CS 354 - Machine Organization & Programming
Tuesday, March 7 and Thursday March 9, 2023

Print paper copies of this outline for best use.

Week 07 Activity: Heap Practice Assignment

Project p3: DUE on or before Friday March 25

Homework 3: DUE on or before Friday March 11

Last Week

Placement Policies (finish) Free Block - Too Large/Too Small Coalescing Free Blocks Free Block Footers	Explicit Free list Explicit Free List Improvements --- Heap Caveats Memory Hierarchy

This Week: MEMORY MANAGEMENT via CACHING blocks of memory for fast access

Heap Caveats Memory Hierarchy Locality & Caching Bad Locality Caching: Basic Idea & Terms Designing a Cache: Blocks Rethinking Addressing	Designing a Cache: Sets and Tags Basic Cache Lines Basic Cache Operation Basic Cache Practice
Next Week after Spring Break: Vary cache set size and Cache Writes B&O 6.4.3 Set Associative Caches 6.4.4 Fully Associative Caches 6.4.5 Issues with Writes 6.4.6 Anatomy of a Real Cache Hierarchy 6.4.7 Performance Impact of Cache Parameters	

DO NOT TAKE THIS WEEK OFF if you plan to take Spring Break off
Note: p4A and p4B will be released on Monday after Spring Break

Get p3 done this week. It is possible and will make for an actual Spring Break for you!
p3 - implement and test alloc (partA) and free (partB) by Monday and submit progress
p3 - implement coalesce by Wednesday and submit progress
p3 - complete testing and debugging by Friday and complete final submission

Heap Caveats

Consecutive heap allocations don't result in contiguous payloads!

→ Why?
- Payloads are interspersed with heap structure and possibly padding.
- Placement policies & heap structure can scatter allocations throughout heap.

Don't assume heap memory is initialized to 0!

OS initially clears heap pages for security,
but your recycled heap memory will have your old data
unless you use calloc()

Do free all heap memory that your program allocates!

→ Why are memory leaks bad?
They slowly kill your program's performance by cluttering heap with garbage blocks.
Bad leaks could ultimately consume your entire heap!

→ Do memory leaks persist when a program ends?
No, heap pages are returned to the OS for other uses.

Don't free heap memory more than once!

→ What is the best way to avoid this mistake?NULL freed pointers.

Don't read/write data in freed heap blocks!

→ What kind of error will result? Intermittent error

Don't change heap memory outside of your payload!

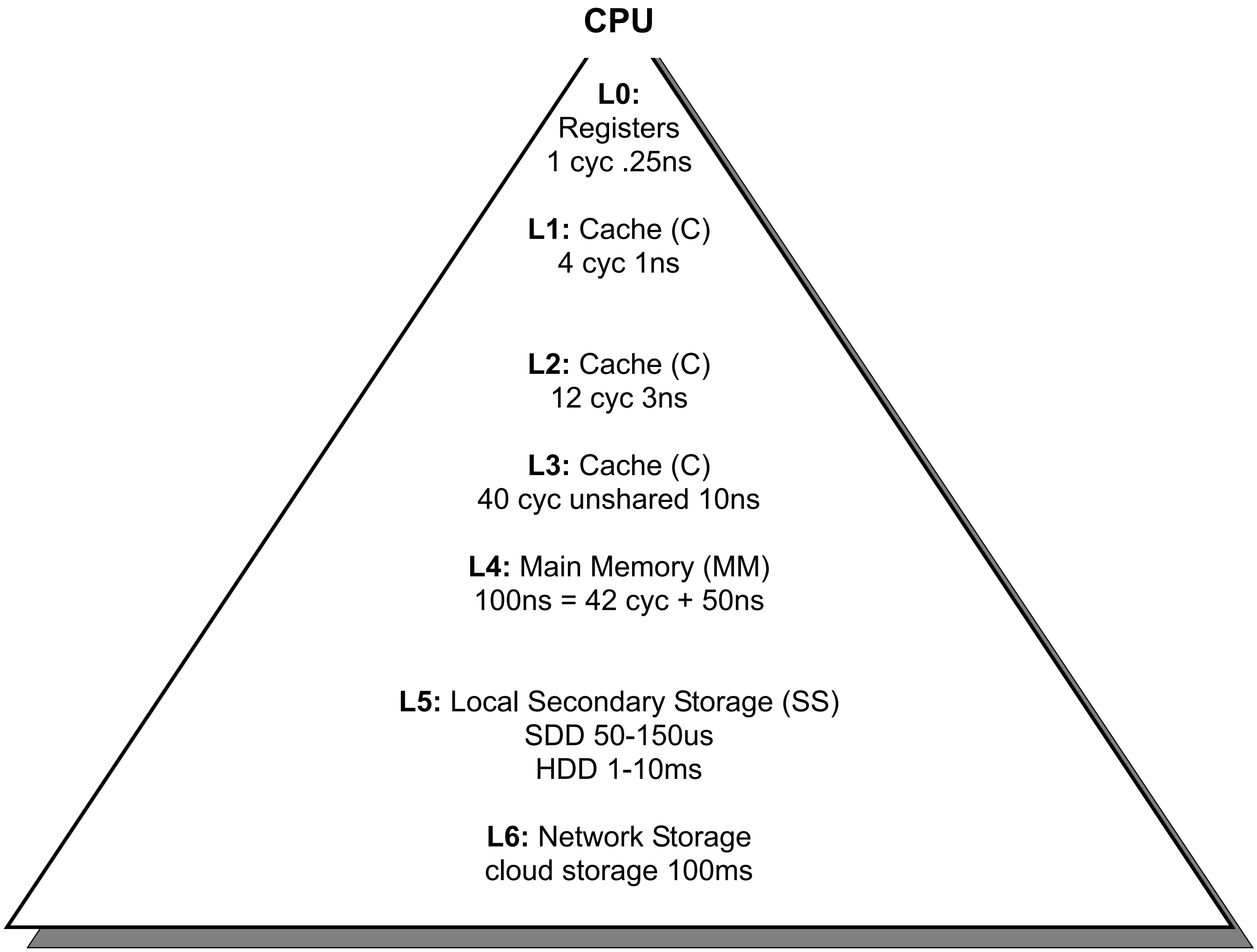
→ Why? You'll trash the heap's internal structure and/or another block's payload.

Do check if your memory intensive program has run out of heap memory!

→ How? Check that allocator's return value is not NULL

Memory Hierarchy

✵ *The memory hierarchygives the illusion of having lots of fast memory.*



<u>Cache</u>

is a smaller faster mem that acts as a staging area for data stored in a larger slower mem

Memory Units

<u>word</u>: size used by CPU transfer betweenL1 & CPU

<u>block</u>: size used by C transfer betweenC levels & MM

<u>page</u>: size used by MM transfer betweenMM & SS

Memory Transfer Time: https://simple.wikipedia.org/wiki/Orders_of_magnitude_(time)

<u>cpu cycles</u>:**used to measure time**

<u>latency</u>:**memory access time (delay)**

Locality & Caching

What?

temporal locality: when a recently accessed memory location is repeatedly accessed in the near future

int i = 0; i< size; i++

spatial locality: when a recently accessed memory location is followed by nearby memory locations being accessed in the near future

locality is designed into Hardware, OS, apps

Example

```
int sumArray(int a[], int size, int step) {
   int sum = 0;
   for (int i = 0; i < size; i += step)
      sum += a[i];
   return sum;
}
```

→ List the variables that clearly demonstrate temporal locality. i, sum, size, step

→ List the variables that clearly demonstrate spatial locality. a (if step is small)

stride: unit in word between sequential accesses

good spatial locality is ~1 word per stride

❋ *The caching system uses locality to predict what the cpu will need in the near future.*

both types

How? The caching system anticipates...

temporal: anticipates data will be reused so it copies and saves in cache

spatial: anticipates nearby data will be used so it copies a BLOCK to the cache

cache block: unit of memory transferred between main memory and cache levels

❋ *Programs with good locality run faster since they work better with the caching system!*

Why? Programs with good locality maximize use of memory at top of memory hierarchy

Bad Locality

Why is this code bad?

3 ?

```
int a[ROWS][COLS];

for (int c = 0; c < COLS; c++)
  for (int r = 0; r < ROWS; r++)
    a[r][c] = r * c;
```

col = 0	1	2		0	1	2		0	1	2	
row 0				**row 1**				**row 2**			
0			...	0			...	0			...
a 1	4			2	5			3	6		

STRIDE

→ How would you improve the code to reduce stride?
rows first then columns, flip for loops

Key Questions for Determining Spatial Locality:

1. What does the memory layout look like for the data?
2. What is the stride of the code across the data? number of columns, smaller stride is better

Why is this code bad?

row 0			row 1		
Col. 0	col 1	col 2	col 0	col 1	col 2
RGBHSL	RGBHSL	RGBHSL	RGBHSL	RGBHSL	RGBHSL
1 2	5 6	9 10	3 4	7 8	11 12

image 13 14

```
struct {
  float rgb[3];
  float hsl[3];
} image[HEIGHT][WIDTH];
```

outer
middle
inner

```
for (int v = 0; v < 3; v++)          v = 0, 1
  for (int c = 0; c < WIDTH; c++)    c = 0, 1, 2, 0
    for (int r = 0; r < HEIGHT; r++) {   r = 0, 1, 0, 1, 0, 1, 0
      image[r][c].rgb[v] = 0;
      image[r][c].hsl[v] = 0;
    }
```

➢ How would you improve the code to reduce stride?

row ---
col ---
v - rgh
v - hsl

Good or bad locality?

- Instruction Flow:
  - sequencing? good spatial locality
  - (if/switch stmts) selection? bad s and t locality
  - (loops) repetition? good spatial if data is contiguous and stride is small, good temporal for loop counters, sentinal values, step index
- Searching Algorithms:
  - linear search good spatial, temporal for match varlue
    array
    linked list bad S
  - binary search array bad spatial

Caching: Basic Idea & Terms

Assume: Memory is divided into 32 byte blocks and all blocks are already in main memory.
Cache L1 has 4 locations to store blocks and L2 has 16 locations to store blocks.

→ Update the memory hierarchy below given blocks are accessed in this sequence:

W = 22,11,22,44,11,33,11,22,55,27,44
mccpp H H H H xcrrv H

(M) *cache miss* - Must fetch
When block is not in the cache

(C) cold miss
when loc is available

(X) capacity miss
when cache is too small for working set

(F) conflict miss
hen two or more blocks map to same location

(H) *cache hit*
When block is in the cache so data can me accesed

placement policies

P1 1. Unrestrictive

P2 2. Restrictive
block num % 16

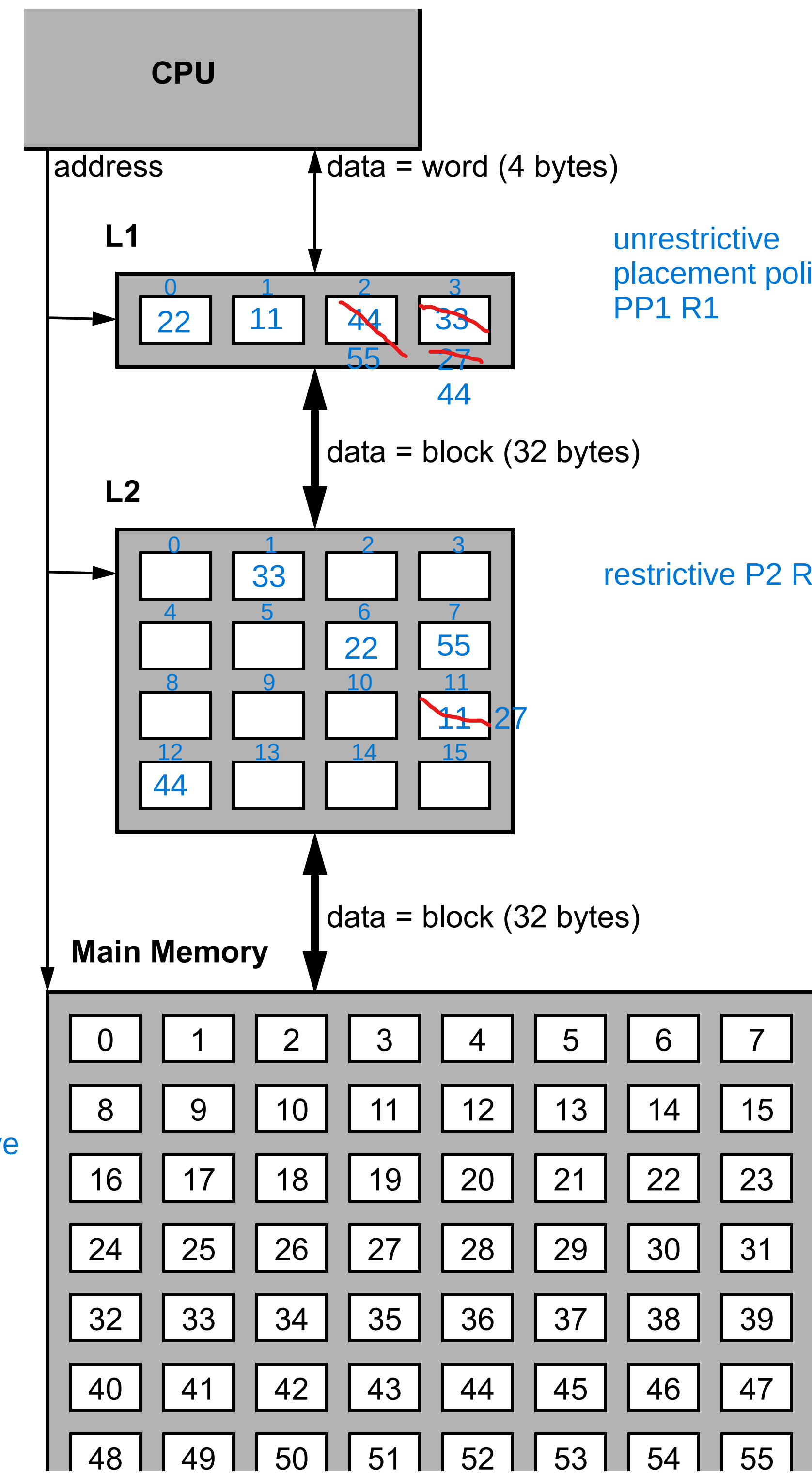
replacement policies

(R) 1. Choose any location

(R2) 2. No choice, because of restrictive placement policy

(V) *victim block*
Cache block chosen for eviction

(W) *working set*
The set of blocks accessed during some interval of time



unrestrictive placement policy, PP1 R1

#%16

restrictive P2 R2

Designing a Cache: Blocks

❋ *The bits of an address* are used to see if block containing address is in cache

IA-32 32 bits in address

How many bytes in an address space?

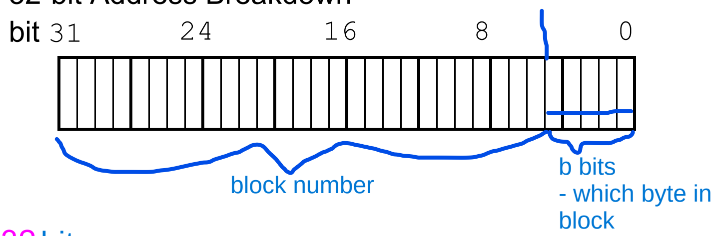
Let M be number of bytes in AS, IA-32 is 4GB

$M = 2^m$ = 4 GB

$m = \log_2 M$ = log2(4GB) = 32 bits

Thus m is number of bits in an address, IA-32 is 32 bits

32-bit Address Breakdown



How big is a block? # bytes

❋ *Cache blocks must be big enough* to capture spatial locality
but small enough to minimize latency

Let B be number of bytes per block, IA-32 is 32 bytes

$B = 2^b$ = 32 bytes

$b = \log_2 B$ = log2(32) = 5 bits

b bits: # of address bits needed to determine which byte in blocks

"*word offset*" identify which work in blocks 4 bytes/word 3 bits

"*byte offset*" identifies which byte in the word 2 bits

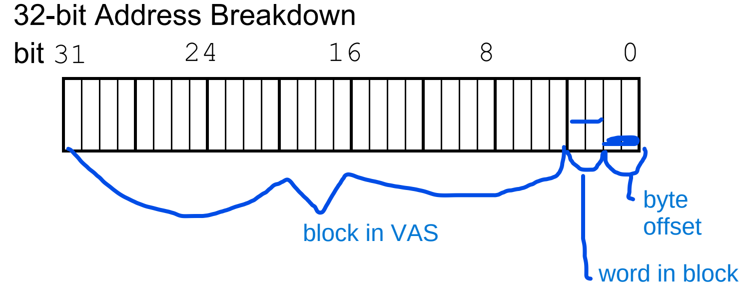
➢ What is the problem with using the most significant bits (left side) for the b bits?
Lose spatial locality, using least significant bits ensures contiguous block of addresses

How many 32-byte blocks of memory in a 32-bit address space?
address space / block size = 2^32 / 2 ^ 5 = 2^27 = 2^7 * 2^20 = 128 Mb

recall: K = 2^10 M = 2^20 G = 2^30

❋ *The remaining bits of an address encode the block number.*

- *An address identifies* *which byte in the VAS to access.*
- *An address is* divided into parts to access memory into steps

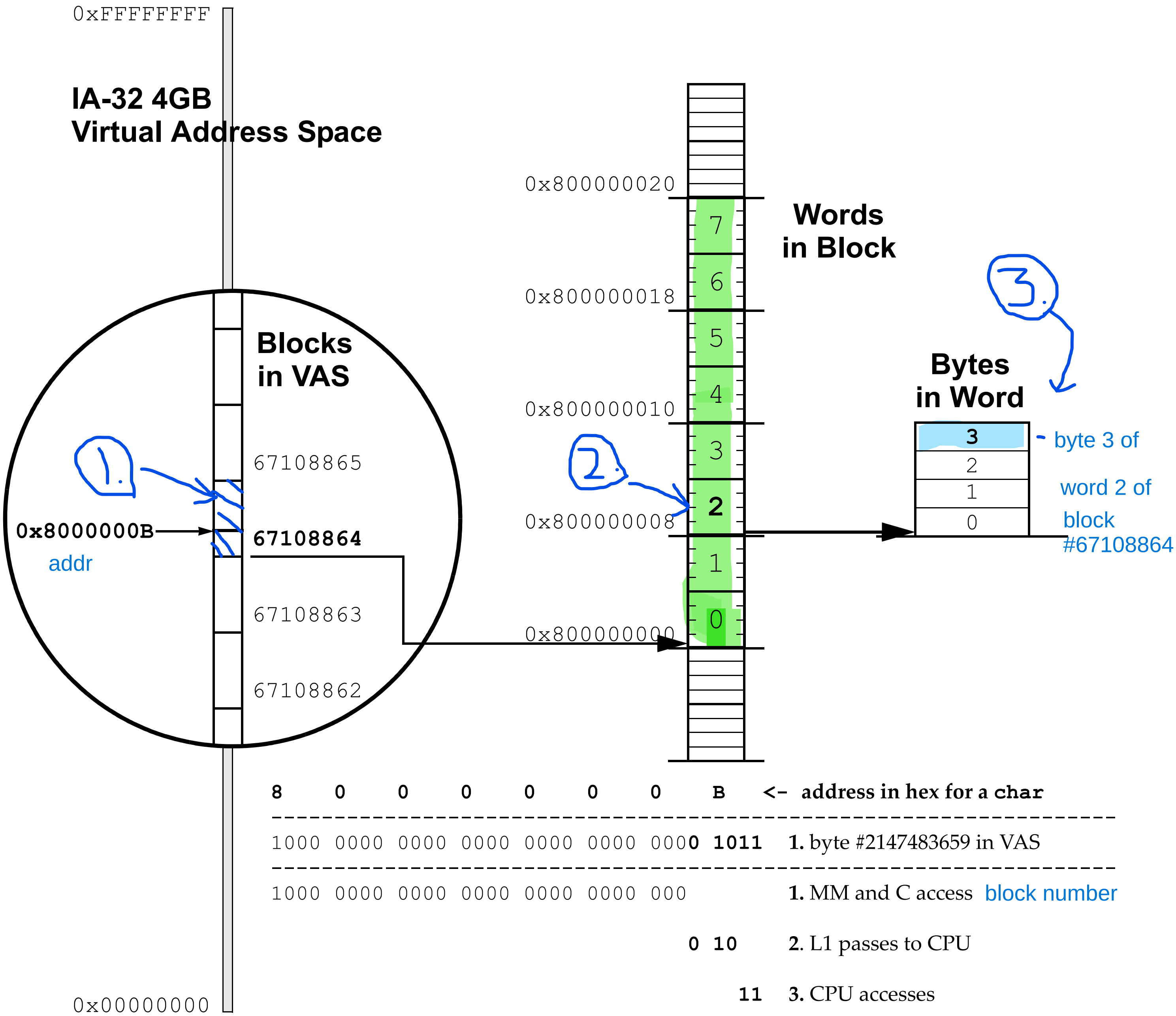


Memory Access in Caching System

step 1. Identify which block in VAS

step 2. Identify which word in the block

step 3. Identify which byte in the word




```
8    0    0    0    0    0    0    B    <-  address in hex for a char
-----------------------------------------------------------------------
1000 0000 0000 0000 0000 0000 0000 1011   1. byte #2147483659 in VAS
-----------------------------------------------------------------------
1000 0000 0000 0000 0000 0000 000         1. MM and C access block number
                                   0 10   2. L1 passes to CPU
                                     11   3. CPU accesses
```

Designing a Cache: Sets & Tags

✵ *A cache must be searched* if unrestrictive placement policy is used

→ Problem? SLOW O(N) where N is number of blocks in cache

Improvement? Limit or restrict location for each block

"*set*:" where block can be found

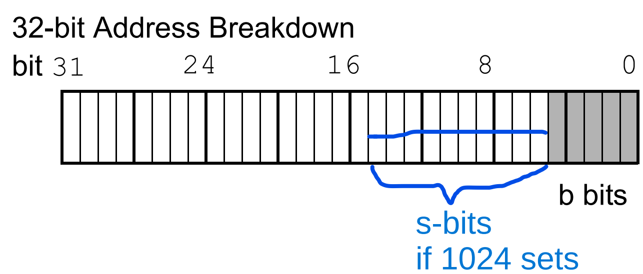
✵ *The block number bits of an address* are divided into two parts

1. maps block to specific set
2. a unique label or "tag" for the block

How many sets in the cache?

Let S bethe number of sets in cache

$S = 2^s$ = 1024 sets in cache
$s = \log_2 S$ = log2(1024) = 10



s bits: identify which set a block belongs to

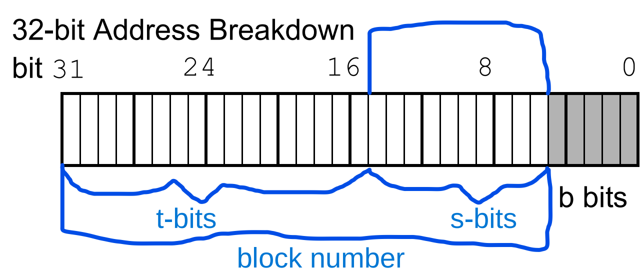
➢ What is the problem with using the most significant bits (left side) for the s bits?
lose spatial locality
choose bits next to b-bits to ensure neighboring blocks go into neighboring sets

→ How many blocks map to each set for a 32-bit AS and a cache with 1024 sets? 8192 sets?

address space / # sets = 2^(32 - 5) / 2^10 = 2^17 = 128 k blocks mapping to same set

Since different blocks map to the same set how do we know which block is in a set?

use remaining bits as a unique tag "label"



t bits: bits of addr that identify block

✵ *When a block is copied into a cache its t bits are also stored as its tag*

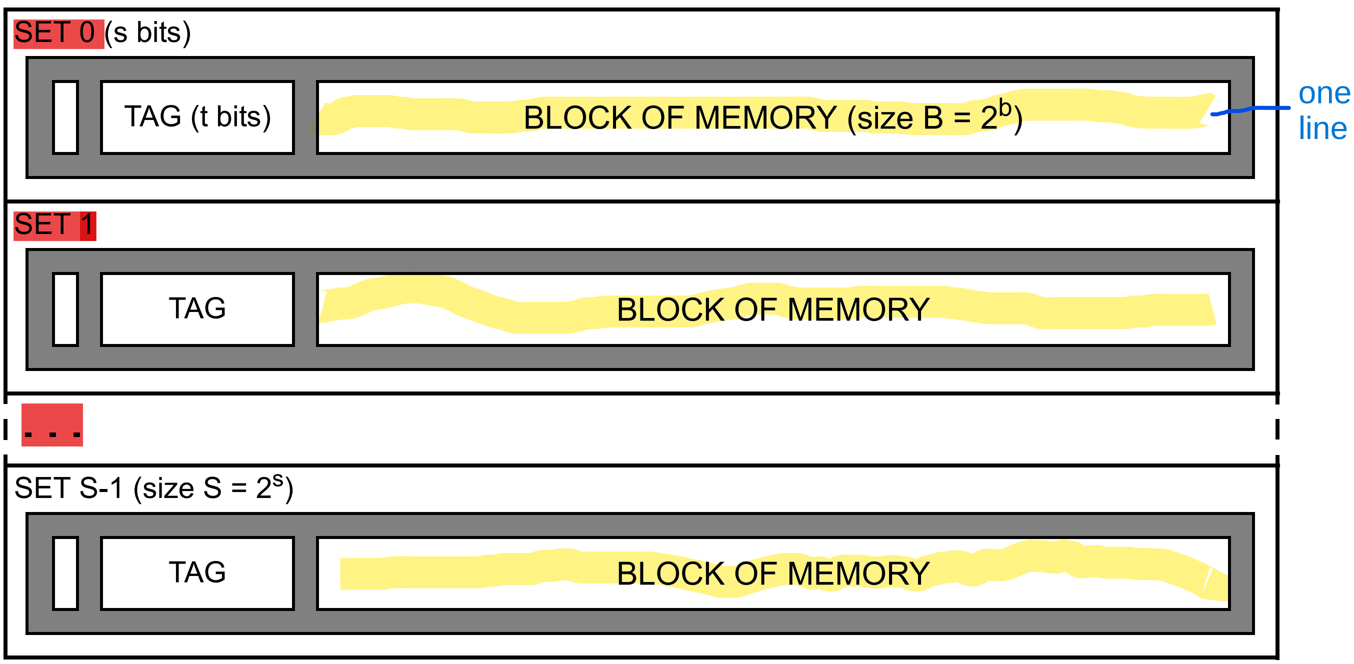
Basic Cache Lines

What? A *line* is

- a location in a cache that stores one block of memory
- composed of storage bits for block and info needed by cache

✵ *In our basic cache each cache set* has only one line

Basic Cache Diagram



→ How do you know if a line in the cache is used or not?

use a status bit
v-bit: 1 = valid 0 = not valid

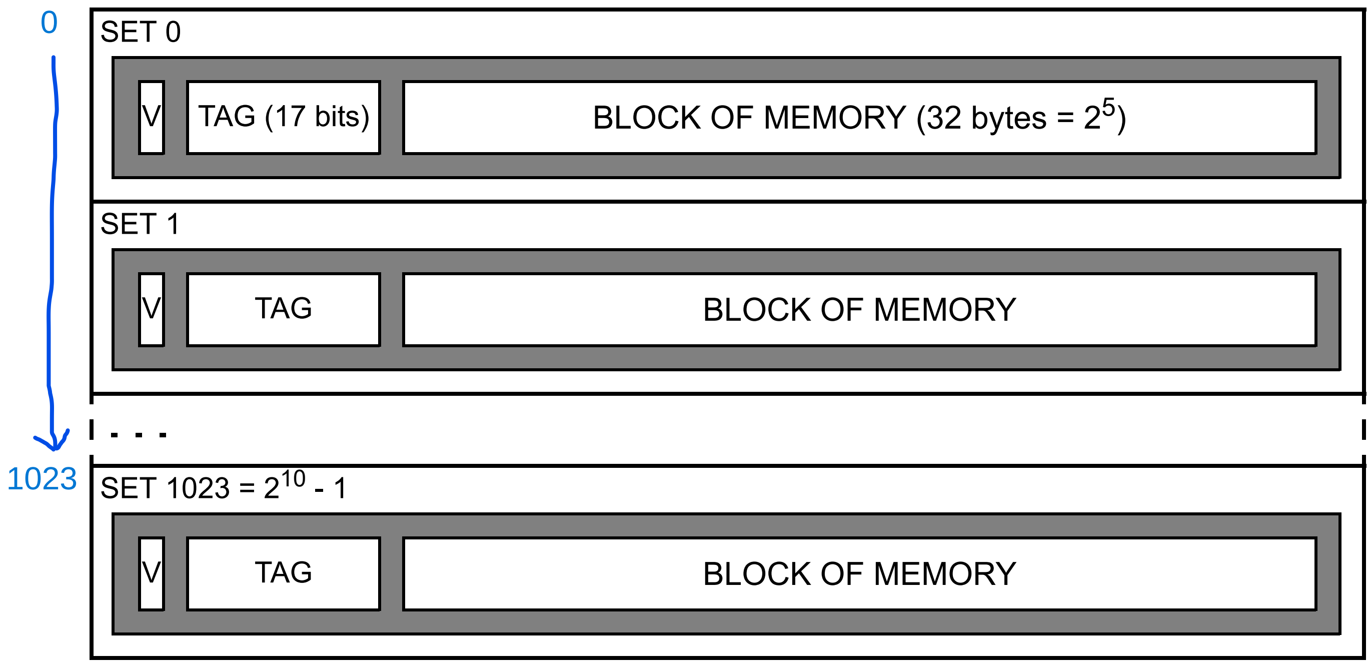
→ How big is a basic cache given S sets with blocks having B bytes?

cache size = C = S x B = # sets times number of bytes per set

$$\# S \times \frac{\#\text{bytes}}{\text{Set}} = \underline{\hspace{2cm}} \text{ bytes}$$

Basic Cache Operation

Basic Cache Diagram

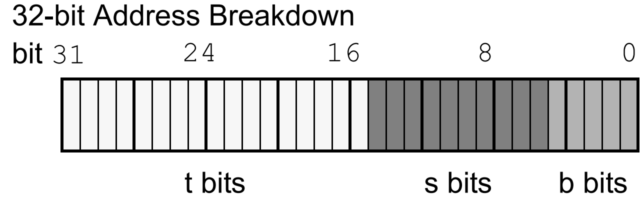


→ How big is this basic cache?

C = S x B = 1024 x 32 = 2^10 x 2^5 = 32 k

How does a cache process a request for a word at a particular address?

32-bit Address Breakdown



1. *Set Selection*

   extract S bits
   use as index

2. *Line Matching*

   extract t bits
   compare t bits with stored tag

   if no match or valid bit is 0, cache MISS!

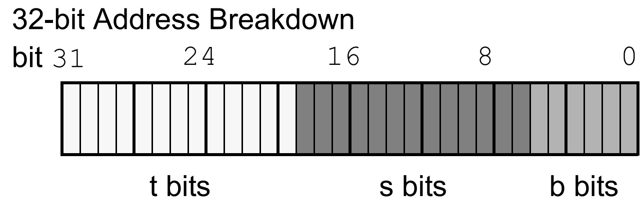
   must fetch block from next lower level

   if match and valid bit is 1, cache HIT

   for L1 cache, must now extract word from block

Basic Cache Practice

You are given the following 32-bit address breakdown used by a cache:



→ How big are the blocks?

6 b-bits   B = 2^b = 2^6 = 64 bytes

→ How many sets?

13 s-bits   S = 2^s = 2^13 = 8 k = 8192 sets

→ How big is this basic cache?

C = S x B = 2^13 x 2^6 = 2^19 = 2^10 x 2^9 = 512 k

Assume the cache design above is given the following specific address: **`0x07515E2B`**

0000 0111 0101 0001 0101 1110 0010 1011

t-bits | s-bits | b-bits

→ Which set should be checked given the address above?

0010101111000

8
16
32
64
256
1024

= 1400
index

→ Which word in the block does the L1 cache access for the address?

$\underbrace{1010}_{\text{word}}\underbrace{11}_{\text{byte}}$ word 10

➢ Which byte in the word does the address specify?

byte 3

Assume address above maps to a set with its line having the following V status and tag.

→ Does the address above produce a hit or miss?

$\underbrace{0}_{0x\,0}\underbrace{0000}_{0}\underbrace{111}_{E}\underbrace{01010}_{A}$

V tag

1.) `1  0x0750` miss

2.) `0  0x0750` miss

3.) `1  0x00EA` HIT

4.) `0  0x00EA` miss