354 Today

Class: Part 3(!)

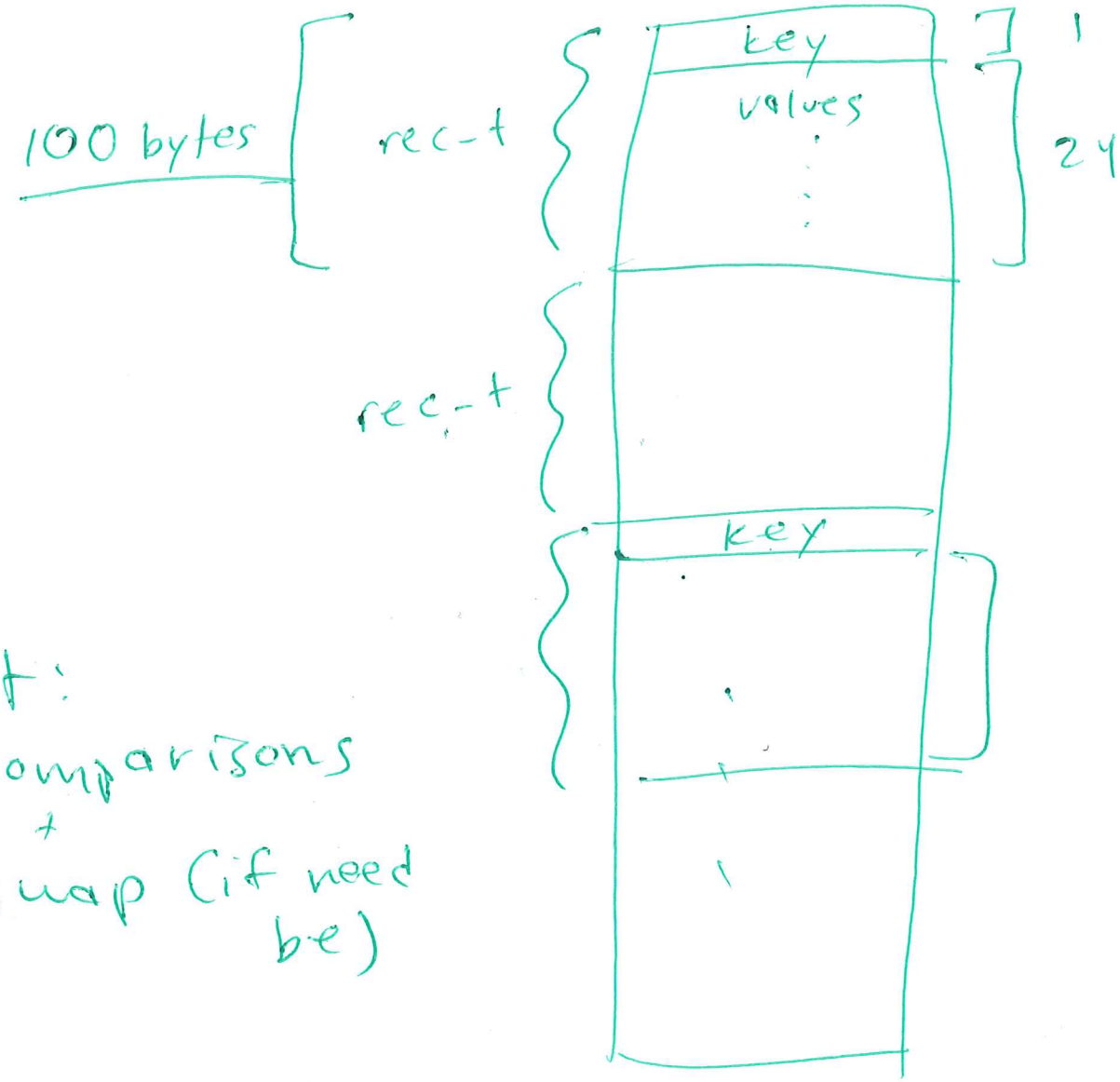
→ Each week:
Some large topic
in one day

→ Today:
Hardware: Hardware Caches

```c
// array.c
int data[DIM][DIM];
int main(int argc, char* argv[]) {
    int i, j;
    // … some init stuff …
    double start = gettime();
    for (i = 0; i < DIM; i++)
        for (j = 0; j < DIM; j++)
            data[i][j] = random() % 256;
    double end1 = gettime();
    for (j = 0; j < DIM; j++)
        for (i = 0; i < DIM; i++)
            data[i][j] = random() % 256;
    double end2 = gettime();
    // … some printing stuff …
}
```

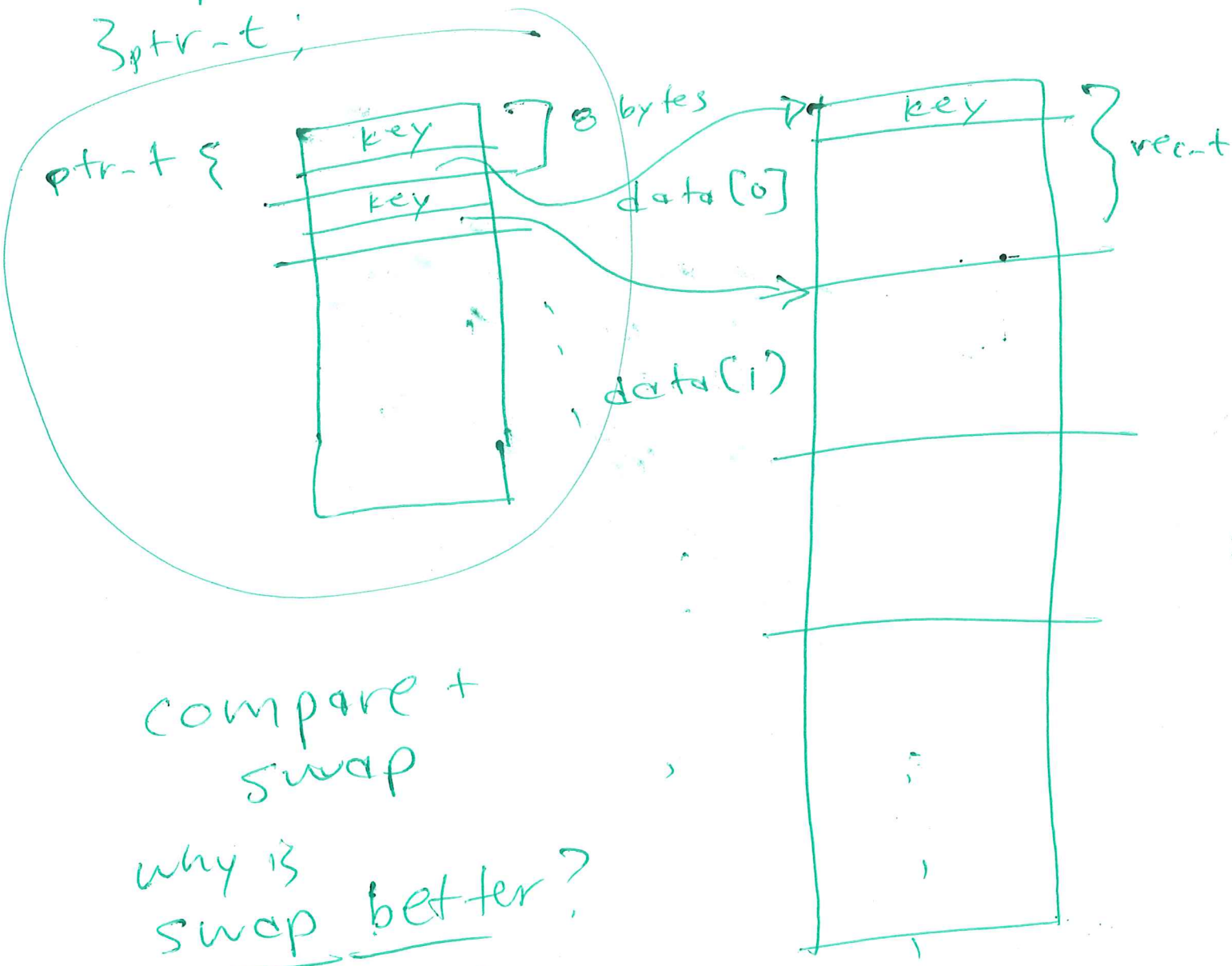
```c
// sort.c
int compare(const void *a, const void *b) {
    return *(int *)a - *(int *)b;
}

int main() {
    // … some init stuff …
    double start = gettime();
#ifdef POINTER_SORT
    printf("pointer sort\n");
    ptr_t* just_keys = malloc(array_size * sizeof(ptr_t));
    for (i = 0; i < array_size; i++) {
        just_keys[i].key = data[i].key;
        just_keys[i].record_ptr = &data[i];
    }
    start = gettime();
    qsort(just_keys, array_size, sizeof(ptr_t), compare);
#else
    printf("record sort\n");
    qsort(data, array_size, sizeof(rec_t), compare);
#endif
    double end = gettime();
    // … some printing stuff …
}
```

record sort



Sort:
comparisons
+
swap (if need be)

```
typedef
  struct ptr {
    int key;
    rect* record-ptr;
  } ptr-t;
```



compare + swap

why is swap better?

less data movement

$O(n)$ steps

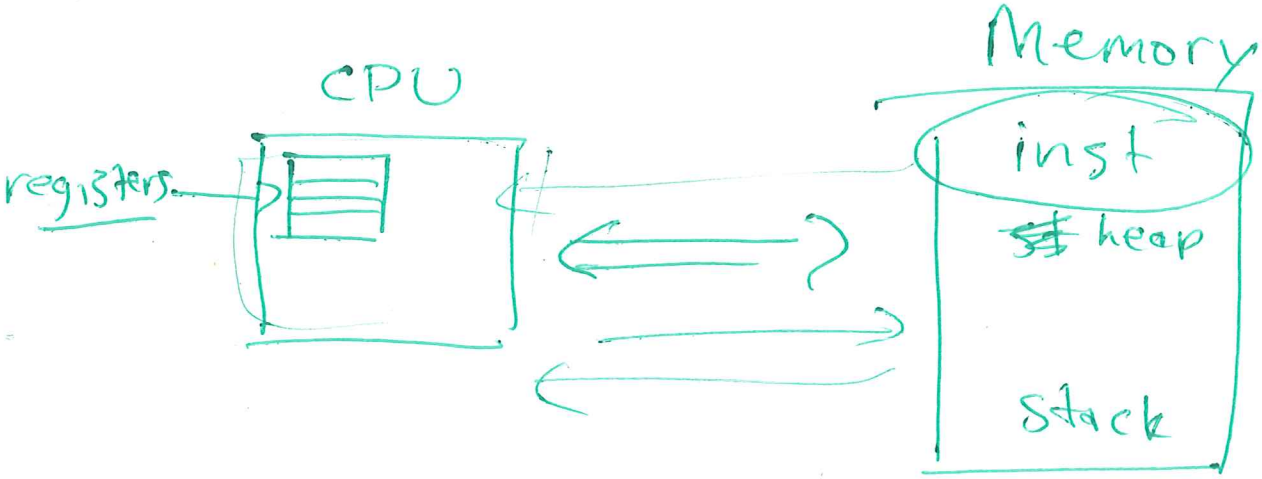
"complexity" of algorithm

for (i=0 ; i<N ; i++)

$\sim N \Rightarrow O(N)$

$\Rightarrow$ # of steps is not enough to predict performance

Memory:



Memory is slow

Solutions?

⇒ (registers) → reg

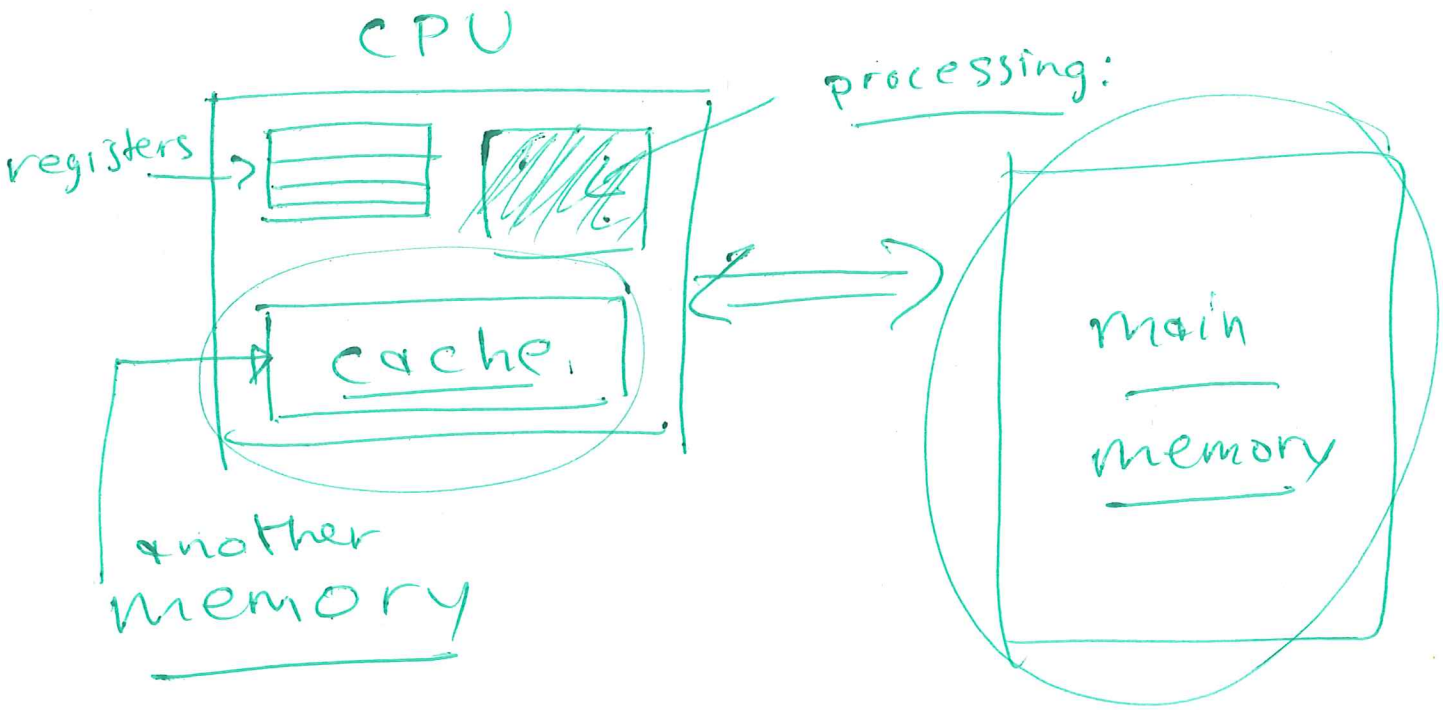
⋮

Problems remain:

⇒ instruction feten

⇒ accessing a lot of data (more than fits into registers)

Hardware cache:



cache: goal

keep copies of frequently used "data" here

assumption:

cache is smaller than main memory but faster

⇒ large memory ⇒ slow

fast memory ⇒ small

Goal: How do caches work?

Memory:

byte address => Memory [load]

data

address, data => Memory [store]

x86:

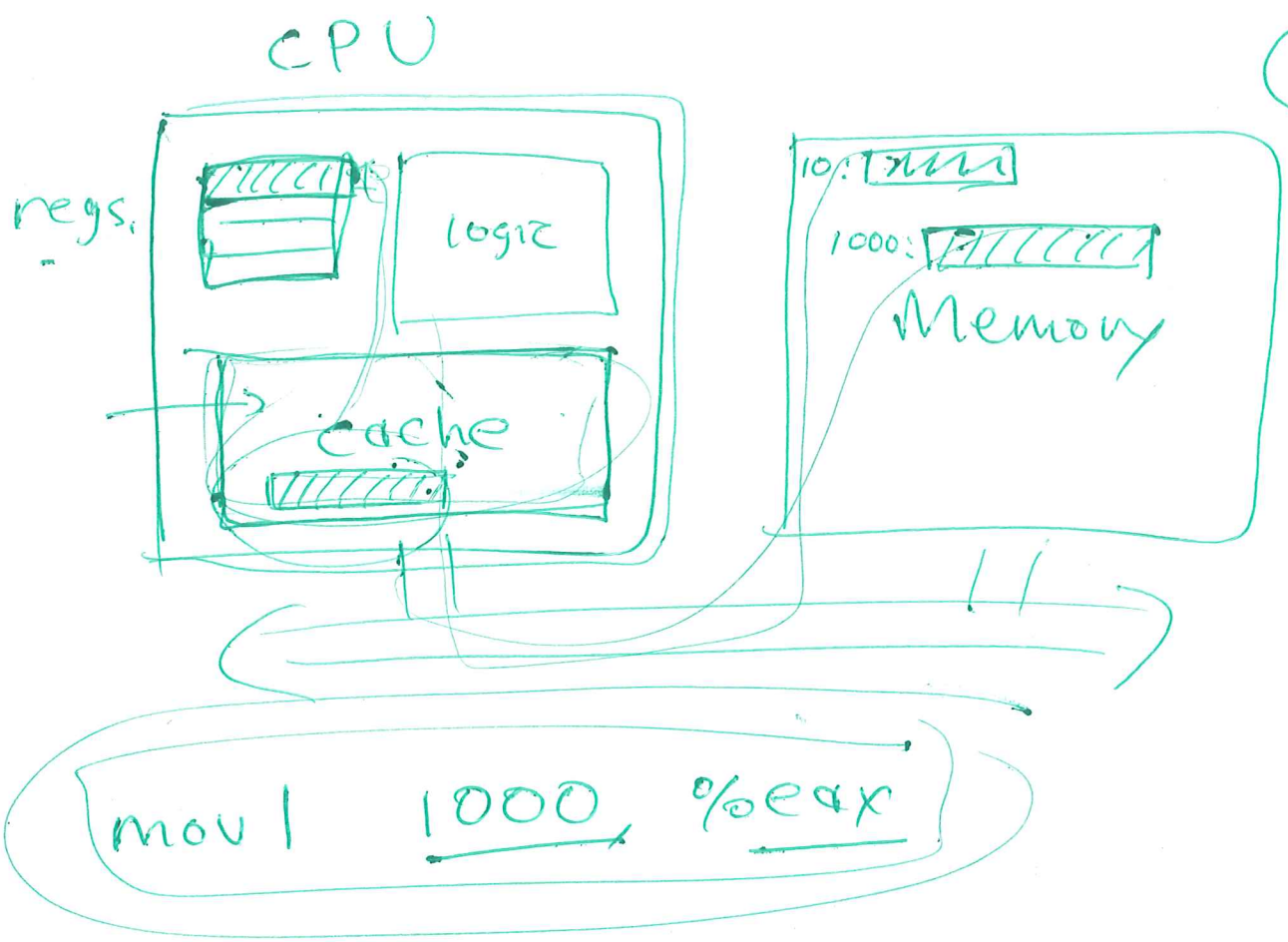
movl 10000, %eax

=> read 4 bytes from mem
put it into eax (register)

explicit data movement/copy

cache: implicit (hidden)

=> make large memory
seem fast



CPU:

fetch

decode - ah, this is a [load]

execute:

check if address = 1000 is in cache

if it is: cache hit

use cached copy to complete instruction

if not in cache: miss

fetch data from memory, put copy in cache, finish instruction

unit / size of access?

mov**l** 1000, %eax

32-bit

4 byte

memory : more efficient

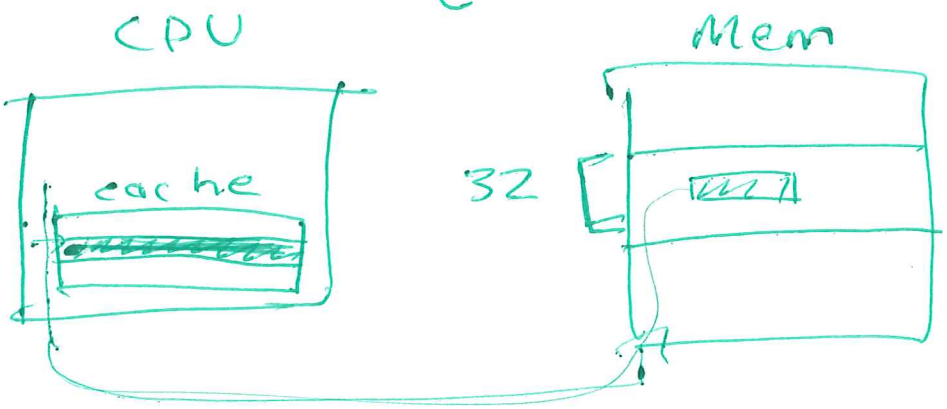
if move more data

(larger than 4 bytes)

cache

block / line:

typical size: [32 bytes, 64 bytes, 128 bytes]



cache blocks:

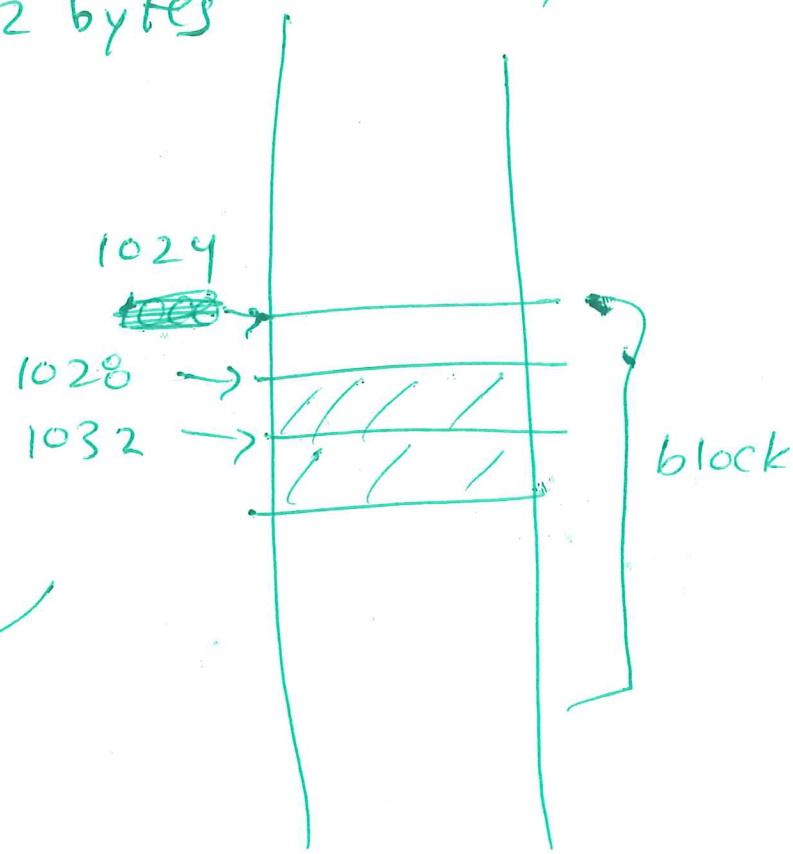
benefits:

movl	1024 [illegible],	%eax	miss
movl	1028 [illegible],	%ebx	hit
movl	1032 [illegible],	%ecx	hit

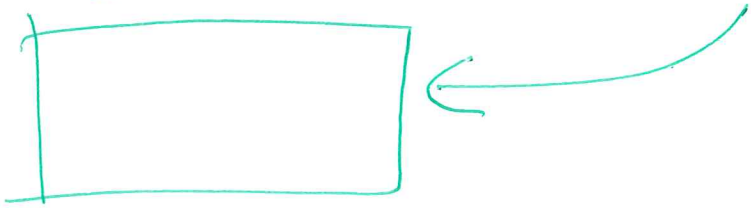
memory

cache block: 32 bytes

⇒ prefetch



cache



Cache organization

cache block: 32 bytes

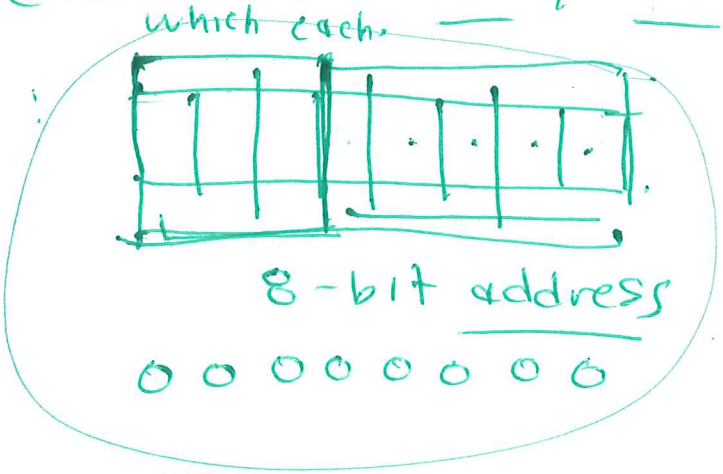
cache entries:

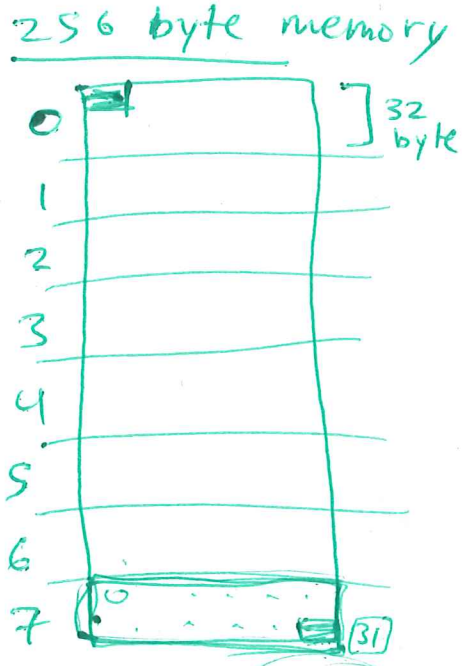
pretend small cache w/ 4 entries

typical size ~ 1000 entries (Level-1 cache)

(problem: given address, determine hit/miss)

address:





address: | which cache block? | which byte in block? |

address: 255 ⇒ | 1 1 1 | 1 1 1 1 1 |

7 31

offset

(32 bytes) main memory

block

block address

256 bytes

cache : 2 entries

single entry:

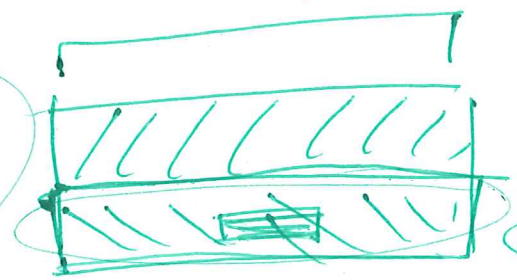
data (block)

32-bytes

tag

2-entry cache :

block



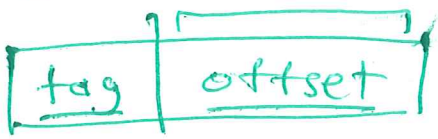
hit:

CPU :

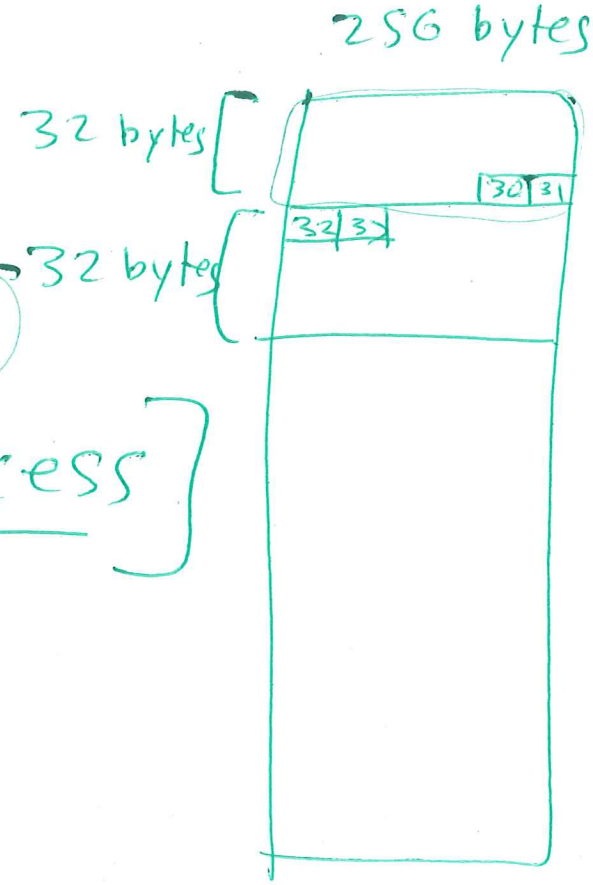
→ examine cache, figure out : hit, miss ?

→ address : mov! 100(%eax), %ebx

100 + [eax] ⇒



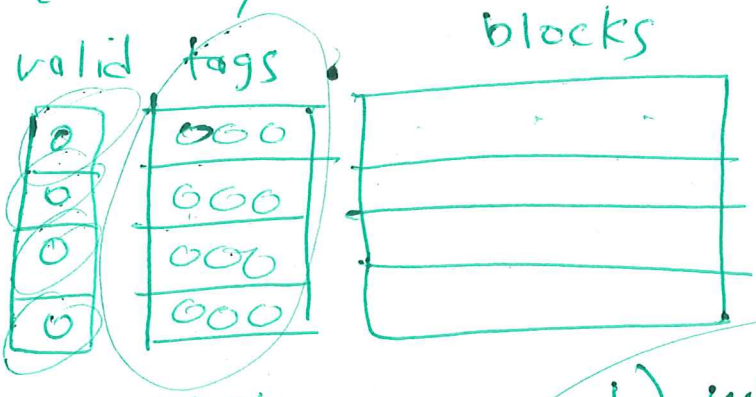
tag ⇒ use tag to determine hit/miss



⇒ movl 30, %eax

[unaligned access]

⇒ 4-entry cache

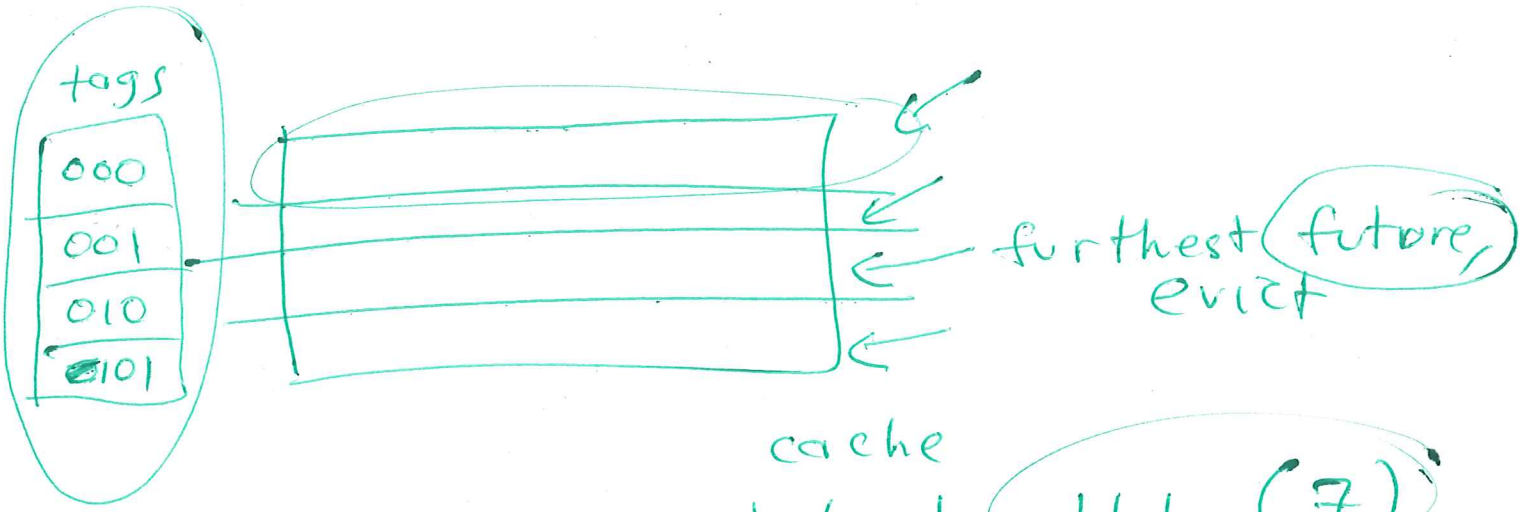


which block is cached?

1) what happens when system first starts executing?

valid bit: one per entry

2) what happens when cache fills up?



cache

access: to block 111 (7)

⇒ miss + cache full

⇒ evict / replace existing entry

eviction/ policy
replacement

heuristic:

eviction / replacement policy:

⇒ furthest away (address distance)

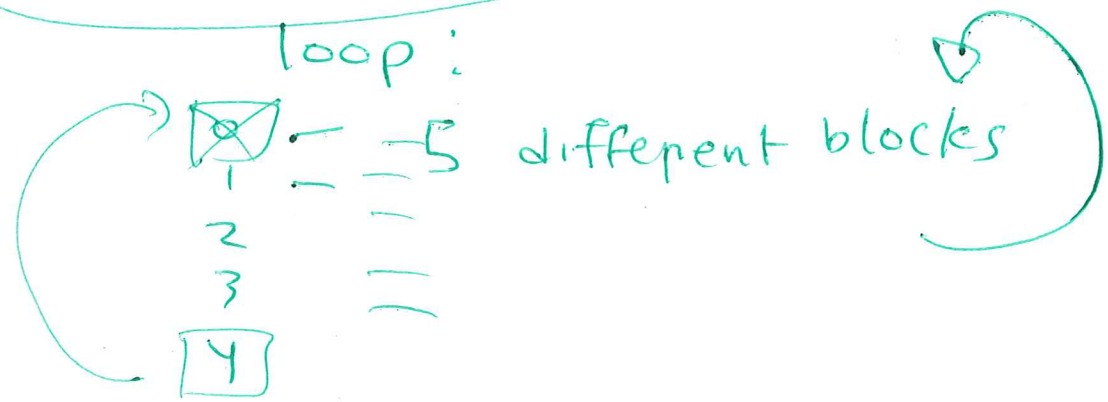
⇒ timing: kick out block least recently used (LRU)

⇒ least frequently used (LFU)

⇒ pseudo-random

4-entry cache:

loop:



Problem w/ cache:

hard to build efficiently

why? caches have 1000's of entries

⇒ 1000's of tags

cache lookup:

⇒ has to look thru all tags to see hit/miss

how to build a cache that doesn't have to check all tags at once?

(faster)

Fully-associative cache

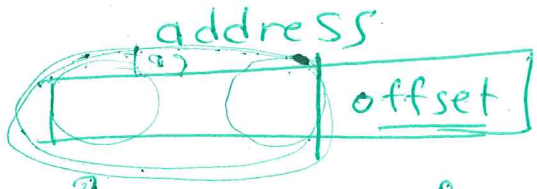
⇒ block from memory
can go anywhere in cache

⇒ slow / complex

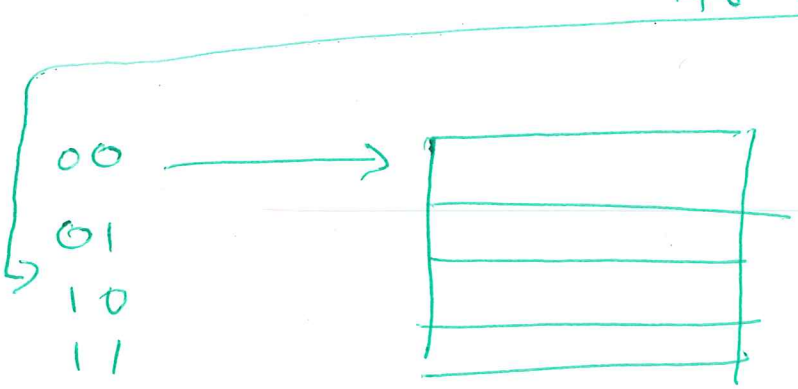
Other extreme: Direct-mapped cache

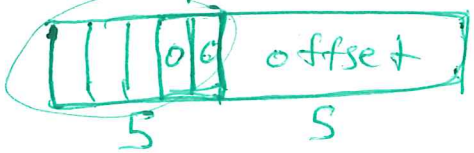
⇒ block, based on address,
can only go into
one particular slot of cache

example:



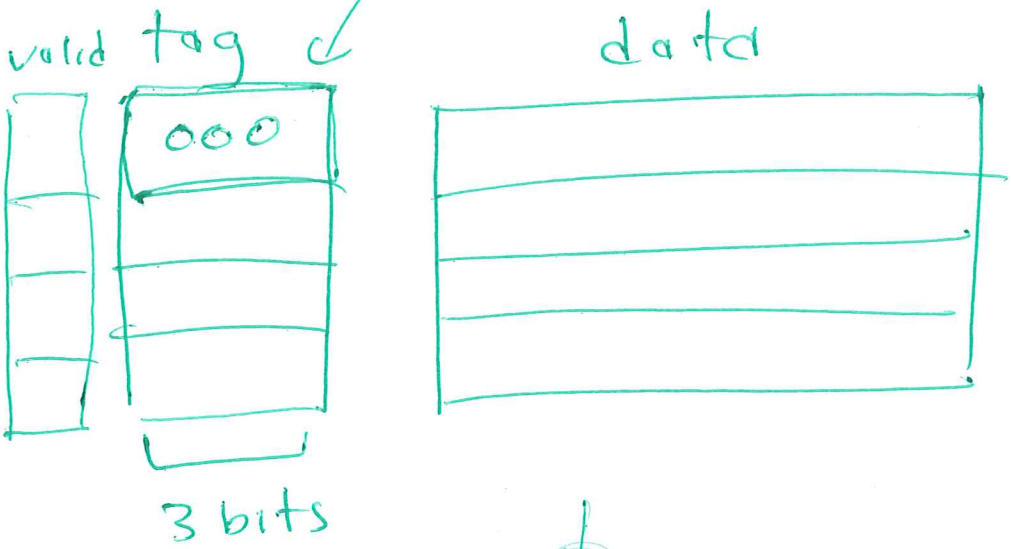
use some of upper bits
to determine slot of cache



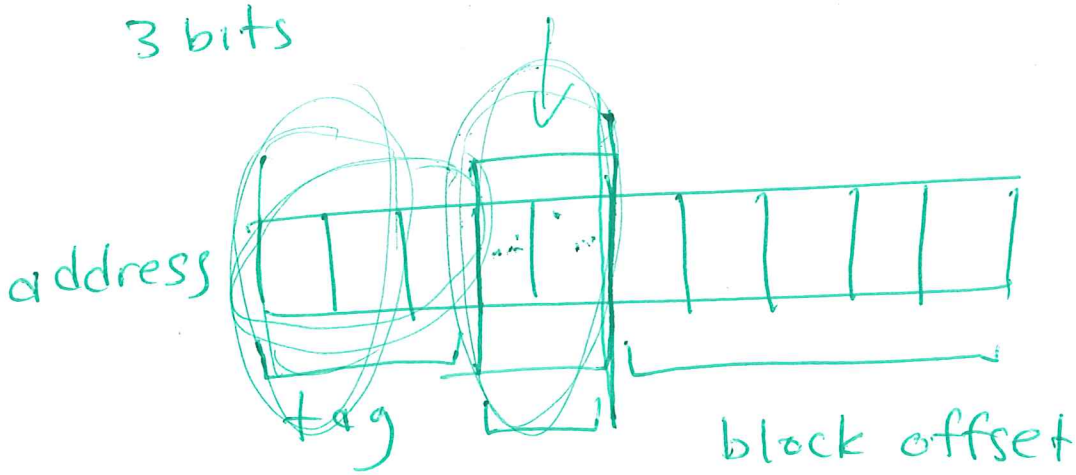


(1024-byte memory) 32-byte block

only look in one place in cache



00
01
10
11



32-byte block

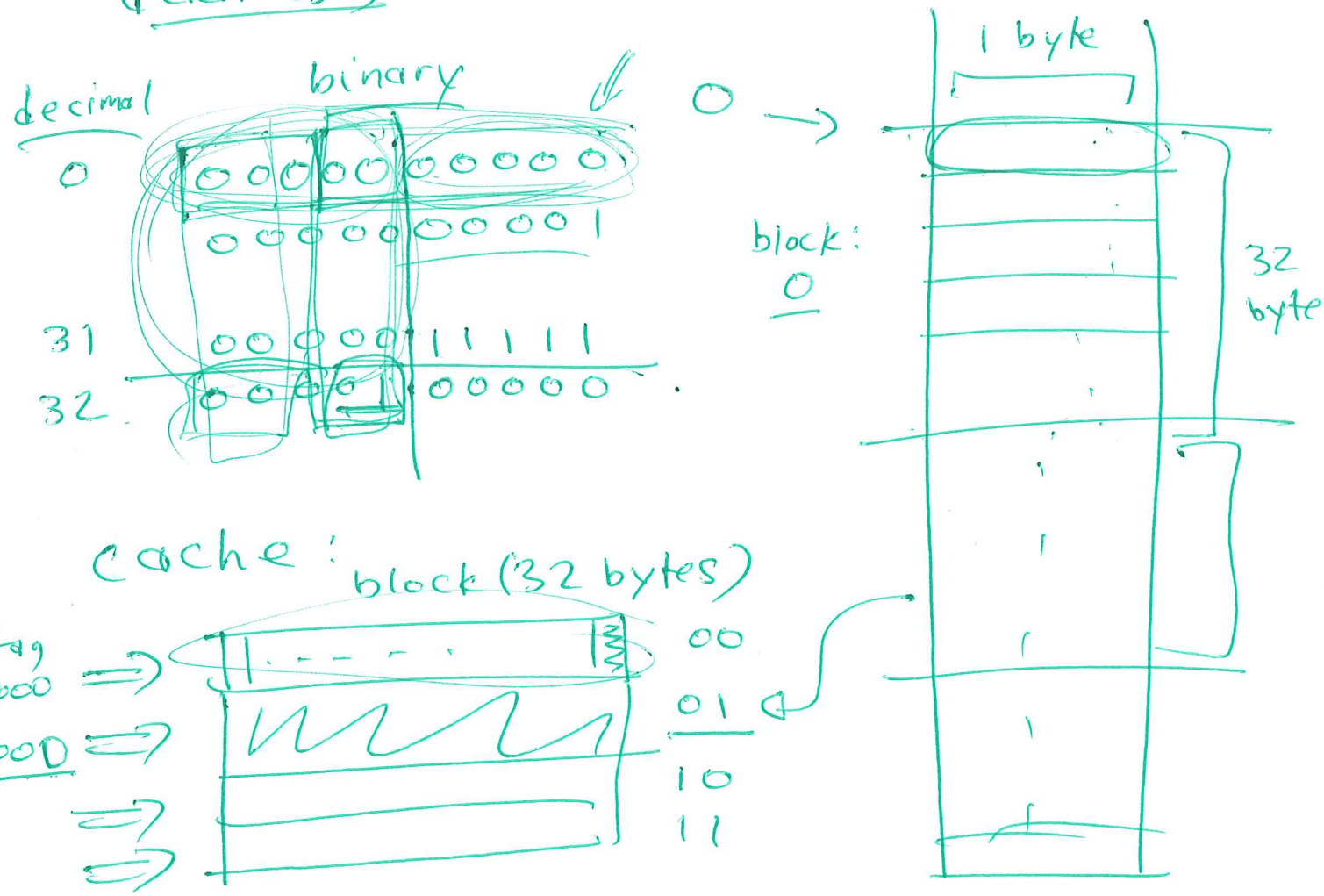
↳ which slot of cache?

address: 000 00 11111 ⇒ 31 (dec)

movb 31, %al

```
char a[SIZE];

for (i=0; i<size; i++)
    a[i] = 0;        movb $0, 0
```

addresses



a[0] => miss
a[1] => hit

sum: %ecx

```
movl 1024, %eax
movl 2048, %ebx
```

memory

```
addl %ebx, %ecx
addl %eax, %ecx
```

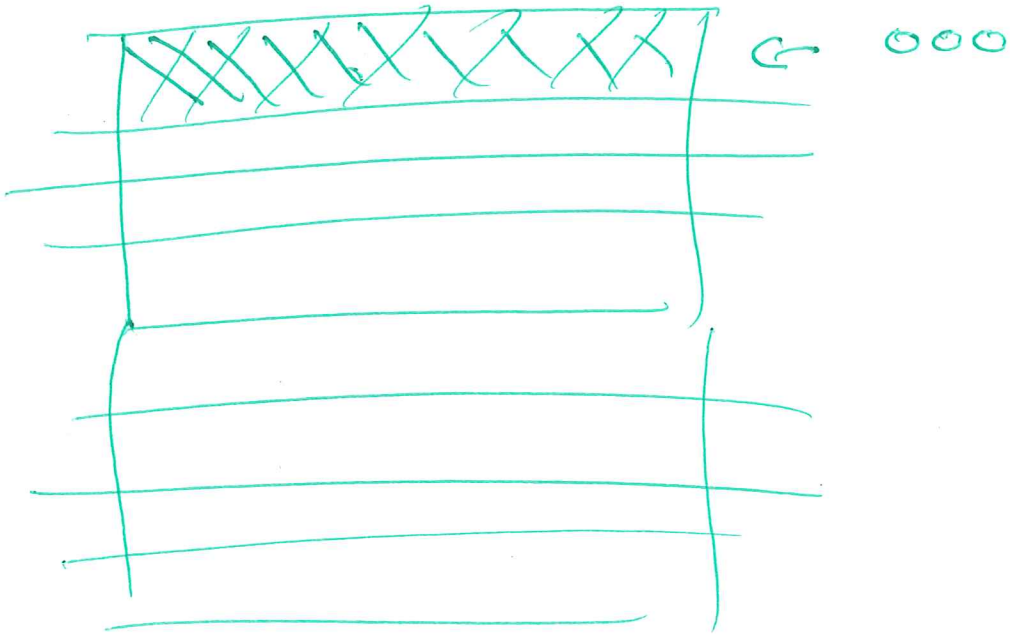
math

CPU:

address:

hit/miss

miss



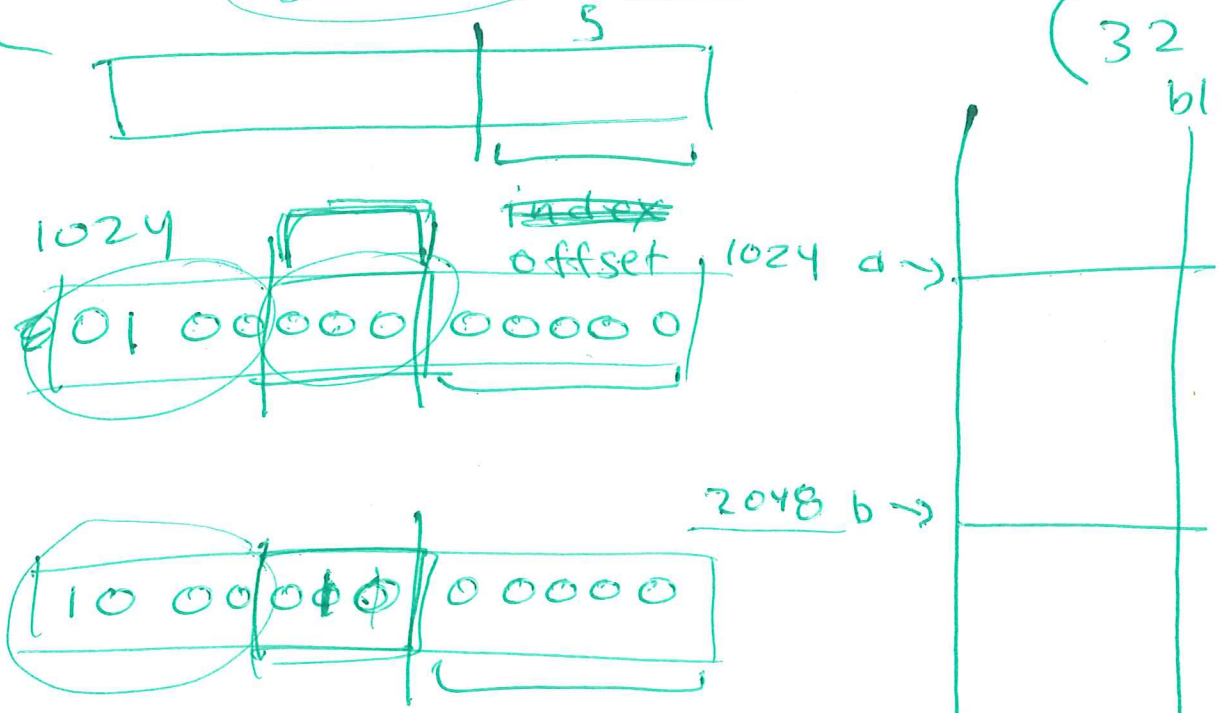
Problems w/ direct mapped cache:

=> conflicts

misses caused by limited mapping to specific slots

```
int a[SIZE];
int b[SIZE];
for (i=0; i<SIZE; i++)
    sum += a[i] + b[i];
```

(32 byte block)



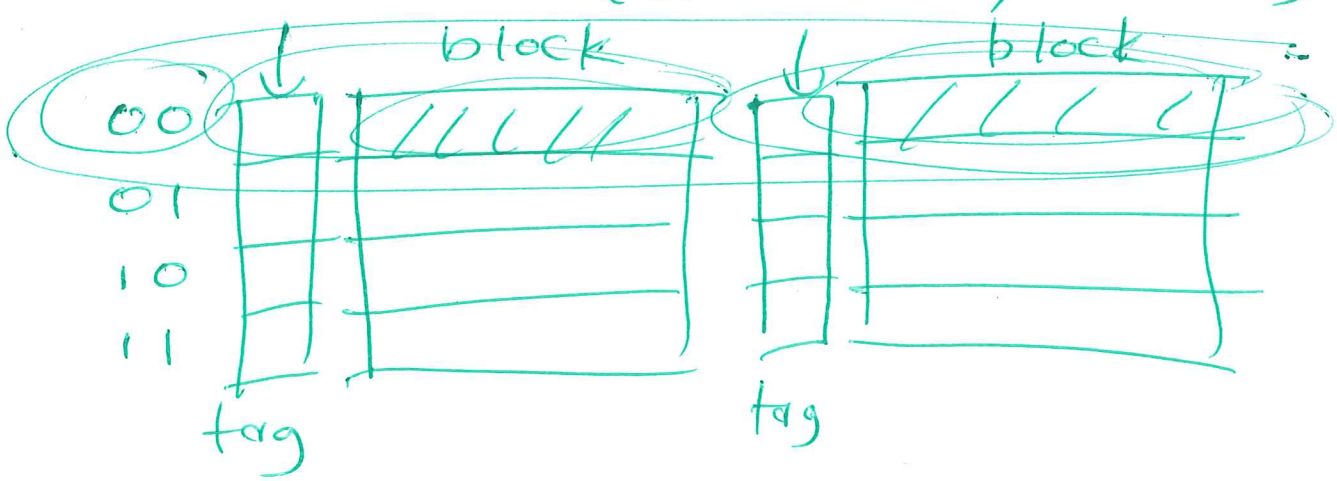
8-entry cache => 3 bits of cache index

N-way set-associative:

⇒ have N slots where block can be put in cache

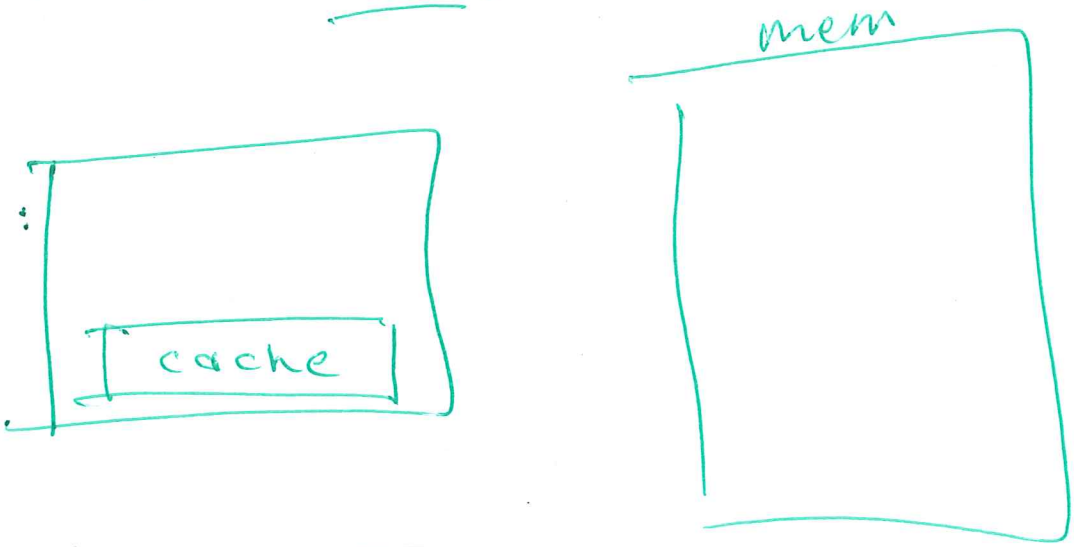
e.g. (2-way set associative)

8 entry cache like (2 4-entry caches)



writes : why different than reads ?

=> movl %eax, 1024



store => addr : 1024 ... 1027

hit: => do we update memory? or just cache?

perf:

options
- write back : update cache upon eviction, flush to memory
- write through : update cache + through to memory

simpler, more predictable / better perf on read misses

another policy choice

⇒ write allocate : on miss, bring cache line \ block into memory (write back)

or

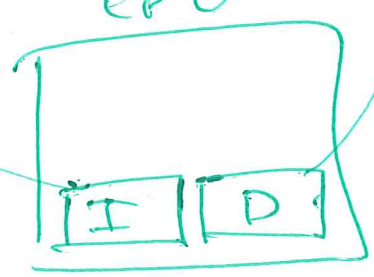
write noallocate : on miss, just update memory (write through)

Everything else about caches in 5 minutes :

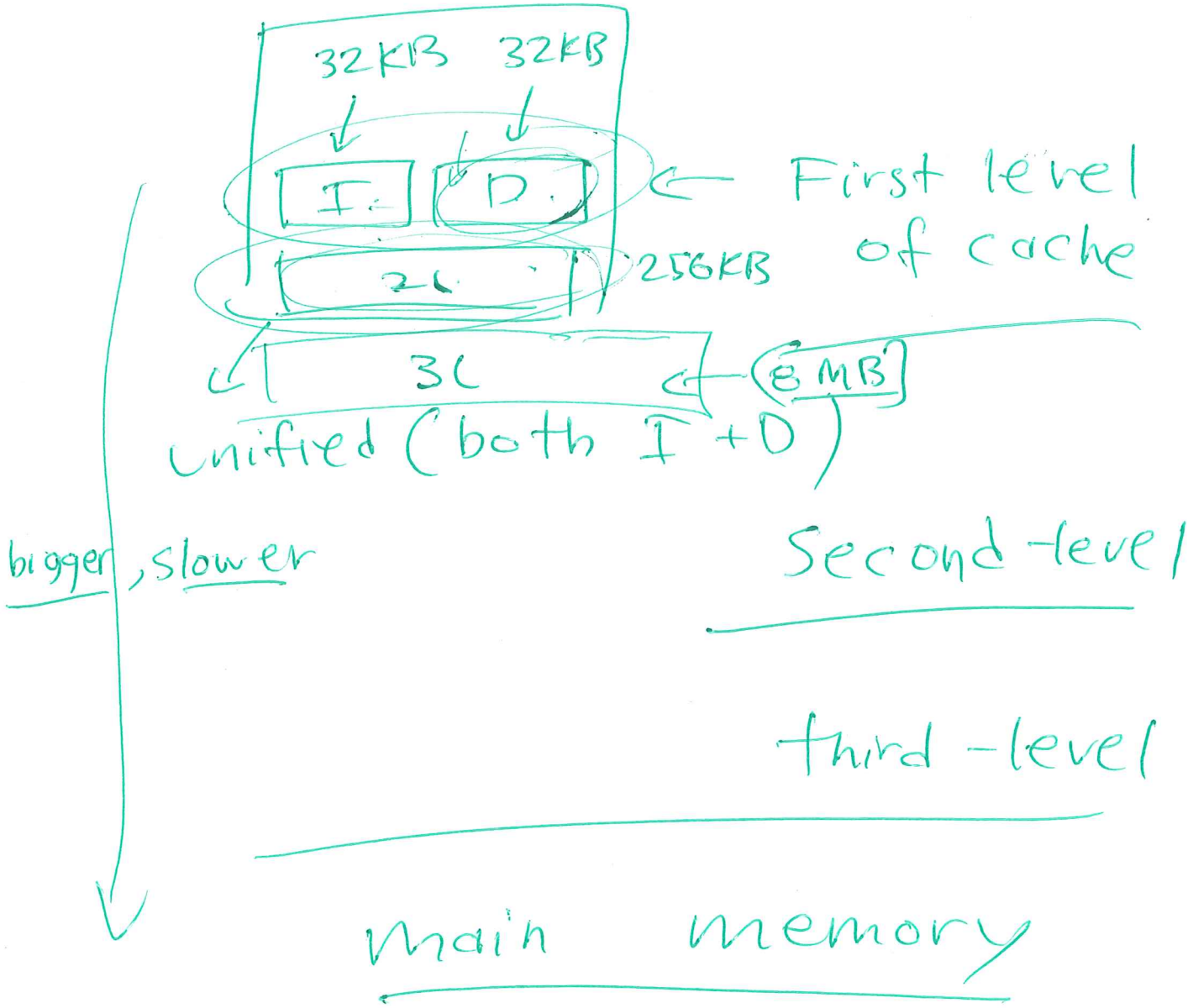
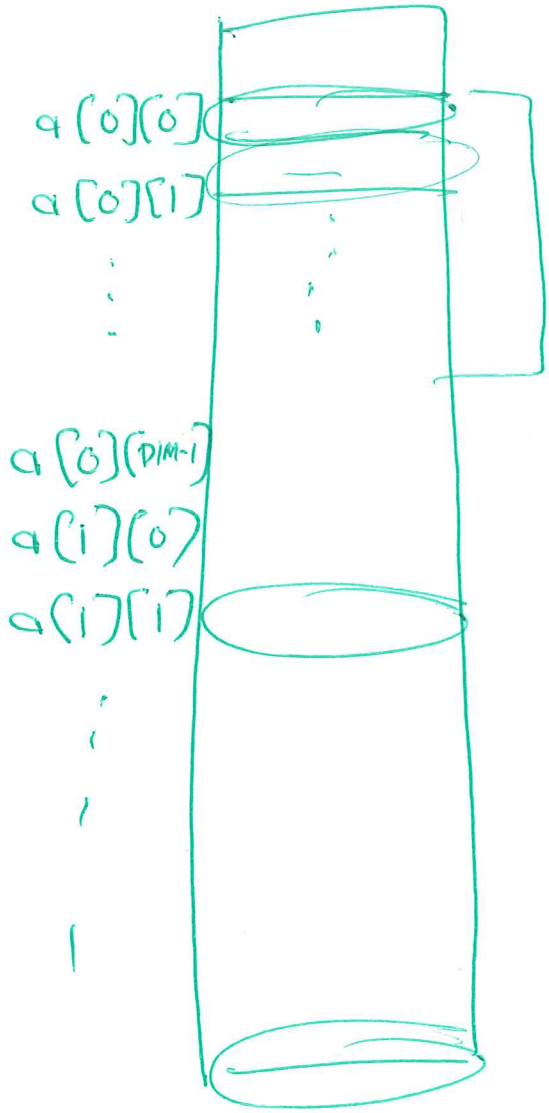
⇒ instructions, data are cached

split cache

CPU



Multiple levels of cache:



```
int a[DIM][DIM]

for (i=0 ....)
    for (j= ....)
        a[i][j]

for (j ....)
    for (i ....)
        a[i][j]
```