

MEMORY HIERARCHY

• Why?

- Memory hierarchies take advantage of "memory locality".

- i.e. future memory accesses are near past memory accesses.

• Types of locality:

- Temporal: we access the same data often.

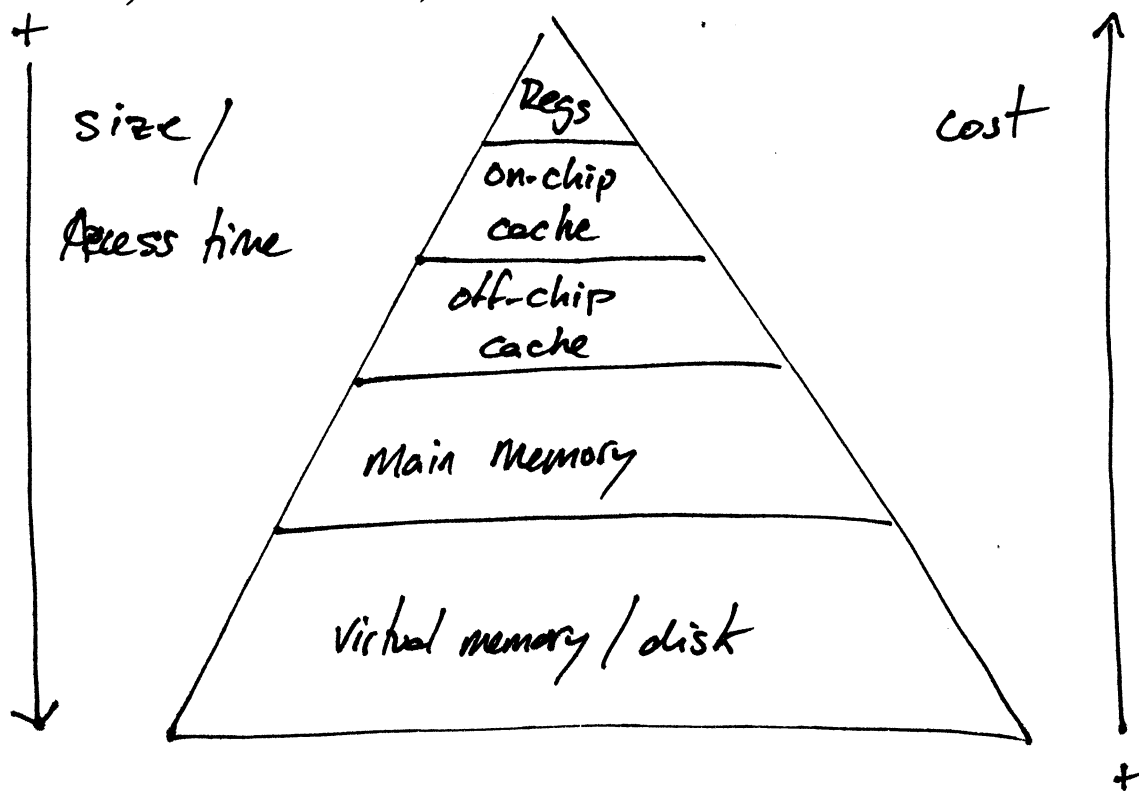
- Spatial: our next mem access is very close to our last accesses.

• So... Why is this important?

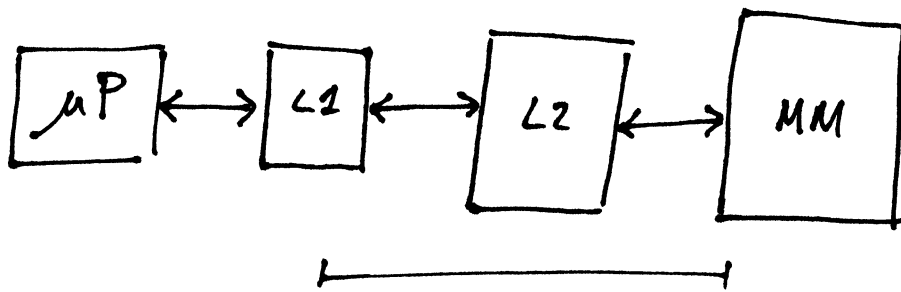
- We exploit locality by "caching-" data that is likely to be used again by the CPU

- think of cache as a place to store/keep data close to the processor.

• Our memory hierarchy will look like:

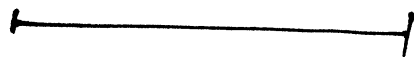


- A memory access will look like:



- Hit: desired data found @ desired level.

Miss: desired data Not found @ desired level.



- Hit time: time to access, if found, data.
(Cur.t. a particular level)

Miss penalty: time to bring desired data to current level after a miss + deliver data to CPU.

i.e. Miss time = Access time + Transfer time

where: * Access time = memory latency, or access to next level.

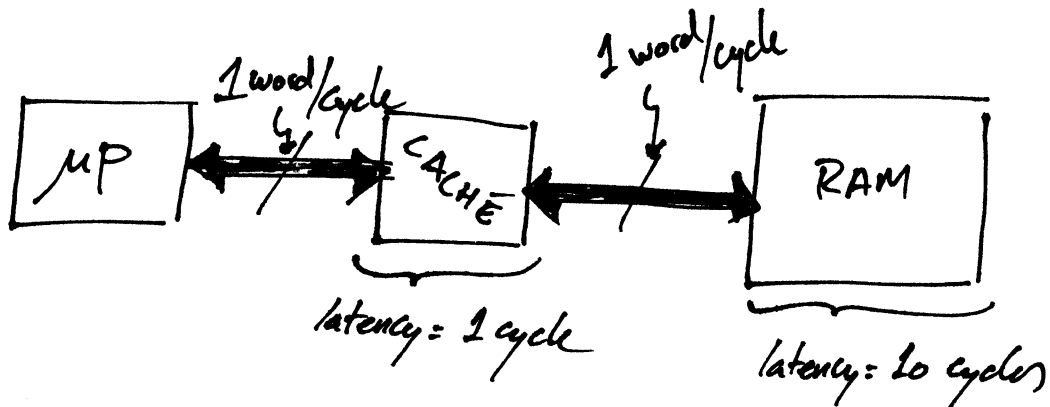
* Transfer time = Multiple transfer for larger data sizes.

- Hit Ratio = % time data is in cache

Miss Ratio = ?

EXAMPLE 1

Given the following architecture:



Assume we have a program with 100 mem references.
With the following distribution: ~~75~~ 75 cache hits
25 cache misses

$$T_{\text{total}} = (\# \text{ hits}) \cdot (T_{\text{hit}}) + (\# \text{ misses}) \cdot (T_{\text{miss penalty}})$$

$T_{\text{RAM ACCESS}} + T_{\text{TRANSFER}}$

So for this example:

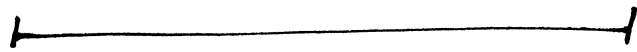
$$T_{\text{total}} = (75) \cdot (1) + (25) \cdot (10 + 1) = \underline{350 \text{ cycles}}$$

$$T_{\text{average}} = \frac{T_{\text{total}}}{100} = \frac{350}{100} = \underline{\underline{3.5 \text{ cycles}}}$$

* YEAH THAT'S INTERESTING, SO WHAT?

- Main problem: Cache is small, and MM is BIG!

we must define a function that maps many elements of the main memory, to any/one element in the cache.

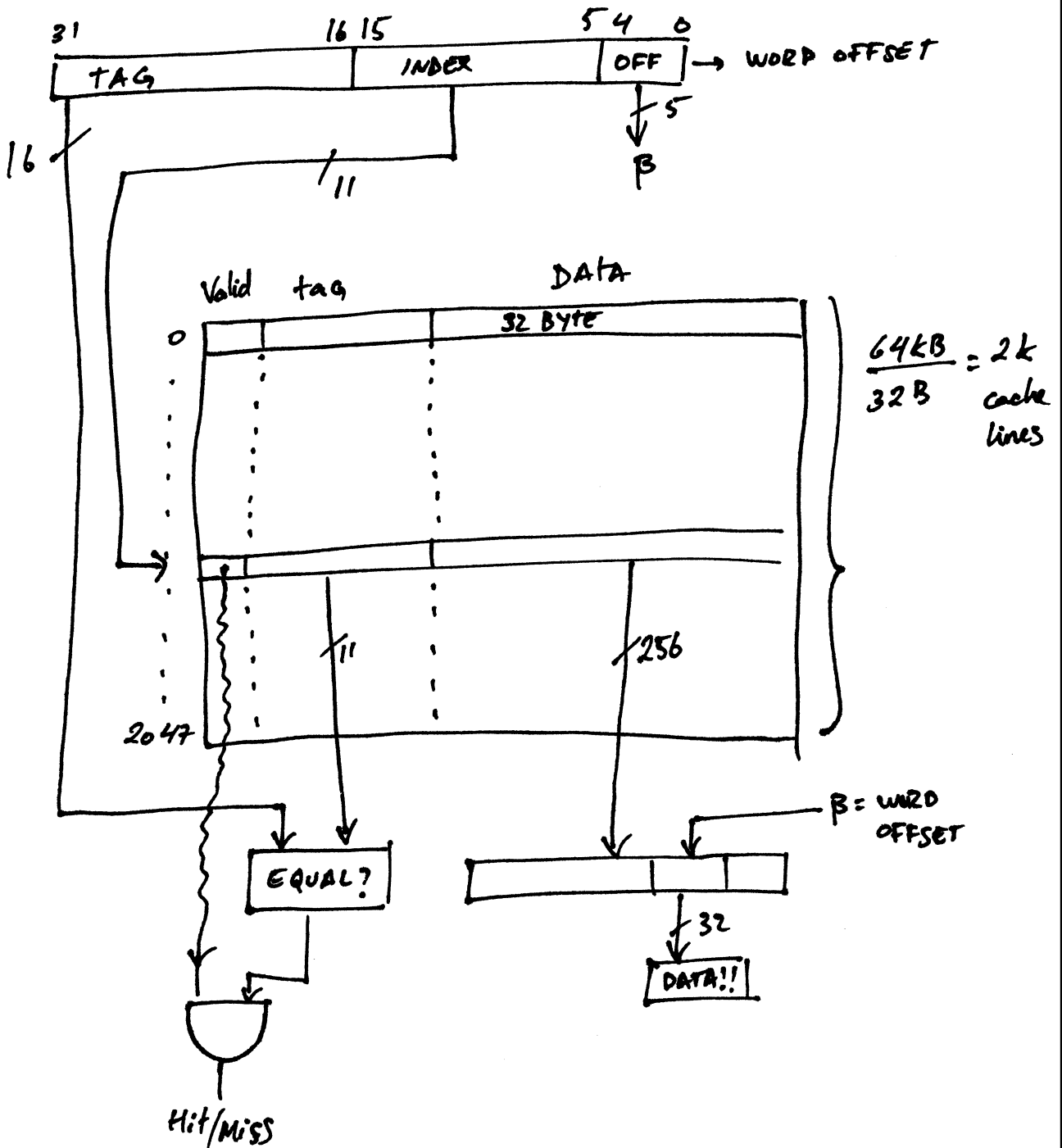


- Cache unit: Block or line
- usually a Block = X words ($X \geq 1$), why?
- Block is usually the atomic unit of transfer from MM.
- the MM \rightarrow cache position mapping policy = Associativity
- TYPES: (assume a cache with C blocks, or S sets of n-blocks)
 - ① FULLY ASSOCIATIVE: Block Y from MM can go to any block in cache.
 - ② DIRECT MAPPED: Block Y from MM can only go to block (Y mod C) in cache.
 - ③ S-SET ASSOCIATIVE: Block Y from MM can go to anywhere in set (Y mod S) in the cache.

SOME EXAMPLES

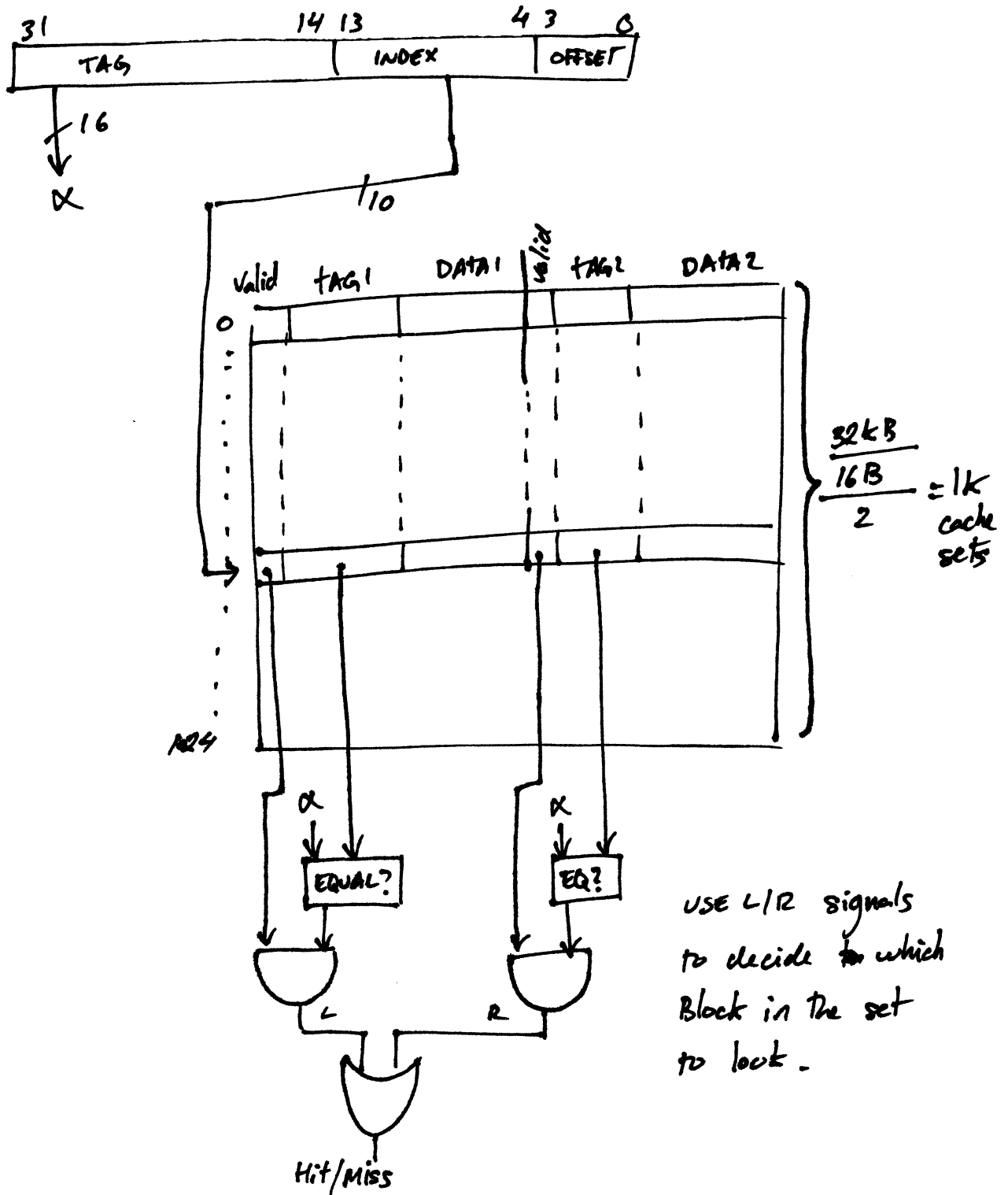
① DIRECT MAPPED :

ASSUME 64KB cache w. 32-Byte block size



⑤ 2-WAY SET ASSOCIATIVE

32kB cache, 16-Byte block size



• ASSOCIATIVE VS. DIRECT MAPPED ?

FOOD FOR THOUGHT:

- Large blocks useful to exploit spatial locality.
- If the block size is too large? We may waste space!
- However the longer the block, the smaller the tag space.
- MODUS OPERANDI:

1.- TAG + INDEX → Access cache & Determine Hit/Miss.

2.- If Hit then return requested data.

3.- If Miss then:

3.1.- select a cache block to be replaced

3.2.- Access next memory level (stall?)

3.3.- load entire missed line into cache

3.4.- Return data (Everyone is happy now!)

4.- If next memory level is also a cache, then go to step 1.- for that cache.

HUM... BUT WHAT ABOUT STORES?

- Stores may not require stalls
- Consistency between cache/memory must be enforced!
 - ① WRITE THROUGH: ALL writes go to cache + memory at the same time.
 - ② WRITE BACK: writes go 1st to cache. lines are sent back to memory only when they're about to be replaced.
- Allocate space in cache if store-miss
 - ③ WRITE ALLOCATE: bring written line into cache
 - ④ WRITE AROUND: ignore cache

REPLACEMENT ISSUES

- What happens to a line in cache when we bring new data to it?

Ⓐ DIRECT MAPPED:

- No choice, we can only replace 1 specific line
- Write data back to memory, (again no choice)

Ⓑ ASSOCIATIVE:

- Several choices for the location of the data
- Must establish a replacement policy.
(i.e. FIFO, LIFO, LRU, Random, etc.)

• WRITE DATA BACK

- write data back always (safest).
- write only "dirty" data back.

- Note: Replacement policy is only required if cache is associative!

CACHE PERFORMANCE

$$CPI_{TOTAL} = CPI_{BASE} + CPI_{MEMORY}$$

CPI_{MEMORY} = # of stall cycles due to memory accesses

$$CPI_{MEMORY} = \left(\frac{\text{accesses}}{\text{Instruction}} \right) \times (\text{miss-rate}) \times (\text{miss-penalty})^*$$

- * assuming:
- pipeline stalls on reads AND write misses
 - miss penalty for read/write misses is the same
 - cache hits have no stalls

VIRTUAL MEMORY!

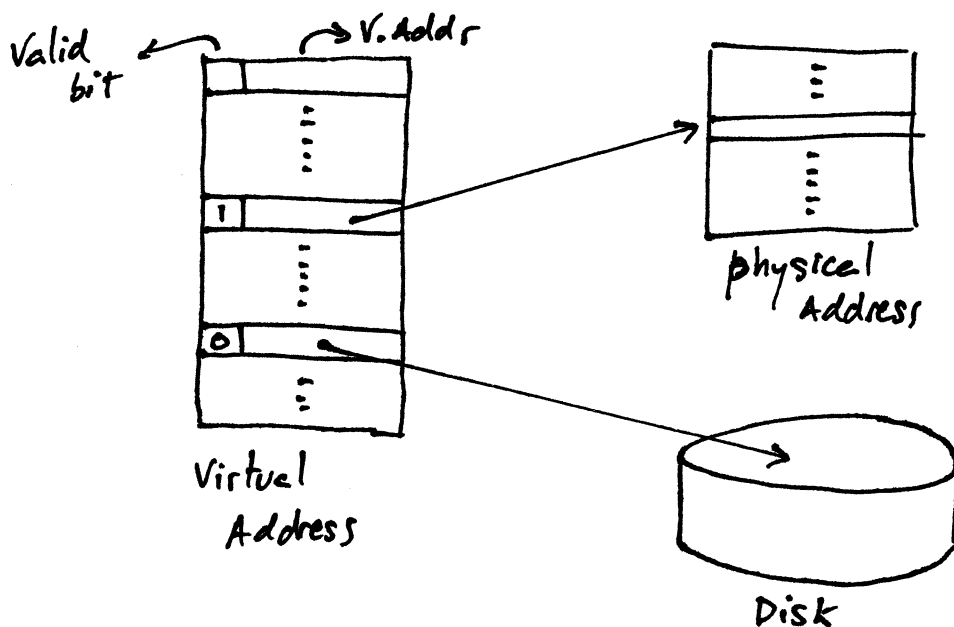
- It's just like caching, I promise!
- Simply use Main Memory as a cache for a larger memory, i.e. the hard disk!
- Of course to make it more complex "they" introduced new terminology:

<u>cache</u>		<u>Virtual Memory</u>
block	→	Page
cache miss	→	Page fault
address	→	virtual Address
index	→	physical Address

- note: the atomic unit of transfer is the page.
its usually determined by the OS.
Normally Page size \gg Cache block size

VIRTUAL MEMORY vs. CACHING

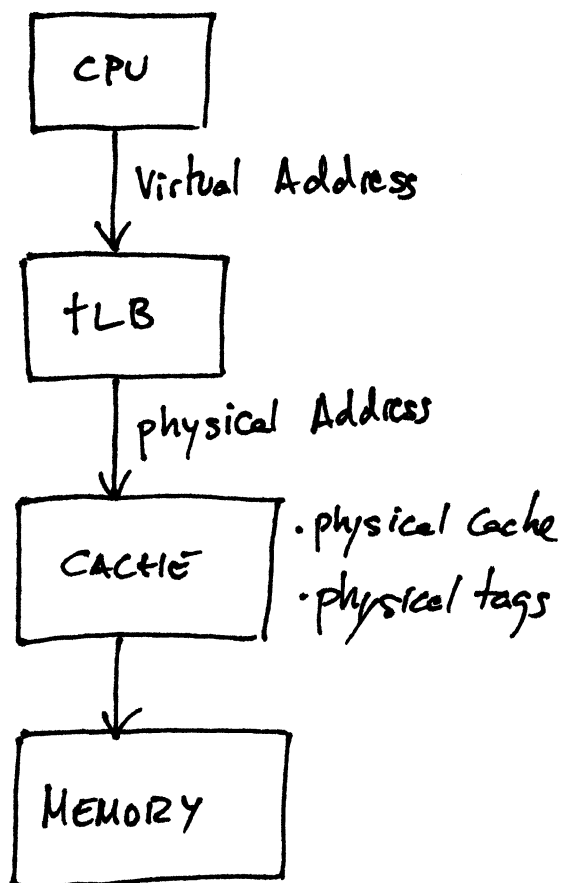
- VM is a functionality enhancement not a performance enhancement
- Pages are at least one order of magnitude larger than cache/memory blocks.
- Usually the mapping of pages to Main Memory is fully associative.
- only write back is allowed, why?
- Miss penalties are way larger.
- Misses are handled by software (but not hits).



MAKING THE COMMON CASE FAST:

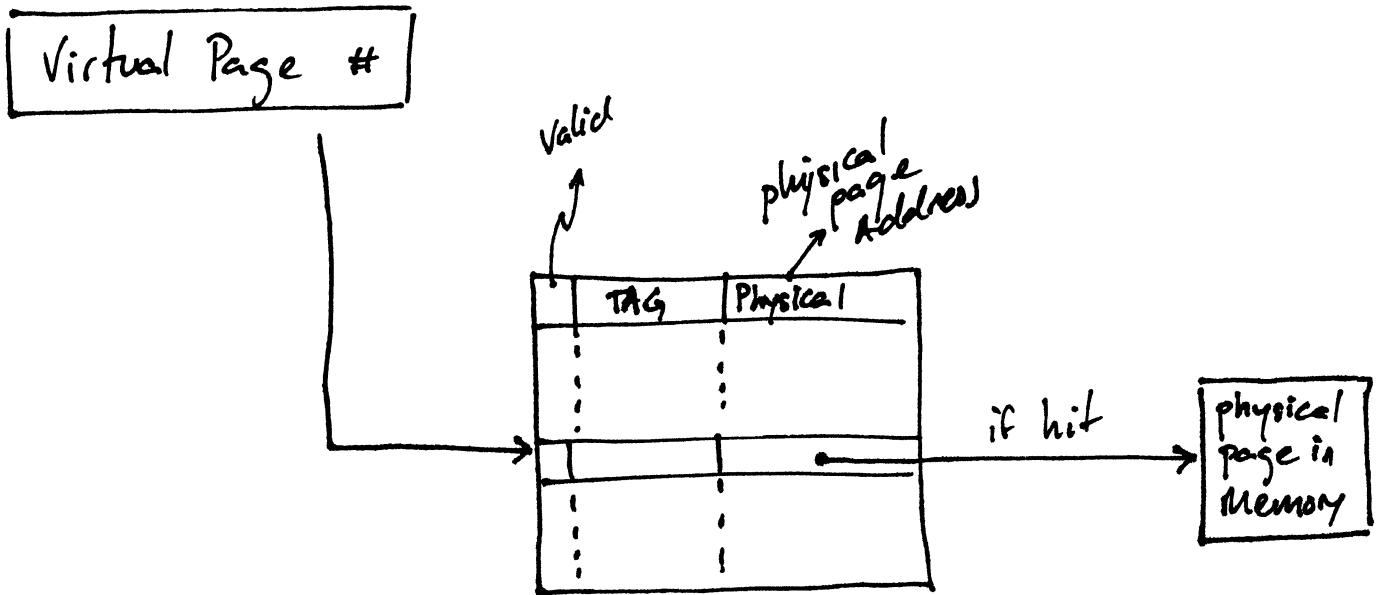
CAN WE SPEED UP ADDRESS TRANSLATION?

- YES! USE TLB!!
- TLB = TRANSLATION LOOK ASIDE BUFFER
- How DOES THE TLB INTERACT WITH CACHE?

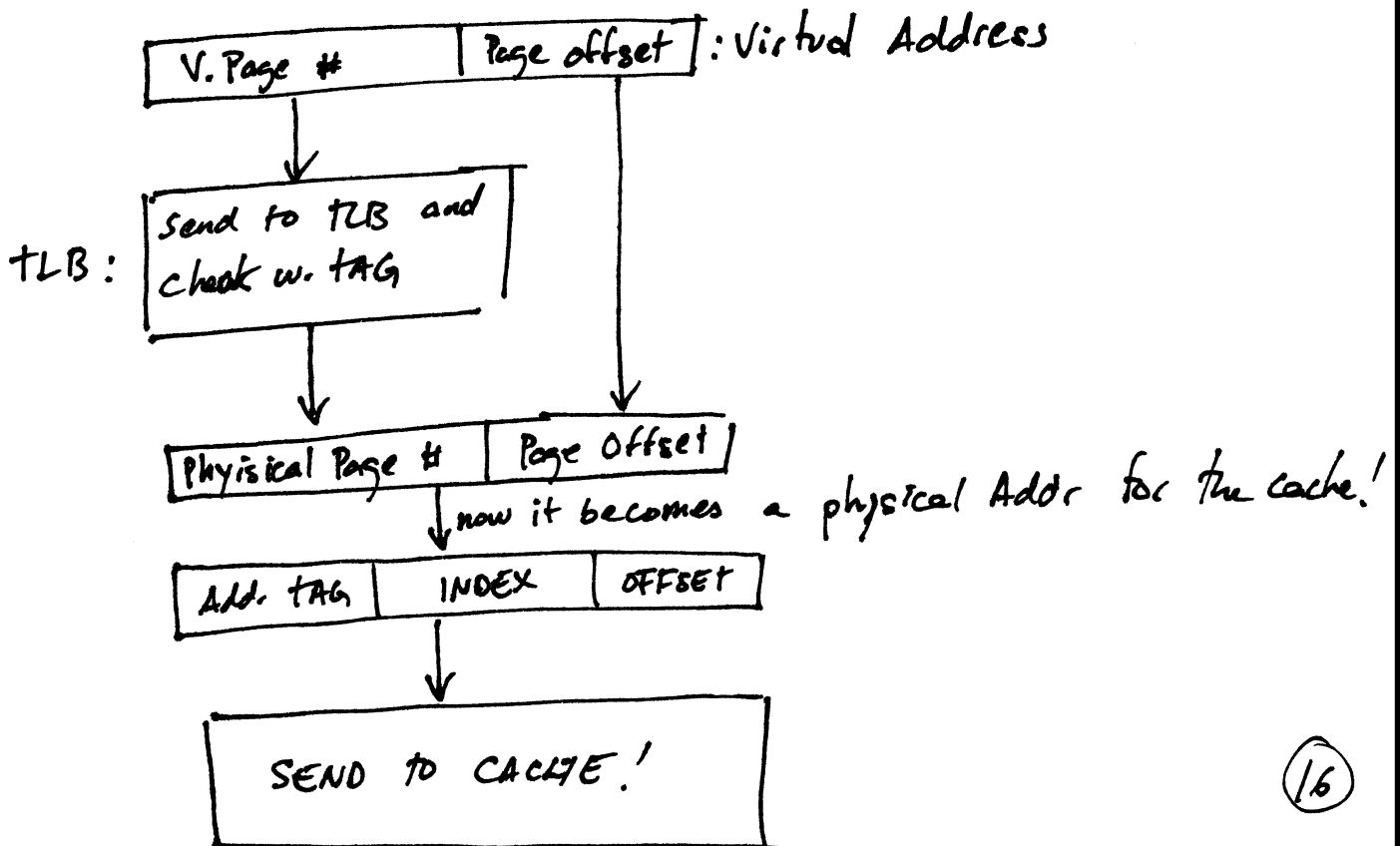


1. CPU sends Virtual Address
2. TLB translates to physical
(we may have a page fault, thus we must reload!)
3. if Hit in TLB, then just pass addr. to Cache
4. Cache returns data
(again we may have a ~~page~~ cache fault/miss)

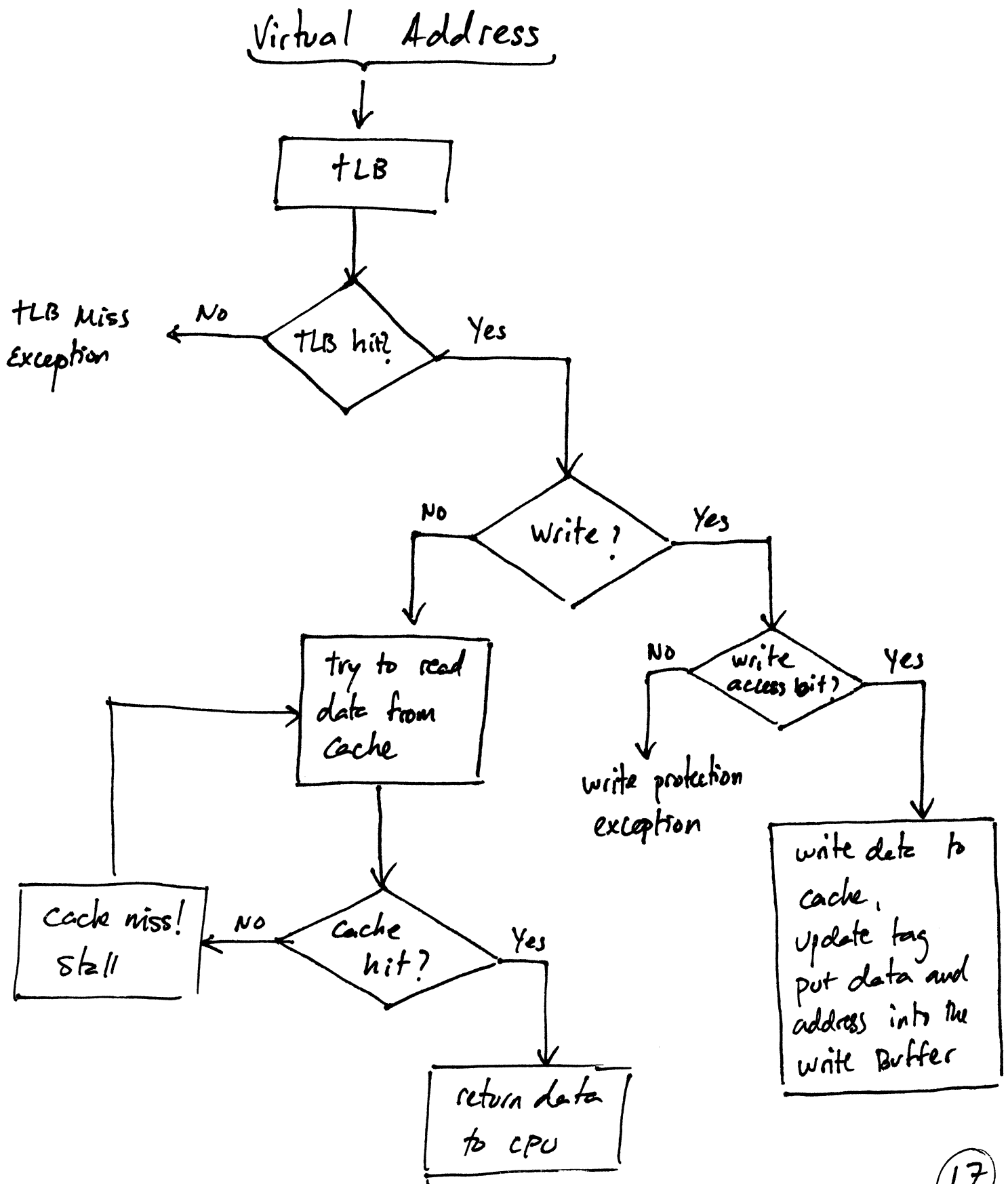
• TLB IS REALLY AN ASSOC. CACHE



• How does it INTERACT WITH CACHE?



DATA FLOW FOR THIS MISS



EXAMPLE :

Given the
following →
machine

	TLB	L1	L2
Size	16	8KB	128KB
Block Size	8KB	32B	32B
Associativity	16	2	2
Miss Penalty	200	60	150
Hit time	0	2	60

CODE 1

```
EIGHTBYTE DOUBLE M[64][64][64];
```

```
int i, j, k;
```

```
for (i=0; i<64; i++)
```

```
    for (j=0; j<64; j++)
```

```
        for (k=0; k<64; k++)
```

```
            M[i][j][k] += 1.0;
```

CODE 2

```
EIGHTBYTE DOUBLE M[64][64][64];
```

```
int i, j, k;
```

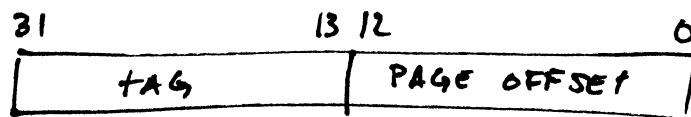
```
for (k=0; k<64; k++)
```

```
    for (j=0; j<64; j++)
```

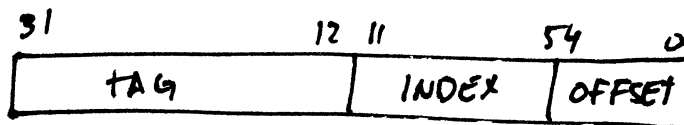
```
        for (i=0; i<64; i++)
```

```
            M[i][j][k] += 1.0;
```

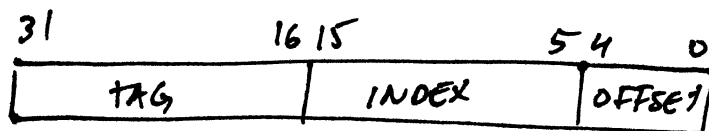
Q. TLB ADDRESS :



L1 ADDRESS :



L2 ADDRESS :



Q. Average access time for Data accesses?

$$T_{\text{ave Data}} = MR_{\text{TLB}} \cdot M_{\text{TLB}} + 2 + MR_{\text{HitL1}} (60 + MR_{\text{HitL2}} \cdot 150)$$

Q: What is the total data memory access time for code 1?

A: Note that the memory accesses are purely sequential!

Thus we will have a miss once per page & once per cache line.

of memory accesses = $2 \cdot 2^6 \cdot 2^6 \cdot 2^6 = \underline{2^{19}}$ total mem accesses

assume: each cache line holds 4 doubles.

each page holds 2^{10} doubles.

$$T_{\text{total}} = 2^{19} \cdot 2 \text{ns} + \frac{2^{18}}{4} \underbrace{(60 + 150)}_{\text{total cache miss}} + \frac{2^{18}}{2^{10}} \underbrace{(200)}_{\text{TLB penalty}}$$

easy!

Q: What is the total data access time for Code 2?

A: Note now data accesses are not sequential!

How much memory are we using?

2^{18} elements in the array, each 2^3 bytes

thus we are using 2^{21} bytes

How much does the TLB fit?

$$2^4 \cdot 2^{13} = 2^{17}$$

However by looking at the access pattern,
we see that we have a TLB miss + Cache miss
EVERY mem access (reads at least

1024 byte strides!

So:

$$T_{\text{total}} = \underline{2^{18} (2 + 60 + 150 + 200)}$$