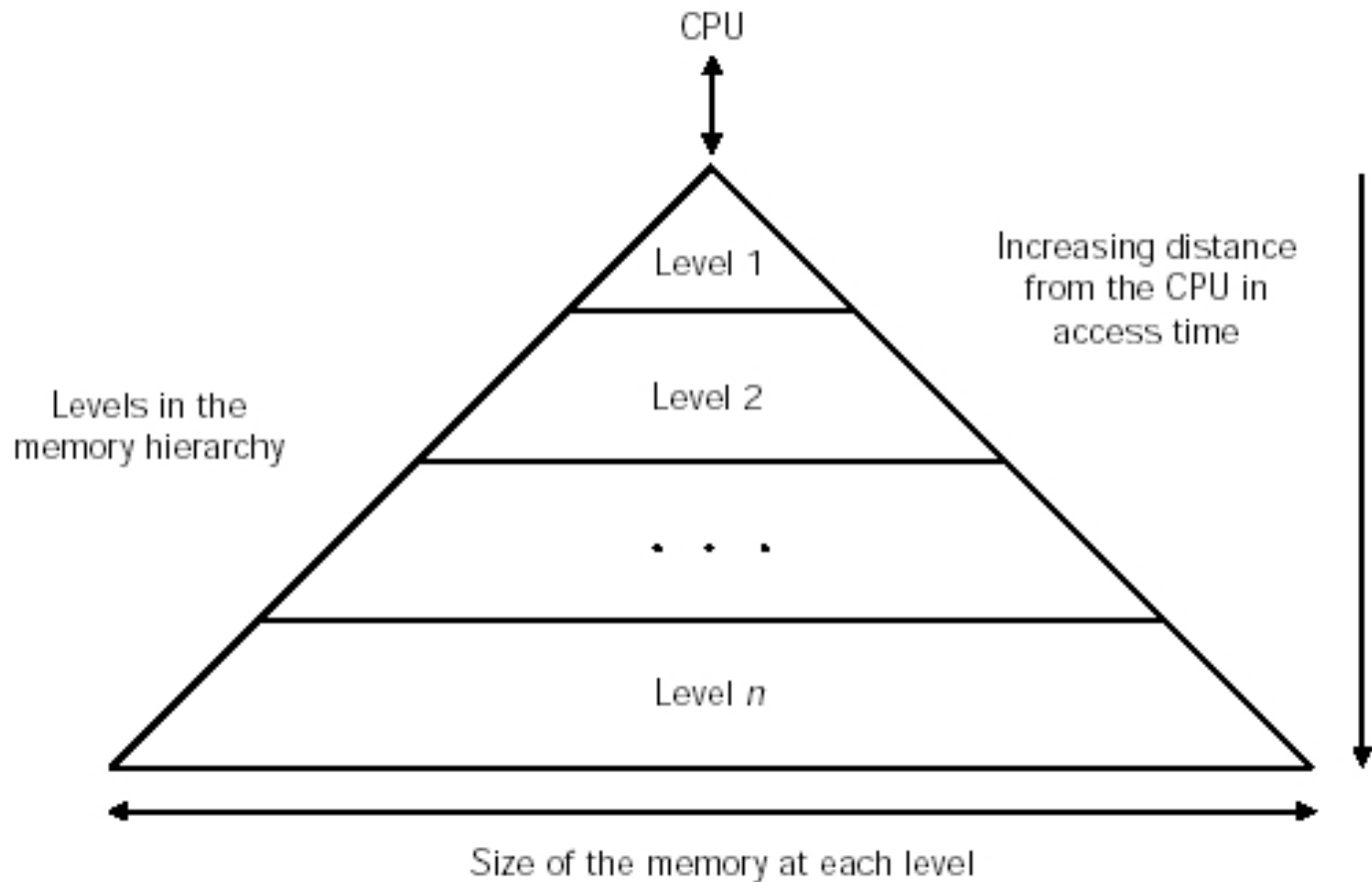


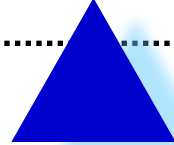
14:332:331

▼ The Memory Hierarchy



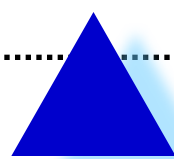


A Typical Memory Hierarchy

- Architectures need to create the illusion of *unlimited* and *fast* memory;
 - To do so we remember that applications do not access all the program memory, or data memory at once
 - Rather programs access a relatively *small portion* of their address space at any moment in time;
- 

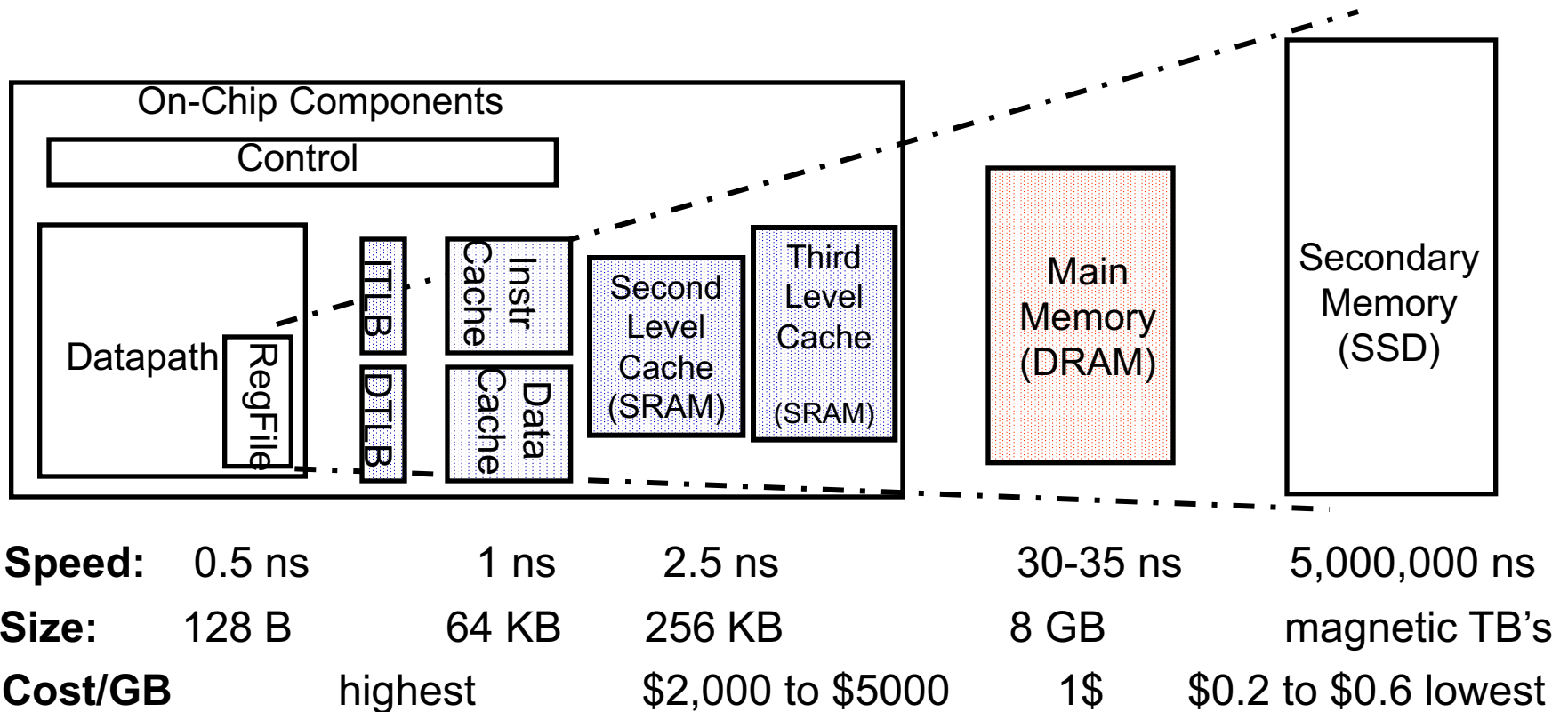


A Typical Memory Hierarchy

- **Temporal locality** - means that a referenced item in memory tends to be referenced again soon; (loops)
 - **Spatial locality** - means that if an item is referenced, those adjacent to it tend to be referenced soon. (arrays)
- 

A Typical Memory Hierarchy

- These principles allow memory to be organized as a hierarchy of multiple levels with different access speeds and sizes

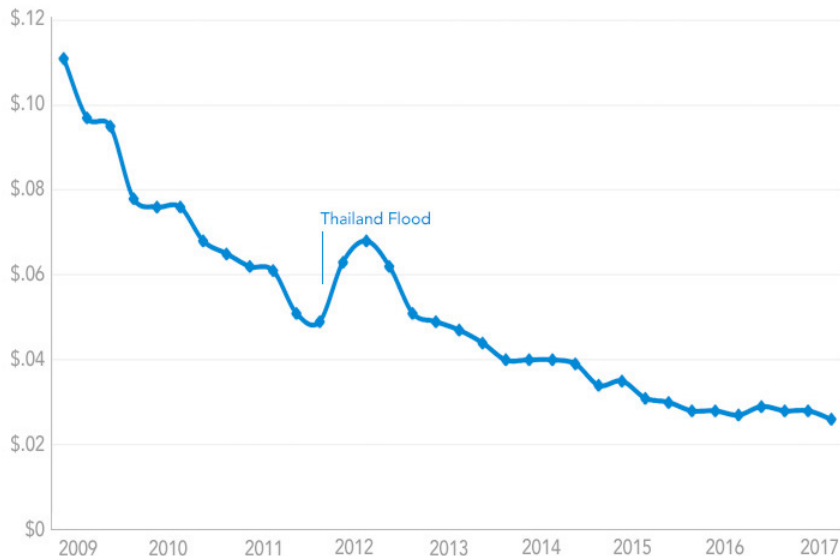


Hard Drive dramatic cost drop

- SSDs gain an increasing share of the market

Backblaze Average Cost per GB for Hard Drives

By Quarter: Q1 2009 - Q2 2017



BACKBLAZE

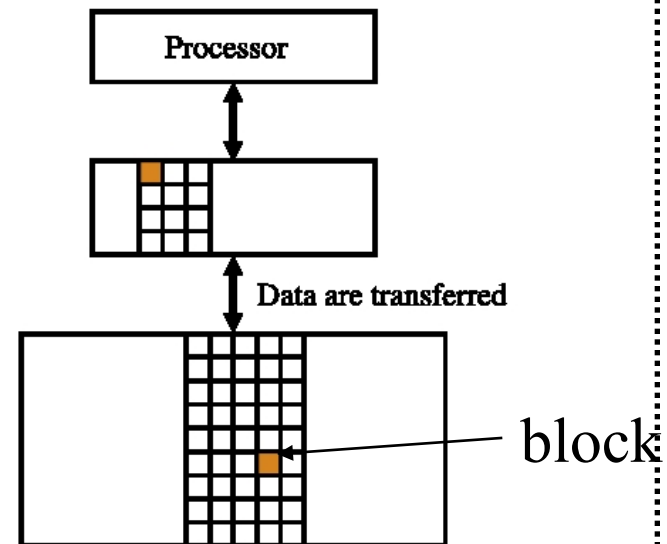
Total HDD + SSD Capacity (Exabytes); SSD as % of Total



Source: IDC; Stifel

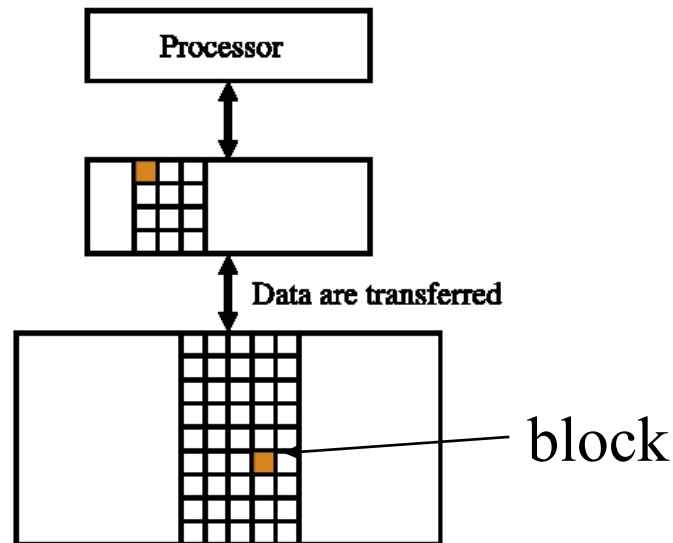
A Typical Memory Hierarchy

- Since main memory is much slower, to avoid pipeline stalls, the data and the instructions that the CPU needs soon should be moved into cache(s).
- Even though memory consists of multiple levels, data is copied only between two *adjacent levels* at a time.



A Typical Memory Hierarchy

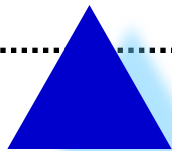
- Within each level, a *unit of information* that is present or not is called a **block**. Blocks can be either single-word (32 bits wide), or multiple-word.





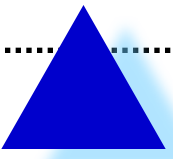
How is the Hierarchy Managed?

- When the data that the CPU needs are present in cache, it is a **hit**. When the needed data are not present, it is a **miss**, and misses have *penalties* -
- When there is a miss, the pipeline is frozen until data is fetched from a lower level cache or from main memory- this affects performance.

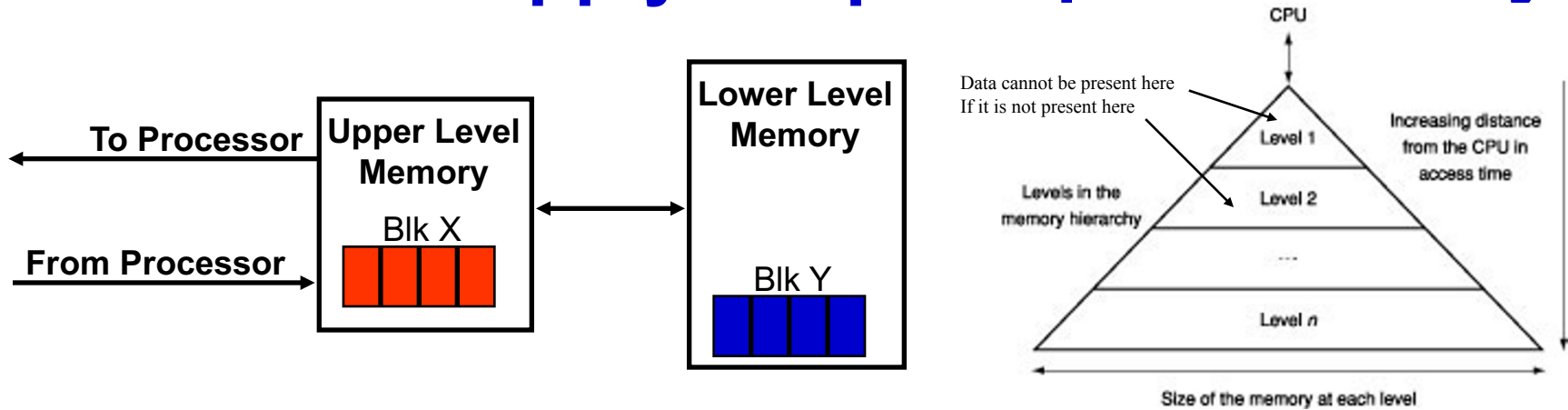




How is the Hierarchy Managed?

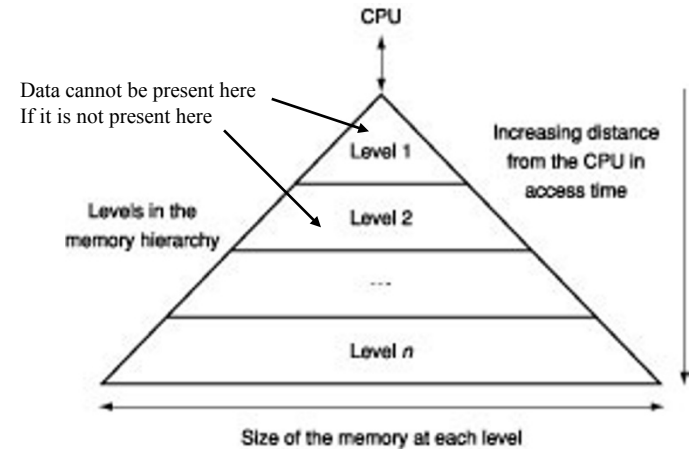
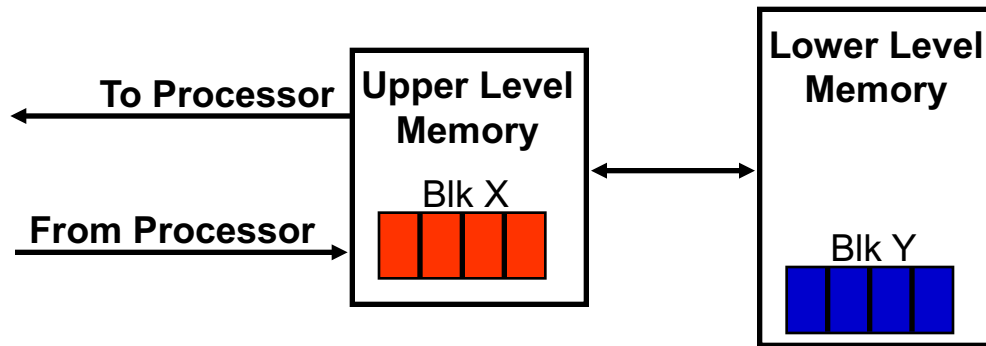
- Data need to always be present in the lowest level of the hierarchy.
 - Temporal Locality - Keep **most recently** accessed data items *closer* to the processor
 - Spatial Locality - Move **blocks** consisting of *contiguous* words to the upper levels
- 

How do we apply the principle of locality?



- **Hit**: data needed by CPU appears in *some* block in the upper level (Block X)
- **Hit Ratio**: the fraction of accesses found in the upper level (or hits/accesses)
- **Hit Time**: Time to access upper level = access time + Time to determine if access is hit/miss

How do we apply the principle of locality?



- **Miss Ratio** = $1 - \text{Hit Ratio}$ (or misses/accesses)
- **Miss penalty**: time for data to be retrieve from a *lower* level memory (needed data is in Block Y) and deliver it to the processor
- Hit Time is much smaller than Miss Penalty

Miss Penalty

- **Average Memory Access Time (AMAT)**
= Hit Time + Miss Penalty x Miss Rate
- Recall that $\text{CPUtime} = \text{CPUexec.time (includes hits)} + \text{MemAccess.time}$
- $\text{MemStall.Access.time} = (\text{Read-Stall.cycles} + \text{Write-stall.cycles}) \times \text{clock cycle.time}$
- $\text{MemStall time} = \text{Reads/Program} \times \text{Read miss rate} \times \text{Miss penalty} + \text{Writes/Program} \times \text{Write miss rate} \times \text{Miss penalty (ignore write buffer stalls)}$
- Or $\text{Mem-stall} = (\text{Misses/program}) \times \text{Miss penalty}$

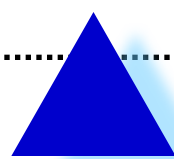


Miss Penalty

- What is the degrading influence of misses (stalls)?

$$\frac{\text{Performance perfect cache}}{\text{Perform. cache with stalls}} = \frac{\text{CPUtime with stalls}}{\text{CPUtime without stalls}}$$

$$= \frac{\text{CPU exec time} + \text{Mem. Stall time}}{\text{CPU exec time}} =$$

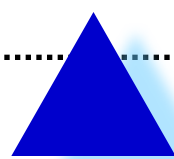
$$= \frac{\# \text{ Instr.} \times \text{CPI}_{\text{withstall}} \times \text{Clock cycle time}}{\# \text{ Instr.} \times \text{CPI}_{\text{perfect}} \times \text{Clock cycle time}} = \frac{\text{CPI}_{\text{withstall}}}{\text{CPI}_{\text{perfect}}}$$


Miss Penalty

- CPIwithstall depends on the application.
- $\text{CPIwithstall} = \text{CPIperfect} + \text{CPImisses} =$
 $= \text{CPIperfect} + \text{CPImiss inst} + \text{CPI miss.data.mem}$
- If miss penalty is 100 cycles, miss rate for instructions is 2%, miss rate for data memory cache is 4%, and data memory access rate is 36% (for gcc example) and CPIperfect is 2, then
- $\text{CPImisses} = (2\% + 36\% \times 4\%) \times 100 = 3.44$
$$\frac{\text{Performance perfect cache}}{\text{Performance with stalls}} = \frac{2 + 3.44}{2} = 2.72$$
- Perfect cache is 2.72 faster thus **172% degradation**



Miss Penalty

- Increasing the clock rate will not solve the problem
 - *doubling the clock rate*, for example, will mean that the miss penalty goes from 100 cycles to 200 cycles for one miss.
 - Lets call performance for double clock rate *Performance.fast.clock*
 - In that case
 - $$\frac{\text{Prfm fastclock w stalls}}{\text{Prfm slowclock w stalls}} = \frac{IC \times CPI_{\text{stall}} \times \text{Clock cycle time}}{IC \times CPI_{\text{stall}} \times \text{Clock cycle time} / 2}$$
- 

Miss Penalty

- $$= \frac{[(2\% + 36\% \times 4\%)100 + 2] \times 2}{(2\% + 36\% \times 4\%)200 + 2} = \frac{10.88}{8.88} = 1.23$$
- Performance improves 23% **not 100%!**
- We need a way to reduce both the miss rate (%) *and* the miss penalty (100-200 cycles)
- Reducing the *miss penalty* is done by doing a multi-level cache - a miss in the primary cache means data is retrieved from secondary cache (faster - less cycles) instead of from RAM.
- Reducing the *miss rate* is dependent on cache architecture (we will see shortly).

Cache Memory

◆ Cache memory

– The level of the memory hierarchy closest to the CPU

◆ Given accesses X_1, \dots, X_{n-1}, X_n

X_4
X_1
X_{n-2}
X_{n-1}
X_2
X_3

a. Before the reference to X_n

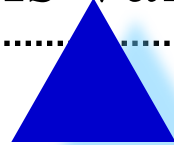
X_4
X_1
X_{n-2}
X_{n-1}
X_2
X_n
X_3

b. After the reference to X_n

- How do we know if the data is present?
- Where do we look?

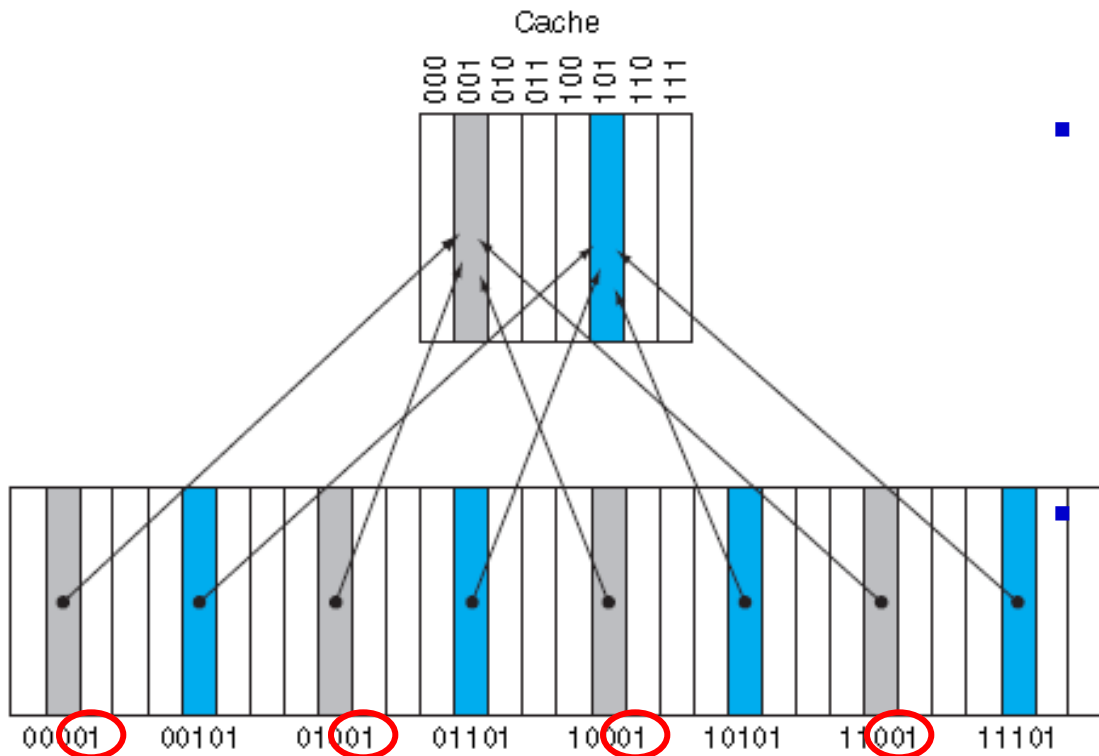
The Simplest Cache



- **Direct-mapped cache** - each memory location (64-bit memory address) maps exactly to *one* cache location
 - But, main memory is much larger than cache, thus *several memory addresses map to one cache location*.
 - Question 1: - How do we know whether the requested data is in the cache or not?
 - Question 2: - How do we know if the data in cache corresponds to the requested word or not?
 - Question 3: - How do we know if the data found in cache is valid?
- 

Cache

- Cache blocks are **indexed**, allowing each block to be addressed when the CPU is looking
- A **tag** identifies which memory location corresponds to that particular block in cache. The tag contains the *upper* portion of the address, while the lower portion is used in the *index*. Bits 0 and 1 are not used.



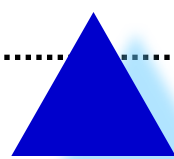
- The first bit in the cache block is a **valid bit** which tells the cache controller if the data in that block are valid or not. If data not present Valid bit is 0

Cache Example



- 8-blocks, 1 word/block, direct mapped
- Initial state is *empty*

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		



Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

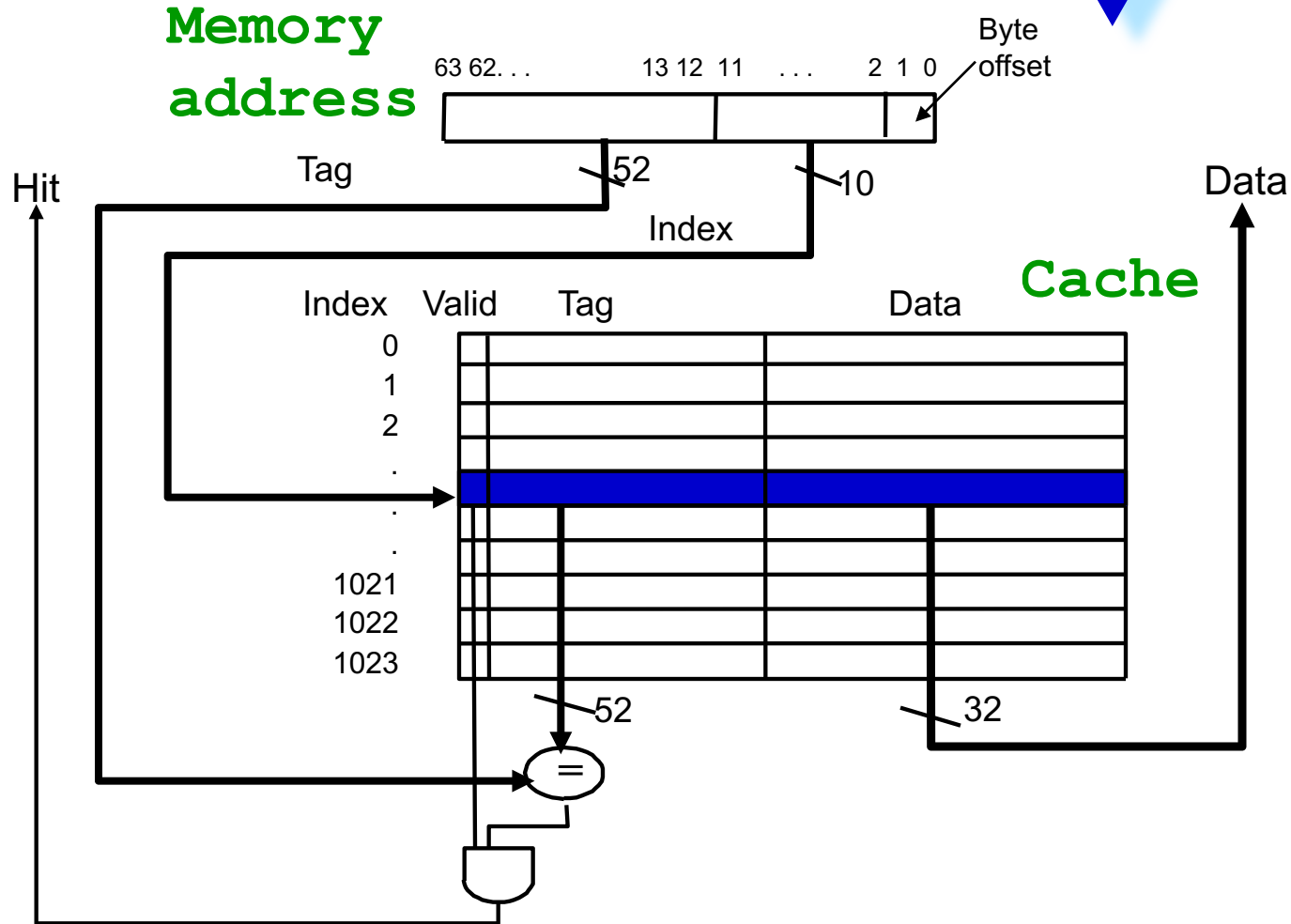
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	10	Mem[10010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

RISC-V Direct Mapped Cache Example



- If there is an instruction miss, Control Block stalls, PC-4 is sent to memory, memory performs read, data is placed in cache in the lower bits index slot, the tag is written with the upper memory address, valid is 1 and instruction is re-fetched

Example:

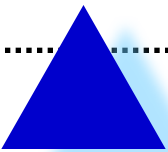
- The series of memory address references as word addresses is **1, 4, 8, 5, 20, 17, 19, 11, 4, 43, 5, 6, 9, 17**. Assume direct-mapped cache with 16 1-word blocks, which is initially empty. Label each reference in the list as Hit or Miss and show the final contents of the cache.

Binary Representation	REFERENCE	HIT/MISS
00 0001	1	Miss
00 0100	4	Miss
00 1000	8	Miss
00 0101	5	Miss
01 0100	20	Miss
01 0001	17	Miss
01 0011	19	Miss
11 1000	56	Miss
00 1001	9	Miss
00 1011	11	Miss
00 0100	4	Miss
10 1011	43	Miss
00 0101	5	Hit
00 0110	6	Miss
00 1001	9	Hit
01 0001	17	Hit

Block	TAG
0000 (0)	
0001 (1)	1 17
0010 (2)	
0011 (3)	19
0100 (4)	4 20 4
0101 (5)	5
0110 (6)	6
0111 (7)	
1000 (8)	8 56
1001 (9)	9
1010 (10)	
1011 (11)	11 43
1100 (12)	
1101 (13)	
1110 (14)	
1111 (15)	

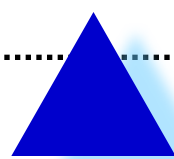


Cache size (direct mapped)

- How large needs the cache be?
 - Each row in the cache is formed of the valid bit + tag bits + data bits
 - Assume that there are n index bits, thus 2^n rows in the cache
 - then the cache size is $2^n \times \text{size of a row}$ or
 - $2^n \times [32 \text{ data} + (64 - n \text{ index} - 2 \text{ byte offset}) + 1]$
cache size = $2^n \times (95 - n)$ bits
- 



Cache size (direct mapped)

- For 64 kB of *data* that needs to be placed in cache which has
 - 1-word blocks, since 1 word = 4 byte, cache needs to hold 16 k words = 16 k blocks. $2^{14} = 16,384$ which means **n** = 14
 - Thus the cache size is $2^n \times (95 - n) = 16,384 \times (95 - 14) = 16,384 \times 81 = 1,327,104$ bits = 156 kB.
- 

32-bit cache with 16-word blocks

