In Lecture 3 .. Quiz 1

- Assignment 1 is due today
- There is a significant speed disparity between processor and main memory
- General purpose registers
  - C register declaration
- Cache memory
  - Design principle: Locality of reference
  - Operation
    - Address sent to cache controller
    - Lookup in cache directory

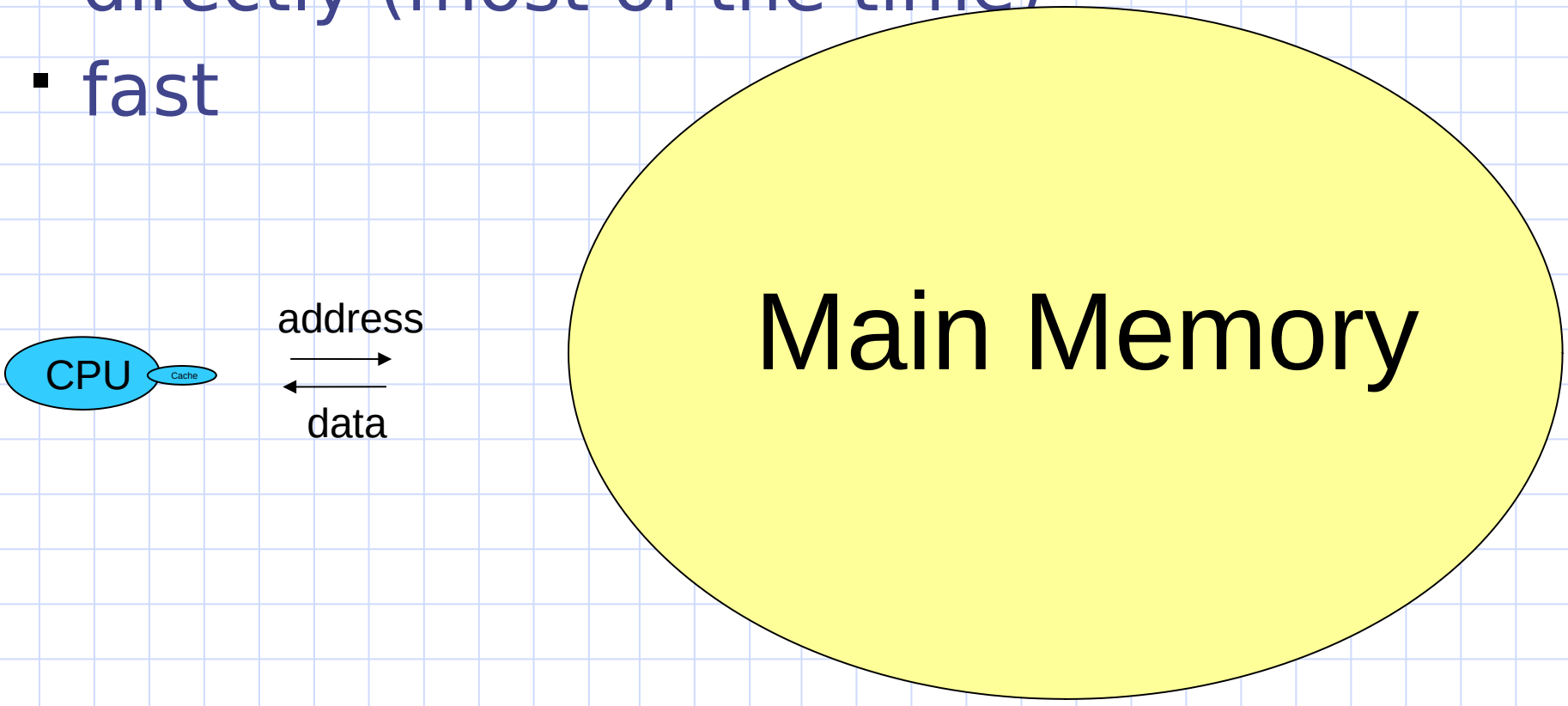
Assignment 2 (due 23/8)

- Solve these problems from B&O
  - 2.59
  - 2.61
  - 2.91(C)

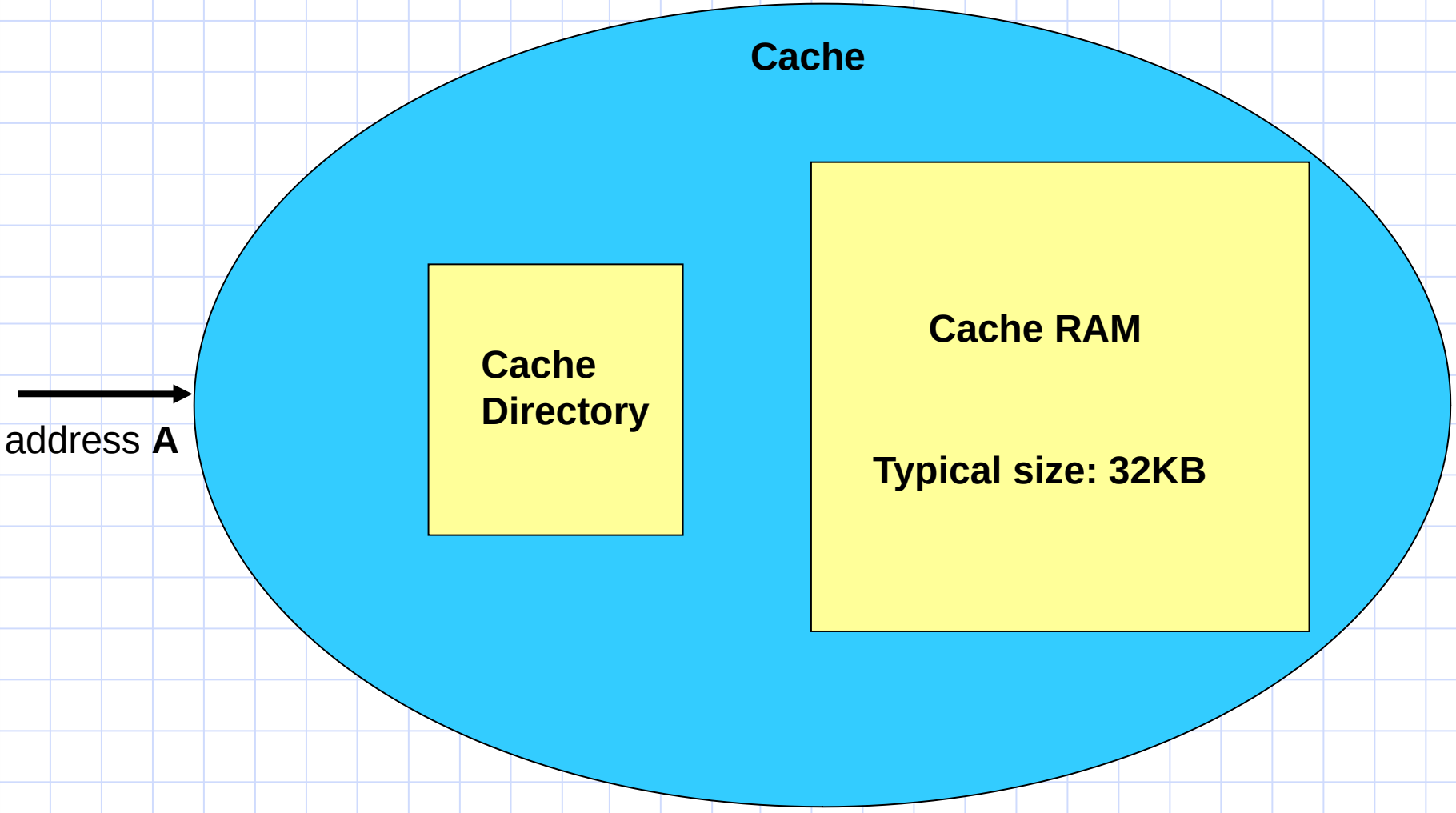
Cache exploits locality

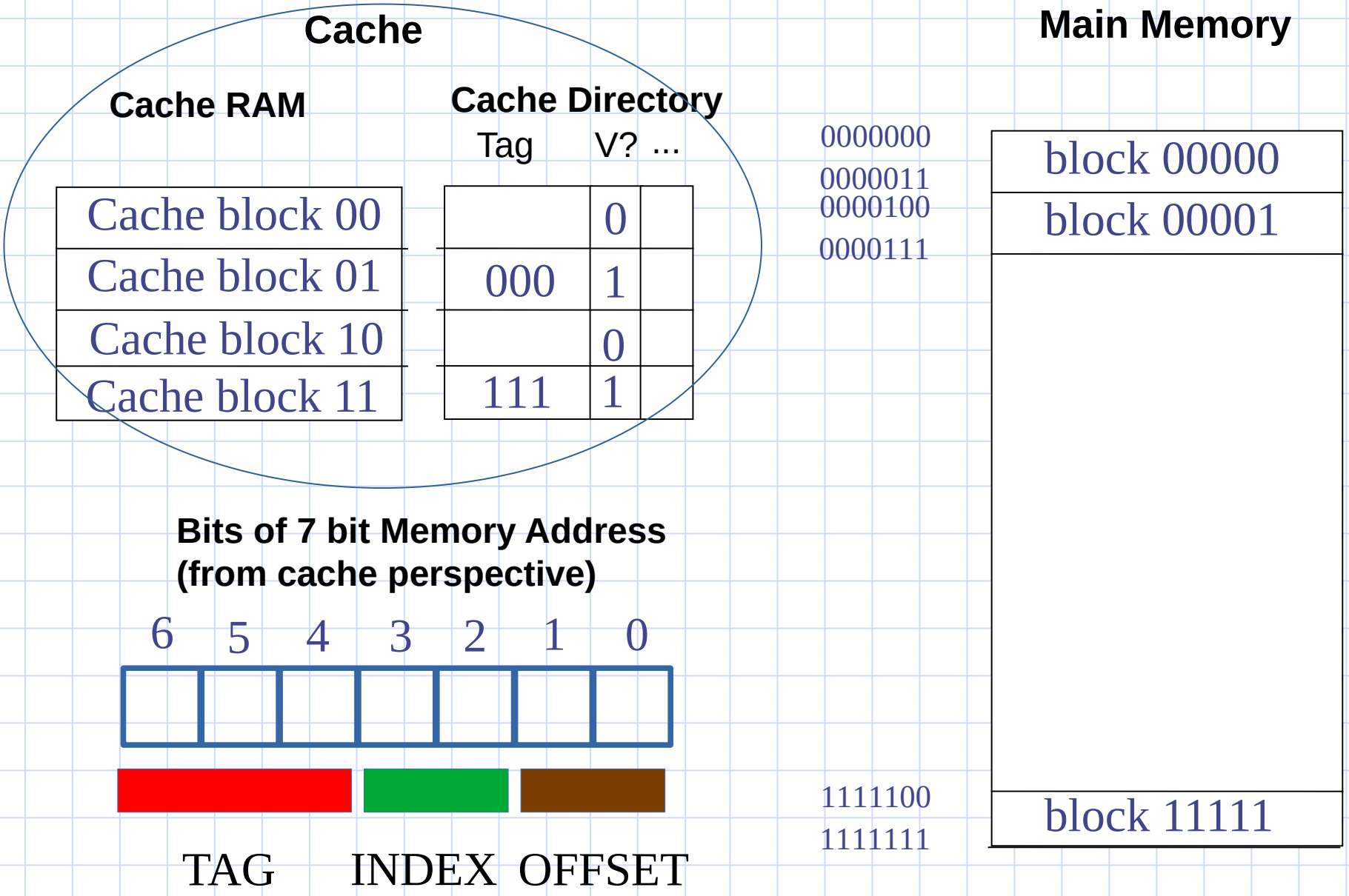
Cache: Hardware structure that provides memory contents that the processor wants to access

- directly (most of the time)
- fast

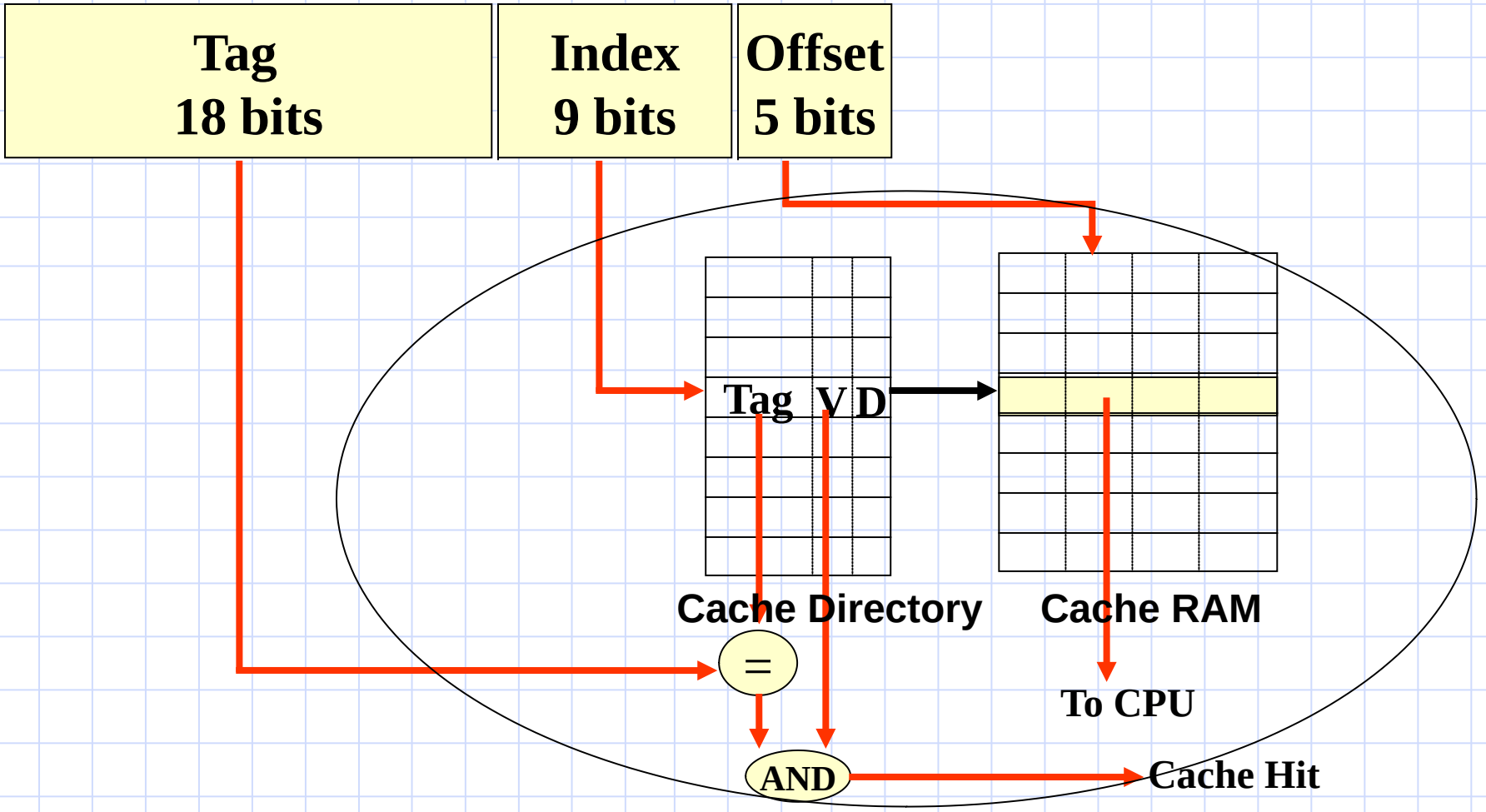


Cache Design

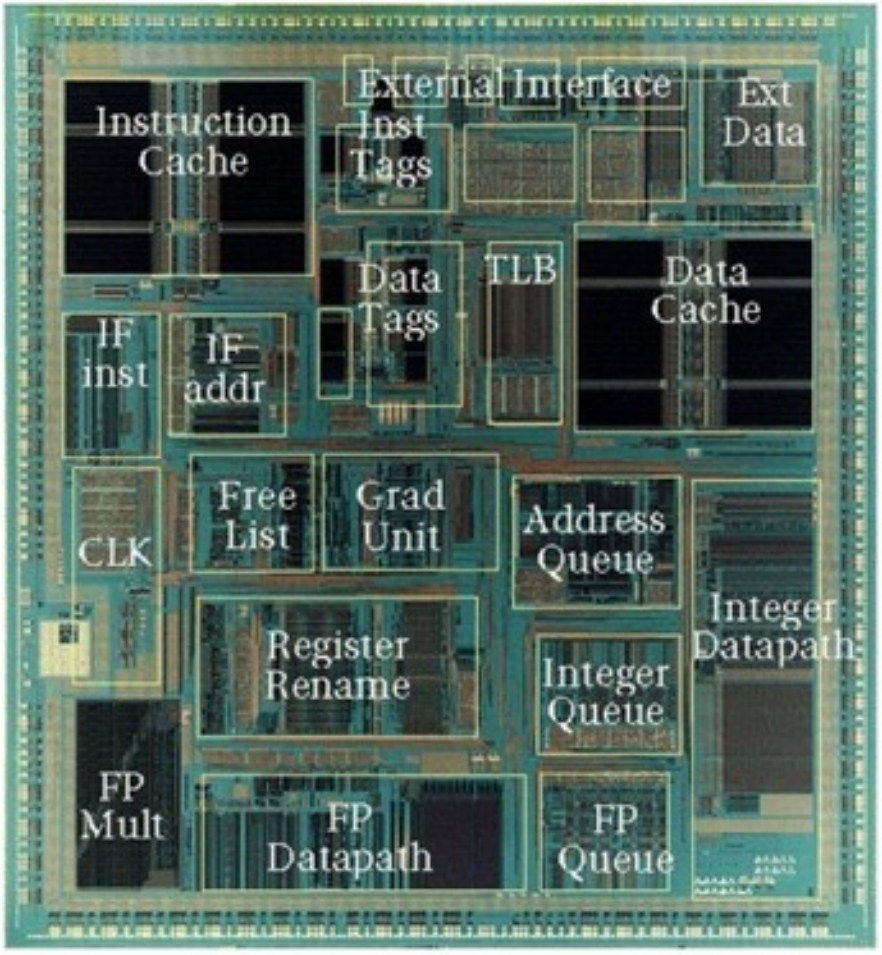



Cache
Cache RAM
Cache Directory
Tag
V?
...
Cache block 00
Cache block 01
Cache block 10
Cache block 11
0
000
1
0
111
1
Main Memory
0000000
0000011
0000100
0000111
block 00000
block 00001
1111100
1111111
block 11111
Bits of 7 bit Memory Address
(from cache perspective)
6 5 4 3 2 1 0
TAG
INDEX
OFFSET

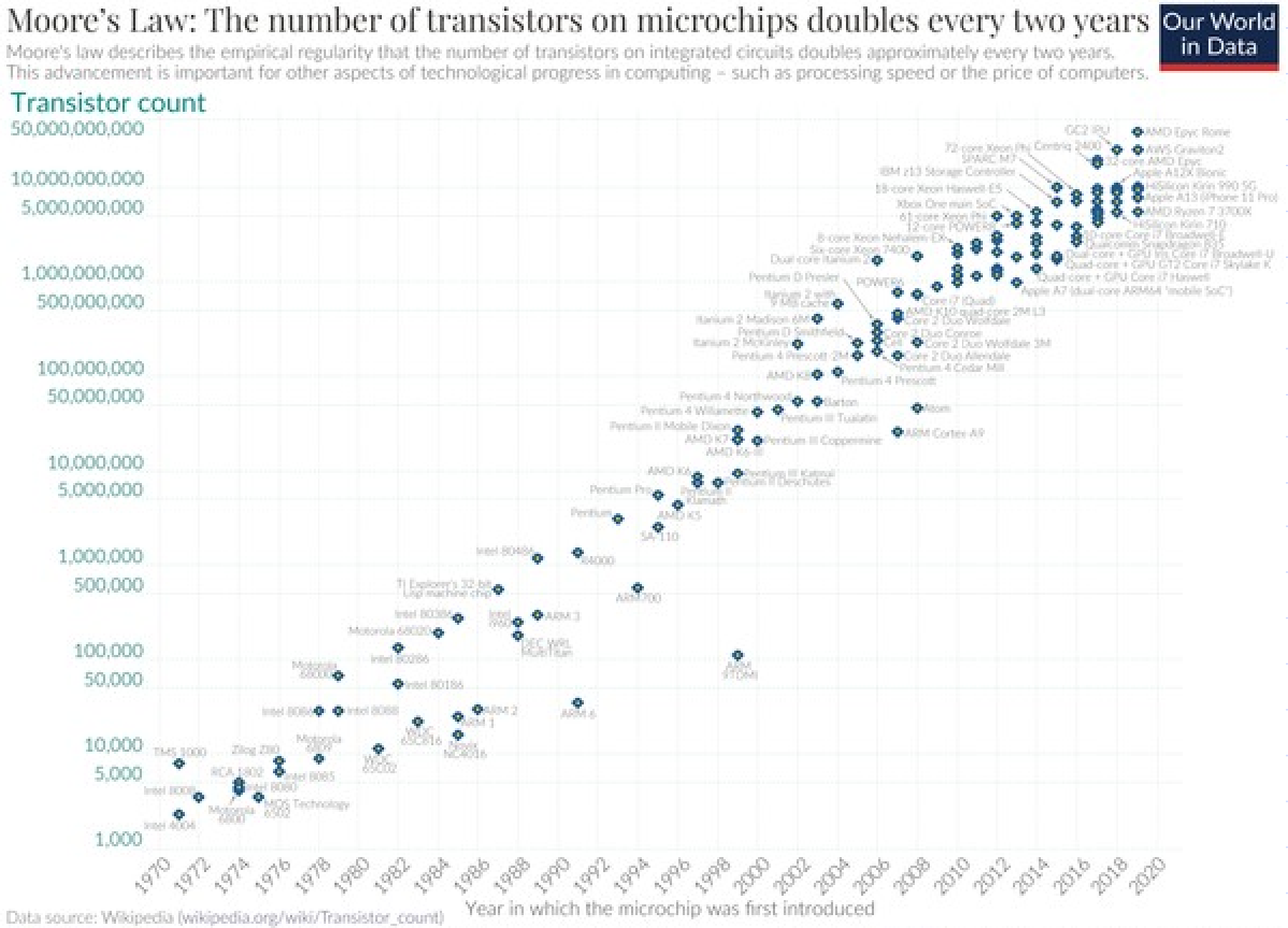
Cache Lookup and Access



MIPS R1000 Die Photo (1995)

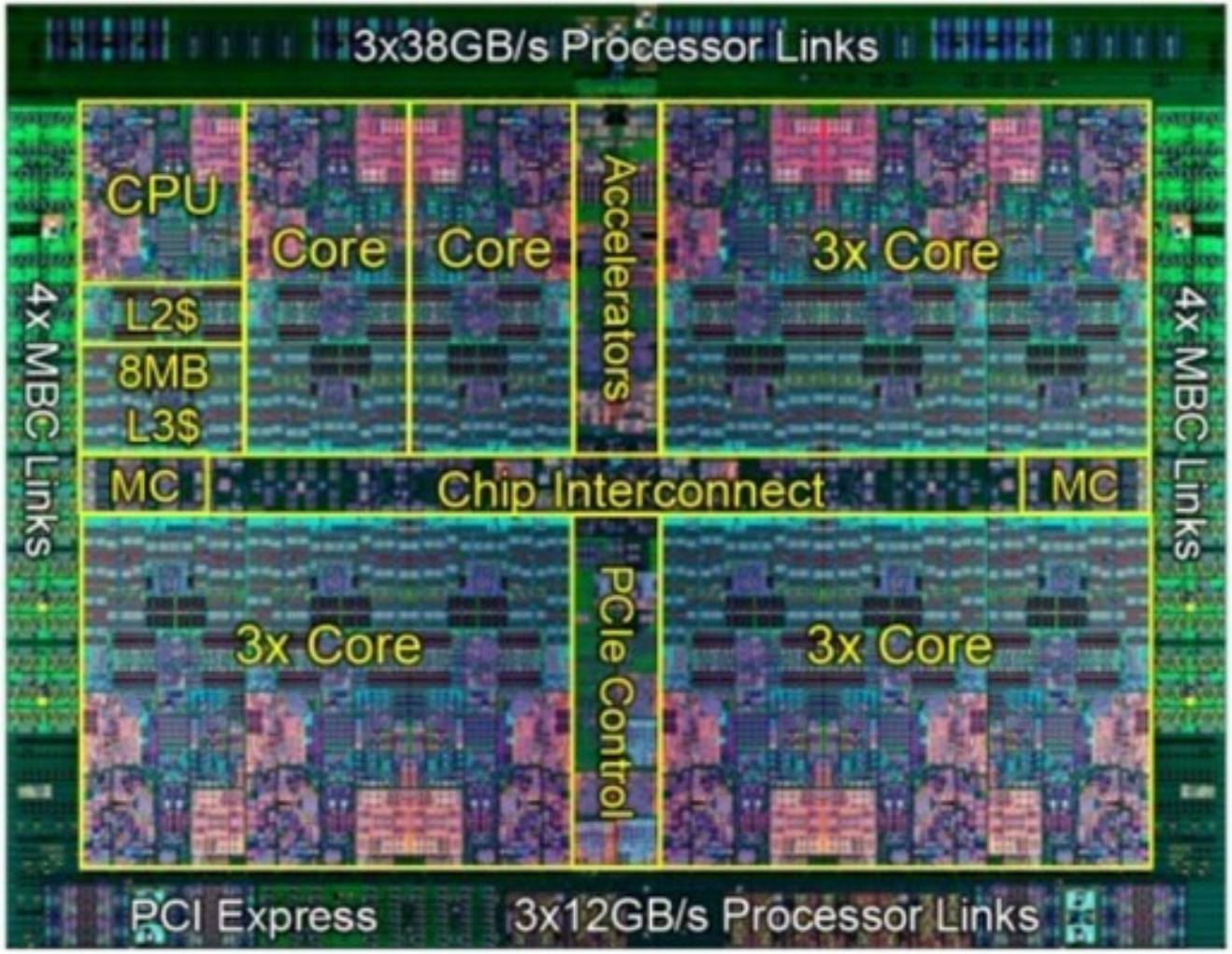


NEC D30700LRS-225 VR10000/ Toshiba TC86R10000-195



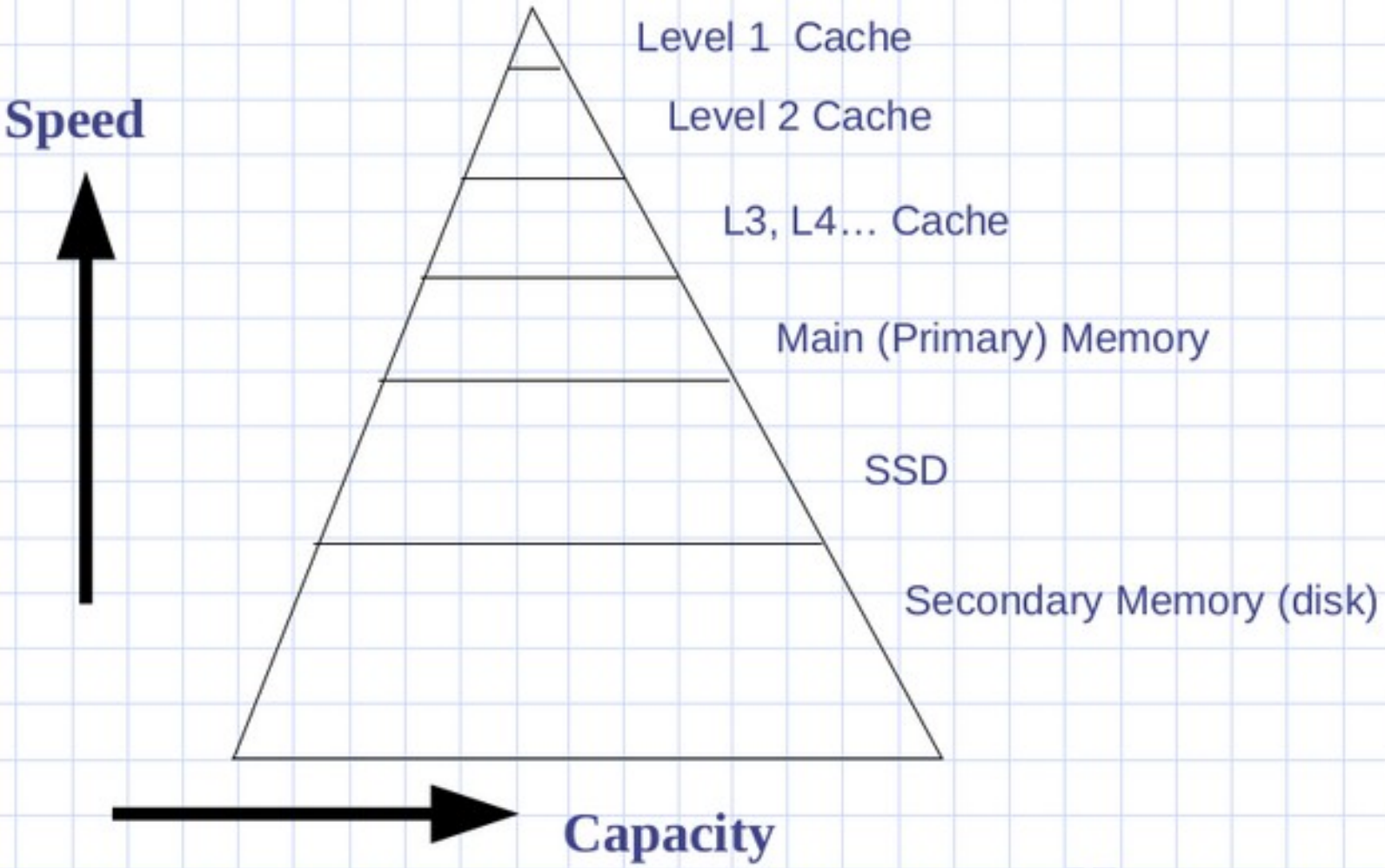
Wikipedia (Data source: OurWorldInData,org)

IBM Power8 Die Photo (2014)



https://www.extremetech.com/computing/181102-ibm-power8-openpower-x86-server-monopoly

Memory Hierarchy

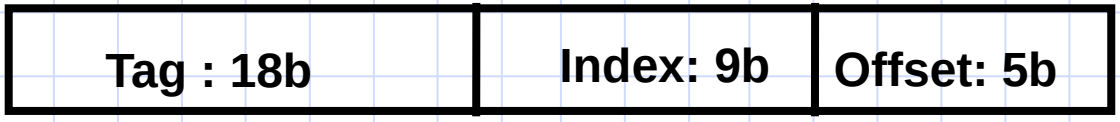


Cache Terminology

- Cache hit: A memory reference where the required data is found in the cache
- Cache Miss: A memory reference where the required data is not found in the cache
- Hit Ratio: Number of cache hits / Number of memory references

Cache and Programming

- Objective: Learn how to assess cache related performance issues for important parts of our programs
- Will look at some examples of programs
- Will consider only data cache, assuming separate instruction and data caches
- Data cache configuration:
  - Direct mapped 16 KB with 32B block size



Example 1: Vector Sum Reduction

```
double A[2048], sum=0.0;
for (i=0; i<2048, i++) sum = sum +A[i];
```

- To do analysis, must view program close to machine code form (to see loads/stores)
  - Will assume that both loop index i and variable sum are implemented in registers
- Will consider only accesses to array elements

Example 1: Reference Sequence

- load A[0] load A[1] load A[2] … load A[2047]
- Assume base address of A (i.e., address of A[0]) is 0xA000, 1010 0000 0000 0000
  - Cache index bits: 100000000 (value = 256)
- Size of an array element (double) = 8B
- So, 4 consecutive array elements fit into each cache block (block size is 32B)
  - A[0] – A[3] have index of 256
  - A[4] – A[7] have index of 257 and so on

Example 1: Cache Misses and Hits

A[0]	0xA000	256	Miss	Cold Start
A[1]	0xA008	256	Hit	
A[2]	0xA010	256	Hit	
A[3]	0xA018	256	Hit	
A[4]	0xA020	257	Miss	Cold Start

```
for (i=0; i<2048; i+=4)
   tmp=A[i];
for (i=0; i<2048, i++)
  sum = sum +A[i];
```

Cold start: we assume that the cache is initially empty

Hit ratio of our loop is 75% -- there are 1536 hits out of 2048 memory accesses

This is entirely due to spatial locality of reference.

What if we precede the loop by a loop that accesses all relevant memory blocks?

Hit ratio of our loop would then be 100%. 25% due to temporal locality and 75% due to spatial locality

Example 1 with double A[4096]

Why should it make a difference?

- Consider the case where the loop is preceded by another loop that accesses all array elements in order
- The entire array no longer fits into the cache – cache size: 16KB, array size: 32KB
- After execution of the previous loop, the second half of the array will be in cache
- Analysis: our loop will see misses as we had calculated

Example 2: Vector Dot Product

```
double A[2048], B[2048], sum=0.0;
for (i=0; i<2048, i++) sum = sum +A[i] * B[i];
```

- Reference sequence:
  - load A[0] load B[0] load A[1] load B[1] …
- Assume base addresses of A and B are 0xA000 and 0xE000
- Again, size of array elements is 8B so that 4 consecutive array elements fit into each cache block

Example 2: Vector Dot Product

Base addresses 0xA000 and 0xE000

.....1010000000000000

Index: 256

.....1110000000000000

Index: 256

Example 2: Cache Hits and Misses

A[0]	0xA000	256	Miss	Cold Start
B[0]	0xE000	256	Miss	Conflict
A[1]	0xA008	256	Miss	Conflict
B[1]	0xE008	256	Miss	Conflict
A[2]	0xA010	256	Miss	Conflict

Conflict: A miss due to conflict in cache block requirements caused by memory accesses of the same program

Hit ratio for our program: 0%

Source of the problem: the elements of arrays A and B are accessed in order and have the same cache index

Hit ratio would be better if the base address of array A was different from that of array B

Is this a contrived example?

```
double A[2048], B[2048], sum=0.0;
for (i=0; i<2048, i++) sum = sum +A[i] * B[i];
```

- How are variable addresses assigned?
- Start with some address, say 0xA000
- Assign addresses to variables in order of their declarations
- Array A: starting at 0xA000
- Array B: starting at 0xA000+ 2048*8

  = 0xE000

```
1010 0000 0000 0000
 100 0000 0000 0000
---------------------
1110 0000 0000 0000
```

Example 2: Cache Hits and Misses

A[0]	0xA000	256	Miss	Cold Start
B[0]	0xE000	256	Miss	Conflict
A[1]	0xA008	256	Miss	Conflict
B[1]	0xE008	256	Miss	Conflict
A[2]	0xA010	256	Miss	Conflict

Conflict: A miss due to conflict in cache block requirements caused by memory accesses of the same program

Hit ratio for our program: 0%

Source of the problem: the elements of arrays A and B are accessed in order and have the same cache index

Hit ratio would be better if the base address of array A was different from that of array B

Example 2 with Packing

- Assume that addresses are assigned as variables are encountered in declarations
- Our objective: to shift base address of B enough to make cache index of B[0] different from that of A[0]

  double A[2052], B[2048];
- Base address of B is now 0xE020
  - 0xE020 is 1110 0000 0010 0000
  - Cache index of B[0] is 257; B[0] and A[0] do not conflict for the same cache block
- Hit ratio of our loop will rise to 75%

Example 2: 0% vs 75%

```
double A[2048], B[2048], sum=0.0;
for (i=0; i<2048, i++) sum = sum +A[i] * B[i];
```

- Suppose there are 20 instructions in the loop, 2 loads + 18 others
- Total instructions executed: 20*2048
- Suppose each instruction takes 1 cycle to execute, and load miss takes 100 cycles
  - 100%: 40,960 cycles
  - 0%: 446,464 cycles
  - 75%: 142,336 cycles

Example 2 with Array Merging

Alternatively, declare the arrays as

```
struct {double A, B;} array[2048];
for (i=0; i<2048, i++)
        sum += array[i].A*array[i].B;
```

Hit ratio: 75%