Logical Not (operator : !)

- result is 0 or 1

if (!x)

if left is 1, then right is not reached

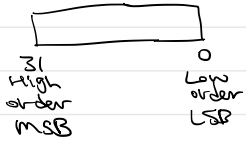
07/31

Masking - Using bitwise operations on integers of various sizes to capture and manipulate bits.

- Bitwise or with a mask pattern to ensure certain bits are turned on (i.e. set to 1)
- Bitwise And with a mask pattern to ensure certain bits are turned off (i.e. set to 0)

ex) capture Low order 23 bits of a 32 bit number

X & 0X007FFFFF



ex) Capture Low order 52 bits of 64 bit number

X & 0X000FFFF FFFF FFFF

*Bitwise and is an identity operator.

Operating on single bits:

- Set bit N=1 in X → X |= (1<<N)
- Set bit N=0 in X → X &= ~(1<<N)
- Toggle bit N in X → X ^= (1<<N)
- Is Bit N in X 1 or 0? → X & (1<<N) → gives zero (false) or nonzero (true)

If bit N was 0, then now 1.
If bit N was 1, then now 0.

ex) 11011001000011

& 00000010000000 ← only one bit is 1

00 0000000000000

Swap values of ints A and B

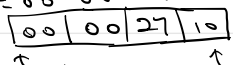
```
temp = A
A = B
B = temp
```

Swap A and B (no temp)

```
A ^= B
B ^= A
A ^= B
```

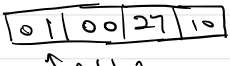
Endianness:

$10001_{10} = 00\ 00\ 27\ 11$



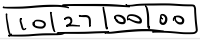
$2710_{16} = 10000_{10}$

$16777216_{10} = 11_{16}$



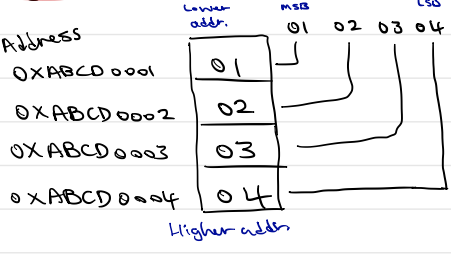
* Big Endian = msb first, decreasing numeric significance as byte address increases

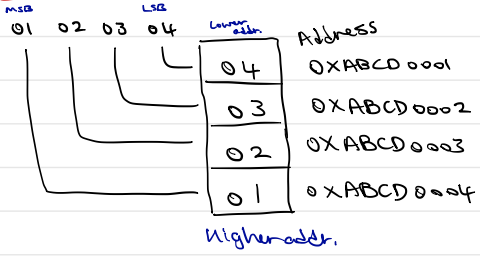
* Little Endian = LSb first, increasing numeric significance as byte address increases

ex) 

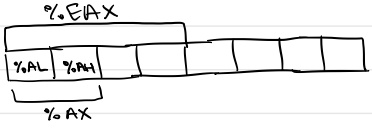
Little Endian 0X2710

Big Endian Ex) 0X01020304 vs Little Endian Ex) 0X01020304





Intel Register (Little Endian)



* Backward compatibility (only in theory, not practical)

%eax occupies the low-order bytes of the %rax register

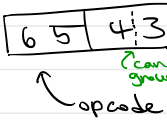
★★ + know sizeof()

C bitfields accessing individual bits or groups of bits

```
Struct BFexample {
    unsigned int F1:10;      <- size in bits
    unsigned int F2:3;
    unsigned int :4;         <- ok to have unnamed bitfield (hardware)
    unsigned int F3:7;
};

Struct BFexample BEX;
BEX F1 = 30;                 <- assign what fits
BEX F2 = 0;
  :
if (BEX.F1 == 20 && BEX.F2 != 4)
```


```
Union {
    unsigned char C;
    Struct RegBite {
        unsigned int RA:4;   <- size in bits
        unsigned int RB:4;
    } RPART
};
```



* Layout of exactly where the bits are implementation dependent.
* Bitfields do not have addresses.

Caching: ★★★

* CPU are faster than memory access
* Faster memory is more expensive.

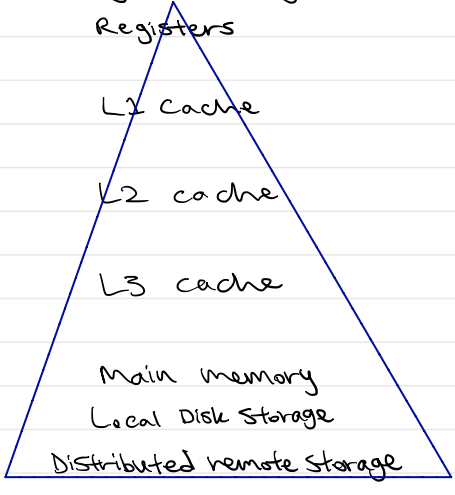
→ Allow faster and cheaper memory cost.

Principles of locality:

1. Spacial locality - If specific memory location was accessed then nearby memory location are more likely to be accessed soon.

2. Temporal locality - If specific memory location was accessed then that same memory location is more likely to be accessed again soon.

Memory Hierarchy:



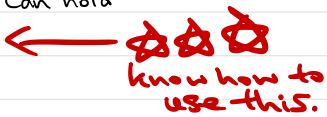
Caching terminology:

- Cache: a small, fast storage device that contains a subset of large and slow storage device
- Block: a fixed-size chunk
- Line: Meta information about one block consisting of tag bits, selector bits, valid bit *
  - what's in the cache
  - boolean

- Set: a collection of one or more lines
- Cache capacity: total number of bytes a cache can hold

Capacity $C = S * E * B$

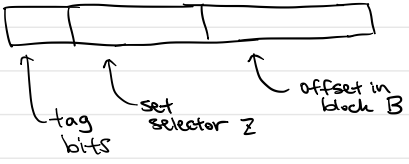
- S: number of sets
- E: number of lines per set
- B: block size in bytes



- Placement Policy: where to put meta information for each block in the cache and where to put the actual data blocks

General Organization of cache:

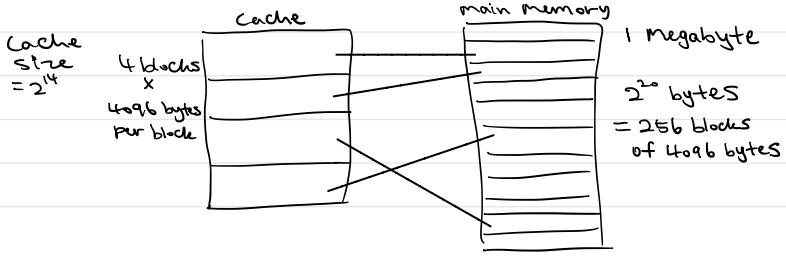
Main memory addresses of m bits are broken into 3 parts



Metainformation for cache block is
- Valid bit
- Tag

Initial state of cache is empty (I.E. valid bit for every block is zero.)

ex) Block size: $4096 = 2^{12}$ => Block offset is 12 bits

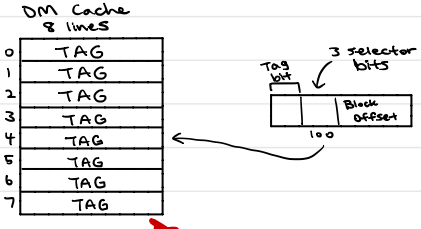


Caching Hits and Misses:

- Hit: Desired block is already in the cache
- Compulsory / Cold Miss: Could not find desired block because cache is empty
- Conflict Miss: Multiple blocks keep mapping to the same line
- Capacity Miss: Number of blocks needed exceed the size of cache

Direct-Mapped Cache:

- One line per set
- Simplest kind of set
- Search cache by mapping selector bits to line of cache.
- z selector bits imply 2^z lines in the direct-mapped cache.
- All Addresses with the same selector bit values go to the same line of cache.



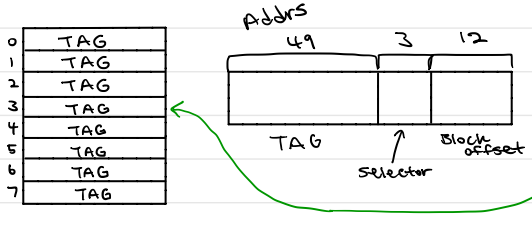
Thrashing = Constantly throwing a block and putting a block in the same block (Swapping in/out Cache lines)

* Tag bit that tells which block to access to
* Selector = what line
* Some lines may not be used even though some get used multiple times

Ex) 64 bit address

4096 byte block

8 blocks in cache



Read Addresses

1234 ABCD 7777 7 3 000 (Block offset)

1234 ABCD 7777 7 3 FFF

All in Same block

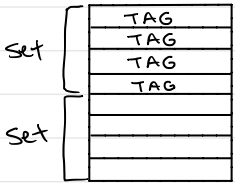
ABCD 1234 7777 7 3 000

FEDCBA90 6666 3 000

Set Associative Cache:

- Main memory addresses broken into tag, selector, block offset bits
- Each set holds more than one line.
- Any line in a set can hold any block that maps to that set.

4 way set Associative



*Hardware is going to do a parallel search.

*Replacement Policy: select victim block to evict { 1. Random Selection, 2. Get rid of least recently used

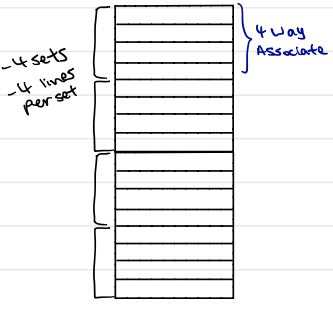
*Better than direct-mapped, b/c less bits needed for selector bits, but way more complicated.

08/02

- Cache tries to find a block that matches an address
- If the requested block is in the cache → "HIT"
- If the requested block is not in the cache → "Miss"
- # of Hits/ #Total requestes = Hit Ratio.

* Goal - to maximize hit ratio.

Set Associative Cache



- Selector bits from address determine set.
- Any line in the set will do.
- Parallel search within a set for matching tag bits.
- Hardware parallel search
- Still some potential underutilization.
- Because selector bits associate to specific sets.

*Direct map cache use underutilization