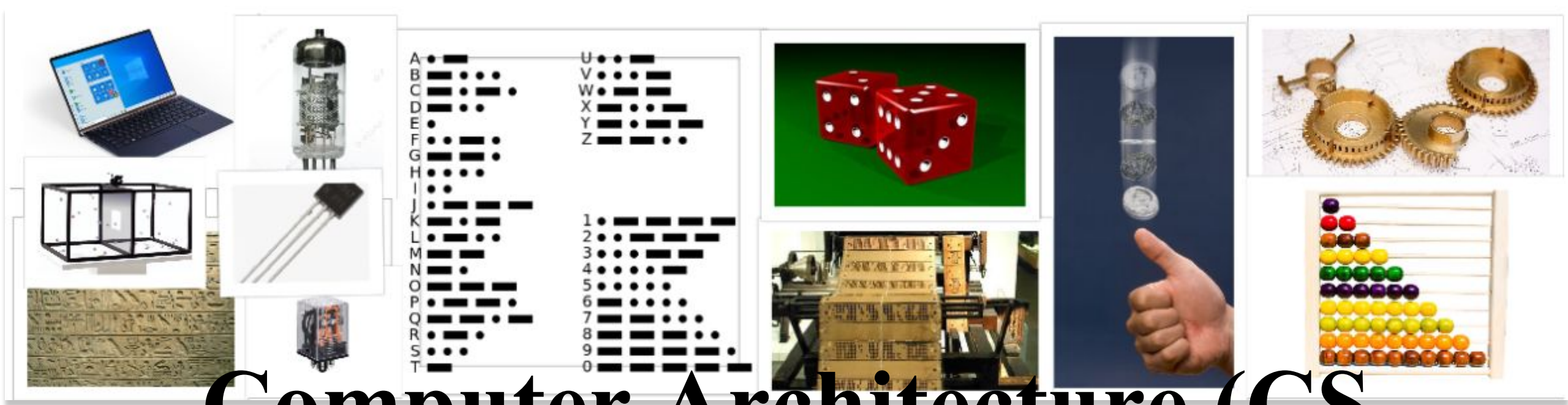



# Computer Architecture (CS F342)

## Fast Storage Unit: Cache Memory Architecture & Organization

## Read, Write & Replacement policies

# Cache Architecture: Read and Write Operations

- Caches have two characteristics: a read architecture and a write policy
- How many ways read and write operation can be done?
- Do we know any related problem?
  - Someone & friend want to buy an item and multiple shops
  - Update of someone's bank passbook

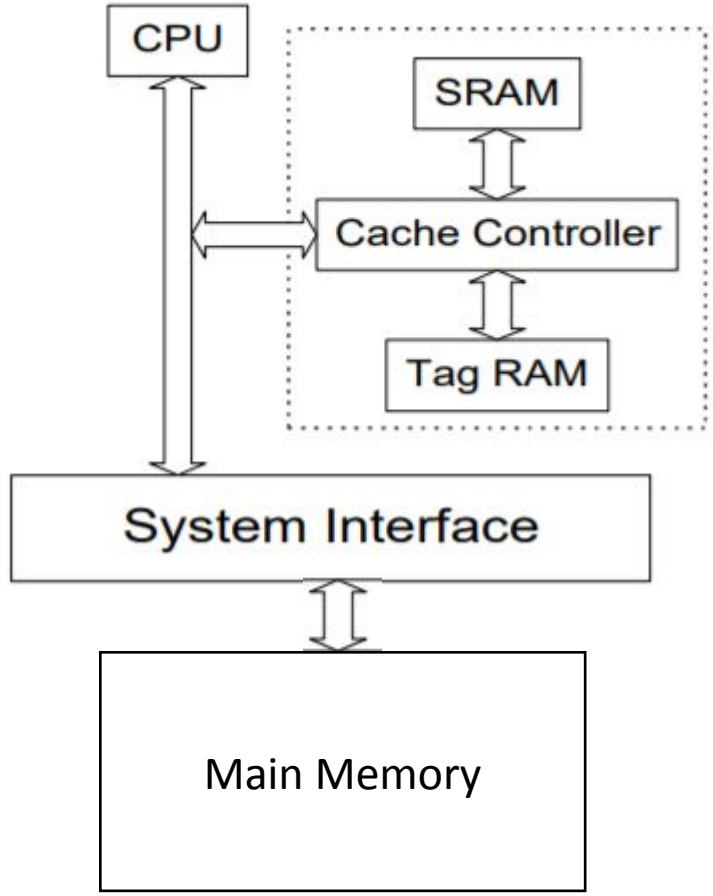
# Cache Architecture: Read and Write Operations

- The read architecture may be either “Look Aside” or “Look Through.”
- The write policy may be either “Write-Back” or “Write-Through.”
- Both types of read architectures may have either type of write policy, depending on the design.

# Cache Read Architecture: Look aside cache

- Cache unit sits in parallel with main memory

- Look aside: Both the main memory and the cache see a bus cycle at the same time



# Cache Read Architecture: Look aside cache

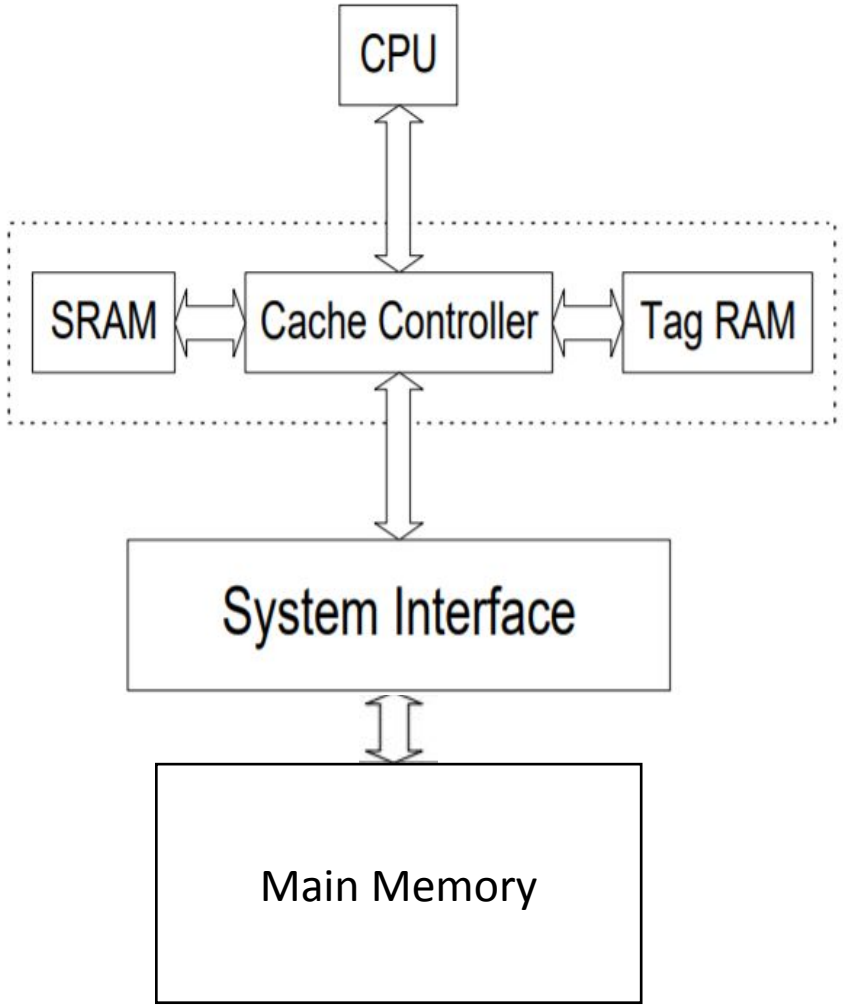
- When the processor starts a memory access, the cache checks to see if that address is a hit
- HIT:
  - The cache responds to the processor's request and terminate the bus cycle
- MISS:
  - Main memory response to the processors request & terminate the bus cycle
  - Cache stores the data for the use of next time

# Cache Read Architecture: Look aside cache

- It is less complex and less expensive
- Provide better response to a cache miss
- Drawback is the processor cannot access cache while another bus master is accessing main memory

# Cache Read Architecture: Look through cache

- Cache unit sits in between the processor & main memory
- Look through: Cache sees the processors bus cycle before allowing it to pass on to the system bus



# Cache Read Architecture: Look through cache

- When the processor starts a memory access, the cache checks to see if that address is a hit
- HIT:
  - The cache responds to the processor's request, without starting an access to main memory
- MISS:
  - The cache passes the bus cycle onto the system bus
  - Main memory response to the processors request
  - Cache stores the data for the use of next time

# Cache Read Architecture: Look through cache

- It allows the processor to run out of cache when another bus master is accessing the main memory
- It is a complex and more expensive
- Provides not better response to a cache miss

# Cache Write Architecture: Write back

- Data are in the main memory as well as in the cache
- How many ways one can write/update the data?
- Do we know any related problem?
  - Update of Bank's passbook
- Can we use that methodology?
  - Directly
  - Modified one

# Cache Write Architecture: Write back

- When processor starts a write cycle
  - Cache receives the data & terminate the cycle
  - Cache write the data back to main memory, when the system bus is available
- Provides the greatest performance
  - Allowing processor to continue its task
  - While main memory is updated at a time later time

# Cache Write Architecture: Write back

- How does this method distinguish among the cache blocks to write and not to write?
  - Extra bit can be added in each cache block called dirty or modified bit
  - Write the cache block in main memory, if dirty bit is 1
  - Don’t write the cache block in main memory, if dirty bit is 0

# Cache Write Architecture: Write through

- Processor write through the cache to main memory
- The write cycle will be completed when the cache as well as memory store the data
  - Write stall
  - Optimization is to minimize the Write stall
- It is less complex and expensive
- Performance is lower as compared to write back strategy

# Cache Write Architecture: Write through

- Processor write through the cache to main memory
- The write cycle will be completed when the cache as well as memory store the data
  - Write stall
  - Optimization is to minimize the Write stall
  - Introduce the write buffer
- It is less complex and expensive
- Performance is lower as compared to write back strategy

# Write misses

- When does the write miss operation happen?
  - Processor wants write the data in cache but it is not present
- How many ways one can implement such write operation?
  - Write allocate method
  - No-write allocate method

# Write misses: Write allocate

- Here write miss will behave as read miss
- The block is allocated on a write miss, followed by the preceding write hit actions

# Write miss: no-write allocate

- Does not affect the cache
- Block is modified only in the lower-level memory
- Here blocks stay out of the cache in no-write allocate until the program tries to read the blocks

# Example:1

- Assume a fully associative write-back cache with many cache entries that starts empty. Following is a sequence of five memory operations (the address is in squarebrackets):
  - Write Mem[100];
  - Write Mem[100];
  - Read Mem[200];
  - Write Mem[200];
  - Write Mem[100].
- What are the number of hits and misses when using no-write allocate versus write allocate?

# Example

- For no-write allocate, the address 100 is not in the cache, and there is no allocation on write, so the first two writes will result in misses. Address 200 is also not in the cache, so the read is also a miss. The subsequent write to address 200 is a hit. The last write to 100 is still a miss. The result for no-write allocate is four misses and one hit.
- For write allocate, the first accesses to 100 and 200 are misses, and the rest are hits because 100 and 200 are both found in the cache. Thus, the result for write allocate is two misses and three hits.

# Write allocate Vs. No-write allocate

- Either write miss policy could be used with
  - write through
  - or write back
- Usually, write-back caches use write allocate, hoping that subsequent writes to that block will be captured by the cache.
- Write-through caches often use no-write allocate. The reasoning is that even if there are subsequent writes to that block, the writes must still go to the lower-level memory, so what's to be gained?

# Emergent of replacement policies

- Cache size is finite and less than the size of main memory
- What do we do when it is full?
  - Cache with direct mapping technique
    - Only one choice & no need of any replacement policy
  - Cache with a set associative mapping technique
    - More choices on block to be replaced
      - Emergence of replacement policies

# Replacement Policies

- Random
- First In First Out (FIFO)
- Last In First Out (LIFO)
- Least Recently Used (LRU)
- Optimal Replacement

# Random & FIFO Replacement policies

- Random replacement policy
  - Replace/Evicts a randomly select block
  - How does one implement it?
    - Need a pseudo-random number generator
  - Does not take advantage of any temporal or spatial locality

- First-in, First-out (FIFO) policy
  - Evicts the block that has been in cache longest period of time
  - How does one implement this strategy?
    - Need the queue data-structure

# Least-Recently Used Replacement Policy

- Replace/Evicts the block that has referenced/used least recently
- How does one implement the strategy?
  - Incorporate 1-bit for each way
  - Consider Cache with degree of associativity = 2
  - On each access
    - Set the (LR)U-bit if it is way-0, otherwise Clear (LR)U-bit if it is way-1
  - Replace the block using LRU-bit

| (LR)U=0 | Way-0 | Way-1 |
|---|---|---|

# Example

- Show the contents of an two-way set associative cache after executing the following code. Assume no. of set is 4, LRU replacement and a block size of one word, and an initially empty cache.

```
lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```

$0x4 = (..0000\mathbf{01}00)_2$
$0x24 = (..0100\mathbf{01}00)_2$
$0x54 = (..1010\mathbf{01}00)_2$

| | | Way 1 | | | Way 0 | | |
|---|---|---|---|---|---|---|---|
| V | U | Tag | Data | V | Tag | Data | |
| 0 | 0 | | | 0 | | | Set 3 (11) |
| 0 | 0 | | | 0 | | | Set 2 (10) |
| 1 | 0 | 00...010 | mem[0x00...24] | 1 | 00...000 | mem[0x00...04] | Set 1 (01) |
| 0 | 0 | | | 0 | | | Set 0 (00) |

Step-1

| | | Way 1 | | | Way 0 | | |
|---|---|---|---|---|---|---|---|
| V | U | Tag | Data | V | Tag | Data | |
| 0 | 0 | | | 0 | | | Set 3 (11) |
| 0 | 0 | | | 0 | | | Set 2 (10) |
| 1 | 1 | 00...010 | mem[0x00...24] | 1 | 00...101 | mem[0x00...54] | Set 1 (01) |
| 0 | 0 | | | 0 | | | Set 0 (00) |

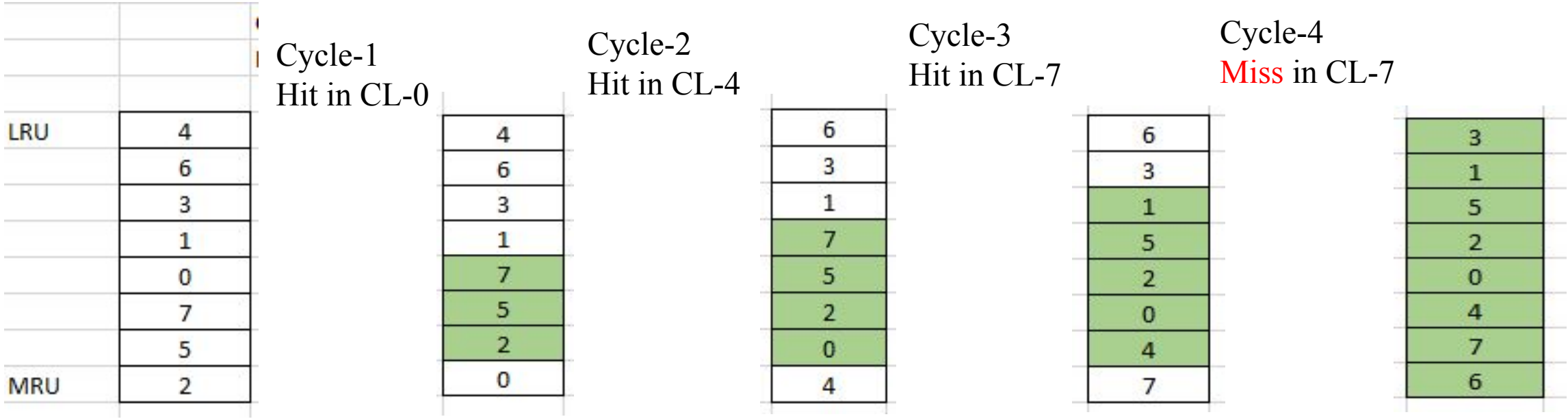
Step-2

# Least-Recently Used Replacement Policy

- What if associativity > 2
- LRU is difficult/expensive
- Record timestamp? How many bits needed for this?
- Search for minimum timestamp on eviction
- Sorted list? Re-sort on every access?
- Shift-register implementation

# Example

- Degree of associativity = 8



# Optimal Replacement Policy

- Replace/Evict block with longest reuse distance
  - Next reference to block is farthest in future
  - Need of knowledge of the future
- Can't be build it, for general purpose computing
- It is better than LRU
  - (X,A,B,C,D,X): LRU 4-ways

# Comprehensive examination



- Syllabus
  - Single-cycle Processor
  - Multi-cycle Processor
  - Pipelined Processor with minimization of hazards
  - Memory hierarchy & Cache memory
- Expected pattern of questions
  - Open book style & emergence of complexity/difficulty in questions
    - Conceptual & problem type questions ($2^{nd}$ or higher order type questions)
  - Closed book style
    - Direct, conceptual & problem type questions ($1^{st}$ & $2^{nd}$ order type questions)

# Summary

- Cache read operations
- Cache write operations
- Cache replacement policies