

Cache

CIT 595
Spring 2007

Cache

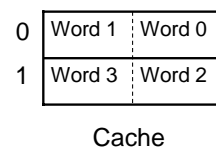
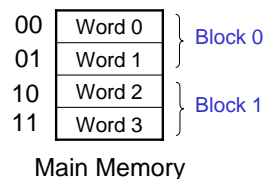
- The purpose of cache memory is to *speed up* data accesses for processor by storing information in faster memory made of SRAM
 - > SRAM access time is 3ns to 10ns
 - > DRAM access time is 30ns to 90ns
- The data stored in cache is data that the processor is *likely to use in the very near future*
 - > SRAM is fast but has less memory, so store only a subset of the data stored in main memory

CIT 595

11 - 2

Basic Terminology

- Memory is divided into *blocks*
- Each block contains *fixed numbers of words*
 - Word = size of data stored in one location e.g. 8 bits, 16 bits etc..
- One block is used as the *minimum unit of transfer* between main memory and cache
- Hence, each *location* in the cache stores **1 block**



CIT 595

11 - 3

Cache Mapping Scheme

- If the CPU generates an address for a particular *word* in main memory, and that data happens to be in the cache, the same main memory address cannot be used to access the cache
- Hence a *mapping scheme* is required that converts the generated main memory address into a cache location
- Mapping Scheme also determines where the block will be placed when it originally copied into the cache

CIT 595

11 - 4

Address Conversion to Cache Location

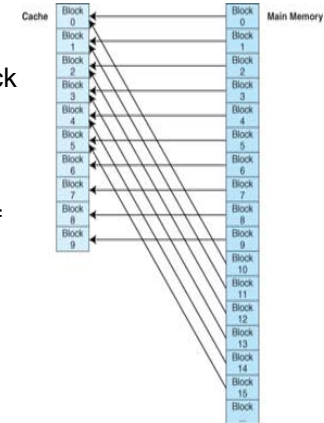
- Address Conversion is done by giving special significance to the *bits of the main memory address*
- The address is split into distinct groups called *fields*
 - Just like instruction decoding is done based on certain bit fields
- The group fields are a way to find:
 - Which cache location ?
 - Which word in the block ?
 - Whether it is the right data are looking for? Some kind of unique identifier

CIT 595

11 - 5

Mapping Scheme 1: Direct Mapped Cache

- In a direct mapped cache consisting of N blocks of cache (i.e. N locations), block X of main memory maps to cache block $Y = X \bmod N$.

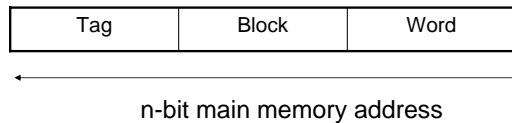


- E.g. if we have 10 blocks of cache, block 7 of cache may hold blocks 7, 17, 27, 37, ... of main memory.

CIT 595

11 - 6

Direct Mapped Scheme: Address Conversion



Word = which block in word?

Block = Which location in Cache?

Tag = unique identifier w.r.t one block

Note: Will explain why the breakup is this way....later

Note: Tag is used to distinguish whether main memory block 7 or 17 is stored in cache block 7

CIT 595

11 - 7

Cache with 4 blocks and 8 words per block

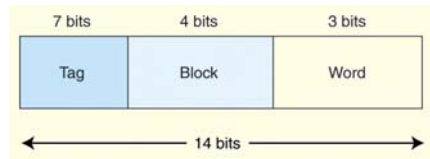
Block No.	Tag	Data							
0									
1									
2									
3									

CIT 595

11 - 8

Example of Direct Mapped Scheme

- Suppose our memory consists of 2^{14} locations (or words), and cache has $16 = 2^4$ blocks, and each block holds 8 words
- Thus main memory is divided into $2^{14} / 2^3 = 2^{11}$ blocks
- Of the 14 bit address, we need 4 bits for the block field, 3 bits for the word, and the tag is what's left over



CIT 595

11 - 9

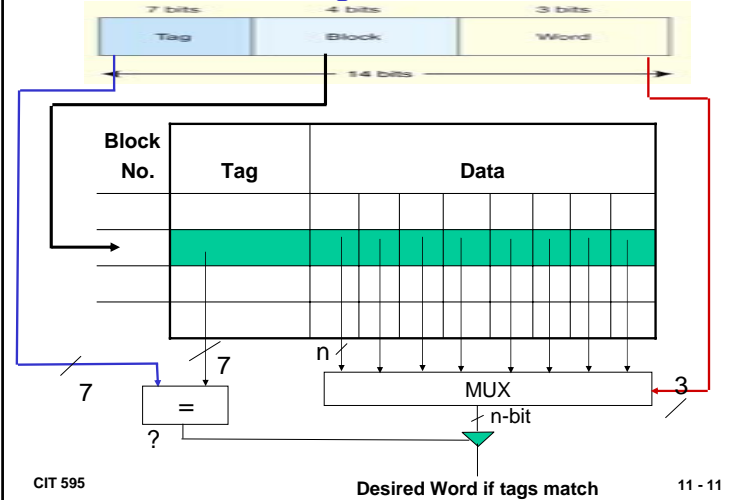
Direct Mapped Cache with 16 blocks

Block No.	Tag	Data							
0									
1									
2									
3									
4									
5									
⋮									
13									
14									
15									

CIT 595

11 - 10

Cache Indexing and Data Retrieval

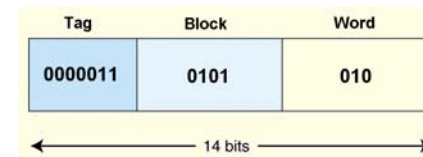


CIT 595

11 - 11

Example of Direct Mapped Cache (contd..)


- Suppose a program generates the address **LAA**
 - In 14-bit binary, this number is: 000001 1010 1010
- The first 7 bits of this address go in the tag field, the next 4 bits go in the block field, and the final 3 bits indicate the word within the block.



CIT 595

11 - 12

Direct Mapped Cache Example

Block No.	Tag	Data
0		
1		
2		
3		
4		
5	0000011	
⋮		
13		
14		
15		

CIT 595

1AB

1AA

11 - 13


Direct Mapped Cache Example (contd..)

- However, if the program generates the address, **3AB**
 - 3AB also maps to block 0101, but we will not find data for 3AB in cache
 - Tags will not match i.e. 0000111 (of addr 3AB) is not equal to 0000011 (of addr 1AB)
 - Hence we get it from main memory
 - The block loaded for address **1AA** would be evicted (removed) from the cache, and replaced by the blocks associated with the **3AB** reference

CIT 595

11 - 14

Direct Mapped Cache with address 3AB

Block No.	Tag	Data
0		
1		
2		
3		
4		
5	0000111	
⋮		
13		
14		
15		

CIT 595

3AB

11 - 15

Disadvantage of Direct Mapped Cache

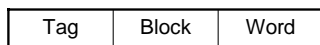
- Suppose a program generates a series of memory references such as: **1AB, 3AB, 1AB, 3AB, ..**
 - The cache will continually evict and replace blocks
- The theoretical advantage offered by the cache is lost in this extreme case
- Other cache mapping schemes are designed to prevent this kind of *thrashing*

CIT 595

11 - 16

Address Breakup

- Why is the address broken up in a particular manner ?



- This is done if we use *high-order memory address interleaving*
 - Due to spatial locality, data from consecutive addresses are brought into cache
- If the higher order bits (i.e. bits used for tag) are used for determining cache location then values from consecutive addresses would map to same location in cache
 - The *middle* bits are preferred for block location as they would cause less thrashing

CIT 595

11 - 17

Valid Cache block

- How do we know whether the block in cache is valid or not?
- For example:
 - When processor just starts up, the cache will be empty and tag fields in each location will be meaningless
 - Thus tag fields must be ignored initially when the cache is starting to fill up
- For validity, another bit called *valid bit* is added to the cache indicate whether the block contains valid information
 - 0 – not valid, 1 – valid
 - All blocks at start up would be not valid
 - If data from main memory is got into cache for a particular block, then valid bit for that field is set
 - Valid bit will contribute to the cache size

CIT 595

11 - 18

Direct Mapped Cache with Valid (V) Field

Block No.	Tag	Data	V
0			0
1			0
2			0
3			0
4			0
5	0000111		1
⋮			
13			0
14			0
15			0

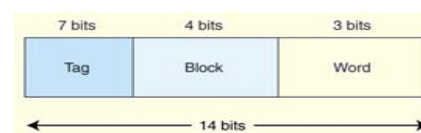
Address 3AB referenced for the first time. Entire block is brought into cache block 5.

CIT 595

11 - 19

Calculating Cache Size

Suppose our memory consists of 2^{14} locations (or words), and cache has $16 = 2^4$ blocks, and each block holds 8 words



- There are 16 locations in the cache
- Each row has 7 bits for tag + 8 words + 1 valid bit
- Assume 1 word is 8 bits, the total bits in row $(8 \times 8) + 7 + 1 = 72$
- 72 bits = 9 bytes
- Cache size = $16 \times 9 \text{ bytes} = 144 \text{ bytes}$

CIT 595

11 - 20

Hit or Miss in the Cache

- *Hit* means that we actually found data in the cache
- A hit occurs when valid bit = 1 *AND* tag in the cache matches the tag field of the address
- If both conditions don't hold then we did not find the data in cache
 - This is known as *miss* in cache
- On a miss, the data is brought from main memory into the cache, and the valid bit is set

CIT 595

11 - 21

Some more Terminology

- The *hit rate* is the percentage of time data is found at a given memory level
- The *miss rate* is the percentage of time it is not
- Miss rate = 1 - hit rate
- The *hit time* is the time required to access data at a given memory level
- The *miss penalty* is the time required to process a miss

CIT 595

11 - 22

Scheme 2: Fully Associative Cache

- Instead of placing memory blocks in specific cache locations based on memory address, we could allow a block to *go anywhere* in cache
- This way, cache would have to fill up before any blocks are evicted
- This is how *fully associative* cache works
- A memory address is partitioned into only two fields: the tag and the word

CIT 595

11 - 23

Fully Associative Cache: Address Conversion

- Suppose, as before, we have 14-bit memory addresses and a cache with 16 blocks, each block of size 8. The field format of a memory reference is:



- When the cache is searched, all tags are searched in *parallel* to retrieve the data quickly
 - More hardware cost than direct mapped
 - Basically we need “n” comparators where n = # of blocks in cache

CIT 595

11 - 24

Fully Associate: Which block to replace if cache is full?

- Recall that direct mapped cache evicts a block whenever another memory reference needs that block
- With fully associative cache, we have no such mapping, thus we must devise an *algorithm* to determine which block to evict from the cache
- The block that is evicted is called the *victim block*
- There are a number of ways to pick a victim, we will discuss them shortly....

CIT 595

11 - 25

Scheme 3: Set Associative

- Set associative cache combines the ideas of *direct* mapped cache and *fully* associative cache
- A set associative cache mapping is like direct mapped cache in that a memory reference maps to a particular location in cache
- But that *cache location can hold more than one* main memory block. The cache location is then called a *set*.
 - Instead of mapping anywhere in the entire cache (fully associative), a memory reference can map only to the subset of cache

CIT 595

11 - 26

Scheme 3: Set Associative

- The number of blocks per set in set associative cache varies according to overall system design
- For example, a 2-way set associative cache can be conceptualized as shown in the schematic below
 - Each set contains two different memory blocks

Set	Tag	Block 0 of set	Valid	Tag	Block 1 of set	Valid
0	00000000	Words A, B, C, ...	1		0
1	11110101	Words L, M, N, ...	1		0
2		0	10110111	P, Q, R, ...	1
3		0	11111100	T, U, V, ...	1

K-way set associate cache will have K blocks per set

CIT 595

11 - 27

Scheme 3: Address Conversion



- Like direct-mapped cache except, middle bits of the main memory address indicate the *set* in cache

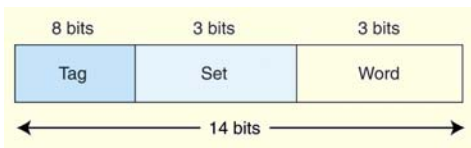
CIT 595

11 - 28

K-Set Associative Cache Example

- Suppose we have a main memory of 2^{14} locations
- This memory is mapped to a **2-way** set associative cache having **16 blocks** where each block contains 8 words
- Number of Sets = Number of Blocks in cache/ K
 - Since this is a 2-way cache, each set consists of 2 blocks, and there are 16 sets i.e. $16/2 = 8$

Thus, we need 3 bits for the set, 3 bits for the word, giving 8 leftover bits for the tag:



CIT 595

11 - 29

Advantage & Disadvantage Set Associative

- Advantage
 - Unlike direct mapped cache, if an address maps to a set, there is **choice** for placing the new block
 - If both slots are filled, then we need an **algorithm** that will decide which old block to evict (like fully associative)
- Disadvantage
 - **Tags** of each block in a set need to be **matched** (in parallel) to figure out whether the data is present in cache. Need **k** comparators.
 - Although, the hardware cost for matching is less than fully associative (need n comparators, where $n = \#$ blocks), but it is more than direct mapped (need only one comparator)

CIT 595

11 - 30

Which block to replace?

- With fully associative and set associative cache, a **replacement algorithm/policy** needs to be used when it becomes necessary to evict a block from cache
- The replacement policy that we choose depends upon the locality that we are trying to optimize
 - E.g. if we are interested in **temporal locality** i.e. referenced memory is likely to be referenced again soon (e.g. code within a loop) then we will keep the most recently used blocks

CIT 595

11 - 31

Replacement Algorithm/Policy

LRU - *Least recently used*

- Evicts the block that has **been unused for the longest period of time**
- The disadvantage of this approach is its complexity: LRU has to maintain an access **history** for each block, which will slow down the cache

CIT 595

11 - 32

Replacement Algorithm/Policy

FIFO - *First-in, first-out*

- In FIFO, the block that has been in the cache the *longest, regardless of when it was last used*

Random Replacement

- Does what its name implies: It picks a block at *random* and replaces it with a new block
- Random replacement can certainly evict a block that will be needed often or needed soon, but it *never thrashes* (like in the case of direct-mapped cache)

CIT 595

11 - 33

What about blocks that have been written to?

- While your program is running, it will modify some locations
- We need to keep main memory and cache *consistent* if we are modifying data
- We have two options
 - Should we update cache and memory at the same time? **OR**
 - Update the cache and then main memory at a later time
 - The two choices are known *Cache Write policies*

CIT 595

11 - 34

Cache Write Policies

Write-Through

- Update cache and main memory simultaneously on every write
- Advantage
 - Keeps cache main memory consistent at the same time
- Disadvantage
 - All writes require main memory access (bus transaction)
 - Slows down the system - If there is another read request for main memory due to miss in cache, the read request has to wait until the earlier write was serviced

CIT 595

11 - 35

Cache Write Policies (contd..)

Write Back or Copy Back

- Data that is modified is *written back* to main memory when the cache block is going to be *evicted* (removed) from cache
- Advantage
 - Faster than write-through, time is not spent accessing main memory
 - Writes to multiple words within a block require only one write to the main-memory
- Disadvantage
 - Need extra bit in cache to indicate which block has been modified
 - Like valid bit, another bit is introduced called *Dirty Bit*, to indicate a modified cache block.
 - 0 – Not Dirty, 1 – Dirty (modified)
 - Adds to size of the cache

CIT 595

11 - 36

Direct Mapped Cache with Valid and Dirty Bit

Block No.	Tag	Data	V	D
0			0	0
1	XXXXX		1	0
2			0	0
3	XXXXX	██████	1	1

██████ Dirty Words within one block

D – Dirty Bit V- Valid Bit

CIT 595

11 - 37

What affects Performance of Cache?

- Programs that exhibit bad locality
- E.g. Spatial Locality with matrix operations
 - Suppose Matrix data kept in memory is by rows (known as row-major) i.e. $\text{offset} = \text{row} * \text{NUMCOLS} + \text{column}$
- Poor code:
 - for (j = 0; j < numcols; j++)
 for(i = 0; i < numRows; i++)
 ➢ i.e. x[i][j] followed by x[i + 1][j]
 ➢ The array is being accessed by column and we going to miss in the cache every time
- Solution: switch the for loops
- C/C++ are row-major, FORTRAN & MATLAB is column-major

CIT 595

11 - 38

Multi-level Cache

- Most of today's systems employ multilevel cache hierarchies
 - The levels of cache form their own small memory hierarchy
- Current day processor uses:
- Level1 cache (8KB to 64KB) is situated on the processor itself
 - Access time is typically about 4ns
 - Level 2 cache (64KB to 2MB) located external to the processor
 - Access time is usually around 15 - 20ns

CIT 595

11 - 39

Multi-level Caches (contd..)

- In a multi-level cache:
 - If the cache system used an *inclusive cache*, the same data may be present at multiple levels of cache
 - *Strictly inclusive* caches guarantee that all data in a smaller cache also exists at the next higher level.
 - *Exclusive* caches permit only one copy of the data
- The tradeoffs in choosing one over the other involve weighing the variables of access time, memory size, and circuit complexity

CIT 595

11 - 40

Instruction and Data Caches

- A *unified* or *integrated* cache is one where both instructions and data are cached
- Many modern systems employ *separate* caches for data and instructions
 - This is called a *Harvard* cache
- Advantage:
 - Allows accesses to be less random and more clustered
 - Less access time than unified cache (typically larger)

CIT 595

11 - 41

Review of Cache Organization

Q1: Where can a block be placed in the cache level?

Mapping scheme

Q2: How is a block found if it is in the cache?

Mapping Scheme

Q3: If cache is full, then where do we put the new block i.e. which old block should we replace?

Block replacement policy

Q4: If we write to a block in cache, should we update the main memory at the same time?

Write Policy

CIT 595

11 - 42

Looking Forward

- Studied interaction between cache and main memory
 - The memory is managed by *hardware*
- Next study the interaction between main memory and disk
 - The memory is managed by hardware and *compiler*

CIT 595

11 - 43