

## Set Associative Caches

Similar to Direct Mapped, except that multiple blocks can be stored at each Index. Must look at ALL tags at a given Index to determine if hit or miss. Must invoke the **replacement policy** on a miss if a given set is full. LRU = Least Recently Used, a replacement policy where you evict the item accessed longest ago.

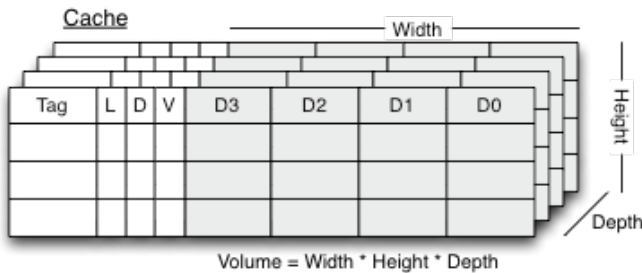


Figure 1: A 12 block, 4 way set associative cache (3 sets) can be visualized as 4 direct mapped caches stacked on top of each other.

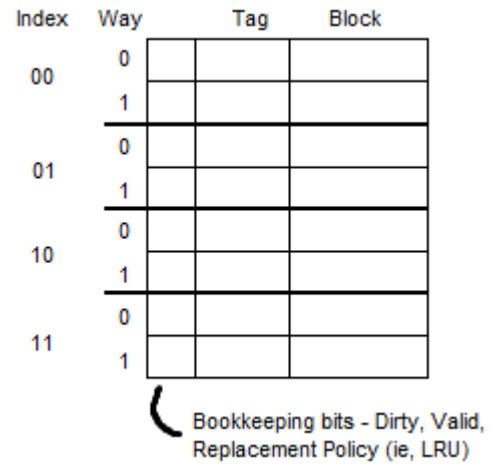


Figure 2: An 8 block, 2 way set associative cache (4 sets)

- How big should the T, I, and O fields be on a system with:
  - 32-bit addressed memory, 64KB fully associative cache, 4-byte blocks.  
**30/0/2**
  - 8-bit addressed memory, 32 B 2-way set associative cache, 4-byte blocks  
**4/2/2**
  - 8-bit addressed memory, 32 B 4-way set associative cache, 4-byte blocks.  
**5/1/2**
  - 32-bit addressed memory, 32 KB 4-way set-associative, 16-byte blocks.  
**19/9/4**
- In a 2-way set-associative cache, the addresses A, B, and C, all have the same index but distant tags.
  - What is the minimum sequence of accesses, which, if repeated, will maximize the miss rate in the cache if it uses the LRU replacement policy?  
**A B C. Every time the newly loaded element will kick the oldest element out. We always get a miss in this scheme**
  - What is the miss rate if this happens?  
**100% miss rate**
  - If the hit time is 1 cycle, and the miss penalty is 3 cycles, what will be the average memory access time (in clock cycles)?  
**AMAT = hit time + miss rate \* miss penalty = 1 + 1\*3 = 4.**
  - What would be a better replacement policy?  
**MRU (most recently used), or even random.**
  - What will the miss rate be for LRU replacement when the sequence is ABC CBA ABC CBA ...?  
**Amortized over time, the miss rate will be 1/3 = 33% (first two of the three hit, last misses)**

## C Memory Management

You can allocate memory on the heap by calling `void* malloc(int num_bytes)` which returns a pointer to the allocated space. However, you must remember to always eventually call `void free(void* ptr)` on that pointer so that you don't have a **memory leak**!

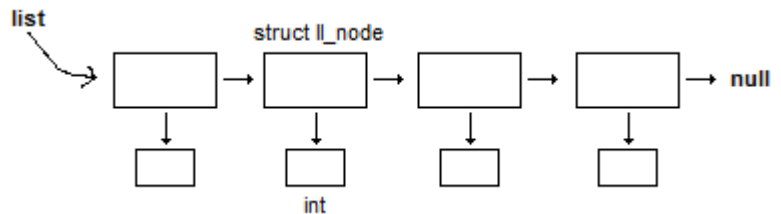
3. What is wrong with the C code below:

```
int* ptr = (int*) malloc(4 * sizeof(int));
if(extra_large) {
    ptr = (int*) malloc(10 * sizeof(int));
}
return ptr;
```

If `extra_large` is true, we have a memory leak (we lose the pointer to the memory initially allocated).

4. For the singly linked list implementation below, fill out `free_ll`, which frees all of the memory allocated for the linked list.

```
struct ll_node {
    struct ll_node* next;
    int *element;
}
```



```
void free_ll(struct ll_node* list) {
    /* YOUR CODE HERE */
    if(list) {
        free_ll(list->next);
        free(list->element);
        free(list);
    }
}
```