CprE 381: Computer Organization and Assembly Level Programming

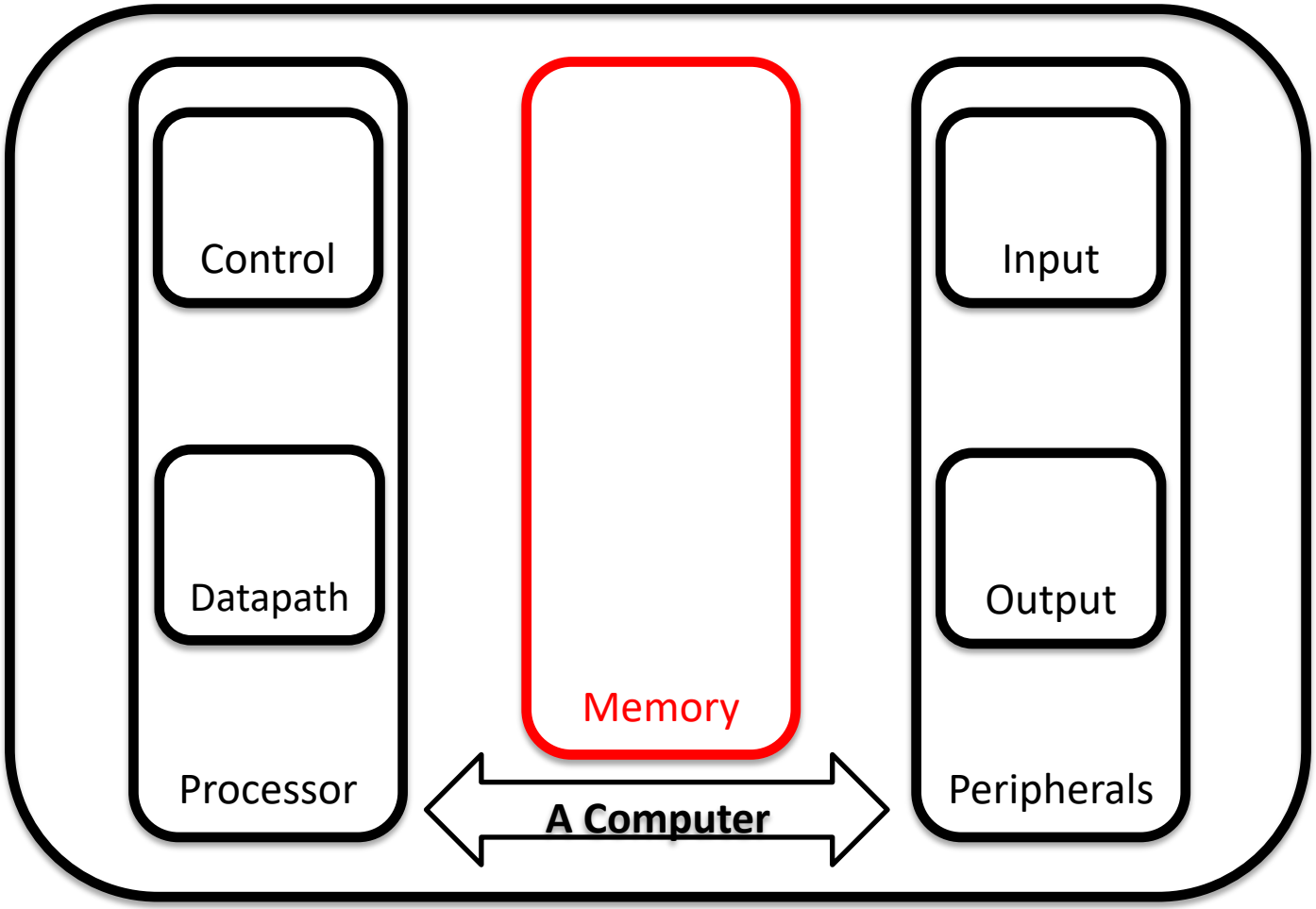
Cache Design

Henry Duwe
Electrical and Computer Engineering
Iowa State University

Administrative

- **WARNING**: Part 3b can be tricky! Get started early!
- HW9 due on Mon April 15

Remember the System View!



Review: Small or Slow

- Unfortunately there is a tradeoff between speed, cost and capacity

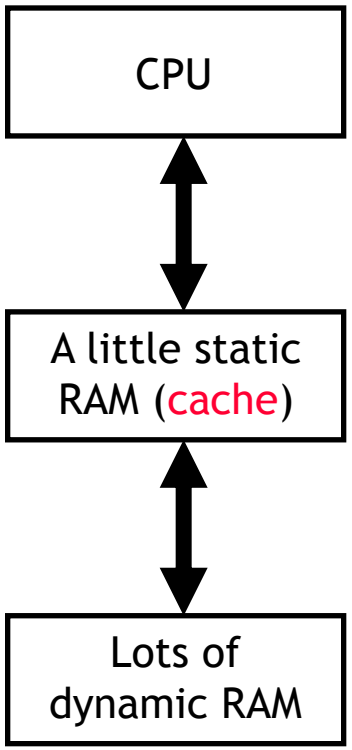
Storage	Speed	Cost	Capacity
Static RAM	Fastest	Expensive	Smallest
Dynamic RAM	Slow	Cheap	Large
Hard disks	Slowest	Cheapest	Largest

- Fast memory is too expensive for most people to buy a lot of
- But dynamic memory has a much longer delay than other functional units in a datapath. If every `lw` or `sw` accessed dynamic memory, we'd have to either increase the cycle time or stall frequently
- Here are ***rough*** estimates of some current storage parameters

Storage	Delay	Cost/MB	Capacity
Static RAM	1-10 cycles	~$1	128KB-128MB
Dynamic RAM	100-200 cycles	~$0.005	256MB-512GB
Hard disks	10,000,000 cycles	~$0.00005	512GB-10TB

Review: Introducing Caches

- Wouldn't it be nice if we could find a balance between fast and cheap memory?
- We do this by introducing a cache, which is a small amount of fast, expensive memory
  - The cache goes between the processor and the slower, dynamic main memory
  - It keeps a copy of the most frequently used data from the main memory
- Memory access speed increases overall, because we've made the common case faster
  - Reads and writes to the most frequently used addresses will be serviced by the cache
  - We only need to access the slower main memory for less frequently used data



Review: The Principle of Locality

- It's usually difficult or impossible to figure out what data will be "most frequently accessed" before a program actually runs, which makes it hard to know what to store into the small, precious cache memory.
- But in practice, most programs exhibit *locality*, which the cache can take advantage of.
  - The principle of temporal locality says that if a program accesses one memory address, there is a good chance that it will access the same address again.
  - The principle of spatial locality says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.

Definitions: Hits and Misses

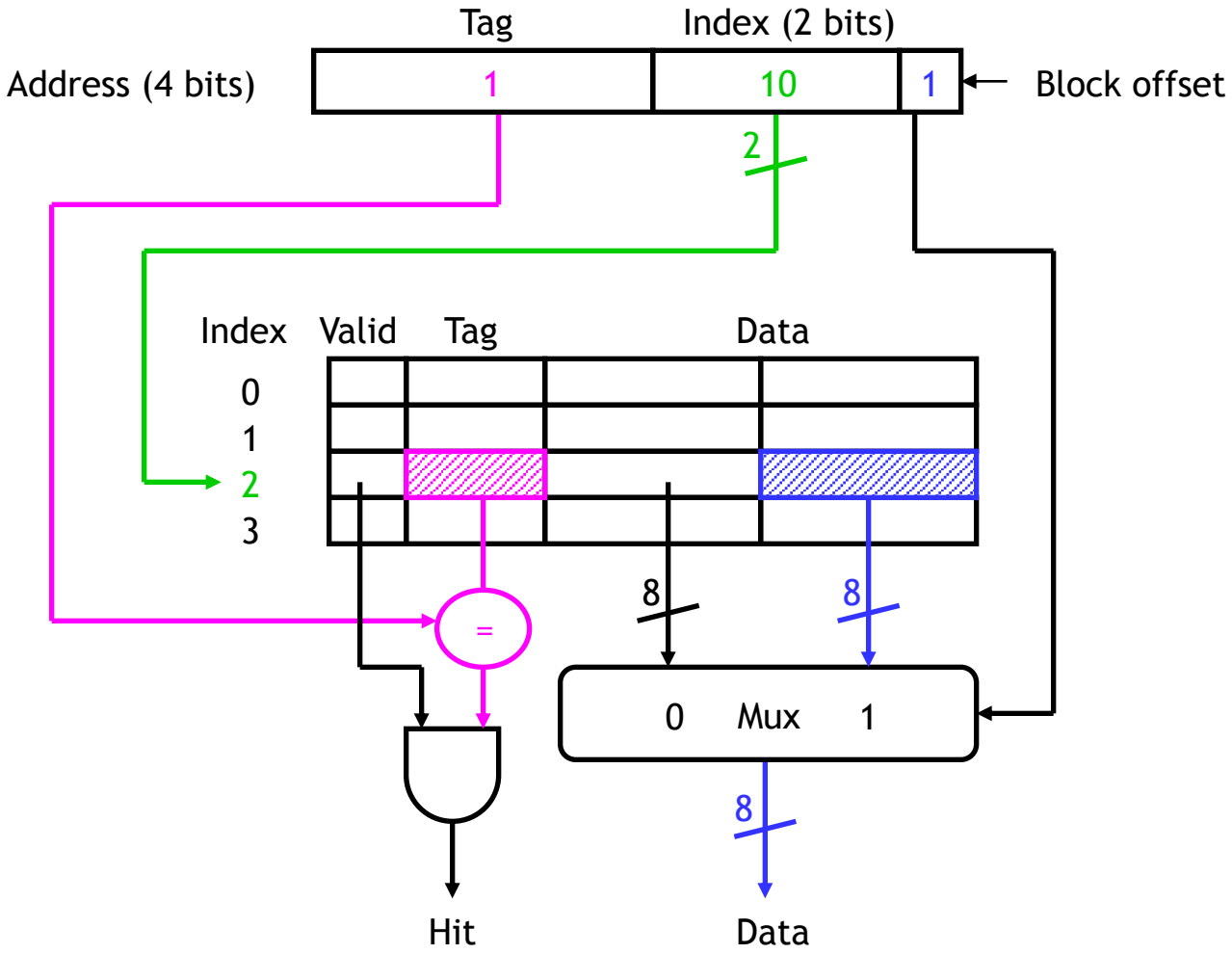
- A cache hit occurs if the cache contains the data that we're looking for. Hits are good, because the cache can return the data much faster than main memory.
- A cache miss occurs if the cache does not contain the requested data. This is bad, since the CPU must then wait for the slower main memory.

- There are two basic measurements of cache performance.
  - The hit rate is the percentage of memory accesses that are handled by the cache.
  - The miss rate (1 – hit rate) is the percentage of accesses that must be handled by the slower main RAM.
- Typical caches have a hit rate of 95% or higher, so in fact most memory accesses will be handled by the cache and will be dramatically faster.

Review: A Direct-Mapped Cache

- Each memory address maps to exactly one location in the cache
- Tag indicates which of the many memory addresses is currently mapped into cache location
- Valid bit indicates if the data stored in the cache entry has been loaded before

Review: Multi-Element Cache Block

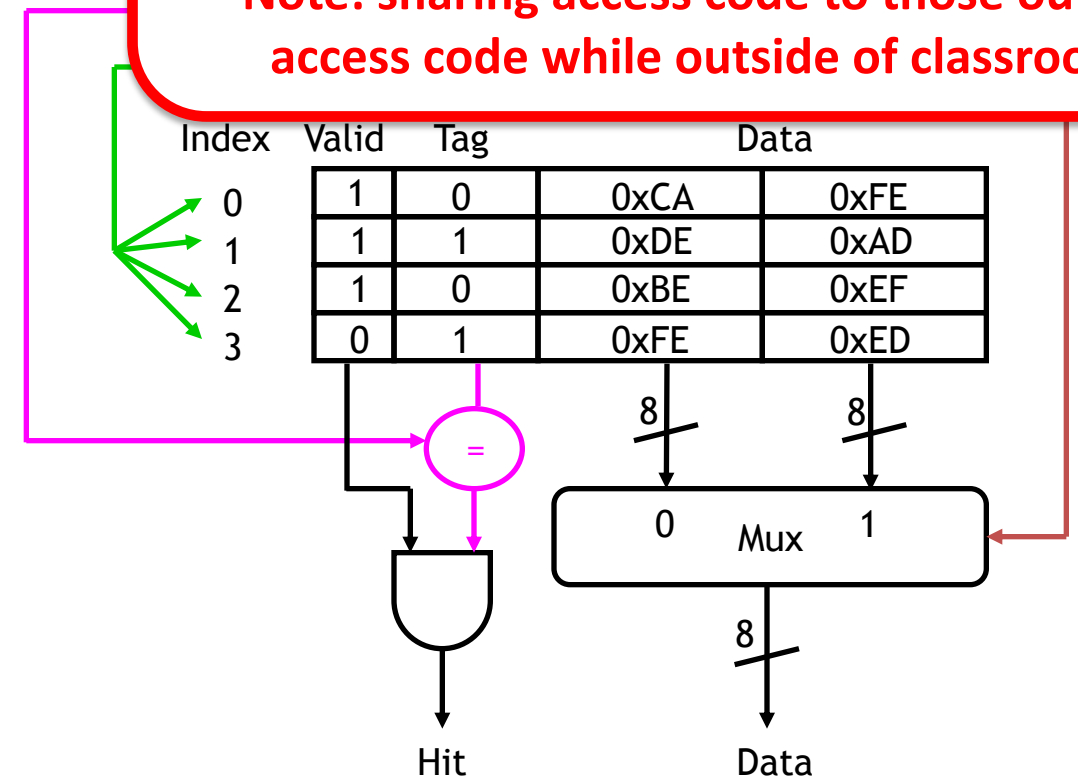


To Hit or not to Hit…

In-class Assessment!

Access Code: De$pth

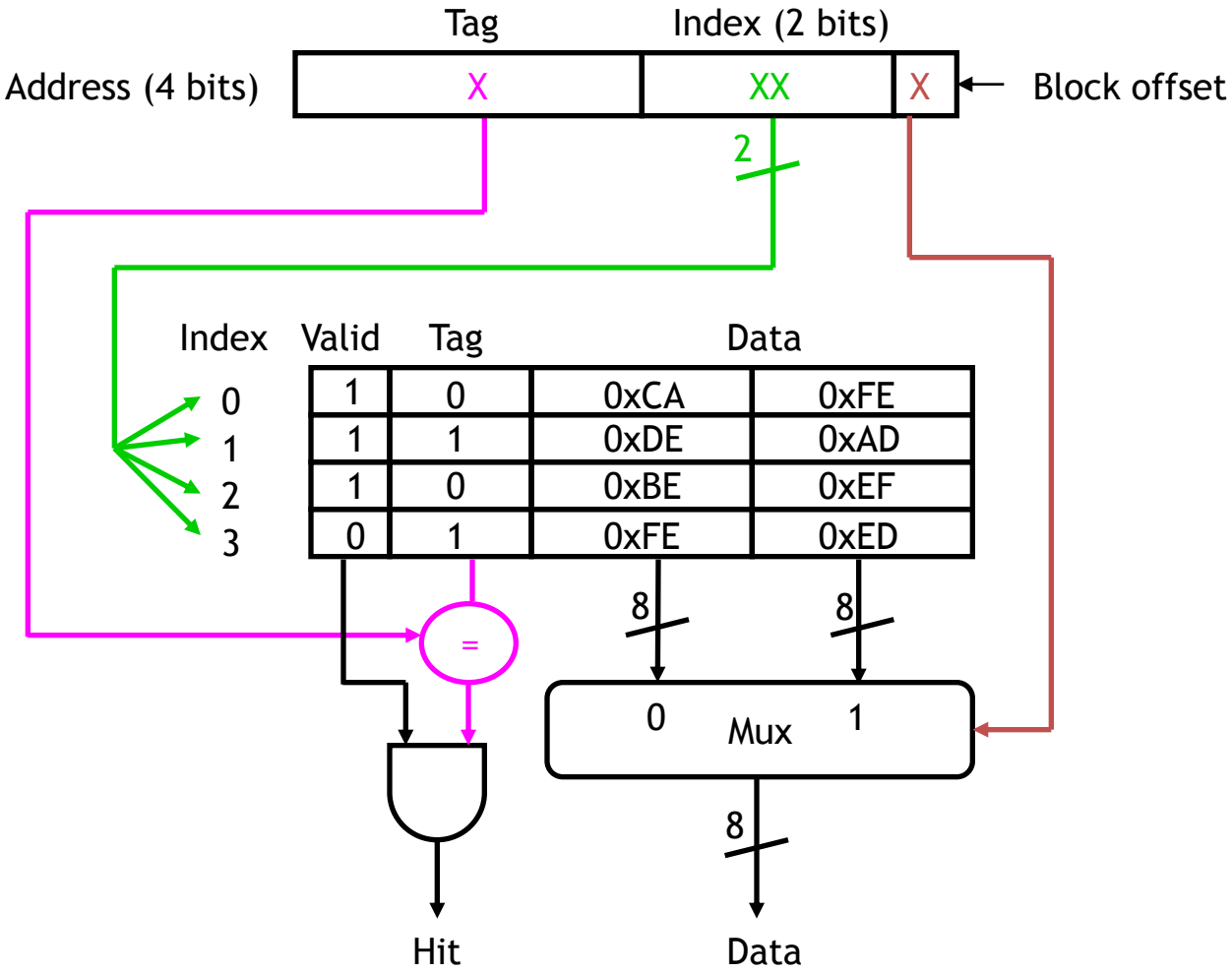
Note: sharing access code to those outside of classroom or using access code while outside of classroom is considered cheating



cache (or is there a miss)?

- 1010
- 1110
- 0001
- 1101

To Hit or not to Hit…

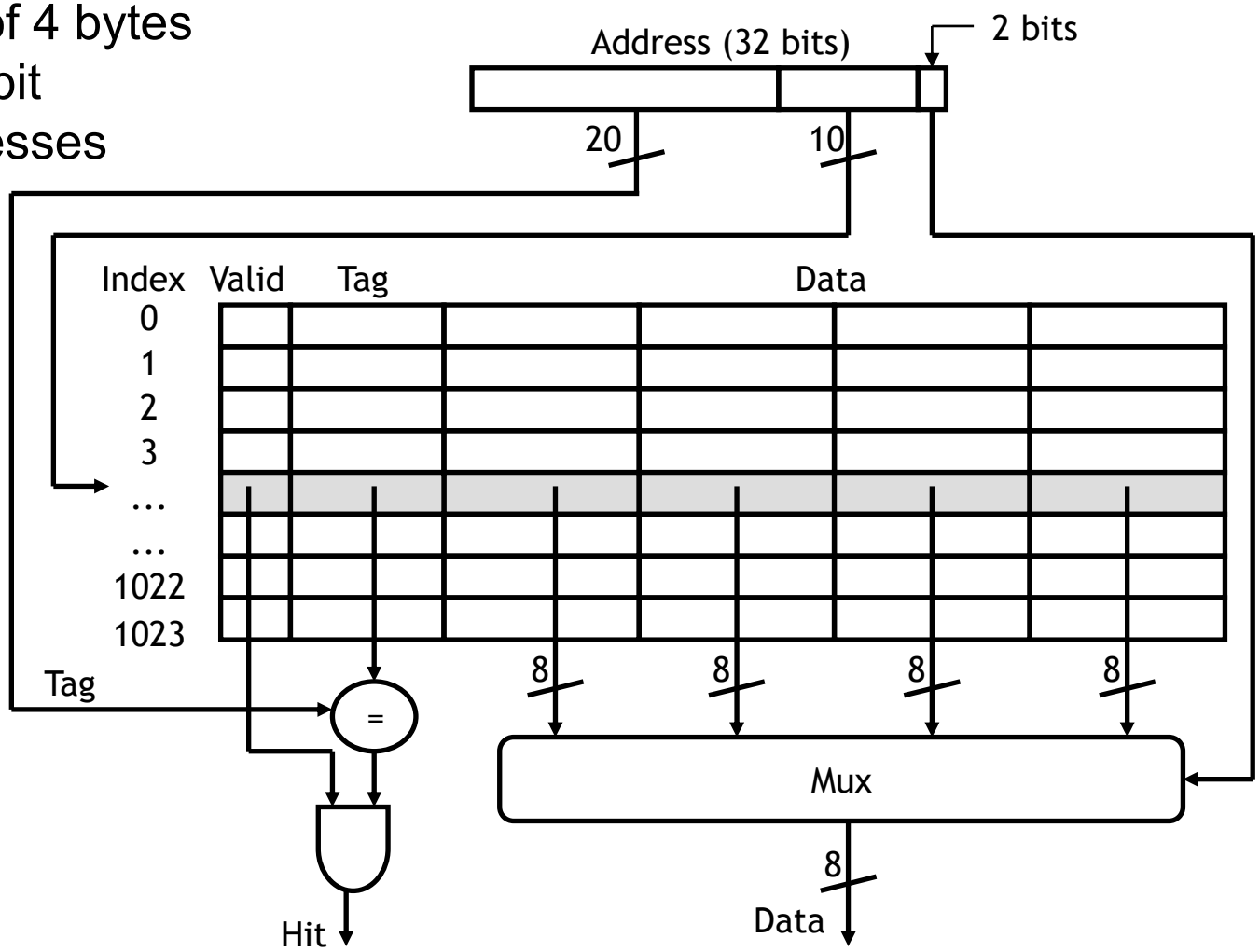


For the addresses below, what byte is read from the cache (or is there a miss)?

- 1010
- 1110
- 0001
- 1101

Larger Example Cache

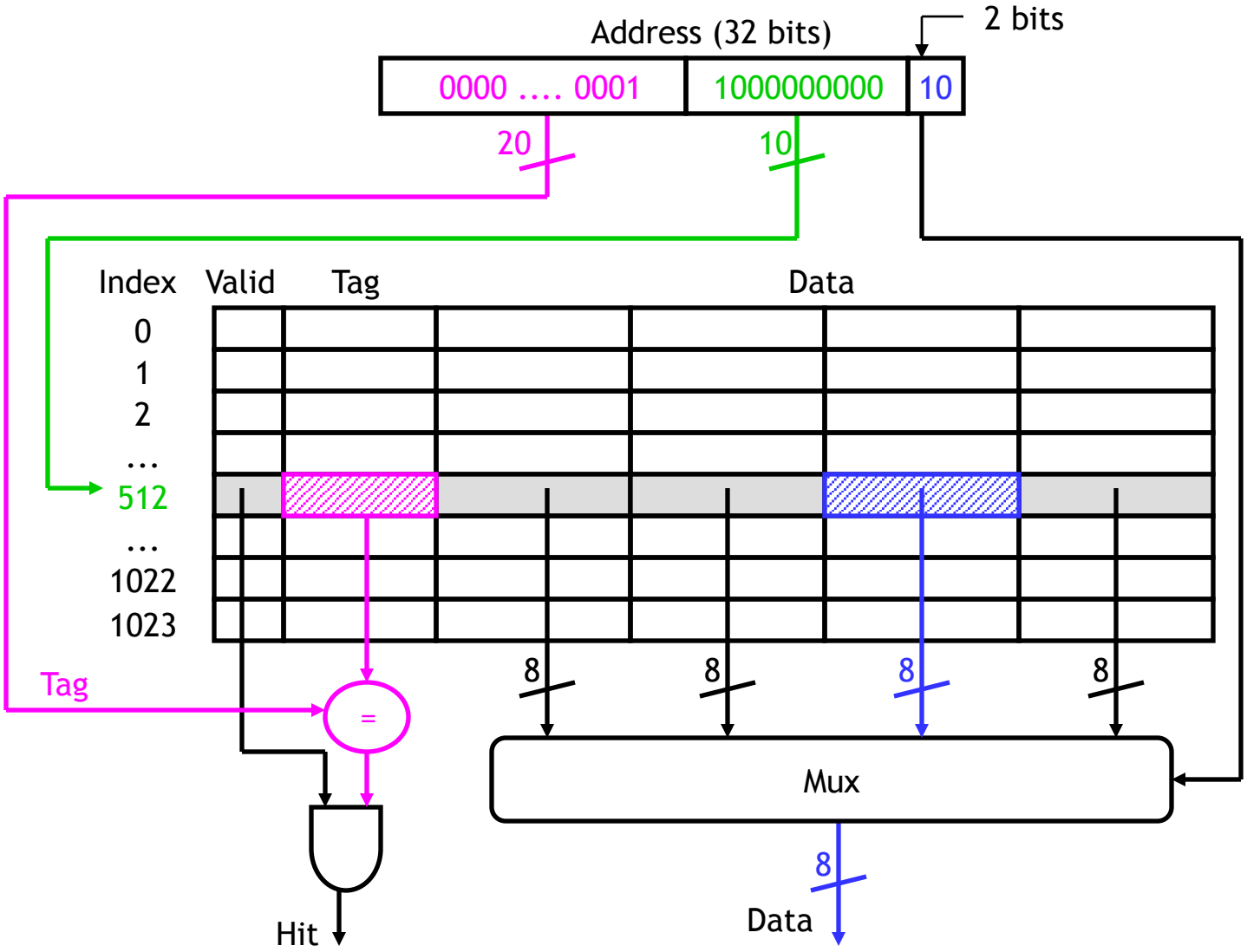
- Here is a cache with 1,024 blocks of 4 bytes each, and 32-bit memory addresses



A Larger Example Cache Mapping

- Where would the byte from memory address 6146 be stored in this direct-mapped 2^{10}-block cache with 2^2-byte blocks?
- We can determine this with the binary force
  - 6146 in binary is 00...01 1000 0000 00 10
  - The lowest 2 bits, 10, mean this is the second byte in its block
  - The next 10 bits, 1000000000, are the block number itself (512)
- Equivalently, you could use your arithmetic mojo instead
  - The block offset is 6146 mod 4, which equals 2
  - The block address is 6146/4 = 1536, so the index is 1536 mod 1024, or 512

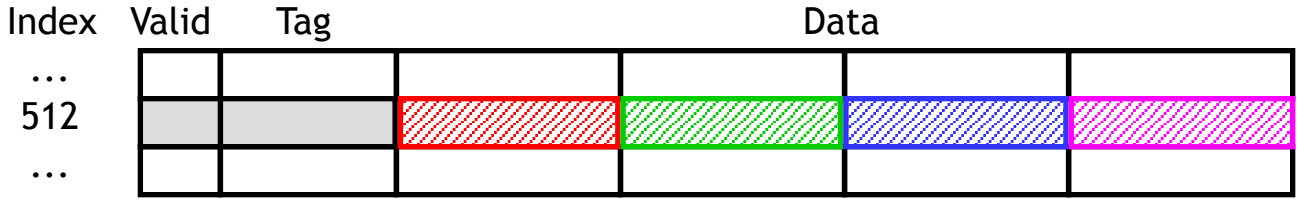
A Larger Example Cache Mapping



The Rest of That Cache Block

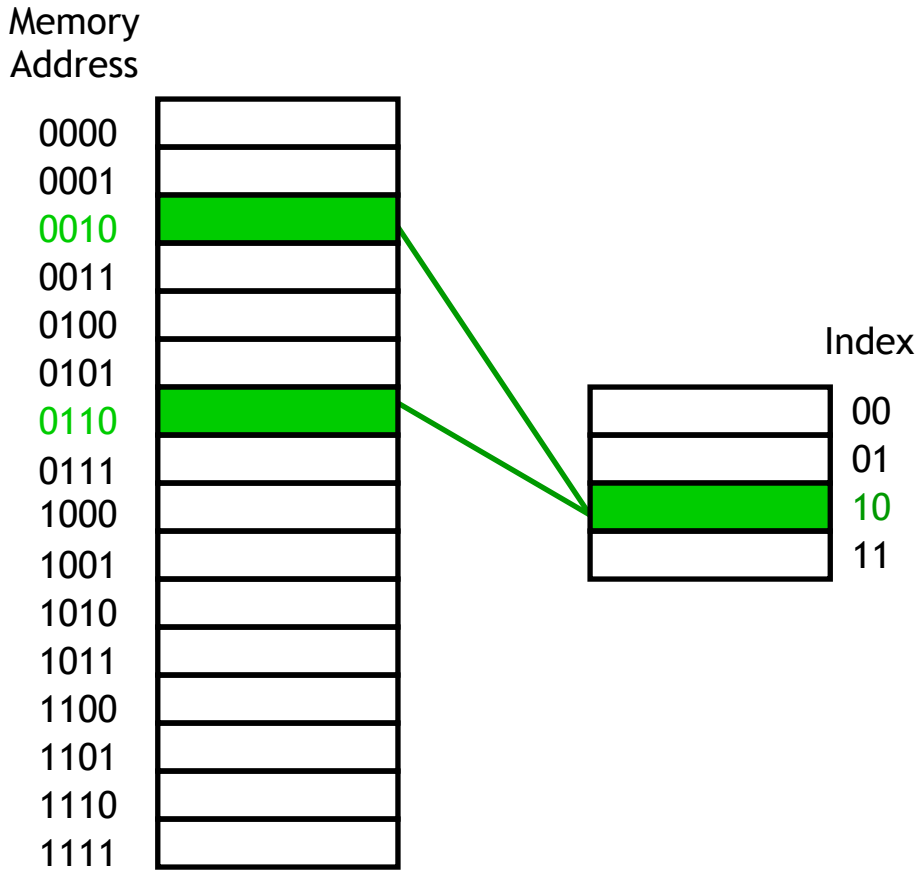
- Again, byte *i* of a memory block is stored into byte *i* of the corresponding cache block
  - In our example, memory block 1536 consists of byte addresses 6144 to 6147. So bytes 0-3 of the cache block would contain data from address 6144, 6145, 6146 and 6147 respectively
  - You can also look at the lowest 2 bits of the memory address to find the block offsets

Block offset	Memory address	Decimal
00	00..01 1000000000 00	6144
01	00..01 1000000000 01	6145
10	00..01 1000000000 10	6146
11	00..01 1000000000 11	6147



Disadvantage of Direct Mapping

- The direct-mapped cache is easy: indices and offsets can be computed with bit operators or simple arithmetic, because each memory address belongs in exactly one block.
- But, what happens if a program uses addresses 2, 6, 2, 6, 2, …?



Disadvantage of Direct Mapping

- But, what happens if a program uses addresses 2, 6, 2, 6, 2, …? Cache Thrashing!

Ech. 0m 020 P.M.

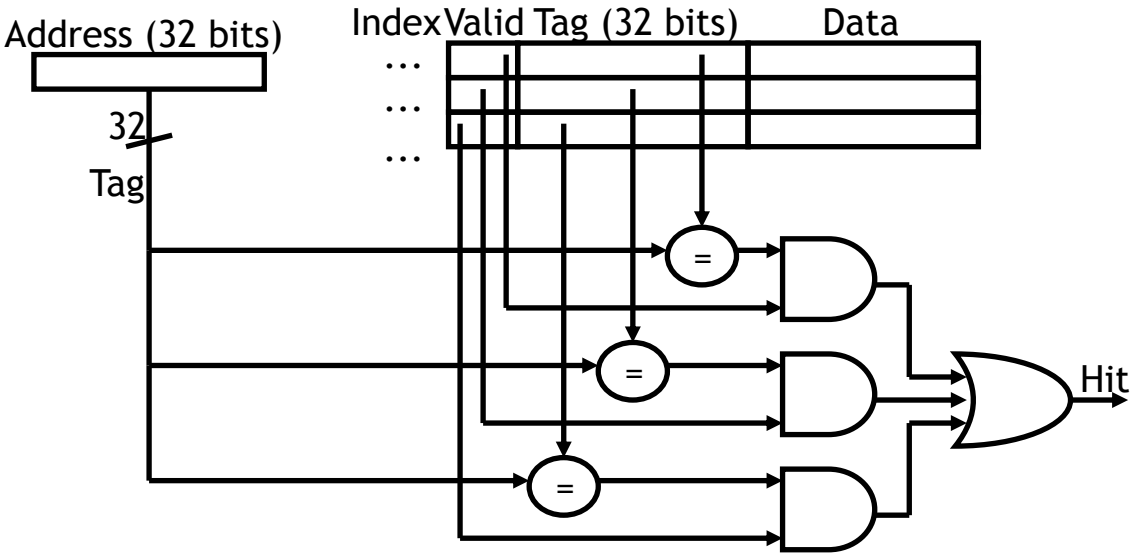
Fig. 349. — *Batteuse Damey à manège direct placé sous la batteuse.*

A Fully-Associative Cache

- A fully-associative cache permits data to be stored in *any* cache block, instead of forcing each memory address into one particular block
  - When data is fetched from memory, it can be placed in *any* unused block of the cache
  - This way we'll never have a conflict between two or more memory addresses that map to a single cache block
- In the previous example, we might put memory address 2 in cache block 2, and address 6 in block 3. Then subsequent repeated accesses to 2 and 6 would all be hits instead of misses
- If all the blocks are already in use, it's usually best to replace the least recently used one, assuming that if it hasn't used it in a while, it won't be needed again anytime soon

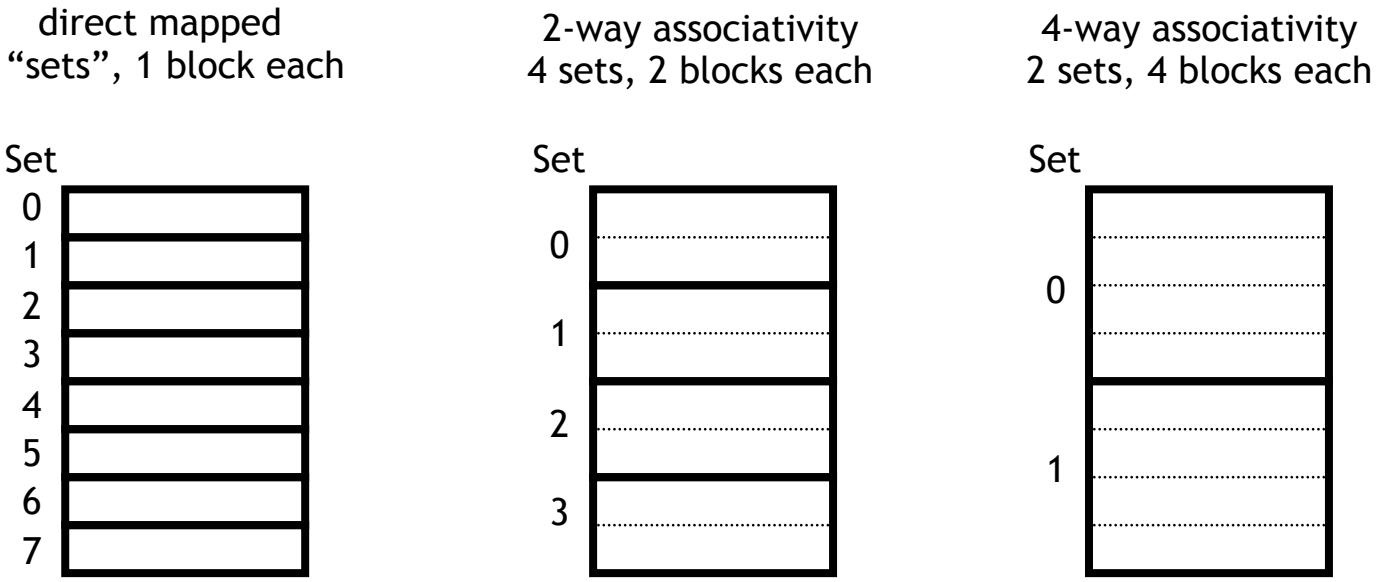
The Price of Full Associativity

- However, a fully-associative cache is expensive to implement
  - Because there is no index field in the address anymore, the *entire* address must be used as the tag, increasing the total cache size
  - Data could be anywhere in the cache, so we must check the tag of *every* cache block. That's a lot of comparators!



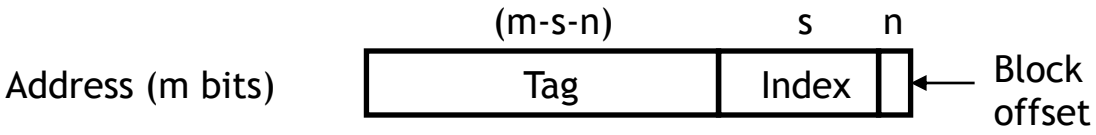
Set Associativity

- An intermediate possibility is a set-associative cache
  - The cache is divided into *groups* of blocks, called sets
  - Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set
- If each set has 2^x blocks, the cache is a 2^x-way associative cache
- Here are several possible organizations of an eight-block cache



Locating a Set Associative Block

- We can determine where a memory address belongs in an associative cache in a similar way as before
- If a cache has 2^s sets and each block has 2^n bytes, the memory address can be partitioned as follows



- Our arithmetic computations now compute a set index, to select a *set* within the cache instead of an individual block

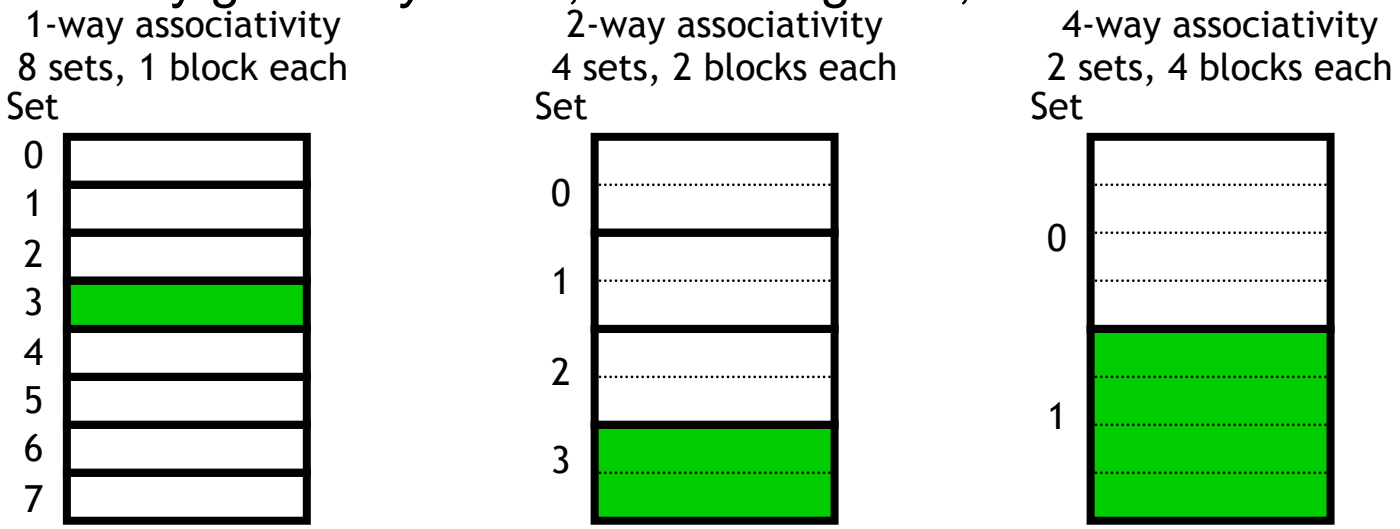
Block Offset = Memory Address mod 2^n

Block Address = Memory Address / 2^n

Set Index = Block Address mod 2^s

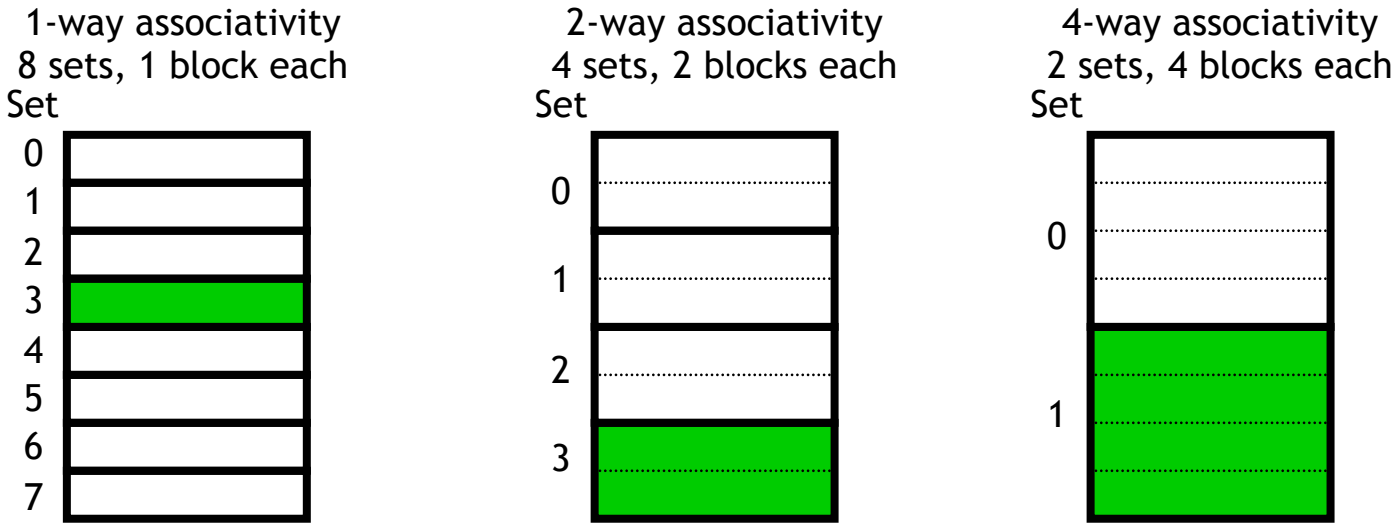
Extra Set-Associative Placement

- Where would data from memory byte address 6195 be placed, assuming the eight-block cache designs below, with 16 bytes per block?
- 6195 in binary is 00...0110000 011 0011.
- Each block has 16 bytes, so the lowest 4 bits are the block offset
  - For the 1-way cache, the next three bits (011) are the set index
  - For the 2-way cache, the next two bits (11) are the set index
  - For the 4-way cache, the next one bit (1) is the set index
- The data may go in *any* block, shown in green, within the correct set



Block Replacement

- Any empty block in the correct set may be used for storing data
- If there are no empty blocks, the cache controller will attempt to replace the least recently used block, just like before
- For highly associative caches, it's expensive to keep track of what's really the least recently used block, so some approximations are used. We won't get into the details…



Preview: Four Important Questions

1. When we copy a block of data from main memory to the cache, where exactly should we put it?
2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?
3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we'd have to replace one of the existing blocks in the cache... which one?
4. How can *write* operations be handled by the memory system?

- Previously we answered the first 3. Now, we consider the 4th.

Acknowledgments

- These slides contain material developed and copyright by:
  - Joe Zambreno (Iowa State)
  - Akhilesh Tyagi (Iowa State)
  - David Patterson (UC Berkeley)
  - Mary Jane Irwin (Penn State)
  - Christos Kozyrakis (Stanford)
  - Onur Mutlu (Carnegie Mellon)
  - Krste Asanović (UC Berkeley)
  - Morgan Kaufmann