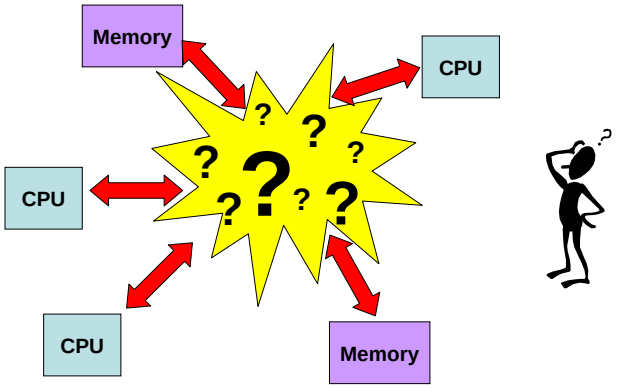
Shared Memory Bus for Multiprocessor Systems

Mat Laibowitz and Albert Chiou

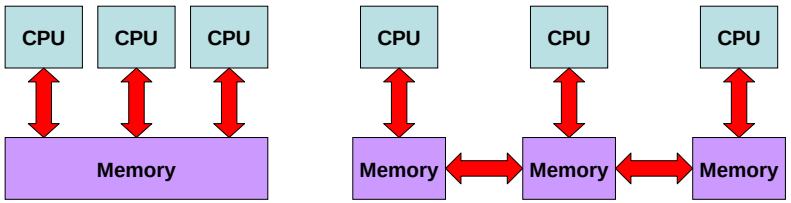
Group 6

Shared Memory Architecture



- We want multiple processors to share memory
  - Question: How do we connect them together?

Shared Memory Architecture

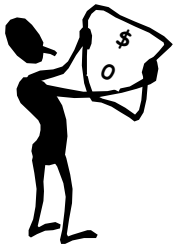


Single, large memory **Multiple smaller memories**

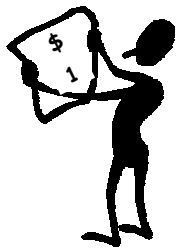
Issues

- Scalability
- Access Time
- Cost
- Application: WLAN vs Single chip multiprocessor

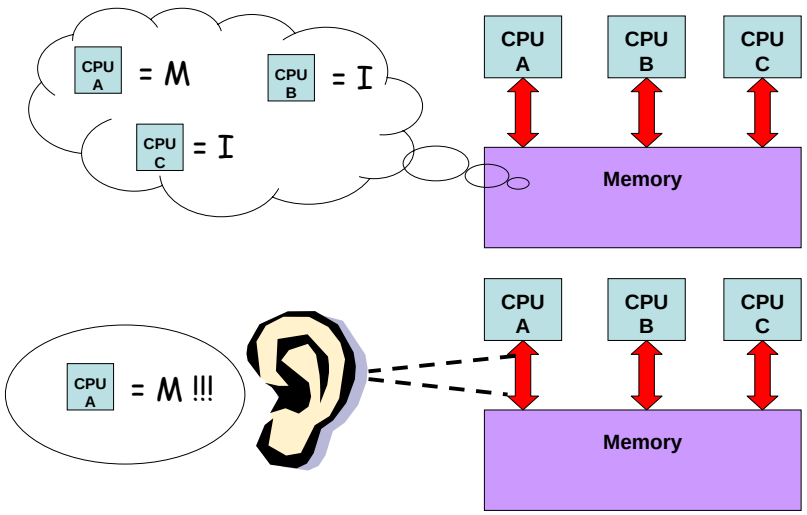
Cache Coherency Problem



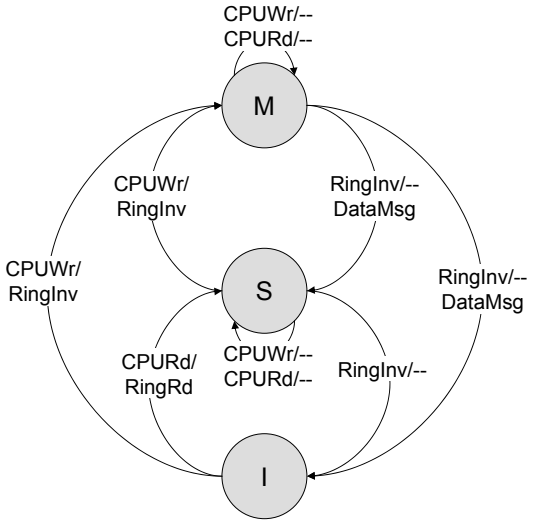
- Each cache needs to correctly handle memory accesses across multiple processors
- A value written by one processor is eventually visible by the other processors
- When multiple writes happen to the same location by multiple processors, all the processors see the writes in the same order.



Snooping vs Directory



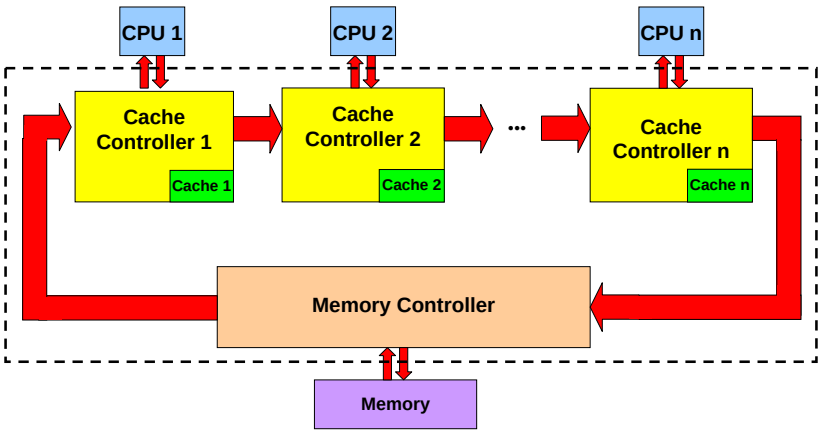
MSI State Machine



MSI Transition Chart

Cache State	Pending State	Incoming Ring Transaction	Incoming Processor Transaction	Actions
& Miss	0	-	Read	Pending->1; SEND Read
& Miss	0	-	Write	Pending->1; SEND Write
& Miss	0	Read		PASS
& Miss	0	Write		PASS
& Miss	0	WriteBack		PASS
& Miss	1	Read		DATA/S->Cache; SEND WriteBack(DATA)
& Miss	1	Write (I/S)		DATA/M->Cache, Modify Cache; SEND WriteBack(DATA)
& Miss		Write (M)		DATA/M->Cache, Modify Cache; SEND WriteBack(DATA), SEND WriteBack(data), Pending->2
S	0	-	Read(Hit)	
S	0	-	Write	Pending->1; SEND Write
S	0	Read(Hit)	-	Add DATA; PASS
S	0	Read(Miss)	-	PASS
S	0	Write(Hit)	-	Add DATA; Cache->I & PASS
S	0	Write(Miss)	-	PASS
S	0	WriteBack	-	PASS
S	1	Write	-	Modify Cache; Cache->M & Pass Token
S	1	WriteBack	-	Pending->0, Pass Token
M	0	-	Read(Hit)	
M	0	-	Write(Hit)	
M	0	Read(Hit)		Add DATA; Cache->S & PASS
M	0	Read(Miss)		PASS
M	0	Write(Hit)		Add DATA; Cache->I & PASS
M	0	Write(Miss)		PASS
M	0	WriteBack		PASS
M	1	WriteBack	-	Pending->0 & Pass Token
M	2	WriteBack	-	Pending->1

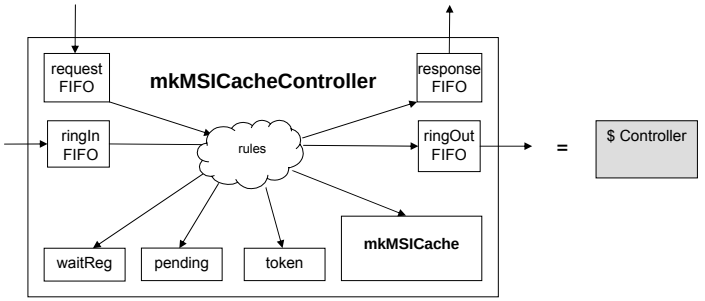
Ring Topology



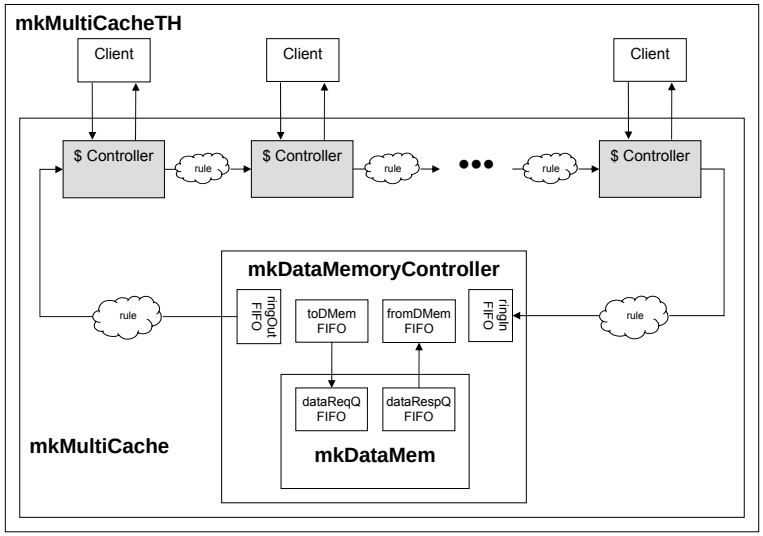
Ring Implementation

- A ring topology was chosen for speed and its electrical characteristics
  - Only point-to-point
  - Like a bus
  - Scaleable
- Uses a token to ensure sequential consistency

Test Rig



Test Rig



Test Rig (cont)

• An additional module was implemented that takes a single stream of memory requests and deals them out to the individual cpu data request ports.

•This module can either send one request at a time, wait for a response, and then go on to the next cpu or it can deal them out as fast as the memory ports are ready.

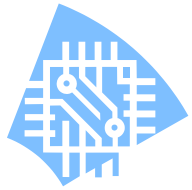
•This demux allows individual processor verification prior to multi-processor verification.

•It can then be fed set test routines to exercise all the transitions or be hooked up to the random request generator

```
=> Cache 2: toknMsg op->Tk8
=> Cache 5: toknMsg op->Tk2
=> Cache 3: ringMsg op->WrBk addr->0000022c data->aaaaaaaa valid->1 cache->1
=> Cache 3: getState I
=> Cache 1: newCpuReq St { addr=00000230, data=ba4f0452 }
=> Cache 1: getState I
=> Cycle =  56
=> Cache 2: toknMsg op->Tk7
=> Cache 6: ringMsg op->Rd addr->00000250 data->aaaaaaaa valid->1 cache->6
=> DataMem: ringMsg op->WrBk addr->00000374 data->aaaaaaaa valid->1 cache->5
=> Cache 6: getState I
=> Cache 8: ringReturn op->Wr addr->000003a8 data->aaaaaaaa valid->1 cache->7
=> Cache 8: getState I
=> Cache 8: writeLine state->M addr->000003a8 data->4ac6efe7
=> Cache 3: ringMsg op->WrBk addr->00000360 data->aaaaaaaa valid->1 cache->4
=> Cache 3: getState I
=> Cycle =  57
=> Cache 6: toknMsg op->Tk2
=> Cache 3: toknMsg op->Tk8
=> Cache 4: ringMsg op->WrBk addr->0000022c data->aaaaaaaa valid->1 cache->1
=> Cache 4: getState I
=> Cycle =  58
=> dMemReq:  St { addr=00000374, data=aaaaaaaa }
=> Cache 3: toknMsg op->Tk7
=> Cache 7: ringReturn op->Rd addr->00000250 data->aaaaaaaa valid->1 cache->6
=> Cache 7: writeLine state->S addr->00000250 data->aaaaaaaa
=> Cache 7: getState I
=> Cache 1: ringMsg op->WrBk addr->00000374 data->aaaaaaaa valid->1 cache->5
=> Cache 1: getState I
=> Cache 4: ringMsg op->WrBk addr->00000360 data->aaaaaaaa valid->1 cache->4
=> Cache 4: getState I
=> Cache 9: ringMsg op->WrBk addr->000003a8 data->aaaaaaaa valid->1 cache->7
=> Cache 9: getState I
=> Cycle =  59
=> Cache 5: ringMsg op->WrBk addr->0000022c data->aaaaaaaa valid->1 cache->1
=> Cache 5: getState I
=> Cache 7: toknMsg op->Tk2
=> Cache 3: execCpuReq Ld { addr=000002b8, tag=00 }
=> Cache 3: getState I
=> Cache 4: toknMsg op->Tk8
=> Cycle =  60
=> DataMem: ringMsg op->WrBk addr->000003a8 data->aaaaaaaa valid->1 cache->7
=> Cache 2: ringMsg op->WrBk addr->00000374 data->aaaaaaaa valid->1 cache->5
=> Cache 2: getState I
=> Cache 8: ringMsg op->WrBk addr->00000250 data->aaaaaaaa valid->1 cache->6
=> Cache 8: getState I
=> Cache 5: ringReturn op->WrBk addr->00000360 data->aaaaaaaa valid->1 cache->4
=> Cache 5: getState S
=> Cycle =  61
=> Cache 5: toknMsg op->Tk8
```

Trace Example

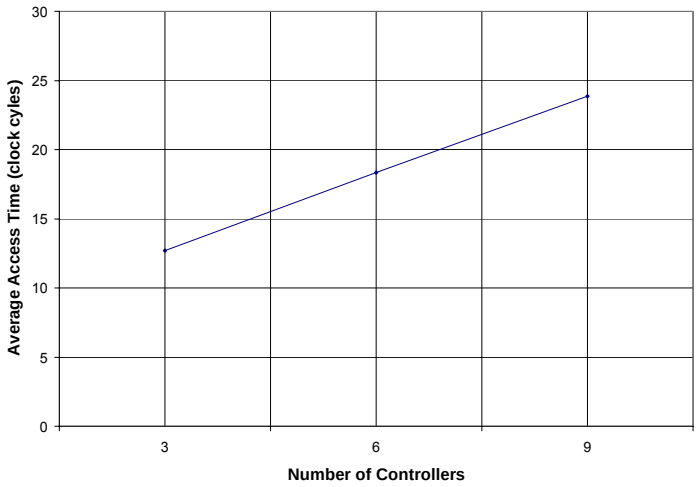
Design Exploration



- Scale up number of cache controllers
- Add additional tokens to the ring allowing basic pipelining of memory requests
  - Tokens service disjoint memory addresses (ex. odd or even)
- Compare average memory access time versus number of tokens and number of active CPUs

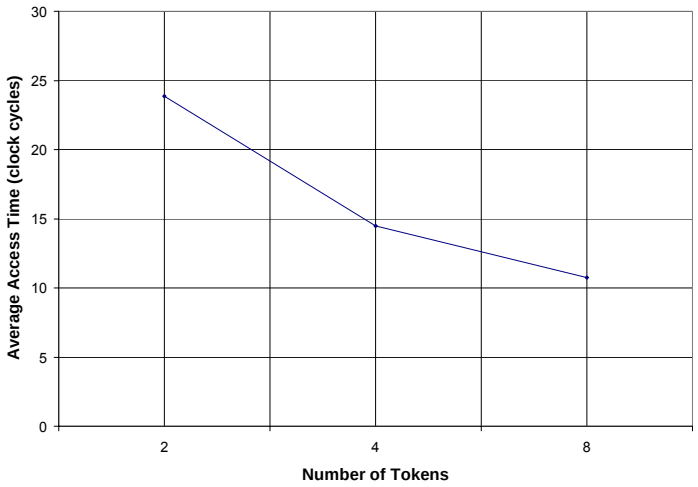
Test Results

Number of Controllers vs. Avg. Access Time (2 Tokens)

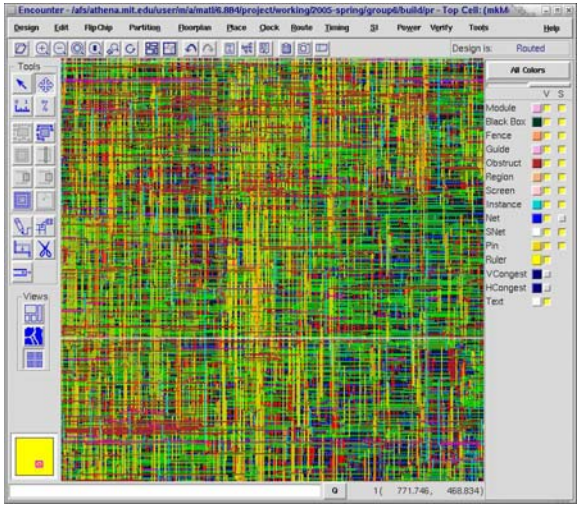


Test Results

Number of Tokens vs. Avg. Access Time (9 Controllers)



Placed and Routed



Stats (9 cache, 8 tokens)

- Clock speed: 3.71ns (~270 Mhz)
- Area: 1,296,726 μm^2 with memory
- Average memory access time: ~39ns