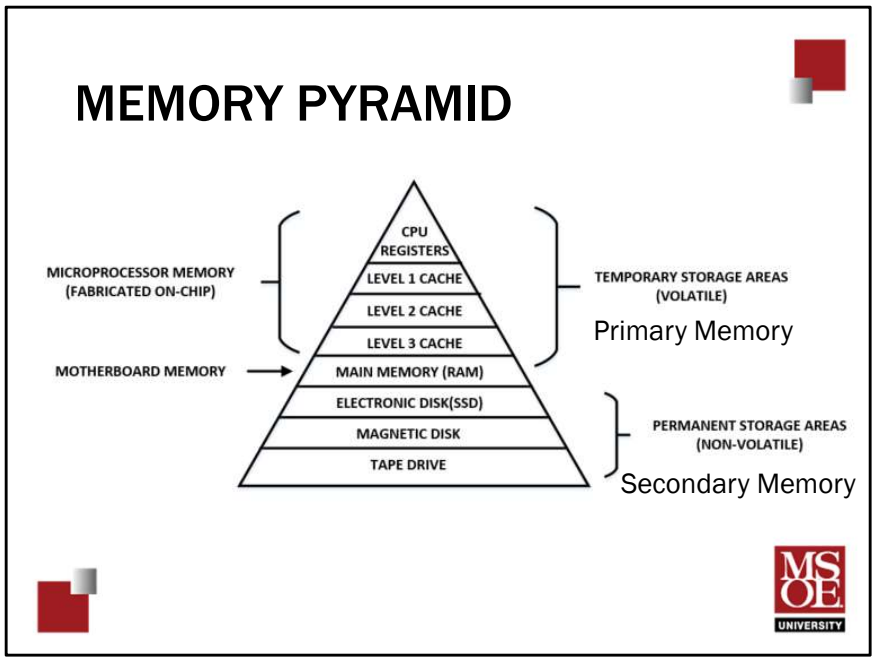

MSOE
UNIVERSITY
MEMORY
Dr. Russ Meier
Milwaukee School of Engineering



- Primary memory is the working electronic memory used by a running program. It is volatile – meaning that it loses its values when the power is turned off.
- Secondary memory is long term storage to hold data and programs for later use. It is non-volatile – meaning that it retains its values when the power is turned off.
- Speed is fastest at the top of the pyramid. These memories are also the most expensive per bit. Speed and cost both decrease as you go down the pyramid.
- Size is largest at the bottom of the pyramid. These technologies are also the cheapest per bit. Size decreases as you go up the pyramid.

MEMORY PYRAMID

LEVEL	NAME	TYPE	PRICE (2019)
L0	CPU REGISTERS	REGISTER	On-chip Microprocessor Memories
L1	CPU I-CACHE, CPU D-CACHE	STATIC RAM	
L2	LEVEL 2 SHARED CACHE	STATIC RAM	
L3	LEVEL 3 SHARED CACHE	STATIC RAM	
L4	MAIN MEMORY (RAM)	DYNAMIC RAM	$5/GB
L5	ELECTRONIC DISK (SSD)	NAND FLASH ROM	$0.13/GB
L6	MAGNETIC HARD DRIVE (HDD)	MAGNETIC PLATTER	$0.05/GB
L7	MAGNETIC TAPE DRIVE	MAGNETIC TAPE	$0.006/GB

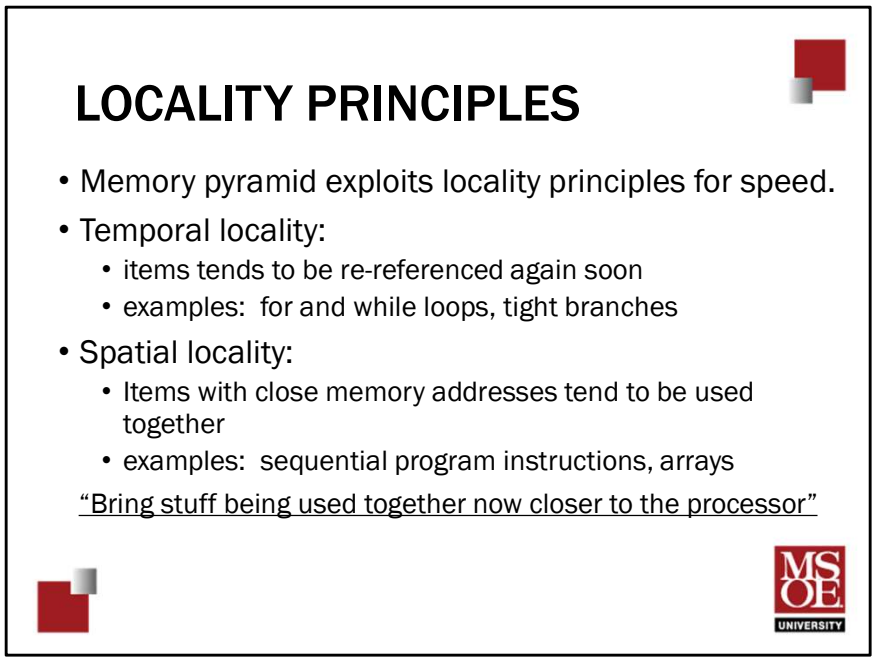
Pricing pulled from multiple electronic warehouses and averaged on May 15, 2019

IC MEMORY COMPARISON

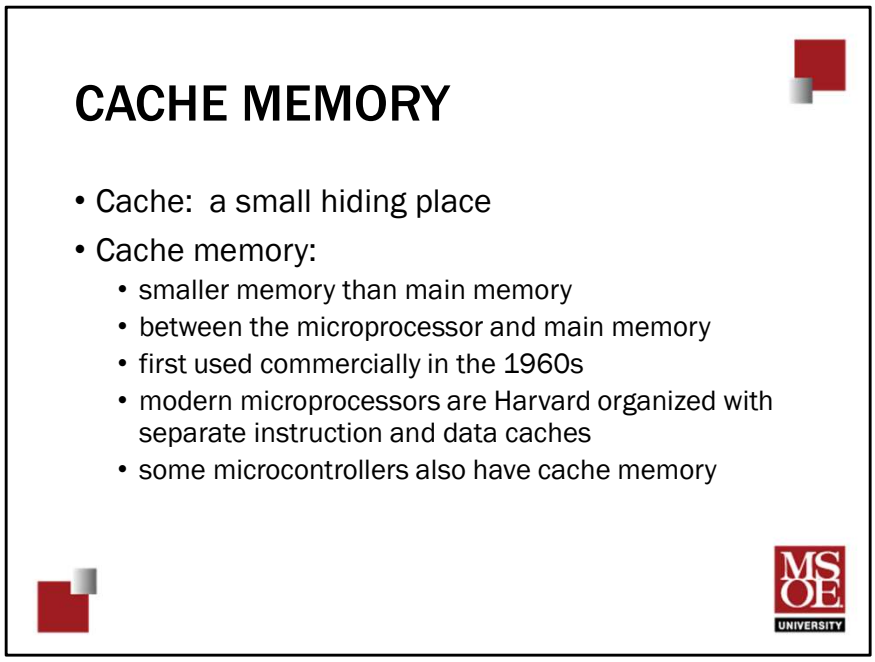
- Static RAM
  - fast: 10-50 ns delay
  - space per bit: 6 transistors per bit
  - expensive: $300-$500 per GB (2019 price: digikey.com)
  - use today: cache memory
  - refresh: not required : static storage

IC MEMORY COMPARISON

- Dynamic RAM
  - slower: 20-50 ns delay
  - space per bit: 1 transistor per bit
  - cheap: $10-15 per GB
  - use today: main memory
  - refresh: required : dynamic storage



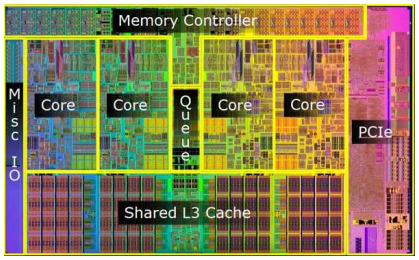
The locality principles led to the design of the memory pyramid. Smaller, faster, more expensive memories could be used close to the microprocessor to ensure the fastest access when the CPU needs the data. The goal would be to bring stuff being used together now closer to the processor.



- The register file is part of the central processing unit. It can be manipulated directly by assembly language programmers, but is usually not directly manipulated by high-level language programmers.
- Most programmers write in high-level languages. To these programmers the "***main memory***" is what they think of as their working storage.
- The locality principles suggested that a hidden smaller memory be placed between the larger main memory and the CPU registers. This hidden **cache** of data could be accessed more quickly by the CPU than main memory because it was smaller and faster.
- Caches are faster because they are fabricated today on the same silicon die as the central processing unit – avoiding the pin delay. They are faster because they are built from static-RAM rather than the forgetful DRAM that must be refreshed. They are faster because they operate at the speed of the CPU clock, while main memory usually operates at the speed of the memory bus protocol. These are some of the examples of why cache is faster. The Computer Architecture 3 elective examines memory systems in detail.
- Separate instruction and data caches support pipelines in modern central processing units.

CACHE MEMORY

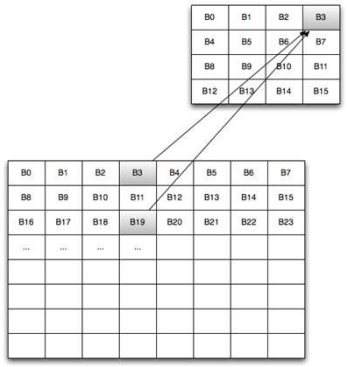
- Cache memory is also hierarchically split
- Traditional split
  - Level 1: microprocessor instruction and data
  - Level 2: separate silicon packaged in same chip
  - Level 3: motherboard
- Today
  - microprocessors generally have L1, L2, and L3 on-chip



Intel Core I5-750

CACHE MEMORY

- Memory address blocks map to cache blocks
- Multiple memory blocks will map to the same cache location
- Modulo-n arithmetic hash function



MSOE UNIVERSITY

- In its most basic form, a cache memory implements a hash table.
- Memory addresses are ***hashed*** into the smaller memory using a module-n arithmetic hash function.
- Remember that in computer architecture, a memory address represents a particular byte in memory.
- In this example, addresses are shown by their byte number.
- The example shows that byte B3 and byte B19 both map to location B3 in the smaller cache memory. This bin, bin B3, is the has location for both bytes.
- The hash location, also called the cache address, can be calculated as the remainder of taking the byte number and dividing by the size of the cache.
- The remainder is the **modulo** result.
- 3 / 16 = 0 remainder 3. Thus, byte B3 hashes into cache bin B3.
- 19 mod 16 = 3. Thus, byte B19 also hashes into cache bin B3.

DIRECT MAPPED CACHES

- simplest cache organization
- each memory address maps to exactly one cache
- classic mapping equation
  - Memory address bits form a binary address = b
  - Number of blocks in cache = cache size = n
  - cache address = b mod n

CACHE TERMINOLOGY

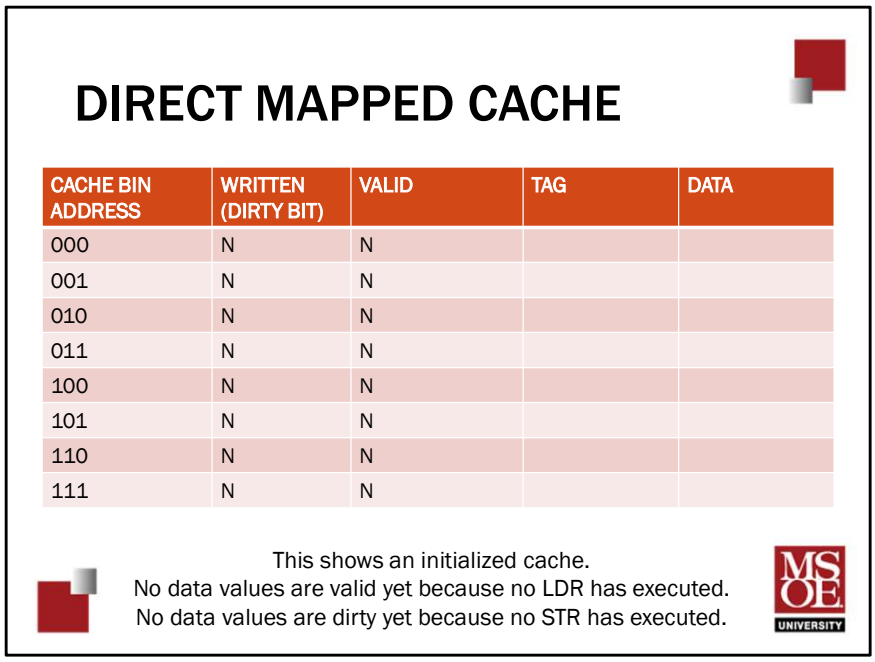
- block
  - smallest data size transferred between levels
- blocksize
  - size of the block in bytes
- memory hit
  - address access finds data in memory
- memory miss
  - address access does not find data in memory

CACHE TERMINOLOGY

- hit rate
  - fraction of times a memory hit occurs
- miss rate
  - fraction of times a memory miss occurs
  - Miss rate = 1 – hit rate
  - m = 1 – h
- hit time
  - examine address, determine validity, retrieve

CACHE TERMINOLOGY

- miss penalty
  - examine address, determine invalid, replace item by reading from lower level, retrieve
  - significantly longer delay than hit time
- coherency
  - process of maintaining consistent data across memory levels as misses occur
  - coherency replacement strategies are important



The next few slides will illustrate a cache memory filling bin locations with data. Loads and stores will change valid and dirty bits. Tags will be inserted. Fake data will be used to represent “something that came from the lower levels of memory.”

DIRECT MAPPED CACHE READ LOCATION 0xFBCA

Miss causes read into cache location 2, tag updated, valid bit changed to yes

CACHE ADDRESS	WRITTEN (DIRTY BIT)	VALID	TAG	DATA
000	N	N		
001	N	N		
010	N	Y	1111 1011 1100 1	0x49
011	N	N		
100	N	N		
101	N	N		
110	N	N		
111	N	N		

The LDR for this read presents address FBCA. It hashes to bin 2.
The cache does not find it. The location is not dirty.
The cache initiates a read from the lower level.

DIRECT MAPPED CACHE READ LOCATION 0x96AF

Miss causes read into cache location 7, tag updated, valid bit changed to yes

CACHE ADDRESS	WRITTEN (DIRTY BIT)	VALID	TAG	DATA
000	N	N		
001	N	N		
010	N	Y	1111 1011 1100 1	0x49
011	N	N		
100	N	N		
101	N	N		
110	N	N		
111	N	Y	1001 0110 1010 1	0xAB

The LDR for this read presents address 96AF. It hashes to bin 7.
The cache does not find it. The location is not dirty.
The cache initiates a read from the lower level.

DIRECT MAPPED CACHE READ LOCATION 0x0003

Miss causes read into cache location 3, tag updated, valid bit changed to yes

CACHE ADDRESS	WRITTEN (DIRTY BIT)	VALID	TAG	DATA
000	N	N		
001	N	N		
010	N	Y	1111 1011 1100 1	0x49
011	N	Y	0000 0000 0000 0	0xFF
100	N	N		
101	N	N		
110	N	N		
111	N	Y	1001 0110 1010 1	0xAB

The LDR for this read presents address 0003. It hashes to bin 3.
The cache does not find it. The location is not dirty.
The cache initiates a read from the lower level.

DIRECT MAPPED CACHE READ LOCATION 0x6826

Miss causes read into cache location 6, tag updated, valid bit changed to yes

CACHE ADDRESS	WRITTEN (DIRTY BIT)	VALID	TAG	DATA
000	N	N		
001	N	N		
010	N	Y	1111 1011 1100 1	0x49
011	N	Y	0000 0000 0000 0	0xFF
100	N	N		
101	N	N		
110	N	Y	0110 1000 0010 0	0xCC
111	N	Y	1001 0110 1010 1	0xAB

The LDR for this read presents address 6826. It hashes to bin 6.
The cache does not find it. The location is not dirty.
The cache initiates a read from the lower level.

DIRECT MAPPED CACHE READ LOCATION 0x992E

Miss causes conflict, not dirty, read overwrites location 6, tag updated, valid bit stays yes

CACHE ADDRESS	WRITTEN (DIRTY BIT)	VALID	TAG	DATA
000	N	N		
001	N	N		
010	N	Y	1111 1011 1100 1	0x49
011	N	Y	0000 0000 0000 0	0xFF
100	N	N		
101	N	N		
110	N	Y	1001 1001 0010 1	0x00
111	N	Y	1001 0110 1010 1	0xAB

The LDR for this read presents address 992E. It hashes to bin 6.
The cache does not find it. The location is not dirty.
The cache initiates a read from the lower level and overwrites.

DIRECT MAPPED CACHE
WRITE LOCATION 0x992E WITH 0x8E

Hit, mark dirty, write overwrites location 6, valid bit stays yes,

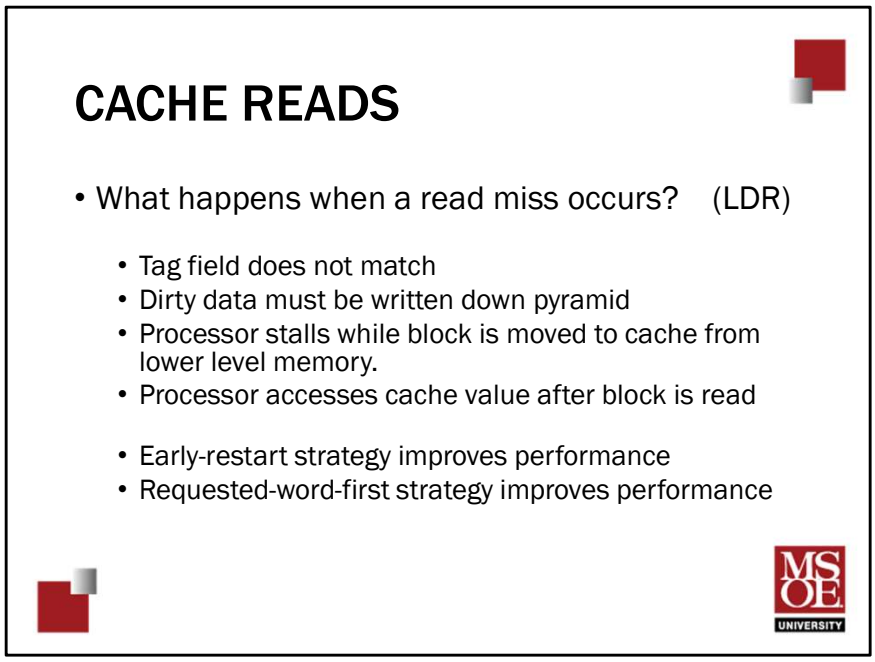
CACHE ADDRESS	WRITTEN (DIRTY BIT)	VALID	TAG	DATA
000	N	N		
001	N	N		
010	N	Y	1111 1011 1100 1	0x49
011	N	Y	0000 0000 0000 0	0xFF
100	N	N		
101	N	N		
110	Y	Y	1001 1001 0010 1	0x8E
111	N	Y	1001 0110 1010 1	0xAB

The STR for this read presents address 992E. It hashes to bin 6. The cache finds the address tag. The data location is written. The dirty bit is set because of the STR of new data.

Remember that the data ***lives in main memory*** during execution of a program. **Copies**, or **clones**, of the data live in the cache memory. The STR sets the dirty bit because this clone of the data will no longer match what is in main memory. It is now dirty.

CACHE READS

- What happens when read hit occurs? (LDR)
  - tag field matches
  - valid bit is yes
  - processor finishes access from cache



Early-restart and requested word first are algorithmic techniques used when the block size is not exactly one byte. In this case, multiple bytes are being moved from the lower level memory while the processor is stalled.

- Early-restart **releases the processor** as soon as the desired byte arrives in the cache memory. The cache controller continues moving the rest of the block after terminating the stall.
- Requested-word-first **moves the desired word from the lower memory first** and then releases the processor from stall. The cache controller continues moving the rest of the block after terminating the stall.

Stalls because of cache miss are significant. Cache stalls might hold the pipeline for hundreds of clock cycles because memory down the pyramid is so much slower. Any technique to reduce bubbles in the pipeline is important.

CACHE WRITES

- Writes result in different values across levels. (STR)
  - cache value
  - next lower level value
- A write strategy must be implemented. Why?
  - data must be kept consistent in all places

CACHE WRITES

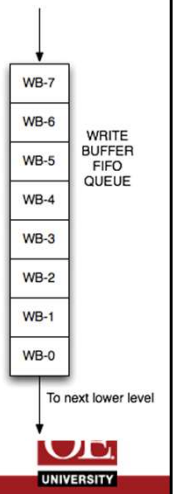
- Three classic cache write strategies:
  - write-through
  - write-buffer
  - write-back
- Advanced techniques exist and are in use

CACHE WRITES

- Classic write-through strategy:
  - Every write updates cache
  - Cache controller updates the next lower level
  - Each cache controller continues the updates
  - Results in main memory matching the clones at all times
  - Data and clones are always consistent
  - This has less desirable performance as the processor is stalled for the entire pyramid write-through

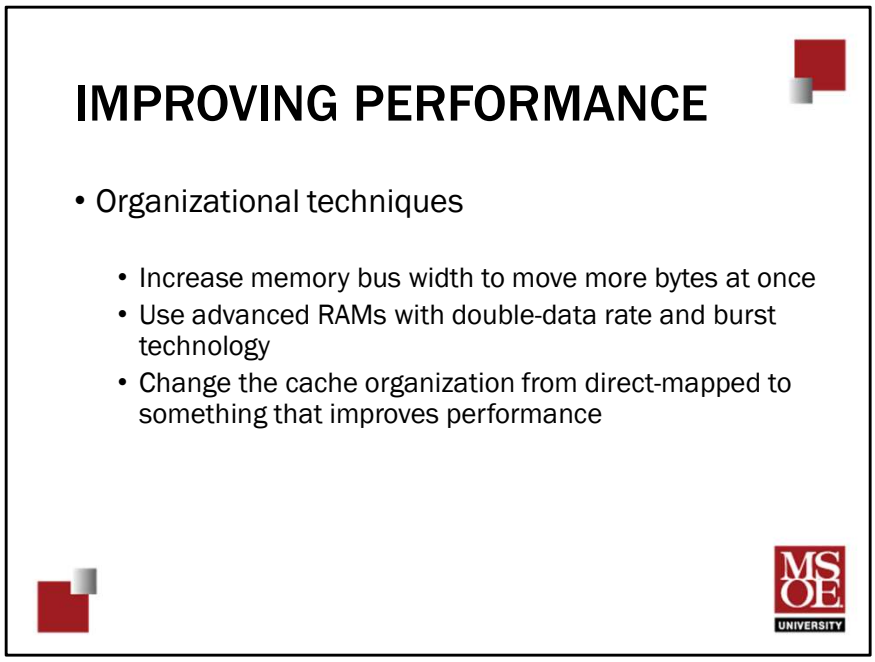
CACHE WRITES

- Classic write-buffer strategy:
  - Variation of the write-through strategy
  - Every write updates a cache buffer
  - The stall is released after the buffer write
  - Write-buffer controller updates next lower level
  - This technique has improved performance than when no buffer because the stalls are released. But, clones do not update in the same “time chunk” and thus can be inconsistent as processor instructions execute.

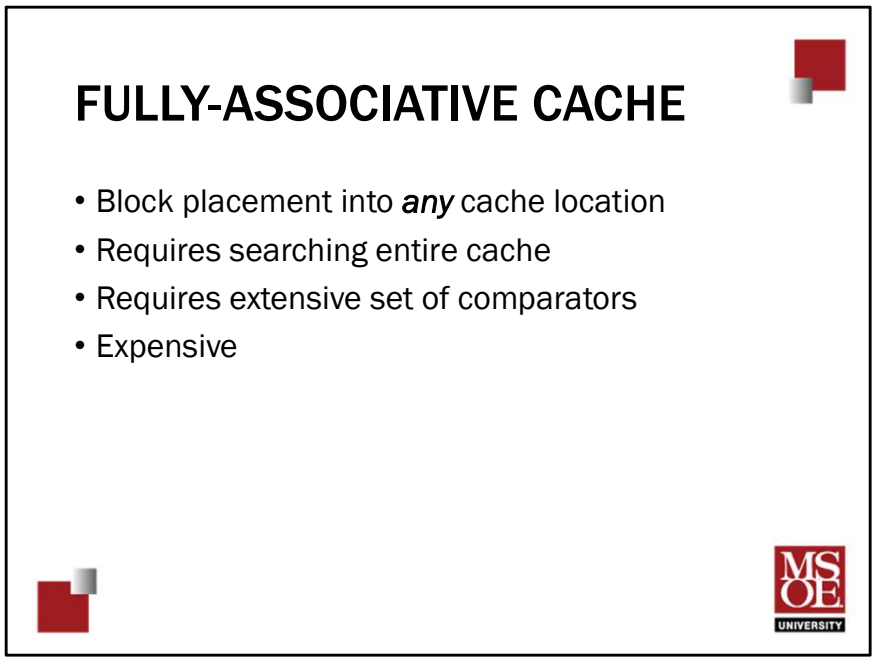


CACHE WRITES

- Classic write-back strategy:
  - Every write updates **only** cache
  - Blocks must be written back when replaced in the bin
  - The **write-bit** flags dirty data that must move down pyramid
  - More complex but higher performance / less stall



- Double-data rate memories provide values on both edges of the clock.
- Burst technology allows presentation of a starting address and then rapidly provides all sequential values beginning at the address on each clock edge. This avoids the overhead of providing each individual address.



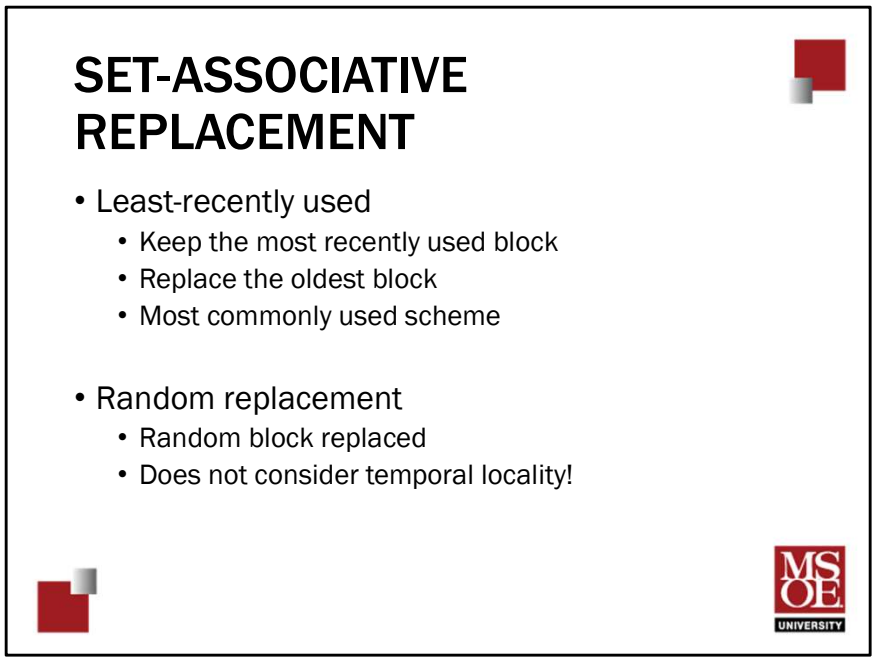
In a fully-associative cache, the hash equation is not used. Instead, the value moving into the cache can be placed into any bin. This requires the complete address to be stored as the tag field. It also requires comparators at every bin that compare the stored tag with the presented cache address. If any bin comparator matches, then a hit occurs. If no comparator matches, then a miss occurs. These comparators add space and thus expense.

SET-ASSOCIATIVE CACHE

- Keep multiple entries at index
- N-way set-associative means a set of size n at each index
- This example is 2-way set-associative
- Advantages: decreased miss rate
- Disadvantages: increased hit time and more comparators that direct mapped

INDEX	TAG	DATA	TAG	DATA
0				
1				
2	FBC 1	49		
3	000 0	FF		
4				
5				
6	682 0	CC	992 1	00
7	96A 1	AB		

In contrast to the fully-associative cache, a simpler solution that improves upon direct cache is the set-associative cache. This type of cache places multiple tag+data fields in each bin. The diagram shows a 2-way set-associative cache.



Because multiple things can now be held in a bin, some algorithmic policy is needed to determine which of the bin items gets evicted when something else needs to be stored there.