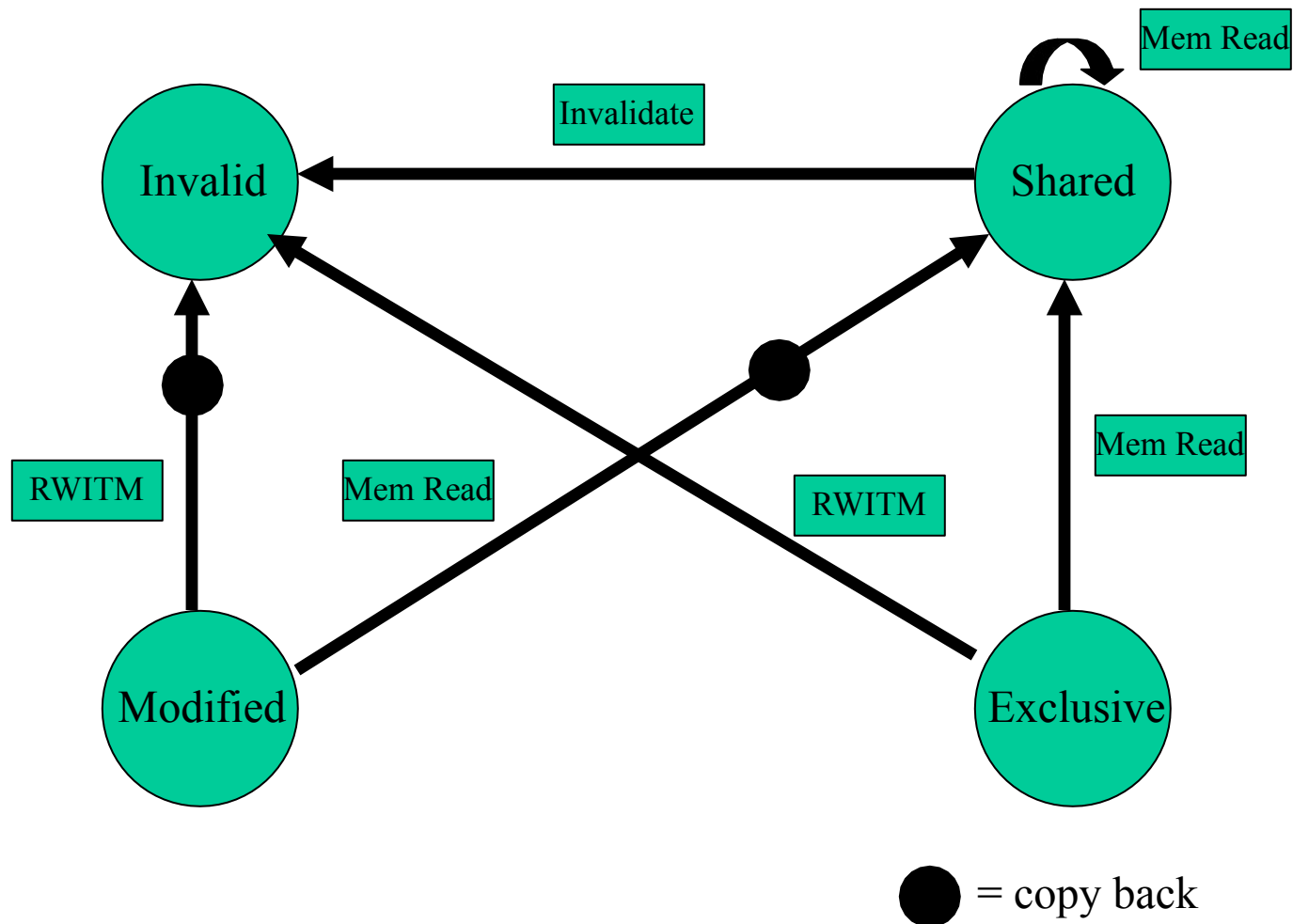
Putting it all together

- All of this information can be described compactly using a state transition diagram
- Diagram shows what happens to a cache line in a processor as a result of
 - memory accesses made by that processor (read hit/miss, write hit/miss)
 - memory accesses made by other processors that result in bus transactions observed by this snoopy cache (Mem read, RWITM, Invalidate)

MESI – locally initiated accesses



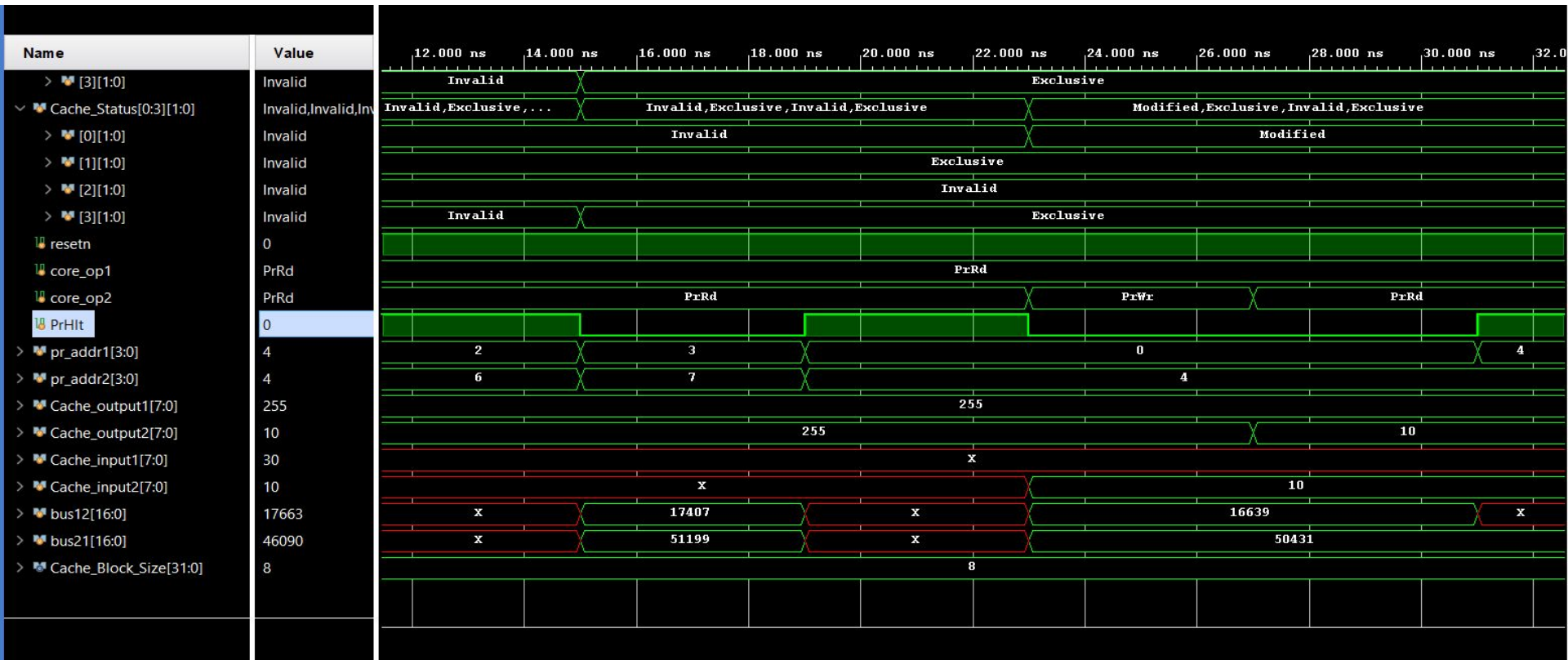
MESI – remotely initiated accesses



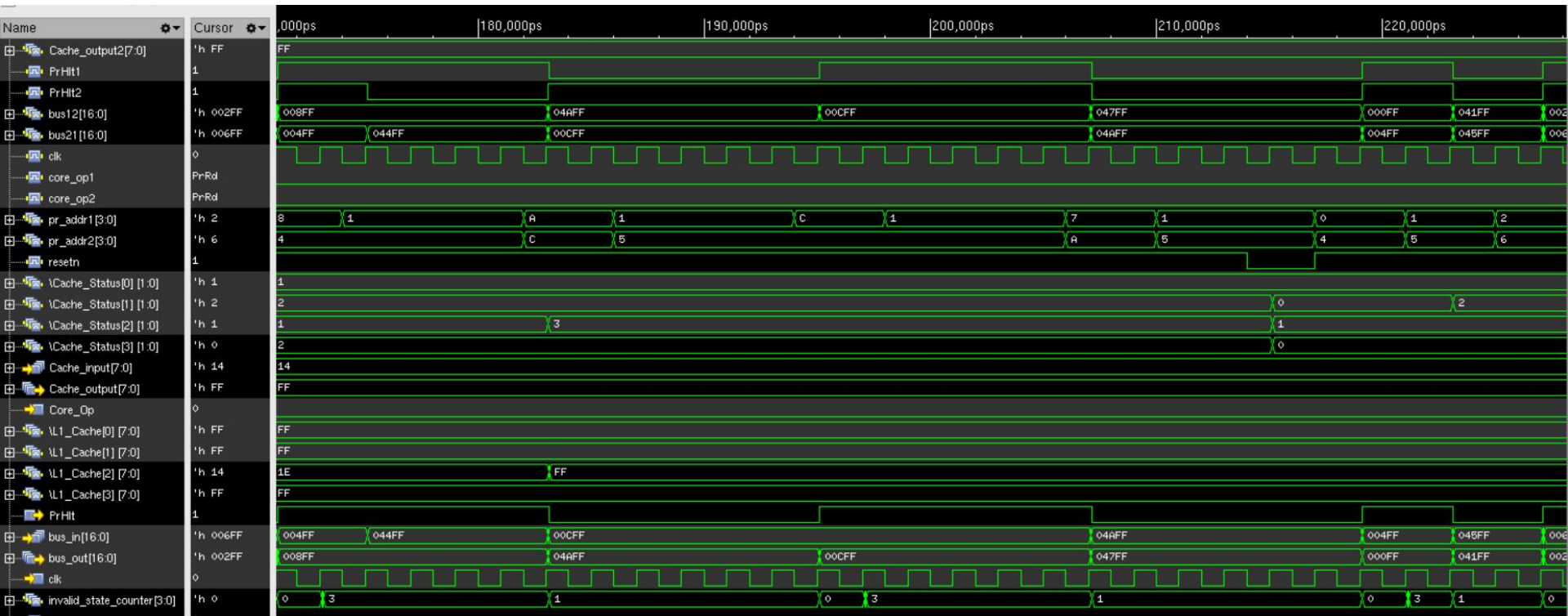
MESI notes

- There are minor variations (particularly to do with write miss)
- Normal ‘write back’ when cache line is evicted is done if line state is M
- Multi-level caches
 - If caches are inclusive, only the lowest level cache needs to snoop on the bus

Pre-Synthesis Simulation Results

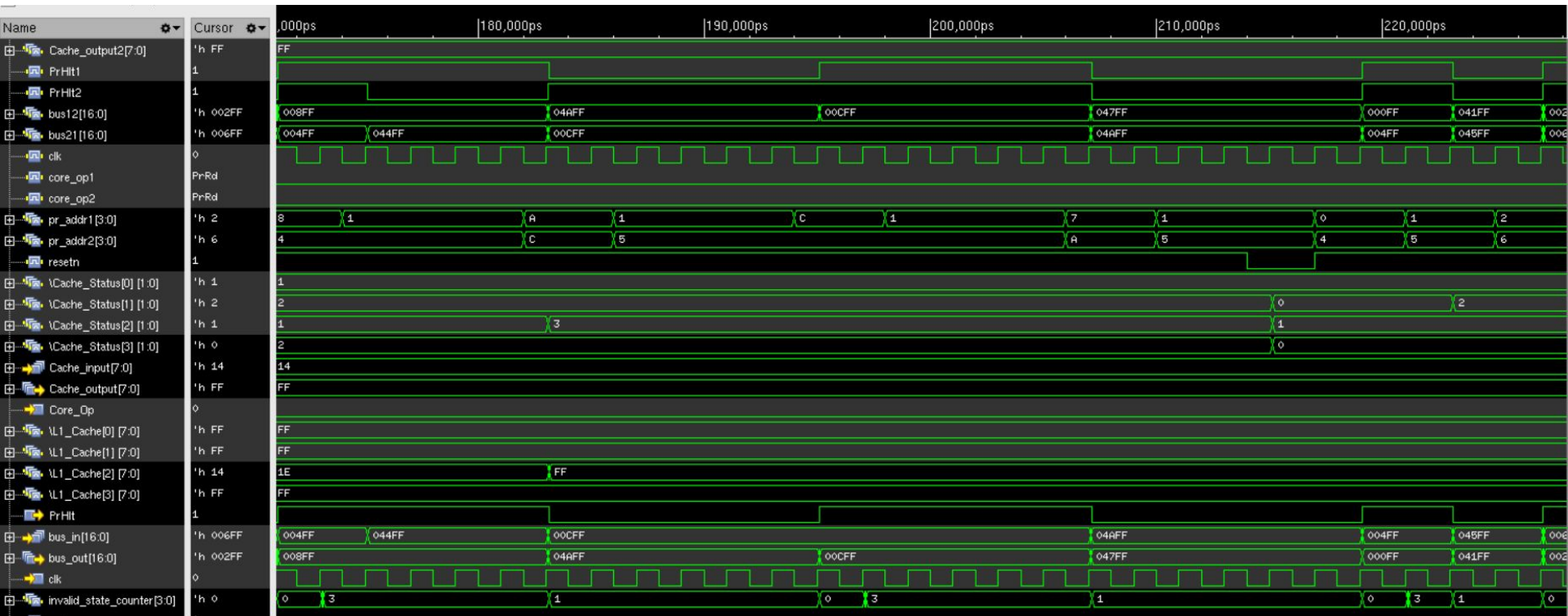


Post-Synthesis Simulation Results



The Design is free of any Setup or Hold Violations

Post-Synthesis Simulation Results



The Design is free of any Setup or Hold Violations

Post-Synthesis Power Report

1	Instance: /MESI_FSM					
2	Power Unit: W					
3	PDB Frames: /stim#0/frame#0					
4	-----					
5	Category	Leakage	Internal	Switching	Total	Row%
6	-----					
7	memory	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.00%
8	register	3.98281e-08	3.96270e-04	3.63466e-05	4.32656e-04	58.76%
9	latch	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.00%
10	logic	5.46895e-08	1.33174e-04	1.29903e-04	2.63132e-04	35.74%
11	bbox	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.00%
12	clock	0.000000e+00	0.000000e+00	4.04640e-05	4.04640e-05	5.50%
13	pad	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.00%
14	pm	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.00%
15	-----					
16	Subtotal	9.45177e-08	5.29444e-04	2.06714e-04	7.36252e-04	100.00%
17	Percentage	0.01%	71.91%	28.08%	100.00%	100.00%
18	-----					

Post-Synthesis Area Report

```
1 =====
2  Generated by:      Genus(TM) Synthesis Solution 21.10-p002_1
3  Generated on:     Apr 10 2023  01:19:38 am
4  Module:           MESI_FSM
5  Technology library: uk65lscllmvbbr_120c25_tc
6  Operating conditions: uk65lscllmvbbr_120c25_tc (balanced_tree)
7  Wireload mode:    top
8  Area mode:        timing library
9 =====
10
11 Instance Module  Cell Count  Cell Area  Net Area  Total Area  Wireload
12 -----
13 MESI_FSM         463      1418.760    0.000     1418.760    wl0 (D)
14   (D) = wireload is default in technology library
15   (T) = wireload mode is 'top'
```


Post-Synthesis Timing Report

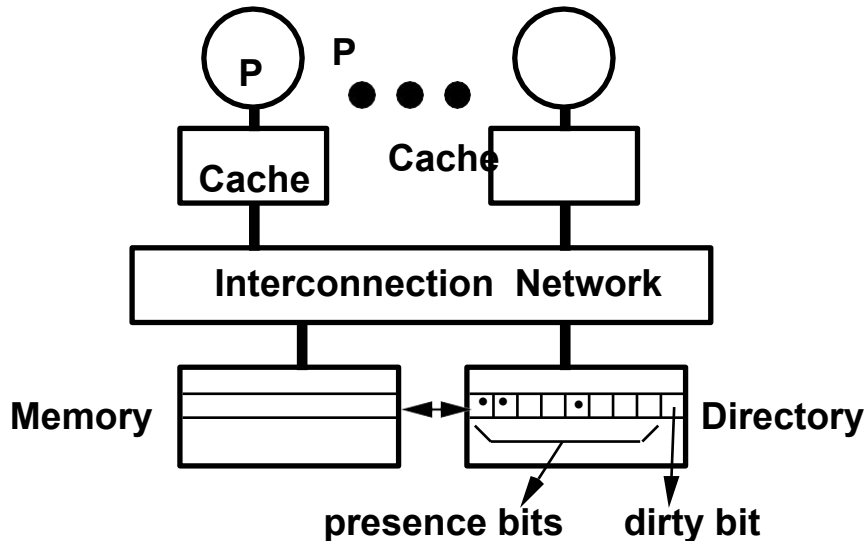
```
1 =====
2  Generated by:      Genus(TM) Synthesis Solution 21.10-p002_1
3  Generated on:     Apr 10 2023  01:19:38 am
4  Module:           MESI_FSM
5  Technology library: uk65lscllmvbbr_120c25_tc
6  Operating conditions: uk65lscllmvbbr_120c25_tc (balanced_tree)
7  Wireload mode:    top
8  Area mode:        timing library
9 =====
10
11          Pin                Type          Fanout Load Slew Delay Arrival
12                                (fF) (ps)  (ps)  (ps)
13 -----
57 - - - - -
58 (clock clk)                capture                2000 R
59                          uncertainty                -10  1990 R
60 -----
61 Cost Group   : 'clk' (path_group 'clk')
62 Timing slack :      409ps
63 Start-point  : tag_directory_reg[1][0]/CK
64 End-point    : Cache_output_reg[5]/D
65
```

THANK YOU

Directory Schemes

- Snoopy schemes do not scale because they rely on broadcast
- Directory-based schemes allow scaling.
 - avoid broadcasts by keeping track of all PEs caching a memory block, and then using point-to-point messages to maintain coherence
 - they allow the flexibility to use any scalable point-to-point network

Basic Scheme (Censier & Feautrier)



- Assume "k" processors.
- With each cache-block in memory: k presence-bits, and 1 dirty-bit
- With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit

– Read from main memory by PE-i:

- If dirty-bit is OFF then { read from main memory; turn p[i] ON; }
- if dirty-bit is ON then { recall line from dirty PE (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to PE-i; }

– Write to main memory:

- If dirty-bit OFF then { send invalidations to all PEs caching that block; turn dirty-bit ON; turn P[i] ON; ... }

Key Issues

- Scaling of memory and directory bandwidth
 - Can not have main memory or directory memory centralized
 - Need a distributed memory and directory structure
- Directory memory requirements do not scale well
 - Number of presence bits grows with number of PEs
 - Many ways to get around this problem
 - limited pointer schemes of many flavors
- Industry standard
 - SCI: Scalable Coherent Interface

Overview

- We have talked about optimizing performance on single cores
 - Locality
 - Vectorization
- Now let us look at optimizing programs for a shared-memory multiprocessor.
- Two architectures:
 - Bus-based shared-memory machines (small-scale)
 - Directory-based shared-memory machines (large-scale)

Update or Invalidate?

- Update looks the simplest, most obvious and fastest, but:-
 - Multiple writes to same word (no intervening read) need only one invalidate message but would require an update for each
 - Writes to same block in (usual) multi-word cache block require only one invalidate but would require multiple updates.

Update or Invalidate?

- Due to both spatial and temporal locality, previous cases occur often.
- Bus bandwidth is a precious commodity in shared memory multi-processors
- Experience has shown that invalidate protocols use significantly less bandwidth.
- Will consider implementation details only of invalidate.

Implementation Issues

- In both schemes, knowing if a cached value is not shared (copy in another cache) can avoid sending any messages.
- Invalidate description assumed that a cache value update was written through to memory. If we used a ‘copy back’ scheme other processors could re-fetch old value on a cache miss.
- We need a protocol to handle all this.

Snooping Protocols

- Write Update
 - CPU wanting to write grabs bus cycle and broadcasts new data as it updates its own copy
 - All snooping caches update their copy
- Note that in both schemes, problem of simultaneous writes is taken care of by bus arbitration - only one CPU can use the bus at any one time.

MESI Local Write Miss (4)

Another copy in state M (continued)

- Original local processor re-issues RWITM request
- Is now simple no-copy case
 - Value read from memory to local cache
 - Local copy value updated
 - Local copy state set to M

Name	Value
> [3][1:0]	Invalid
Cache_Status[0:3][1:0]	Invalid,Invalid,Inv
> [0][1:0]	Invalid
> [1][1:0]	Invalid
> [2][1:0]	Invalid
> [3][1:0]	Invalid
reseth	0
core_op1	PrRd
core_op2	PrRd
PrHit	0
> pr_addr1[3:0]	4
> pr_addr2[3:0]	4
> Cache_output1[7:0]	255
> Cache_output2[7:0]	10
> Cache_input1[7:0]	30
> Cache_input2[7:0]	10
> bus12[16:0]	17663
> bus21[16:0]	46090
> Cache_Block_Size[31:0]	8

