Memory Read and write operations

Read (using Multiplexers):

1. Transfer the binary address of the desired word to the address line (tells the memory where to read from)
2. Activate read control line (tells memory to perform read operation)

Write (using decoder):

1. Transfer the binary address of the desired word to the address lines (tells the memory where to store the data)
2. Transfer data bits that must be stored in memory to the data output line (data we want to write)
3. Activate the write control line (tells memory to perform write operation)

Temporal Locality: When an instruction is executed or data is accessed, it is stored in the cache because there is a high probability it will be accessed again.

eg. (loop variables)

```
while (condition) {
   i++; // access variable
} // for 100 iterations, the variable would be referenced again and again
```

Spatial Locality: When an instruction is executed or data is accessed, nearby items are also loaded into the cache because there's a high probability they'll be accessed soon

eg. Arrays and vectors

Cache Hit: a memory access where the data is already in cache
Cache Miss: a memory access where data isn't in the cache
Hit ratio: (# of cache accesses)/(# of total accesses)

eg.

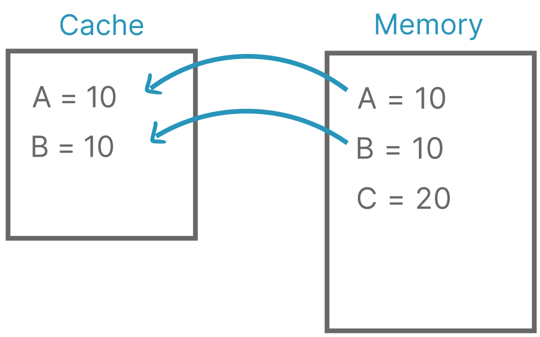
× ✓ ✓ ✓ × × ✓ ✓
Data to access → 5, 5, 5, 5, 6, 7, 6, 5

Hit ratio: 5/8

× Cache Miss
✓ Cache Hit

Data gets into the cache by a read operation only.

Cache/Memory Example: C = A + B (read A, read B, write C)



What we bring in-to the cache is based on the principle of locality.

Direct Mapping: Each block of main memory maps to only one cache line:

(block #) mod (# of lines)

Direct mapped cache example:

Cache size 4 (r = 2) , memory size: 32 blocks (s = 5)
Line 0: block 0, 4, 8, 12
Line 1: block 1, 5, 9, 13

Replacement Policies:

- When cache is full, a line must be replaced
- Most common strategy: Least Recently Used (LRU)

Write Policies:

- Write-through: Update RAM every time cache is updated
- Write-back: Delay RAM update until block is evicted from cache

Cache: Direct Mapping Address Structure

Direct Mapping Fundamentals

- Main Memory Structure:
  - Memory is divided into blocks (eg. 64 blocks)
  - Cache is made of lines

- Cache Mapping Formula:
  - A memory block maps to a specific cache line
    cache line index = (block #) mod (# of lines)

Example:
- Main Memory has 32 blocks (0 - 31), cache has 4 lines (0-3) * Unrealistic Example
- Block 25 maps to line 1 (25 mod 4 = 1)
- Block 4 and block 16 both map to line 0
- Consequence: If block 4 is in line 0, and block 16 is loaded, then block 4 gets evicted

Tag and Line Number: Address Breakdown

- When we store to cache, we want to store the block # alongside the data data, to better be able to check the cache for hits/misses
- But we don't want to store the entire block # since that would take up too much space

If we have a 4-line cache, any block # that is a multiple of 4 will map to line 0
Likewise, any mem block # that is a (multiple of 4) + 1 will map to cache line 1

line 0:

Mem block #	block # binary
block 0	000 : 00
block 4	001 : 00
block 8	010 : 00
block 12	011 : 00
⋮	

line 1:

Mem block #	block # binary
block 1	000 : 01
block 5	001 : 01
block 9	010 : 01
block 13	011 : 01
⋮	

We notice that the last 2 digits are the same, so we can simply store the first 3 digits and add back the last 2 digits on the fly when we want to access the cache. This "truncated" block # is called the tag

When a block is loaded into a cache line, its identity must be memoized using a tag
- Tag = top bits of the address (unique identifier)
- Line Number = lower bits (used to index into the cache)

General Formula
- Address size: S bits (memory size = 2^S)
- Cache size: 2^R lines (line index size = R bits)
- Tag size = $S - R$

Example:
- Memory has 32 blocks = 5 bit address ($S = 5$)
- Cache has 4 lines = 2 bit line # ($R = 2$)
- Tag = 5 - 2 = 3 bits

Cache Structure

Each cache line contains:
- Valid/invalid bit (V/I): Indicates if the data is valid
- Tag: Used for identifying the block stored
- Data: The contents of the block

Cache Line Format

V/I	Tag	Data
1 bit	x bits	y bits

Tag and data bit width will depend on the system being described

Direct Mapping Address Structure

Address size (S bits)

Tag	line index	Word Index	Byte Index
S - R - W - B bits	R bits	W bits	B bits

- S = Total address bits
- R = Bits for line index
- W = Bits for word index (If multiple words per block. If only 1 word per block, we don't include a partition for the word index)
- B = Bits for byte index (if the system uses byte addressing, otherwise we don't include this partition for the byte index)

Address Decomposition for Word and Byte Addressing

Scenarios:
- Word Addressing, 1 word per block

Tag	line index

- Byte Addressing, 1 word per block, word size = 4 bytes

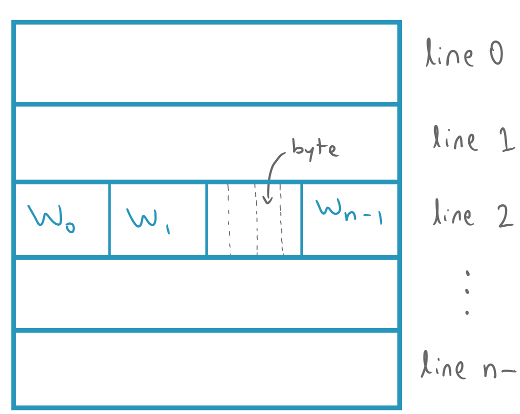
Tag	line index	byte offset (2 bits)

- Word Addressing, multiple words per block

Tag	line index	word index

- Byte Addressing, multiple words per block

Tag	line index	word index	byte offset



Cache Initialization and Context Switching

- On program startup: all cache lines are marked invalid
- On context switch:
  - Previous process's cache data becomes irrelevant
  - All cache lines are again marked invalid
- Valid bit is set to 1 only when new block data is loaded

Total Cache Size Calculation

Formula:

$$\text{Total cache size} = \#\text{ of lines} \times (1 + \text{Tag Bits} + \text{Data bits})$$

- Find the total # of bytes for a direct mapping cache to store 64 KB in 1-word blocks assuming a word size of 32 bits and MIPS addressing

↳ Break down the specifications:
- Word size = 32 bits → 32 bit address
- Block size = 1 word
- Addressing mode: byte addressing (from what we know of MIPS)
- Data: 64 KB

* b = bits

* B = bytes

Block size = 1 word → Since we only have 1 word per block, we don't need to allocate any bits for word select

Word size = 32 bits → 4 B = 2^2 B → Allocate 2 bits for addressing byte # (byte addressing)

Data = 64 KB = 64×2^{10} B = $2^6 \times 2^{10}$ B = 2^{16} B

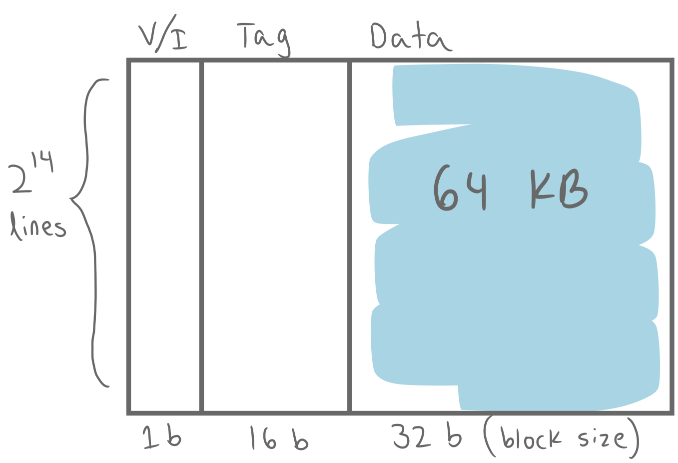
↳ $\#\text{ of lines} = \dfrac{\text{Data size}}{\text{block size}} = \dfrac{2^{16}\text{ B}}{2^2\text{ B}} = 2^{14}$ lines

Tag = 32 − (14 + 2) = 16 bits

↳ Address Structure

Tag	line #	byte #
16 b	14 b	2 b

Cache Structure



→ 49 b

Total cache size

2^{14} ~~lines~~ × 49 bits/~~line~~ = $(49 \cdot 2^{14})$ bits

= $(49 \cdot 2^{11})$ Bytes = $(2 \cdot 49 \cdot 1k)$ B = $\boxed{98\text{ KB}}$