

USN



Internal Assessment Test II – Sept. 2017

Sub:	Advanced Computer Architecture(ACA)				Sub Code:	10CS74	Branch:	CSE		
Date:	8/11/2017	Duration:	90 min's	Max Marks:	50	Sem / Sec:	VII/A,B,C			
<u>Answer any FIVE FULL Questions</u>										
								MARKS	CO	RBT
1	With a neat diagram describe the structure of Tomasulo based MIPS FP unit and explain the fields of reservation station.					[10]		CO3	L2	
2 (a)	What is the drawback of 1-bit dynamic branch prediction method? Clearly state, how to overcome in 2-bit prediction. Give the state transition diagram of 2-bit predictor.					[08]		CO3	L1	
(b)	To achieve a speedup of 80 with 100 processors what fraction of original computation can be sequential?					[02]		CO3	L3	
3	<p>Explain the principles of loop unrolling. Demonstrate the normal loop execution and loop unrolling concept for the following C-code segment by translating the given code segment to MIPS assembly language code.</p> <p>C-code: for (i=1000; i>0; i--) X[i] = X[i] +s where s=scalar value</p> <p>a) Calculate the number of clock cycles required per element for both unscheduled and scheduled loops in normal case considering stalls/idle clock cycles.</p> <p>b) Repeat the above step for loop unrolled execution with and without scheduling.</p> <p>c) Calculate the average value of clock cycle per element for (a) and (b).</p>					[10]		CO3. CO6	L3	

USN



Internal Assessment Test II – Sept. 2017

Sub:	Advanced Computer Architecture(ACA)				Sub Code:	10CS74	Branch:	CSE		
Date:	8/11/2017	Duration:	90 min's	Max Marks:	50	Sem / Sec:	VII/A,B,C			
<u>Answer any FIVE FULL Questions</u>										
								MARKS	CO	RBT
1	With a neat diagram describe the structure of Tomasulo based MIPS FP unit and explain the fields of reservation station.					[10]		CO3	L2	
2 (a)	What is the drawback of 1-bit dynamic branch prediction method? Clearly state, how to overcome in 2-bit prediction. Give the state transition diagram of 2-bit predictor.					[08]		CO3	L1	
(b)	To achieve a speedup of 80 with 100 processors what fraction of original computation can be sequential?					[02]		CO3	L3	
3	<p>Explain the principles of loop unrolling. Demonstrate the normal loop execution and loop unrolling concept for the following C-code segment by translating the given code segment to MIPS assembly language code.</p> <p>C-code: for (i=1000; i>0; i--) X[i] = X[i] +s where s=scalar value</p> <p>a) Calculate the number of clock cycles required per element for both unscheduled and scheduled loops in normal case considering stalls/idle clock cycles.</p> <p>b) Repeat the above step for loop unrolled execution with and without scheduling.</p> <p>c) Calculate the average value of clock cycle per element for (a) and (b).</p>					[10]		CO3. CO6	L3	

4	Define Instruction level Parallelism. Explain data dependence and name dependence. Also explain three types of data Hazard with example.	[10]	CO3	L1
5 (a)	Describe snooping with respect to cache-coherence protocol.	[06]	CO4	L2
(b)	Assume we have a computer where the clock per instruction (CPI) is 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 25 clock cycles and the miss rate is 2%, how much faster would the computer be if all instructions were cache hits?	[04]	CO1, CO3	L3
6	Explain directory based cache coherence for a distributed memory multiprocessor system along with the state transition diagram.	[10]	CO4	L2
7	Explain any three basic cache optimization techniques.	[10]	CO5	L2
8	What are the four common questions for the first level of memory hierarchy?	[10]	CO4, CO5	L1

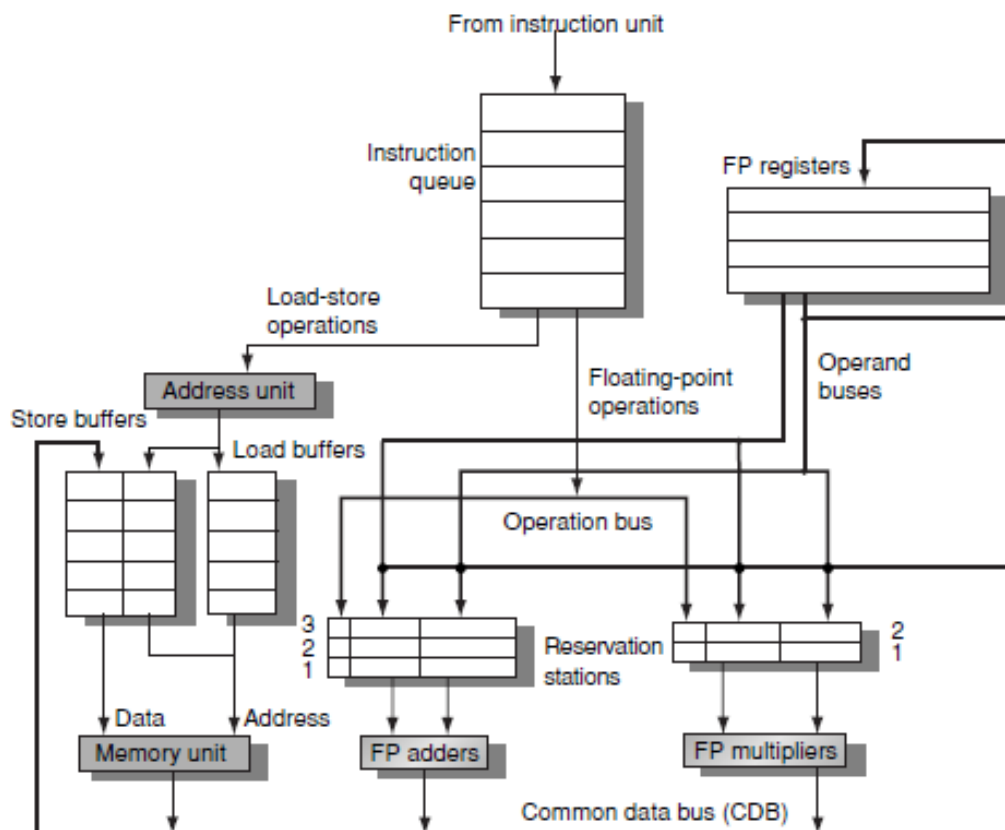
4	Define Instruction level Parallelism. Explain data dependence and name dependence. Also explain three types of data Hazard with example.	[10]	CO3	L1
5 (a)	Explain snooping with respect to cache-coherence protocol.	[06]	CO4	L2
(b)	Assume we have a computer where the clock per instruction (CPI) is 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 25 clock cycles and the miss rate is 2%, how much faster would the computer be if all instructions were cache hits?	[04]	CO1, CO3	L3
6	Explain directory based cache coherence for a distributed memory multiprocessor system along with the state transition diagram.	[10]	CO4	L2
7	Explain any three basic cache optimization techniques.	[10]	CO5	L2
8	What are the four common questions for the first level of memory hierarchy?	[10]	CO4, CO5	L1

Scheme and Solution : Internal Assessment Test II – Nov. 2017

Sub:	Advanced Computer Architecture(ACA)	Sub Code:	10CS74	Branch:	CSE
Date:	8/11/2017	Duration:	90 min's	Max Marks:	50
			Sem / Sec:	VII/A,B,C	

Solution

1. The basic structure of Tomasulo based Floating point unit is shown below in diagram.



- Instructions are sent from the instruction unit into the instruction queue from which they are issued in FIFO order.
- Each reservation station holds an instruction that has been issued and is awaiting execution at a functional unit. The reservation station hold the operand values if they are

already computed else it contains the names of the reservation stations that will provide the operand values.

- Load buffers have three functions: hold the components of the effective address until it is computed, track outstanding loads that are waiting on the memory, and hold the results of completed loads that are waiting for the CDB.
- Similarly, store buffers have three functions: hold the components of the effective address until it is computed, hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and hold the address and value to store until the memory unit is available.
- All results from either the FP units or the load unit are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers.
- The FP adders implement addition and subtraction, and the FP multipliers do multiplication and division.

Seven fields of reservation station are as follows

- Op: The operation to perform on source operands S1 and S2.
- Qj, Qk—It is set when operand values are unavailable. It contains the reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary.
- Vj, Vk— It is set when operand values are available. It contains the value of source operands. For loads, the Vk field is used to hold the offset field.
- A—Used to hold information for the memory address calculation for a load or store. Initially, the immediate field of the instruction is stored here; after the address calculation, the effective address is stored here.
- Busy—indicates that this reservation station and its accompanying functional unit are occupied.

2. A) Dynamic Branch Prediction

- The simplest dynamic branch-prediction scheme is a branch-prediction buffer or branch history table.

- A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not.
- The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back.
- This simple 1-bit prediction scheme has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken, since the misprediction causes the prediction bit to be flipped.

* 1-bit prediction: consider an example where the branch is almost always taken but when once it is untaken. In between there are two mispredictions, since misprediction causes the prediction bits to be flipped.

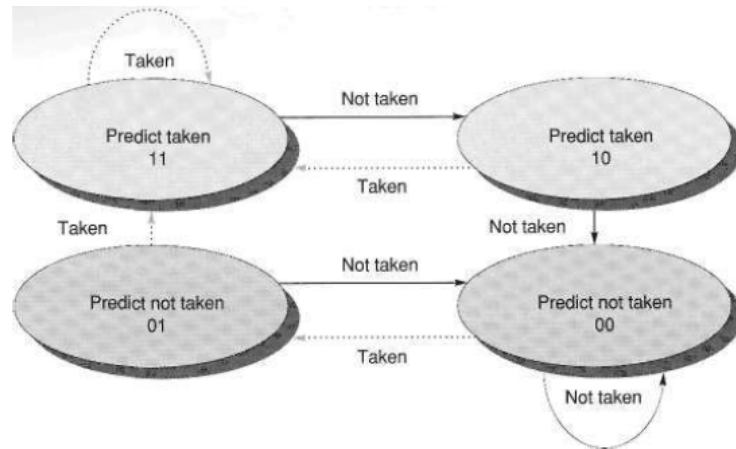
1-e

	T	T	T	T	NT	T	T	T	T
Prediction.	T	T	T	T	T	NT	T	T	T
Misprediction.	No	No	No	No	Yes.	Yes	No	No	No

} two mispredictions.

Number of mispredictions are more in 1-bit prediction scheme as compared to 2-bit prediction scheme.

- To remedy this weakness, 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must miss twice before it is changed.



The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually a specialization of a more general scheme that has an n-bit saturating counter for each entry in the prediction buffer. With an n-bit counter, the counter can take on values between 0 and $2^n - 1$: When the counter is greater than or equal to one-half of its maximum value (2^{n-1}), the branch is predicted as taken; otherwise, it is predicted untaken. Studies of n-bit predictors have shown that the 2-bit predictors do almost as well, and thus most systems rely on 2-bit branch predictors rather than the more general n-bit predictors.

2. B Problem

Amdahl's Law is

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

$$80 = \frac{1}{\frac{\text{Fraction}_{\text{parallel}}}{100} + (1 - \text{Fraction}_{\text{parallel}})}$$

Simplifying this equation yields

$$0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times (1 - \text{Fraction}_{\text{parallel}}) = 1$$

$$80 - 79.2 \times \text{Fraction}_{\text{parallel}} = 1$$

$$\text{Fraction}_{\text{parallel}} = \frac{80 - 1}{79.2}$$

$$\text{Fraction}_{\text{parallel}} = 0.9975$$

Thus, to achieve a speedup of 80 with 100 processors, only 0.25% of original computation can be sequential.

3. A Principles of loop unrolling

- Loop unrolling is very useful when loop iterations are independent.
- Also load and store of different iterations can be easily interchanged if loop iterations are independent.
- But in loop unrolling if same set of registers are used it complicates the scheduling process. Hence different set of registers are used for each iteration.

- **Consider the for loop**

```
for (i=1000; i > 0; i--)
```

```
  x[i] = x[i] + s;
```

- The MIPS code for the above loop is as follows

```
L.D      F0, 0(R1); F0=array element
```

```
ADD.D    F4, F0, F2; add scalar in F2
```

```
S.D      F4, 0(R1); store result
```

```
DADDUI   R1, R1, #-8; decrements pointer by 8 bytes
```

```
BNE      R1, R2, Loop; branch R1! = R2
```

- **In case of Pipeline Scheduling the dependent instruction is separated from the source instruction by some distance equal to pipeline latency of source instruction.**

Instruction producing result: source instruction	Instruction consuming result: dependent instruction	Latency in Clock cycles
FP ALU OP	FP ALU OP	3
FP ALU OP	Store double	2
Load double	FP ALU OP	1
Load double	FP ALU OP	0

- **Without Pipeline Scheduling** the loop will execute as follows. Branches have a delay or latency of 1 clock cycle and integer operations have zero latency. Total clock cycles per iteration = 9. Hence average number of clock cycles=9 per iteration.

Loop	L.D	F0,0(R1)	cc1
	stall		cc2
	ADD.D	F4,F0,F2	cc3
	stall		cc4
	stall		cc5
	S.D	F4,0(R1)	cc6
	DADDUI	R1,R1,#-8	cc7
	stall		cc8
	BNE	R1,R2, Loop	cc9

- **With Scheduling** the loop will execute as follows taking only 7 clock cycles. Hence average number of clock cycles=7 per iteration.

Loop	L.D	F0,0(R1)	cc1
	DADDUI	R1,R1,#-8	cc2
	ADD.D	F4,F0,F2	cc3
	stall		cc4
	stall		cc5
	S.D	F4,0(R1)	cc6
	BNE	R1,R2, Loop	cc7

- Increases the amount of ILP and improves the scheduling.
- In case of loop unrolling the loop body is replicated multiple times and loop termination code is adjusted at the end.
- If loop is unrolled for 4 iterations, then the unrolled loop would contain the loop body of 4 iterations and loop termination code at the end.
 - loop body(1st iteration)
 - loop body(2nd iteration)
 - loop body(3rd iteration)
 - loop body(4th iteration)
 - loop termination code.
- **With Loop Unrolling** : Hence when the loop is unrolled for four iterations there are four copies of loop body and loop termination code is adjusted at the end as shown below.

Loop	L.D	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)
	L.D	F6,-8(R1)
	ADD.D	F8,F6,F2
	S.D	F8,-8(R1)
	L.D	F10,-16(R1)
	ADD.D	F12,F10,F2
	S.D	F12, -16(R1)
	L.D	F14,-24(R1)
	ADD.D	F16,F14,F2
	S.D	F16,-24(R1)
	DADDUI	R1,R1,#-32
	BNE	R1,R2,LOOP

- Each loop body requires 6 clock cycles and loop termination code requires 3 clock cycles.

Total clock cycles for four iterations= $(6*4) + 3 = 27$

Hence average number of clock cycles per iteration= $27/4 = 6.8$ clock cycles.

- Scheduling the unrolled loop:** scheduling unrolled loop significantly reduces the number of clock cycles and increases ILP.

```

Loop:  L.D      F0,0(R1)
       L.D      F6,-8(R1)
       L.D      F10,-16(R1)
       L.D      F14,-24(R1)
       ADD.D    F4,F0,F2
       ADD.D    F8,F6,F2
       ADD.D    F12,F10,F2
       ADD.D    F16,F14,F2
       S.D      F4,0(R1)
       S.D      F8,-8(R1)
       DADDUI   R1,R1,#-32
       S.D      F12,16(R1)
       S.D      F16,8(R1)
       BNE     R1,R2,Loop

```

Thus the total number of clock cycles required for four iterations are 14. Hence average number of clock cycles per iteration= $14/4 = 3.5$ clock cycles.

3. B Define Instruction level Parallelism. Explain data dependence and name dependence. Also explain three types of data Hazard with example.

Instruction level Parallelism: The potential overlap among instructions inside pipeline is called *instruction-level parallelism* (ILP), since the instructions can be evaluated in parallel.

Data Dependence: If two instructions are parallel, they can execute simultaneously in a pipeline without causing any stalls. If two instructions are dependent they are not parallel and must be executed in order. There are three different types of dependences: data dependences (also called true data dependences), name dependences, and control dependences.

Data Dependences: An instruction j is data dependent on instruction i if either of the following holds: • Instruction i produces a result that may be used by instruction j , or • Instruction j is data dependent on instruction k , and instruction k is data dependent on instruction i . The second condition simply states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions. This dependence chain can be as long as the entire program.

Name Dependence: Name dependence occurs when two instructions use the same register or memory location, called a *name*, but there is no flow of data between the instructions associated with that name. There are two types of name dependences between an instruction i that *precede* instruction j in program order.

- An *antidependence* between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved to ensure that i read the correct value.
- An *output dependence* occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j .

Data Hazard and various hazards in ILP.

- a) Data Hazards: A hazard is created whenever there is dependence between instructions, and they are close enough that the overlap caused by pipelining, or other reordering of instructions, would change the order of access to the operand involved in the dependence.

- b) Data hazards may be classified as one of three types, depending on the order of read and write accesses in the instructions. Consider two instructions i and j , with i occurring before j in program order.

RAW (read after write) — j tries to read a source before i writes it, so j incorrectly gets the old value. This hazard is the most common type and corresponds to a true data dependence. Program order must be preserved to ensure that j receives the value from i . In the simple common five-stage static pipeline a load instruction followed by an integer ALU instruction that directly uses the load result will lead to a RAW hazard.

Example

Instr_j tries to read operand before Instr_i writes it

```
I: add r1, r2, r3
J: sub r4, r1, r3
```

WAW (write after write) — j tries to write an operand before it is written by i . The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination. This hazard corresponds to output dependence.

Example

Instr_j tries to write operand *before* Instr_i writes it

```


  ↪ I: sub r1, r4, r3
  ↪ J: add r1, r2, r3
    K: mul r6, r1, r7
```

WAR (write after read) — j tries to write a destination before it is read by i , so i incorrectly gets the new value. This hazard arises from antidependence. WAR hazards cannot occur in most static issue pipelines even deeper pipelines or floating point pipelines because all reads are early (in ID) and all writes are late (in WB). A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source late in the pipeline or when instructions are reordered.

Example

Instr_j tries to write operand *before* Instr_i reads i

- Gets wrong operand
- Called as “anti-dependence”

 I: sub r4, r1, r3
J: add r1, r2, r3
K: mul r6, r1, r7

5. A Explain snooping with respect to cache coherence protocol.

- One method is to ensure that a processor has exclusive access to data item before it writes that item. This style of protocol is called a *write invalidate protocol* because it invalidates other copies on a write. It is by far the most common protocol, both for snooping and for directory schemes. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: All other cached copies of the item are invalidated.
- Figure shows an example of an invalidation protocol for a snooping bus with write-back caches in action. To see how this protocol ensures coherence, consider a write followed by a read by another processor: Since the write requires exclusive access, any copy held by the reading processor must be invalidated (hence the protocol name). Thus, when the read occurs, it misses in the cache and is forced to fetch a new copy of the data.

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches. We assume that neither cache initially holds X and that the value of X in memory is 0. The CPU and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When the second miss by B occurs, CPU A responds with the value canceling the response from memory. In addition, both the contents of B's cache and the memory contents of X are updated.

- The alternative to an invalidate protocol is to update all the cached copies of a data item when that item is written. This type of protocol is called as write update or writes *broadcast* protocol. Because a write update protocol must broadcast all writes to shared cache lines, it consumes considerably more bandwidth.
- To perform an invalidate, the processor simply acquires bus access and broadcasts the address to be invalidated on the bus. All processors continuously snoop on the bus, watching the addresses. The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache are invalidated.
- The first processor to obtain bus access will cause any other copies of the block it is writing to be invalidated. If the processors were attempting to write the same block, the serialization enforced by the bus also serializes their writes.

5. B Assume we have a computer where the clock per instruction (CPI) is 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 25 clock cycles and the miss rate is 2%, how much faster would the computer be if all instructions were cache hits?

First compute the performance for the computer that always hits:

$$\begin{aligned}\text{CPU execution time} &= (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle} \\ &= (\text{IC} \times \text{CPI} + 0) \times \text{Clock cycle} \\ &= \text{IC} \times 1.0 \times \text{Clock cycle}\end{aligned}$$

Now for the computer with the real cache, first we compute memory stall cycles:

$$\begin{aligned}\text{Memory stall cycles} &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty} \\ &= \text{IC} \times (1 + 0.5) \times 0.02 \times 25 \\ &= \text{IC} \times 0.75\end{aligned}$$

where the middle term (1 + 0.5) represents one instruction access and 0.5 data accesses per instruction. The total performance is thus

$$\begin{aligned}\text{CPU execution time}_{\text{cache}} &= (\text{IC} \times 1.0 + \text{IC} \times 0.75) \times \text{Clock cycle} \\ &= 1.75 \times \text{IC} \times \text{Clock cycle}\end{aligned}$$

The performance ratio is the inverse of the execution times:

$$\begin{aligned}\frac{\text{CPU execution time}_{\text{cache}}}{\text{CPU execution time}} &= \frac{1.75 \times \text{IC} \times \text{Clock cycle}}{1.0 \times \text{IC} \times \text{Clock cycle}} \\ &= 1.75\end{aligned}$$

The computer with no cache misses is 1.75 times faster.

6. Explain directory based cache coherence for a distributed memory multiprocessor system along with the state transition diagram.

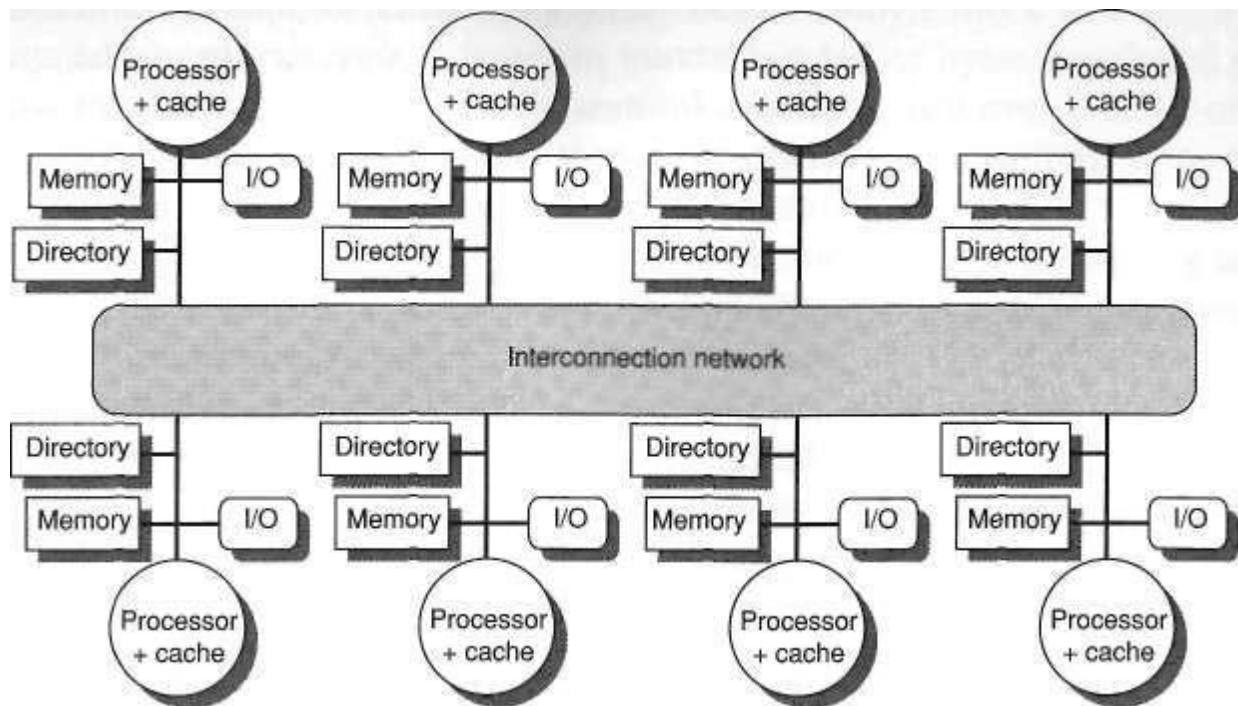
In a simple protocol, the states could be the following:

Shared—One or more processors have the block cached, and the value in memory is up to date (as well as in all the caches).

Un-cached—No processor has a copy of the cache block.

Modified—Exactly one processor has a copy of the cache block, and it has written the block, so the memory copy is out of date. The processor is called the owner of the block.

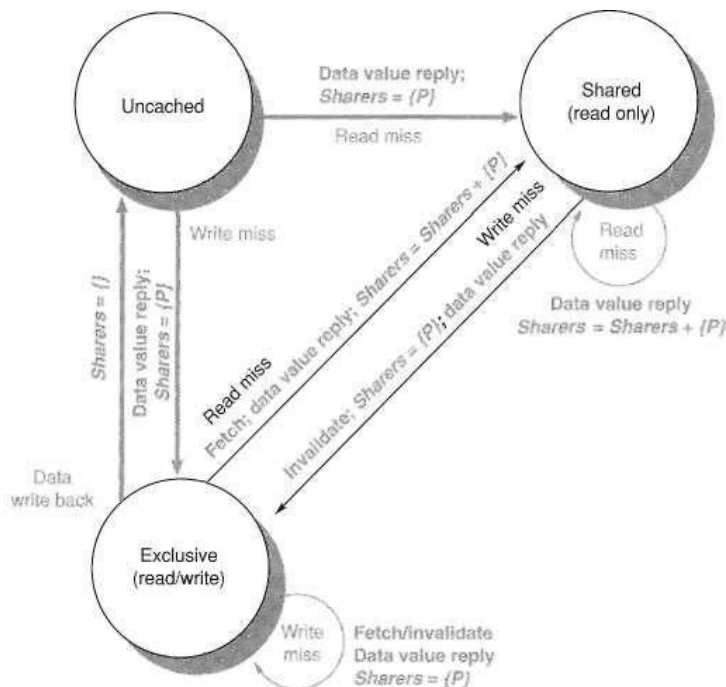
A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor. Each directory is responsible for tracking the caches that share the memory addresses of the portion of memory in the node. The directory may communicate with the processor and memory over a common bus, as shown, or it may have a separate port to memory, or it may be part of a central node controller through which all intra-node and inter-node communications pass.



Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories for the purpose of handling misses and maintaining coherence

The *local node* is the node where a request originates. The *home node* is the node where the memory location and the directory entry of an address reside. The local node may also be the home node. The directory must be accessed when the home node is the local node, since copies may exist in yet a third node, called a *remote node*. A remote node is the node that has a copy of a cache block, whether exclusive (in which case it is the only copy) or shared. A remote node may be the same as either the local node or the home node. In such cases, the basic protocol does not change, but inter-processor messages may be replaced with intra-processor messages.

Message type	Source	Destination	Message contents	Function of this message
Read miss	local cache	home directory	P,A	Processor P has a read miss at address A; request data and make P a read sharer.
Write miss	local cache	home directory	P,A	Processor P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	local cache	home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	home directory	remote cache	A	Invalidate a shared copy of data at address A.
Fetch	home directory	remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	home directory	remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	home directory	local cache	D	Return a data value from the home memory.
Data write back	remote cache	home directory	A,D	Write back a data value for address A.



When a block is in the un-cached state, the copy in memory is the current value, so the only possible requests for that block are

- *Read miss*—The requesting processor is sent the requested data from memory, and the requestor is made the only sharing node. The state of the block is made shared.
- *Write miss*—The requesting processor is sent the value and becomes the sharing node. The block is made exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

When the block is in the shared state, the memory value is up to date, so the same two requests can occur:

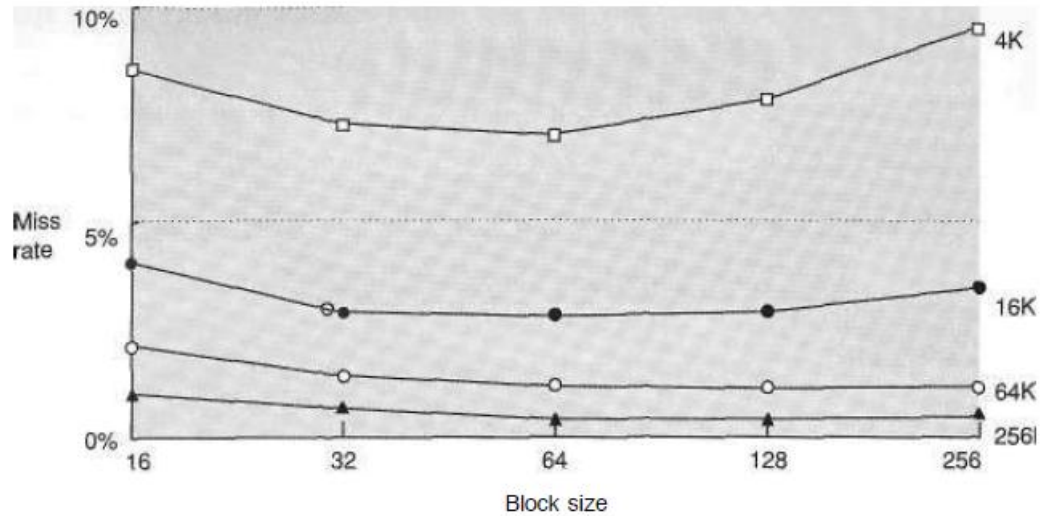
- *Read miss*—The requesting processor is sent the requested data from memory, and the requesting processor is added to the sharing set.
- *Write miss*—The requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, and the Sharers set is to contain the identity of the requesting processor. The state of the block is made exclusive.

When the block is in the exclusive state, the current value of the block is held in the cache of the processor identified by the set Sharers (the owner), so there are three possible directory requests:

- *Read miss*—The owner processor is sent a data fetch message, which causes the state of the block in the owner's cache to transition to shared and causes the owner to send the data to the directory, where it is written to memory and sent back to the requesting processor. The identity of the requesting processor is added to the set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy).
- *Data write back*—The owner processor is replacing the block and therefore must write it back. This write back makes the memory copy up to date (the home directory essentially becomes the owner), the block is now uncached, and the Sharers set is empty.
- *Write miss*—The block has a new owner. A message is sent to the old owner, causing the cache to invalidate the block and send the value to the directory, from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to the identity of the new owner, and the state of the block remains exclusive.

First Optimization: Larger Block Size to Reduce Miss Rate

- The simplest way to reduce miss rate is to increase the block size.
- Larger block sizes will reduce also compulsory misses. This reduction occurs because the principle of locality has two components: temporal locality and spatial locality. Larger blocks take advantage of spatial locality.
- Larger blocks increase the miss penalty and larger blocks may increase conflict misses and even capacity misses if the cache is small.



Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up if the block size is too large relative to the cache size.

Second Optimization: Larger Caches to Reduce Miss Rate

- It reduces the capacity misses.
- The obvious drawback is potentially longer hit time and higher cost and power. This technique has been especially popular in off-chip caches.

Third Optimization: Higher Associativity to Reduce Miss Rate

- It reduces the conflict misses.
- The second observation, called the *2:1 cache rule of thumb*, is that a direct mapped cache of size JV has about the same miss rate as a two-way set-associative cache of size $N/2$.
- But it increases the hit time.

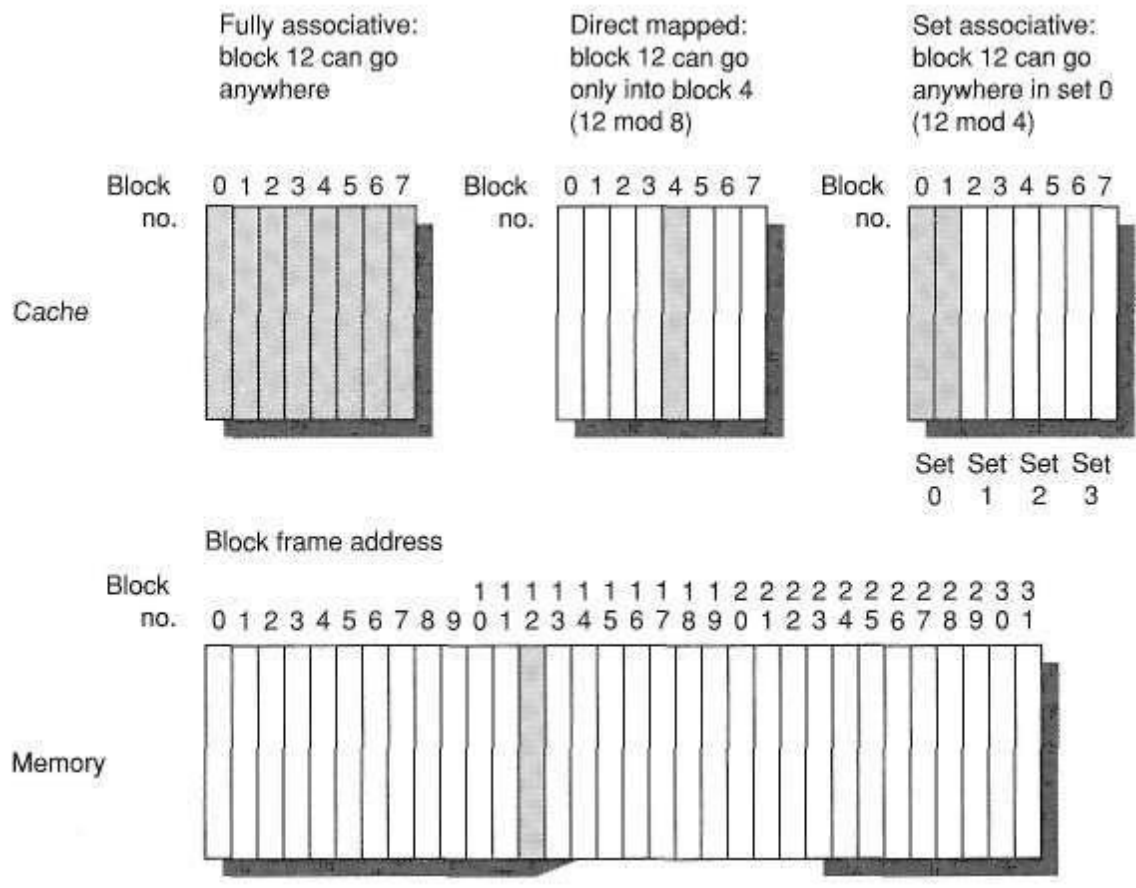
Four Memory Hierarchy Questions

Q1: Where Can a Block Be Placed in a Cache?

If each block has only one place it can appear in the cache, the cache is said to be *direct mapped*. The mapping is usually $(\text{Block address}) \text{ MOD } (\text{Number of blocks in cache})$. If a block can be placed anywhere in the cache, the cache is said to be *fully associative*. If a block can be placed in a restricted set of places in the cache, the cache is *set associative*. A *set* is a group of blocks in the

cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set. The sets are usually chosen by *bit selection*; that is,

$$(Block\ address) \text{ MOD } (Number\ of\ sets\ in\ cache)$$

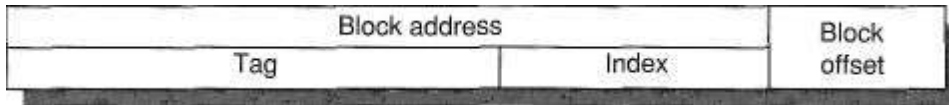


If there are n blocks in a set, the cache placement is called n -way *setassociative*.

Q2: How Is a Block Found If It Is in the Cache?

Caches have an address tag on each block frame that gives the block address. The tag of every cache block that might contain the desired information is checked to see if it matches the block address from the processor. As a rule, all possible tags are searched in parallel because speed is critical.

Figure shows how an address is divided. The first division is between the *block address* and the *block offset*. The block address can be further divided into the *tag field* and the *index field*. The block offset field selects the desired data from the block, the index field selects the set, and the tag field is compared against it for a hit.



Q3: Which Block Should Be Replaced on a Cache Miss?

When a miss occurs, the cache controller must select a block to be replaced with the desired data.

There are three primary strategies employed for selecting which block to replace:

- *Random*—To spread allocation uniformly, candidate blocks are randomly selected. Some systems generate pseudorandom block numbers to get reproducible behavior, which is particularly useful when debugging hardware.
- *Least-recently used (LRU)*—To reduce the chance of throwing out information that will be needed soon, accesses to blocks are recorded. Relying on the past to predict the future, the block replaced is the one that has been unused for the longest time. LRU relies on a corollary of locality: If recently used blocks are likely to be used again, then a good candidate for disposal is the least-recently used block.
- *First in, first out (FIFO)*—Because LRU can be complicated to calculate, this approximates LRU by determining the *oldest* block rather than the LRU.

Q4: What Happens on a Write?

Reads dominate processor cache accesses. All instruction accesses are reads, and most instructions don't write to memory. Making the common case fast means optimizing caches for reads, especially since processors traditionally wait for reads to complete but need not wait for writes.

The block can be read from the cache at the same time that the tag is read and compared, so the block read begins as soon as the block address is available. Such optimism is not allowed for

writes. Modifying a block cannot begin until the tag is checked to see if the address is a hit. Because tag checking cannot occur in parallel, writes normally take longer than reads.

The write policies often distinguish cache designs. There are two basic options when writing to the cache:

- *Write through*—The information is written to both the block in the cache *and* to the block in the lower-level memory.
- *Write back*—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.