

Memory Hierarchy Cache Organization

CIT 595
Spring 2008

Memory

- Von Nuemann Model – stored program concept
- $2^k \times m$ array of stored bits
 - Address: unique (k -bit) identifier of location
 - Contents: m -bit value stored in location
- Memory Access time affects CPU Performance
 - There is bound on how fast we can access data from memory
 - This latency inherently slow down the overall processing speed of the processor

CIT 595

2

Kinds of Memory

Volatile

- Once the power is off, the information is lost
- *RAM - Random Access Memory*
 - Access time is the same for all locations hence *Random Access*
 - Memory can be read and written
 - The instructions and/or data are stored when executing your programs

Non-Volatile

- E.g. Magnetic disk, ROMs, Flash RAM

CIT 595

3

Kinds of RAM: Type I

SRAM – Static RAM

- Memory we studied in chapter 3
 - SR Flip-Flop, D Flip-Flop
- 1-bit information (*memory cell*) needs cross-coupled gates
 - Consists of 8 transistors per cell (1 NAND/NOR gate requires 4 transistors)
 - Can be optimized to use 6 transistors per cell

CIT 595

4

Kinds of RAM: Type II

DRAM - Dynamic RAM

- 1-memory cell consists of one *capacitor* and *transistor*
 - Capacitor is used to store charge
 - Transistor acts as a switch which allows data to be read or written
- DRAM access is slow
 - Charge on capacitor needs to be sensed for 0 or 1
 - Capacitors slowly leak their charge over time and hence must be refreshed every few milliseconds to prevent data loss

CIT 595

5

DRAM vs. SRAM Technology

- DRAM is more *denser*
 - Stores more bits per surface area
 - It cost same to get 4MB SRAM vs. 1GB DRAM
 - DRAM ~250x cheaper than SRAM
- SRAM has faster access time
 - SRAM access time is 3ns to 10ns
 - DRAM access time is 30ns to 90ns
 - ~ 10x slow to SRAM

CIT 595

6

Performance/Cost/Capacity

- In general
 - Slow memory is cheap and has more storage capacity
 - Fast memory is expensive and has less storage capacity
- Ideal Goals
 - Memory that operates at processor speeds
 - Time it takes to compute basic operation
 - Don't want memory access time to dominate the clock cycle time or add to CPI
 - Memory as large as needed for all running programs
 - Memory that is cost effective
- So how do we get best of everything?
 - Use Memory *Hierarchy*

CIT 595

7

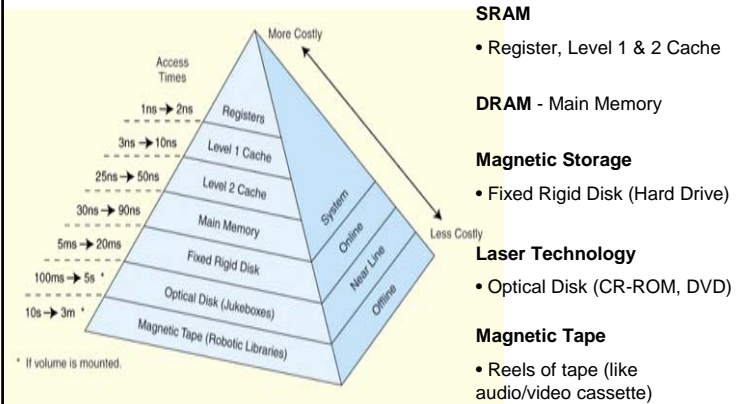
Memory Hierarchy

- To provide the best performance at the lowest cost, memory is organized in a hierarchical fashion
- Small, fast storage elements are *near* the CPU
- Larger, (almost) permanent storage in the form of disk and media storage is still further from the CPU
- Larger, slower memory is *accessed through* the data bus
- Each level of memory keeps a subset of the data contained in the lower memory-level

CIT 595

8

Memory Hierarchy: Pyramid Structure



CIT 595

9

Cache

- The purpose of cache memory is to hide the latency of main memory
- Cache is made from SRAM (as it is fast)
- But SRAM has less density
 - Store only a subset of the data from main memory
 - The data stored in cache is data that the processor is *likely to use in the very near future*
 - Cache – liking cashing (\$\$)

CIT 595

10

Processor<->Cache<->Other Memory

- To access a particular piece of data, the CPU first sends a request to *cache*
- If the data is not in cache, then *main memory* is queried
 - If the data is not in main memory, then the request goes to *disk*
 - More discussion later w.r.t Virtual Memory
- Once the data is located at a level, then the data, and a number of its nearby data elements are fetched into cache memory
 - E.g. If data from address x is requested, then data from address $X + 1$, $X + 2$, etc. is also sent
 - The data is collectively called a *block*

CIT 595

11

Why get nearby data?

Temporal Locality

- Referenced memory is likely to be referenced again soon (e.g. code within a loop)

Spatial Locality

- Memory close to referenced memory is likely to be referenced soon (e.g., data in a sequentially access array)

Sequential locality

- Instructions tend to be accessed sequentially

Use of locality

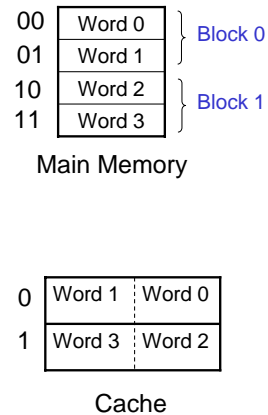
- Access to each lower memory adds to processing time
 - Minimize the access on future access
- Note: Bus-architecture is made to handle block transfer

CIT 595

12

Basic Cache Organization

- 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**
 - Also some extra info – more on it ahead



CIT 595

13

Cache Mapping Scheme

- Main memory address generated by the processor cannot be used to access the cache
- Hence a *mapping scheme* is required that converts the generated main memory address into a cache location
- Also determines where the block will placed when it is originally copied into the cache

CIT 595

14

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

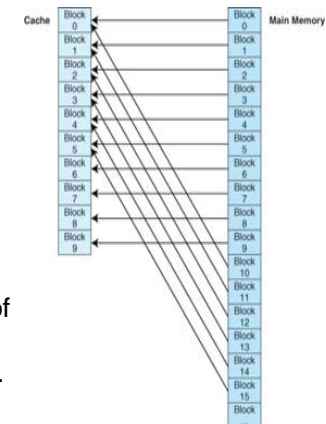
CIT 595

15

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 as

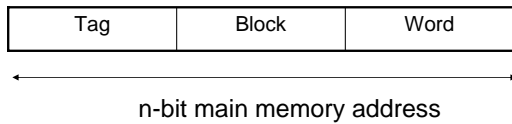
$$Y = X \bmod N$$
- E.g. if we have 10 blocks of cache, block 7 of cache may hold blocks 7, 17, 27, 37, ... of main memory.



CIT 595

16

Direct Mapped Scheme: Address Conversion



Word = which word in block?

Block = Which location in Cache?

Tag = unique identifier w.r.t one block

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

CIT 595

17

Cache Layout

Block No.	Tag	Data							
0									
1									
2									
3									

E.g. Cache with 4 blocks and 8 words per block

CIT 595

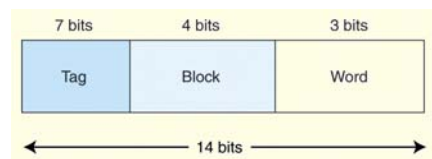
18

Example of Direct Mapped Scheme

- Suppose our memory consists of 2^{14} 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

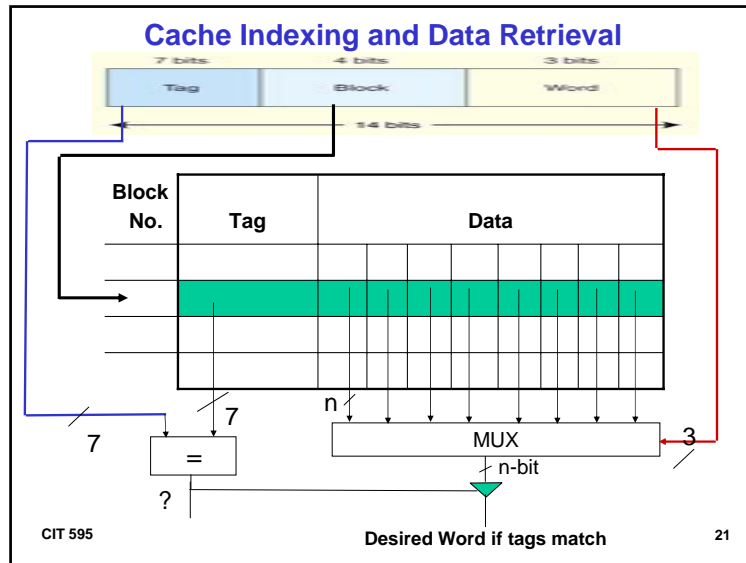
19

Direct Mapped Cache with 16 blocks

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

CIT 595

20



Example of Direct Mapped Cache (contd..)

- Suppose a program generates the address **1AA**
 - 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

Tag	Block	Word
0000011	0101	010

← 14 bits →

CIT 595 22

Direct Mapped Cache Example

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

1AB 1AA

CIT 595 23

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 (3AB) is not equal to 0000011 (1AB)s
 - 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 24

Direct Mapped Cache with address 3AB

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

CIT 595

3AB

25

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
 - Known as thrashing
- 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

26

Calculating Cache Size

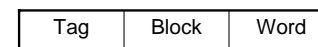
- Whenever Cache Size is mentioned, its stated with capacity of data that it holds
 - Tag storage is considered overhead
- Suppose our memory consists of 2^{14} locations (or words), and cache has $16 = 2^4$ blocks, and each block holds 8 words
- Cache Size = # of Blocks * Block Size
 - There are 16 locations in the cache -> # of Blocks
 - Each block stores 8 words
 - Assume 1 word is 8 bits, then Block size = 8 bytes
 - Cache size = $16 \times 8 \text{ bytes} = 128 \text{ bytes}$

CIT 595

27

Address Breakup

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



- Less variation in higher order bits compared to middle order bits
- If the higher order bits (i.e. bits used for tag) are used for determining cache location (block) then values from consecutive addresses would map to same location in cache
- The middle bits are preferred as they would cause less thrashing

CIT 595

28

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 as overhead bits

CIT 595

29

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

30

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

31

Mapping 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

32

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
- This requires evaluation hardware
 - Basically need a comparator circuit for each block in cache
 - Costly: because we need “n” comparators where n = # of blocks in cache

CIT 595

33

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

34

Mapping 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), map to a set

CIT 595

35

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	10111011	P, Q, R, ...	1
3	-----		0	11111100	T, U, V, ...	1

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

CIT 595

36

Scheme 3: Address Conversion



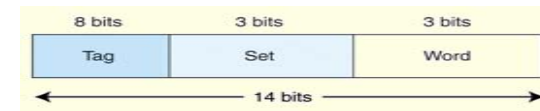
Like direct-mapped cache except, middle bits of the main memory address indicate the *set* in cache

CIT 595

37

K-Set Associative Cache Example

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



CIT 595

38

Advantage & Disadvantage Set Associate

- Advantage
 - Unlike direct mapped cache, there is less trashing
 - If an address maps to a set, there is *choice* for placing the new block and evicting an old block
- Disadvantage
 - **Tags** of each block in a set need to be *matched* (in parallel) to figure out whether the data is present in cache
 - Cost for matching is less than fully associative but it is more than direct mapped i.e. k comparators
 - Contributes to access time
 - If both slots are filled, then we need an *algorithm* that will decide which old block to evict (like fully associate)
 - Adds to design complexity

CIT 595

39

Replacement Algorithm/Policy

Optimal Goal

- Keep blocks required in the near future
- Replace block which is not used for the longest period of time

Least recently used (LRU)

- Evicts the block that has *been unused for the longest period of time*
- Disadvantage: complexity
 - LRU has to maintain an access history for each block, which will slow down the cache
 - Usually some approximation is used
 - E.g. Not Most Recently Used (NMRU)

CIT 595

40

Replacement Algorithm/Policy (contd..)

First-in, first-out (FIFO)

- In FIFO, the block that has been in the cache the *longest, regardless of when it was last used*
- Easy to implement compared to LRU
- Does not always match temporal locality

Random Replacement

- It picks a block at *random* and replaces it with a new block
- Can evict a block that will be needed often or needed soon, but it never thrashes

- Difficult to implement a truly random replacement

CIT 595

41

What about blocks that have been written too?

- While your program is running, it will modify some locations

- We need to keep main memory and cache *consistent* if we are modifying data

- Update cache and memory
 - Both at the same time
 - Update cache and then memory at later time
 - The two choices are known *Cache Write policies*

CIT 595

42

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
 - This is what we were avoiding in the first place when decided to introduce the cache

CIT 595

43

Cache Write Policies (contd..)

Write Back or Copy Back

- Modified data is written back to main memory when the block is going to be evicted (removed) from cache


- Advantage
 - Faster than write-through, time is not spent accessing main memory


- Disadvantage
 - Need extra bit in cache to indicate which block has been modified
 - Like valid bit, a another bit is introduced called *Dirty Bit*, to indicate a modified cache block.
 - 0 – Not Dirty, 1 – Dirty (modified)

CIT 595

44

Direct Mapped Cache with Valid and Dirty Bit

Block No.	Tag	Data	V	D
0			0	0
1	#####		1	0
2			0	0
3	#####		1	1

 Dirty Words within one block

D – Dirty Bit V- Valid Bit

CIT 595

45

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 are column-major

CIT 595

46

Size of block

- How many words must a block contain i.e. block size?
- If block size is too small
 - Cannot hold enough data that is mostly likely needed in the future
 - Need to get data from main memory frequently
- If block size is too large
 - Might benefit data if you are using lot of arrays
 - Might not benefit instructions as usually every few instructions a branch might be encountered and then block brought in will be wasted
 - Also, larger the block size, more circuitry is needed for comparison and muxing data words – will slow down cache access

CIT 595

47

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

48

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

49

Instruction and Data Caches

- The cache we have been discussing is called a *unified* or *integrated* cache where both instructions and data are cached
- Many modern systems employ *separate* caches for data and instructions
 - This is called a *Harvard* cache
- The separation of data from instructions provides better locality, at the cost of more hardware

CIT 595

50

EAT

- The performance of hierarchical memory is measured by its *effective access time* (EAT)
- EAT is a weighted average that takes into account the hit ratio and relative access times of successive levels of memory
- The EAT for a two-level memory is given by:
$$\text{EAT} = H \times \text{Access Time for Level } i + (1-H) \times \text{Access for Level } i+1$$
 - H is the hit rate i.e. % time data is found in level *i*
- This equation can be extended to any number of memory levels

CIT 595

51

Example of EAT

Consider a system with a main memory access time of 200ns supported by a cache having a 10ns access time and a hit rate of 99%.

$$\text{EAT} = 0.99(10\text{ns}) + 0.01(200\text{ns}) = 9.9\text{ns} + 2\text{ns} = 11\text{ns}$$

CIT 595

52

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