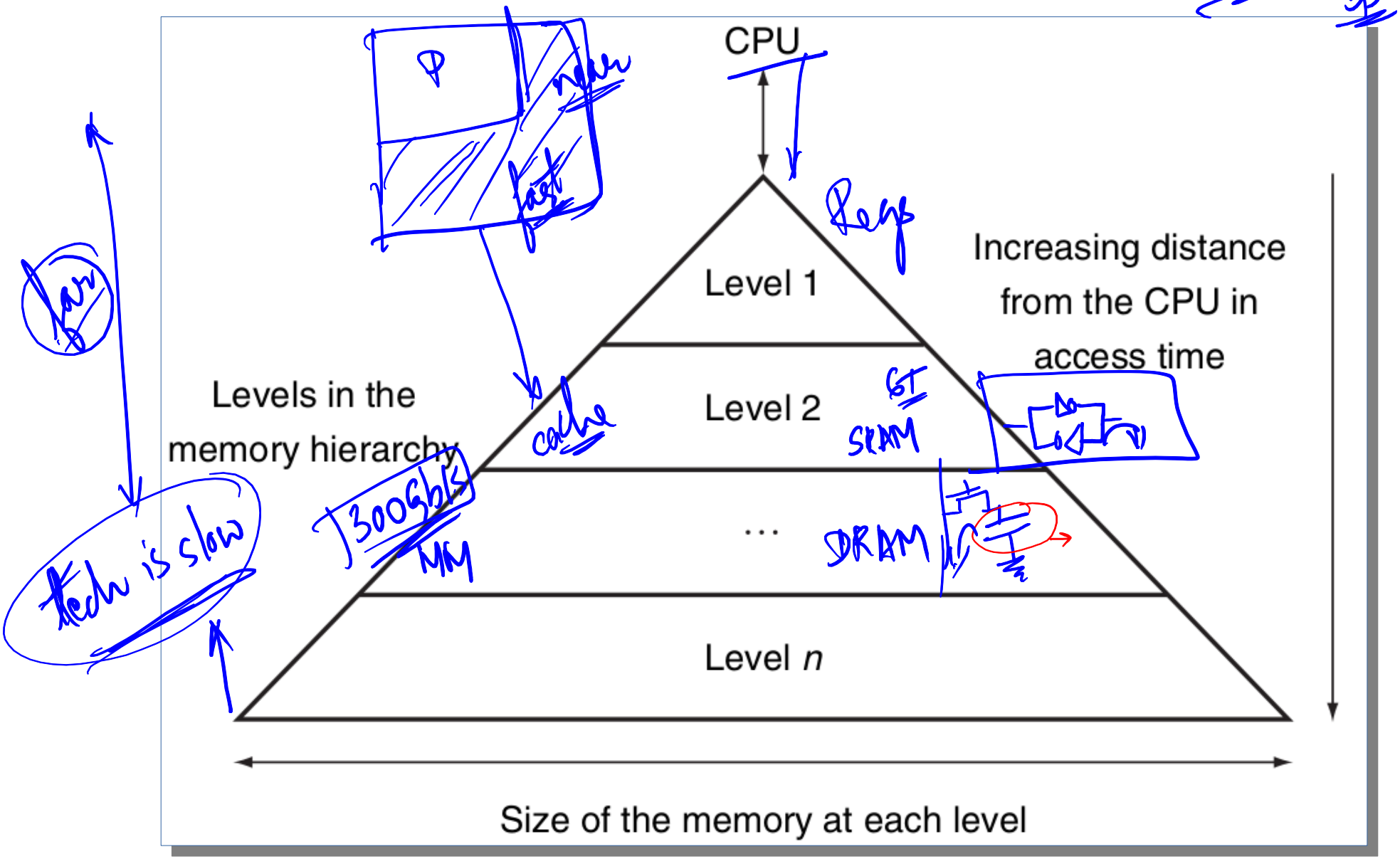
M2 – Memory Systems

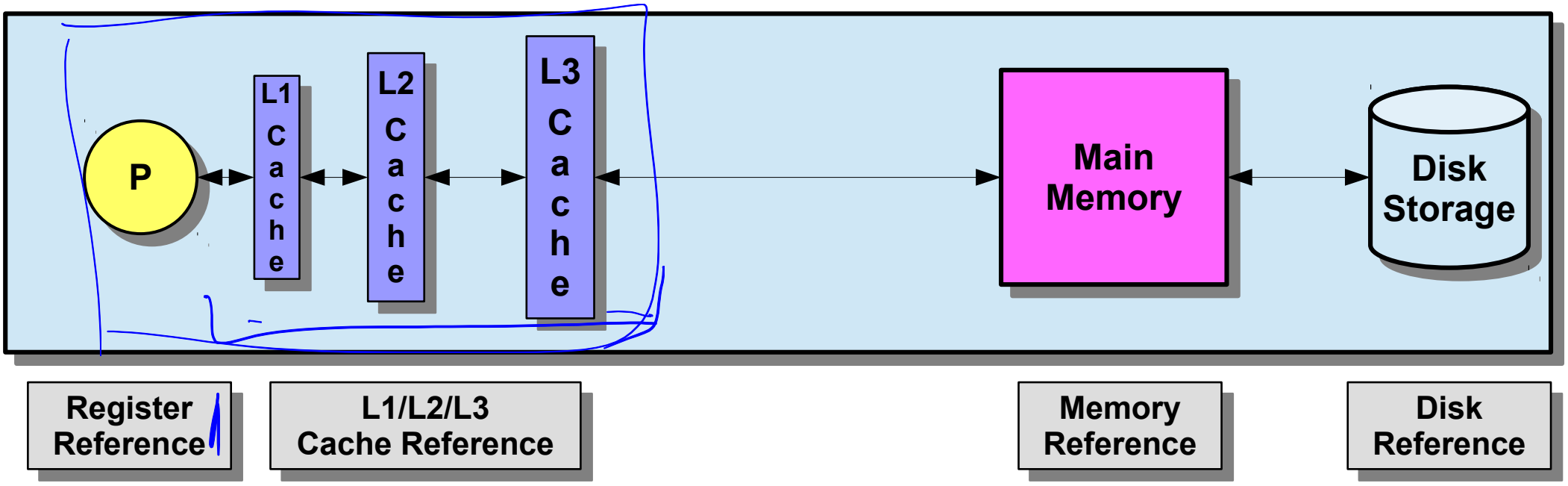
M2 – Outline

- Memory Hierarchy
- Cache Blocking – Cache Aware Programming
- SRAM, DRAM
- Virtual Memory
- Virtual Machines
- Non-volatile Memory, Persistent NVM

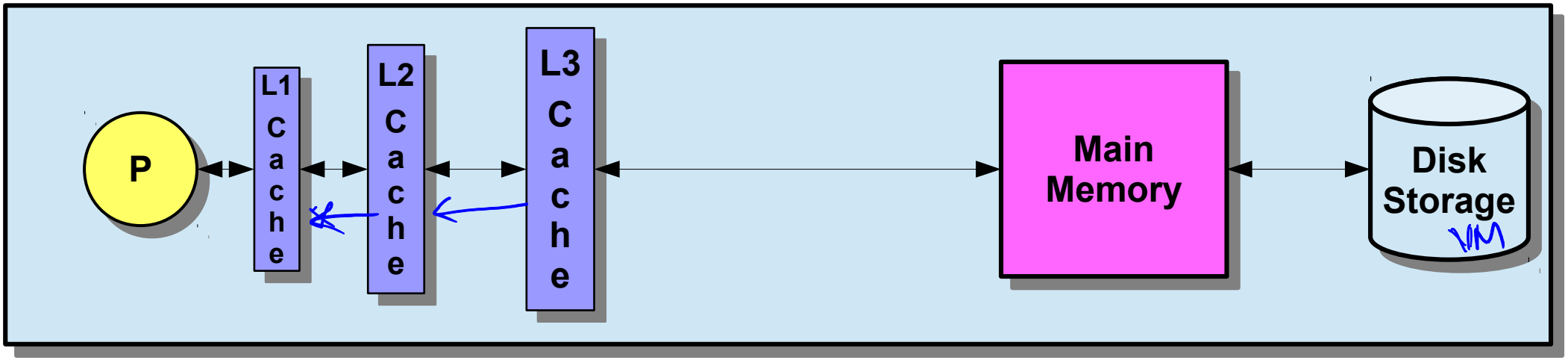
Memory Hierarchy



Memory Hierarchy

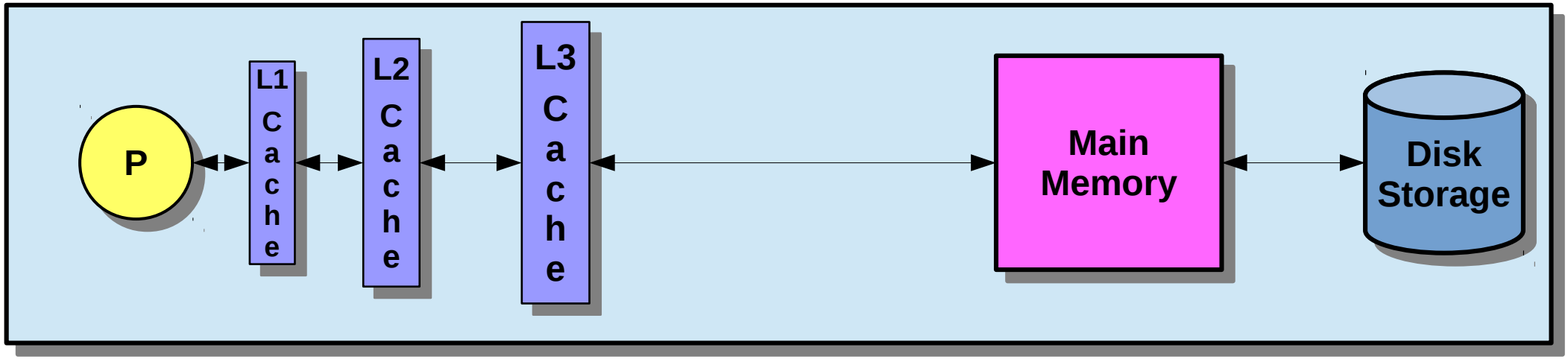


Memory Hierarchy



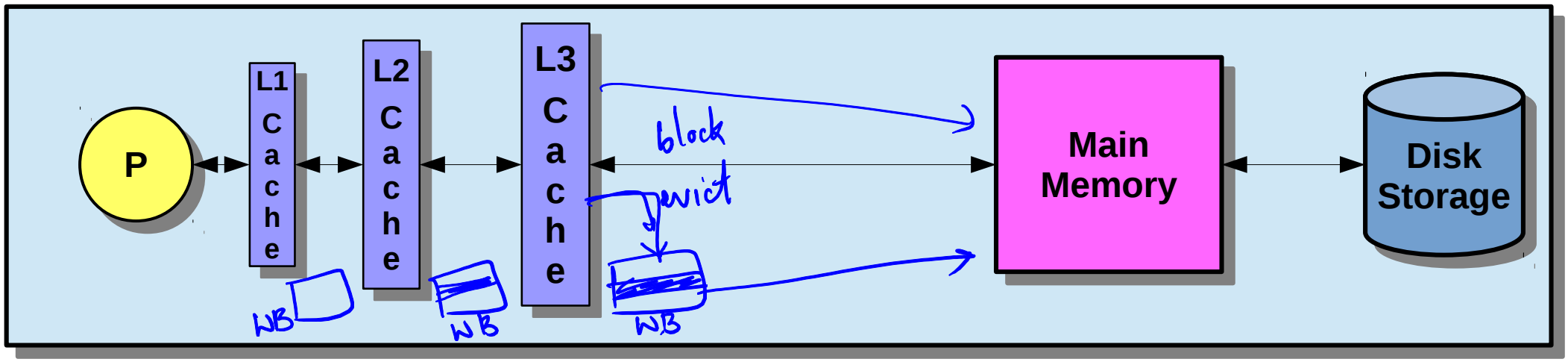
Register Reference	L1/L2/L3 Cache Reference	Memory Reference	Disk Reference
32B – 64B Word	16B – 64B Cache Block		4KB – 16KB Page

Memory Hierarchy



Register Reference	L1/L2/L3 Cache Reference	Memory Reference	Disk Reference
32B – 64B Word	16B – 64B Cache Block	4KB – 16KB Page	
1000 B	64 KB	256KB–1MB	4–16 MB	4-16 GB	4-16 TB

Memory Hierarchy



Register Reference	L1/L2/L3 Cache Reference			Memory Reference	Disk Reference
32B – 64B Word			16B – 64B Cache Block		4KB – 16KB Page
1000 B	64 KB	256KB–1MB	4–16 MB	4-16 GB	4-16 TB
300ps	1 ns	3 - 10 ns	10 - 25 ns	50 - 100 ns	5 - 10 ms

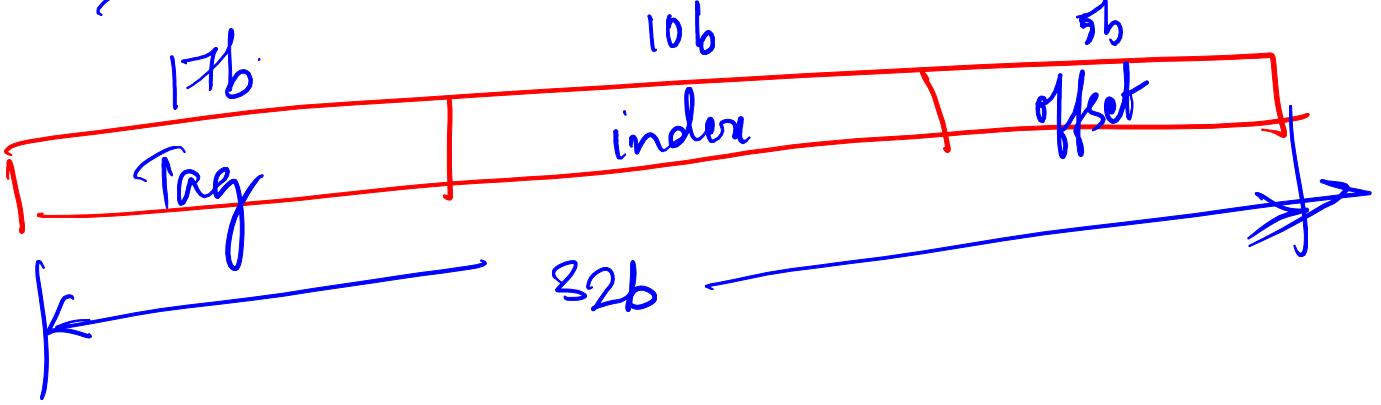
Question

- Identify the Block Index in a 32KB Direct Mapped cache with 32B cache lines, using 32b addresses. Address = 0xABCD1010.

DM

* tag = 17b
* index = 10b
* offset = 5b = 32B cache line
  2^5 B line

No. of cache lines = $\frac{32 \times 2^{10} B}{32 B} = 2^{10}$

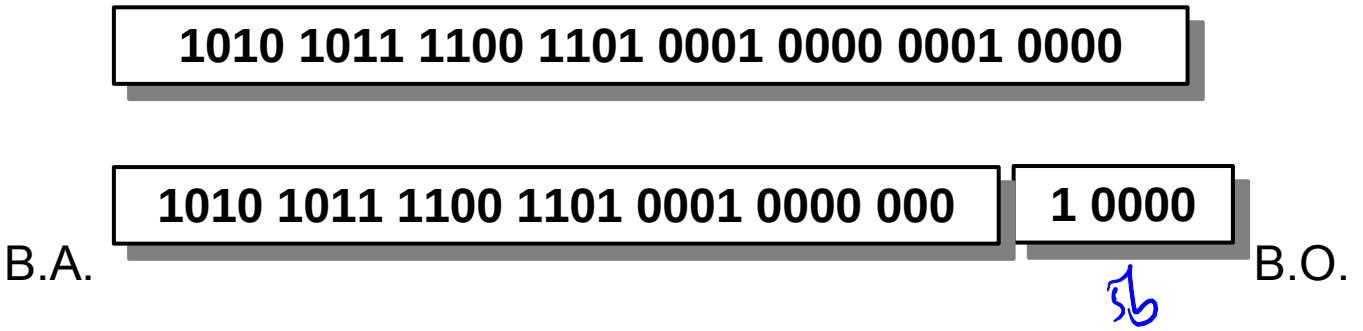


Question

- Identify the Block Index in a 32KB Direct Mapped cache with 32B cache lines, using 32b addresses. Address = 0xABCD1010.

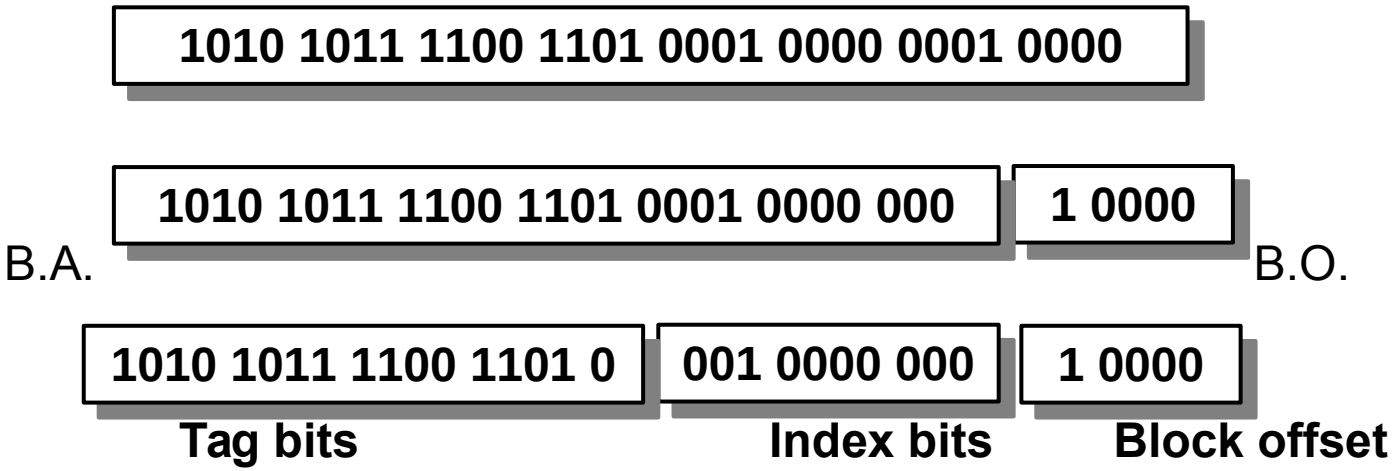
Question

- Identify the Block Index in a 32KB Direct Mapped cache with 32B cache lines, using 32b addresses. Address = 0xABCD1010.
- 1024 cache lines



Question

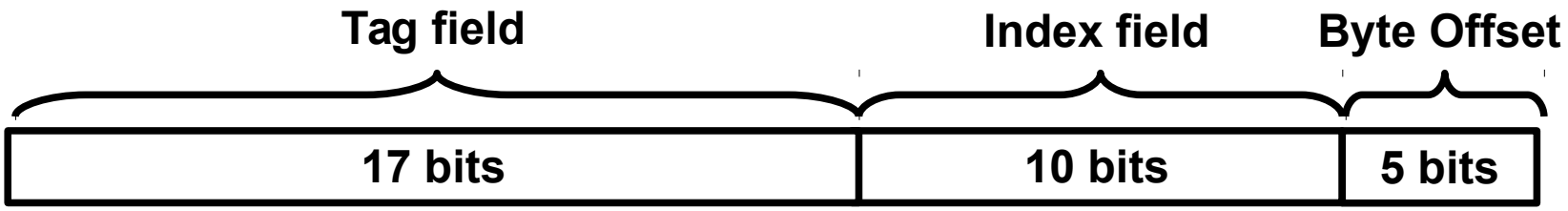
- Identify the Block Index in a 32KB Direct Mapped cache with 32B cache lines, using 32b addresses. Address = 0xABCD1010.
- 1024 cache lines



Question

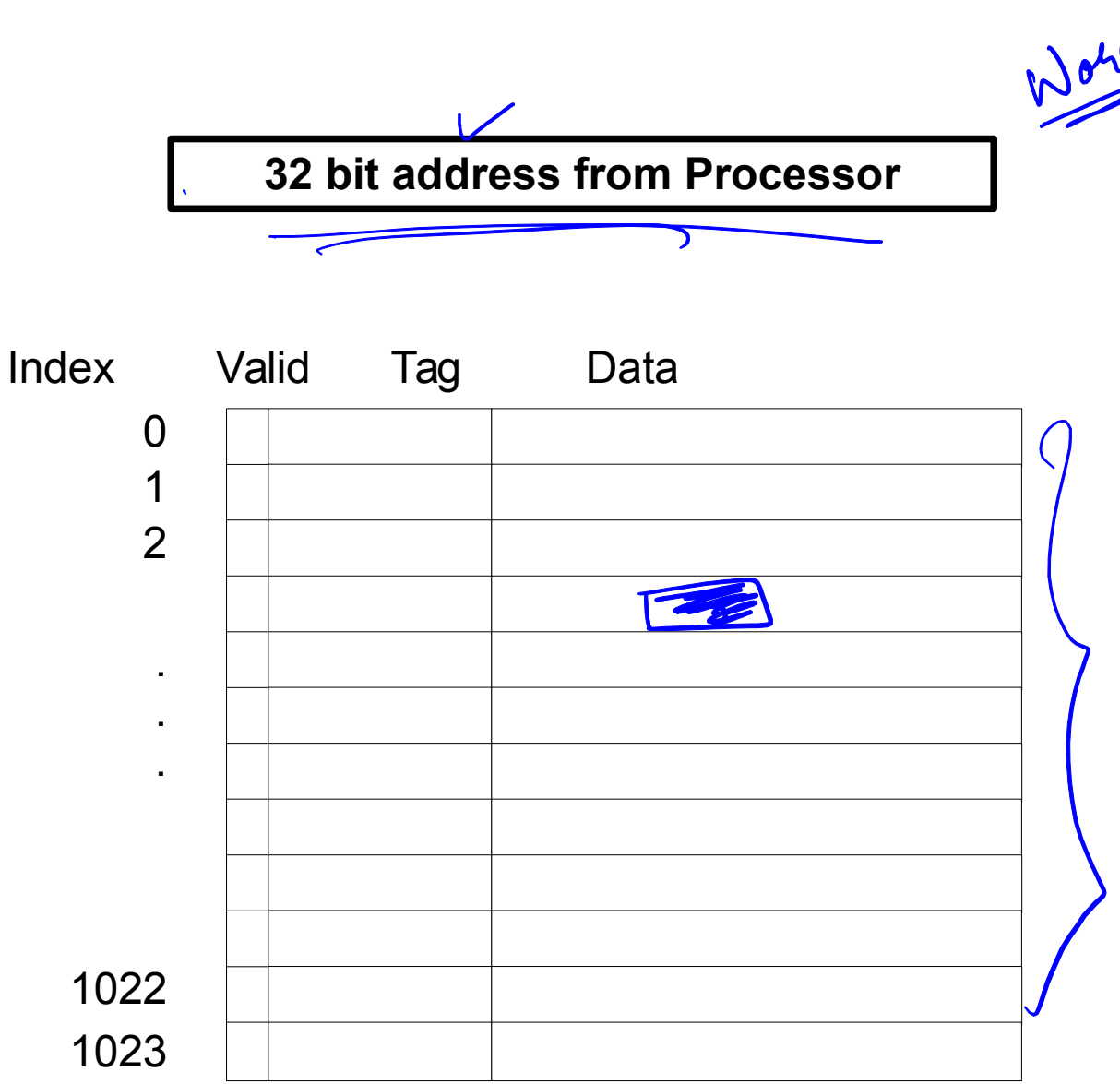
- Identify the Block Index in a 32KB Direct Mapped cache with 32B cache lines, using 32b addresses. Address = 0xABCD1010.
- 1024 cache lines

2^{10}

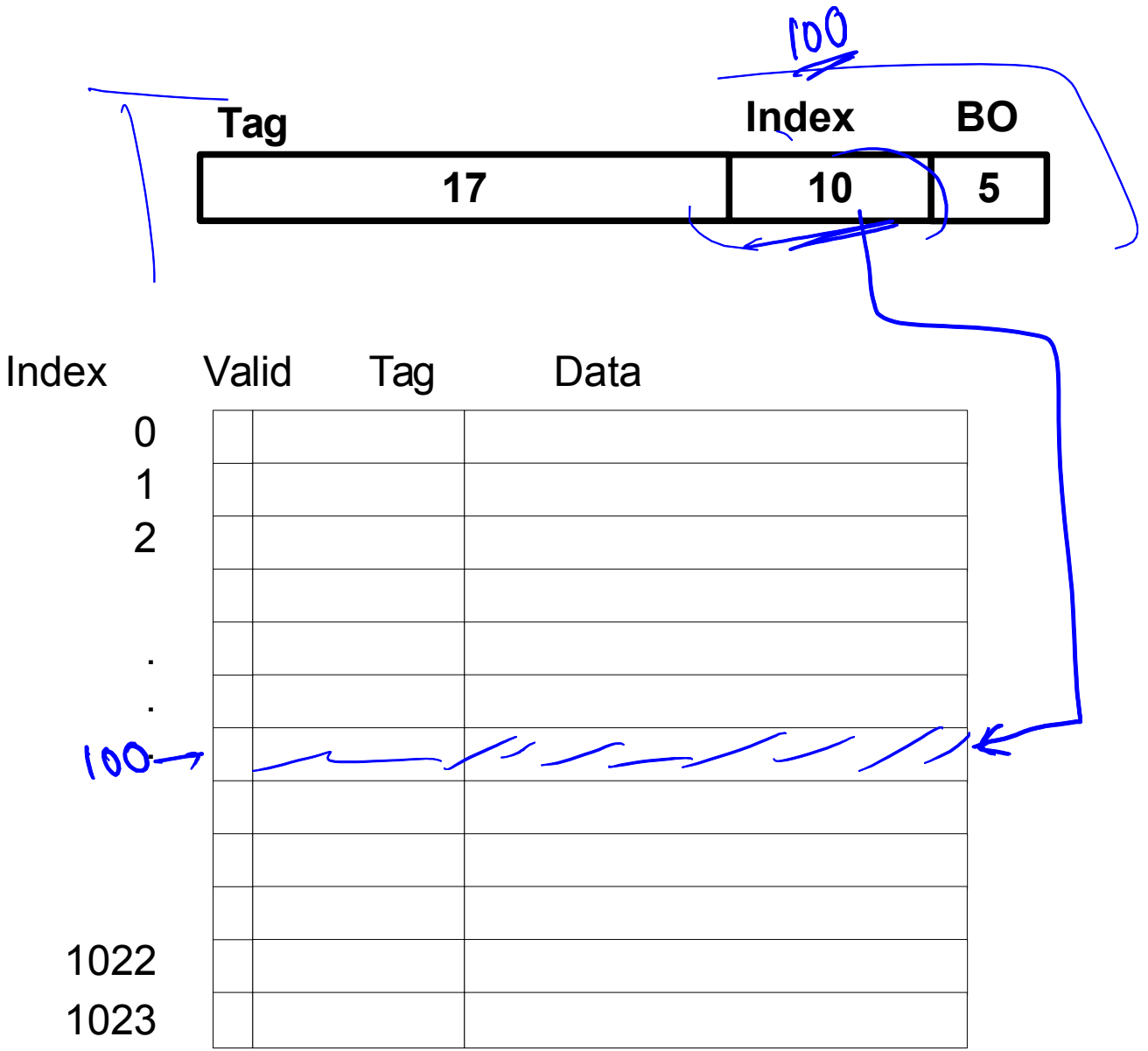


Direct Mapped Cache Organization

Direct Mapped Cache Organization

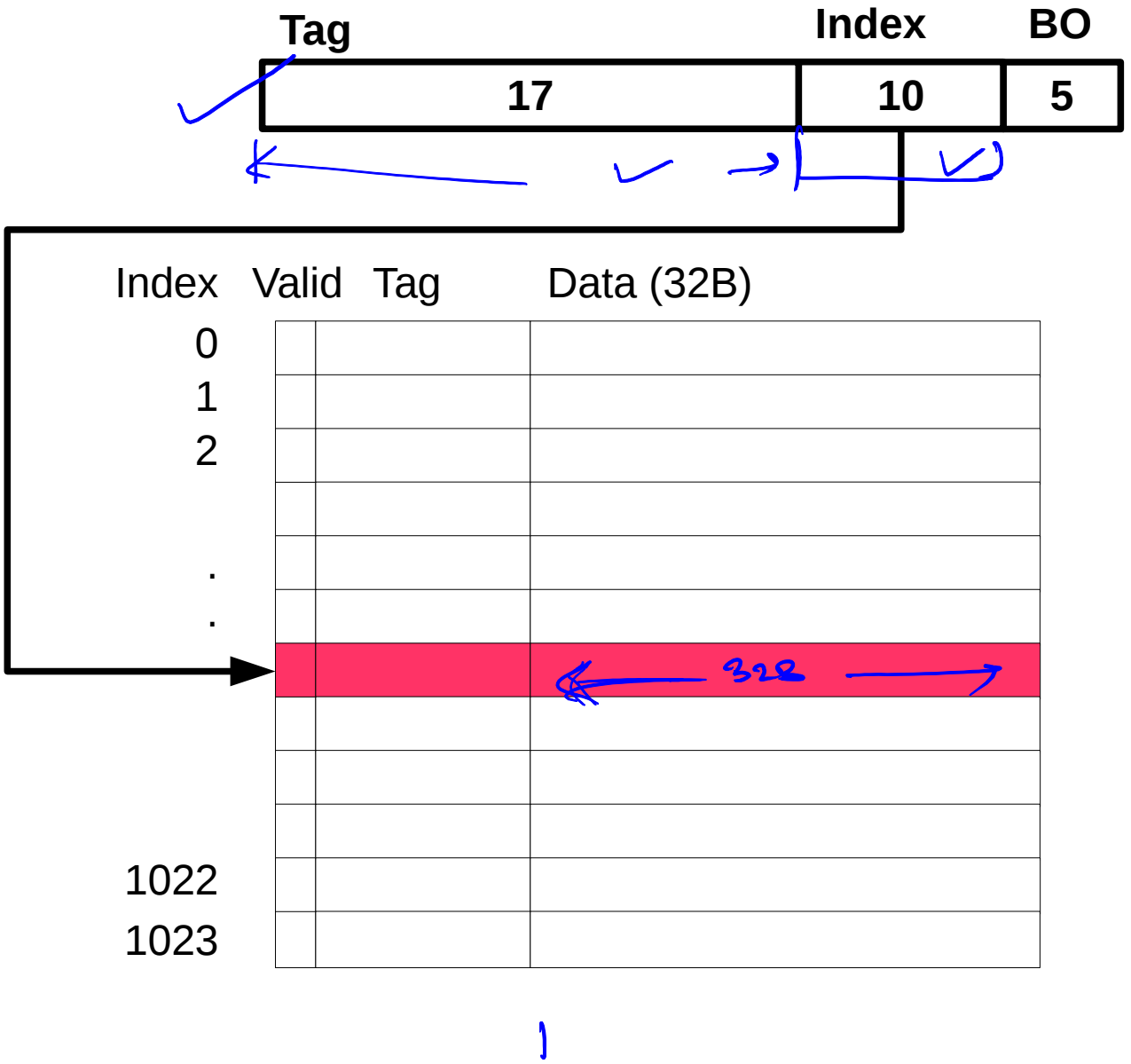


Direct Mapped Cache Organization

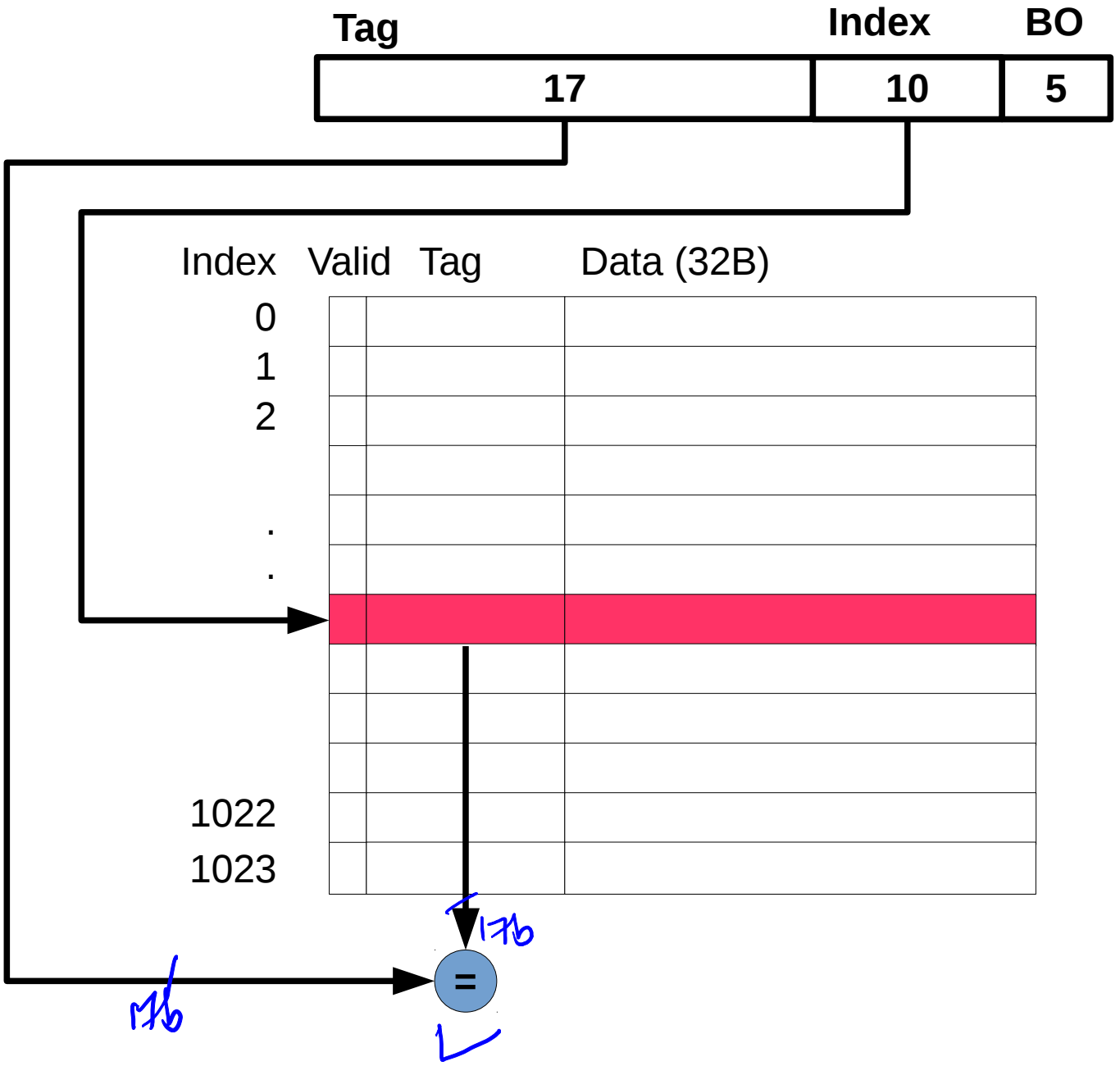


Direct Mapped Cache Organization

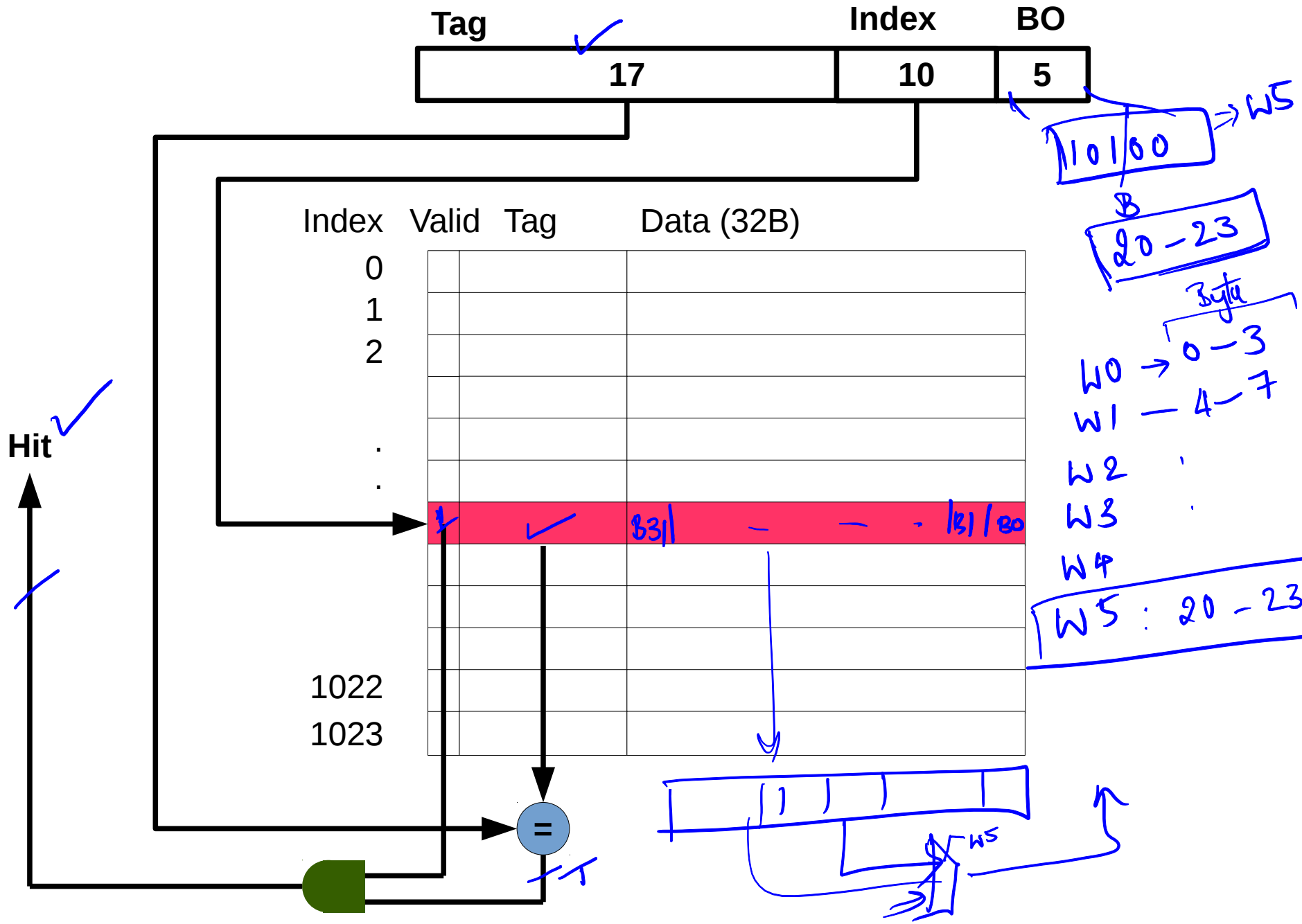
Direct Mapped Cache Organization



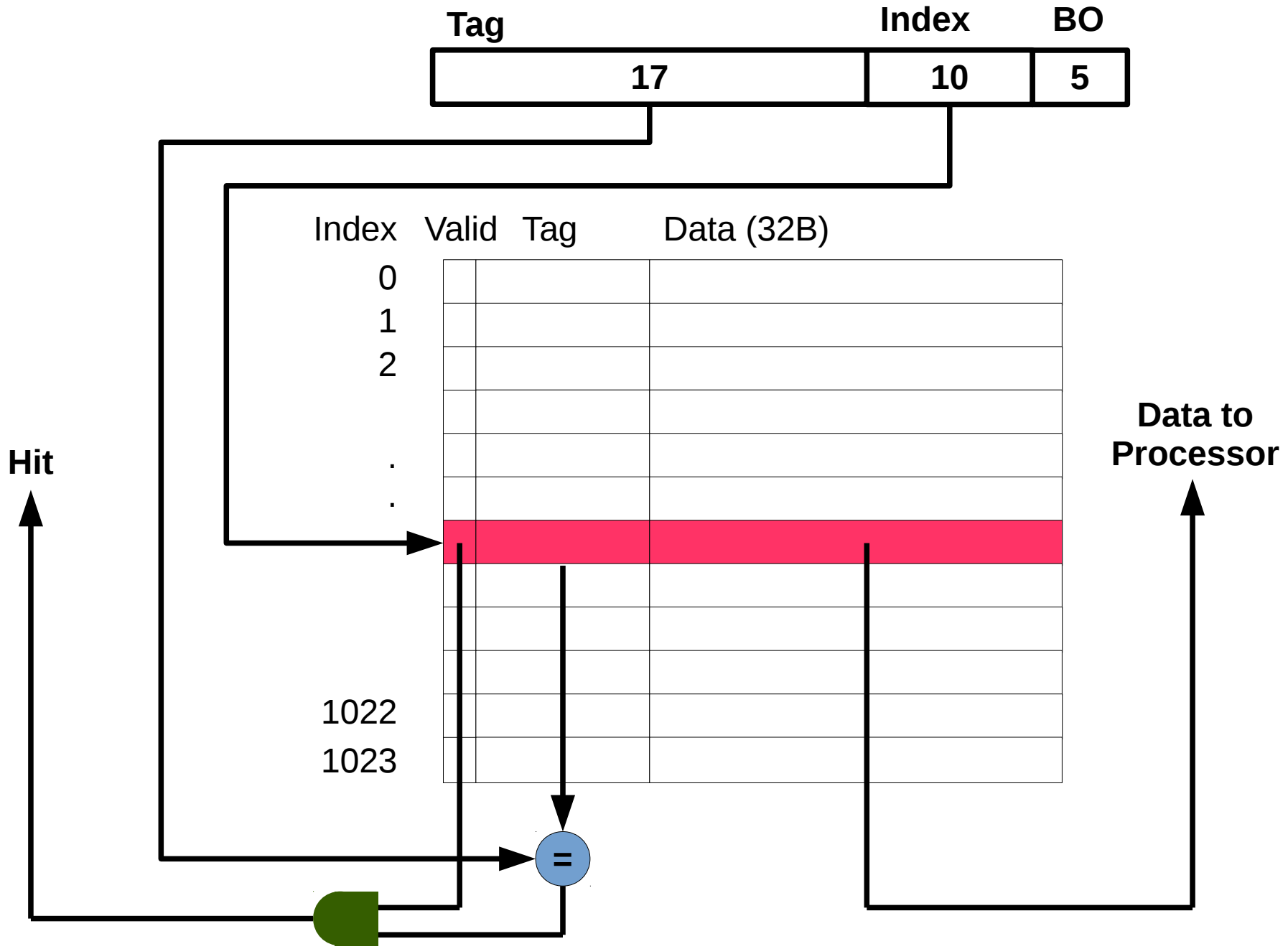
Direct Mapped Cache Organization



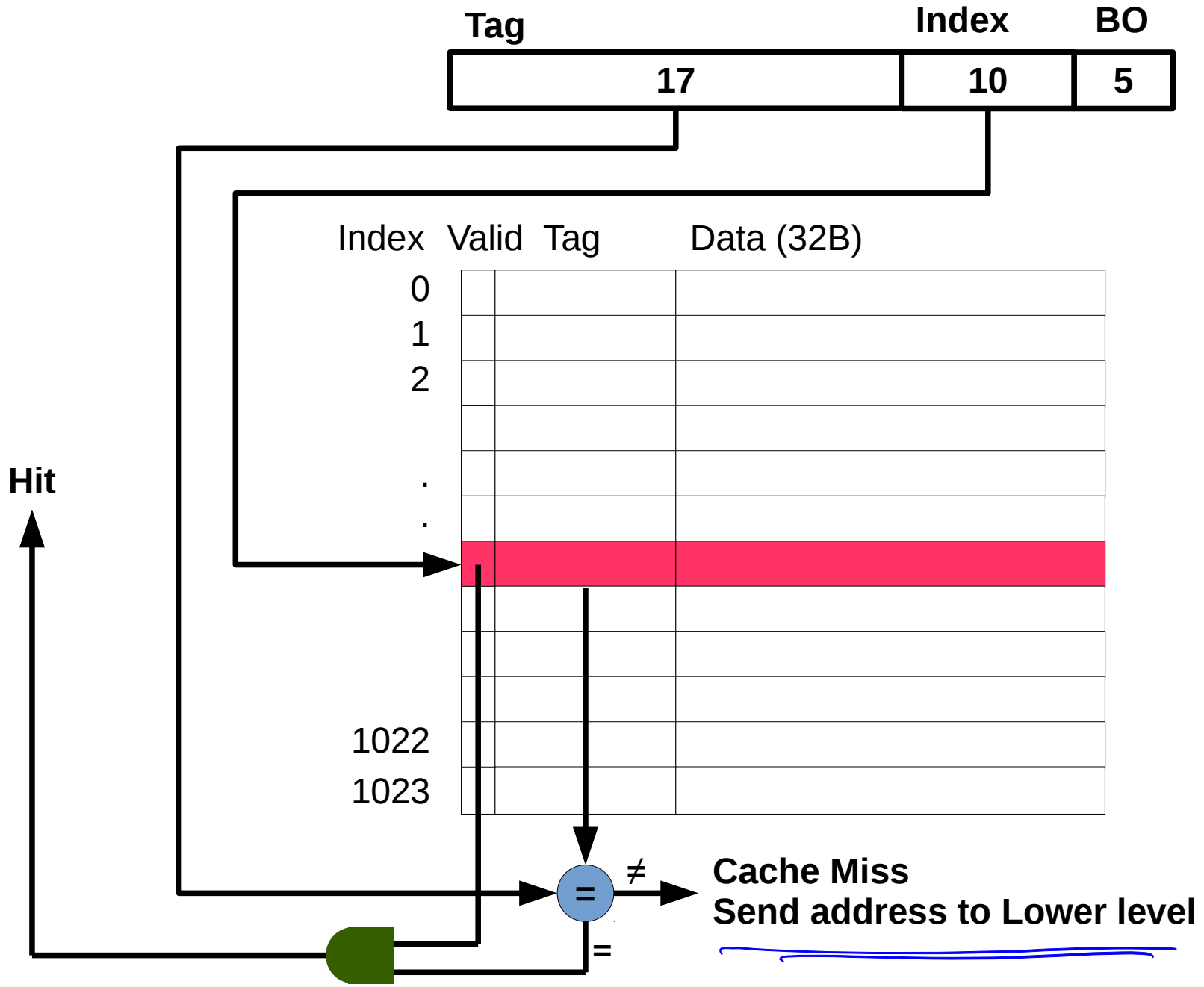
Direct Mapped Cache Organization



Direct Mapped Cache Organization



Direct Mapped Cache Organization



Cache Aware Programming

Cache Aware Programming

- Objective: Learn how to assess cache related performance issues for important parts of our programs

Cache Aware Programming

- Objective: Learn how to assess cache related performance issues for important parts of our programs
- Assume separate instruction and data caches
  - We consider data accesses

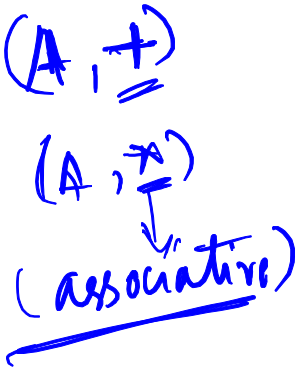
Cache Aware Programming

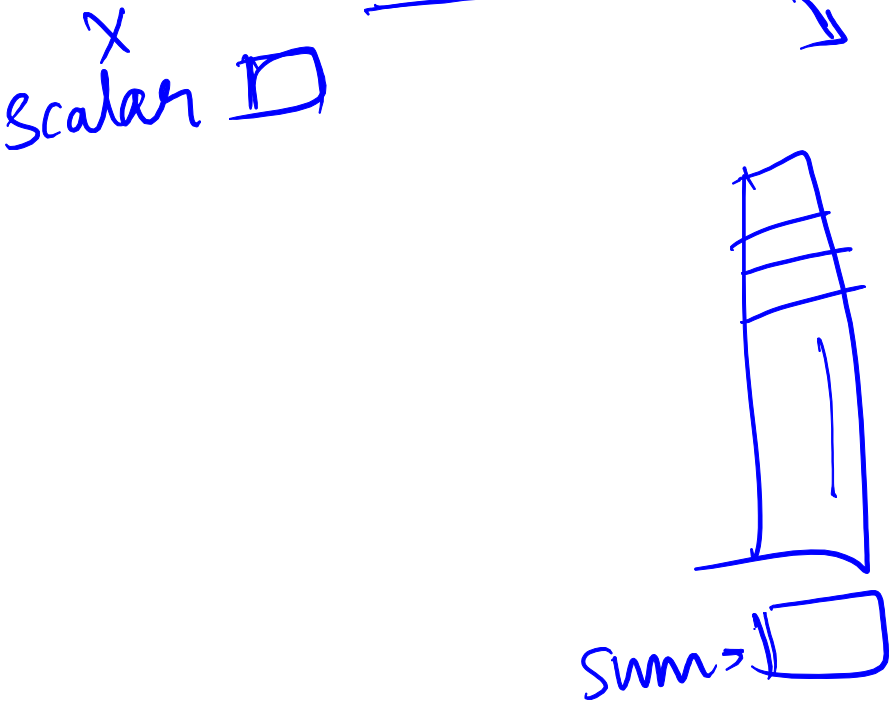
- Objective: Learn how to assess cache related performance issues for important parts of our programs
- Assume separate instruction and data caches
  - We consider data accesses
- Data cache configuration:
  - 16 KB, 32B blocks, Direct Mapped, Write Back

Cache Aware Programming

- Objective: Learn how to assess cache related performance issues for important parts of our programs
- Assume separate instruction and data caches
  - We consider data accesses
- Data cache configuration:
  - 16 KB, 32B blocks, Direct Mapped, Write Back

Vector Sum Reduction





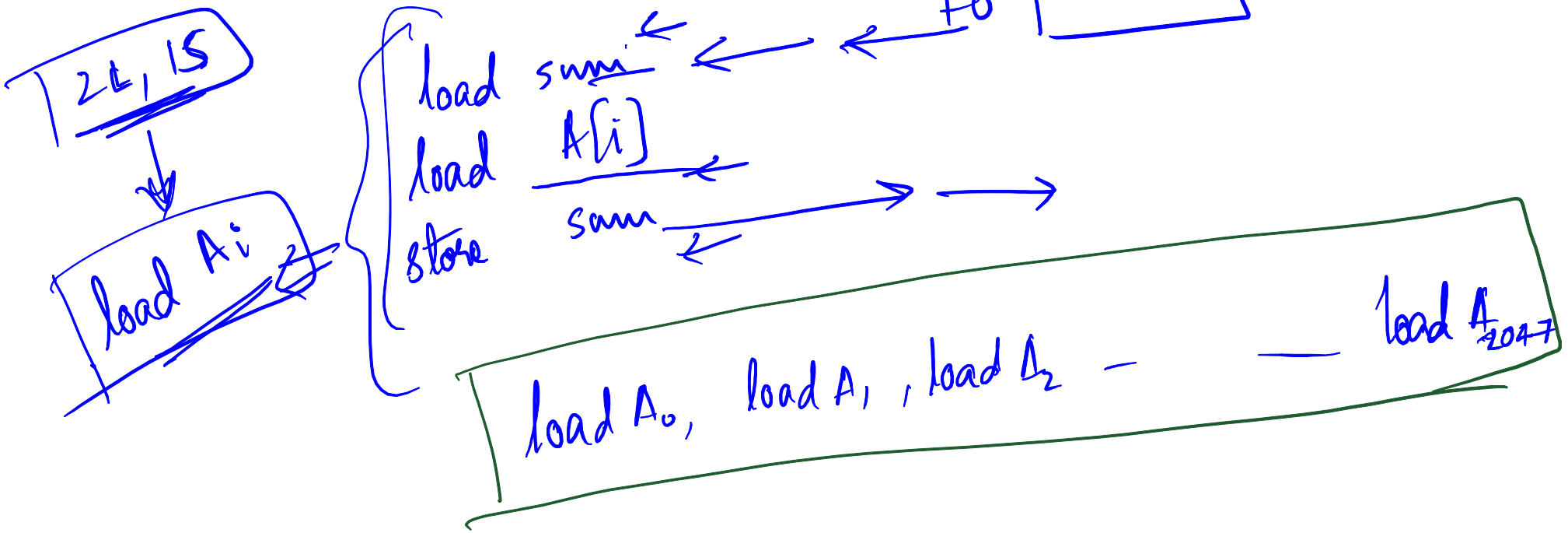
Vector Sum Reduction

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

Vector Sum Reduction

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

- List the data memory accesses generated by this code.



Vector Sum Reduction

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

- List the data memory accesses generated by this code.
- See the Loads and Stores in the loop body

Vector Sum Reduction

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

- List the data memory accesses generated by this code.
- See the Loads and Stores in the loop body
- Loop index i and variable sum are implemented in registers

Vector Sum Reduction

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

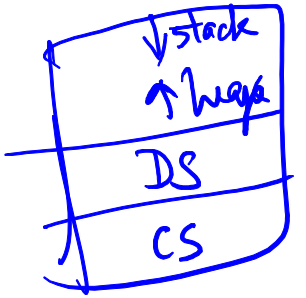
- List the data memory accesses generated by this code.
- See the Loads and Stores in the loop body
- Loop index i and variable sum are implemented in registers
- Consider only accesses to array elements

VSR – Memory References

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

VSR – Memory References

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



- load A[0], load A[1], load A[2], ..., load A[2047]

VSR – Memory References

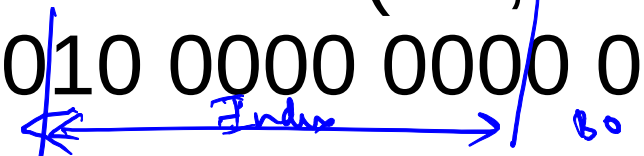
```
double A[2048], sum=0.0;
for (i=0; i<2048, i++) sum = sum +A[i];
```

- 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

VSR – Memory References

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

- 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)

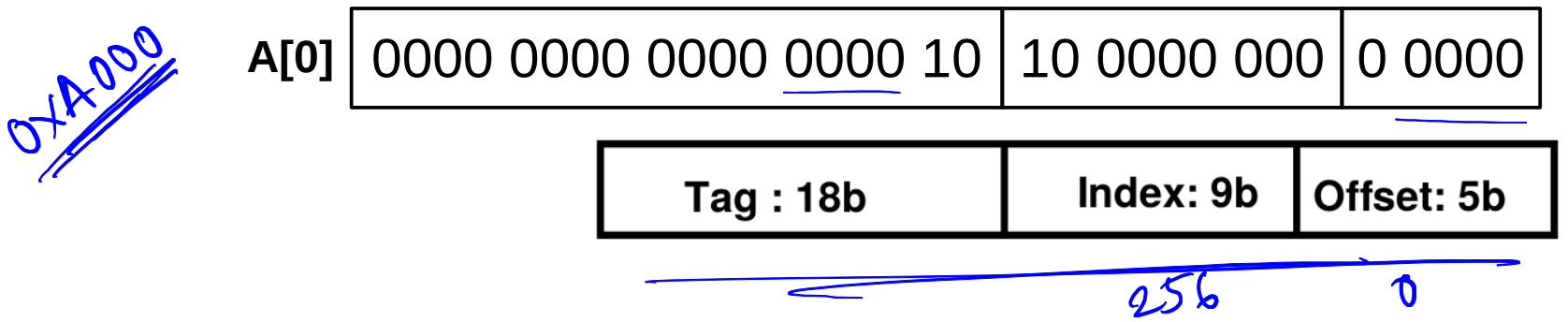


16K, 32B
cache 16K B
1 ins = 32B
= 512
@
= 2

VSR – Memory References

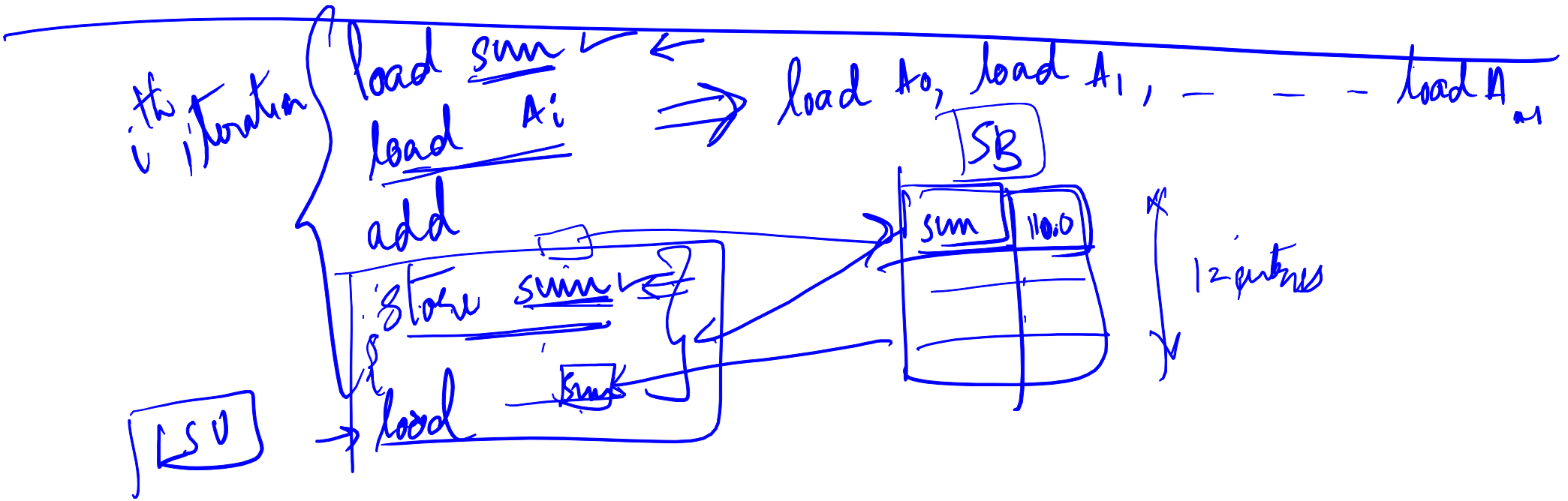
```
double A[2048], sum=0.0;
for (i=0; i<2048, i++) sum = sum +A[i];
```

- 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)



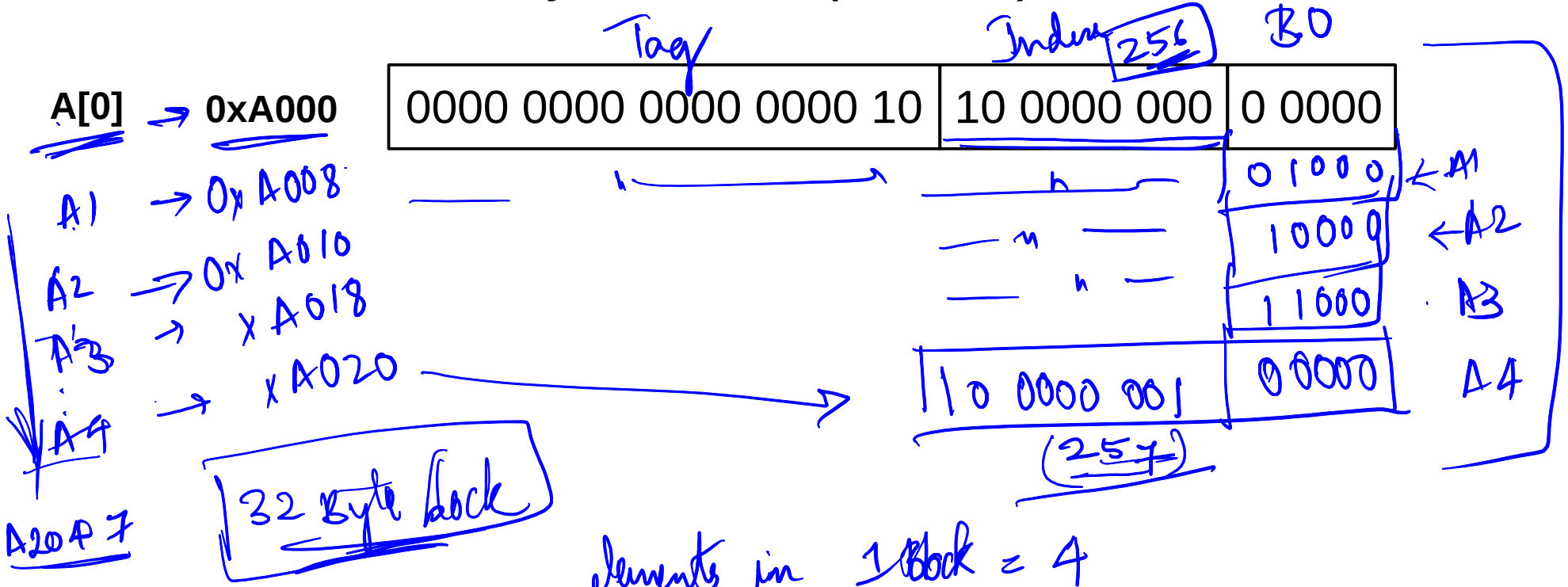
VSR – Memory References

- Base address of A (i.e., address of A[0]) is 0xA000
- What is the address of A[1], A[2], ..., A[2047]?
  - Size of an array element (double) = 8B



VSR – Memory References

- Base address of A (i.e., address of A[0]) is 0xA000
- What is the address of A[1], A[2], ..., A[2047]?
  - Size of an array element (double) = 8B

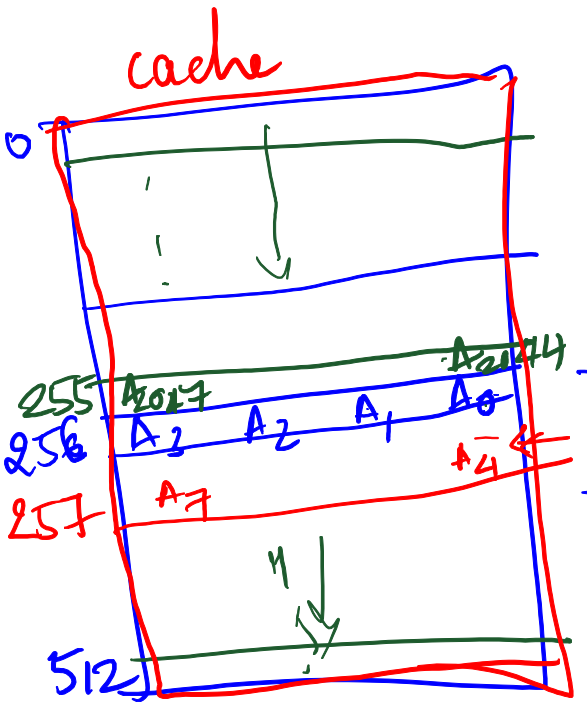


VSR – Memory Addresses

		Tag : 18b	Index: 9b	Offset: 5b
A[0]	0xA000	0000 0000 0000 0000 10	10 0000 000	0 0000
A[1]	A008			
A[2]	A010			
A[3]	A018			
A[4]	A020			
	⋮			
A[2043]				
A[2044]				
A[2045]				
A[2046]				
A[2047]				

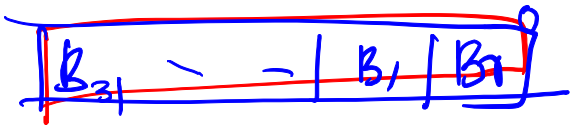
VSR – Memory Addresses

	Tag : 18b	Index: 9b	Offset: 5b
A[0]	0xA000	0000 0000 0000 0000 10	10 0000 000	0 0000
A[1]	0xA008	0000 0000 0000 0000 10	10 0000 000	0 1000
A[2]			
A[3]			
A[4]			
A[2043]			
A[2044]			
A[2045]			
A[2046]			
A[2047]			



VSR – Memory Addresses

		Tag : 18b	Index: 9b	Offset: 5b
A[0]	0xA000	0000 0000 0000 0000 10	10 0000 000	0 0000
A[1]	0xA008	0000 0000 0000 0000 10	10 0000 000	0 1000
A[2]	0xA010	0000 0000 0000 0000 10	10 0000 000	1 0000
A[3]	0xA018	0000 0000 0000 0000 10	10 0000 000	1 1000
A[4]	0xA020	0000 0000 0000 0000 10	10 0000 001	0 0000
...				
A[2043]	0xDFD8	0000 0000 0000 0000 11	01 1111 110	1 1000
A[2044]	0xDFE0	0000 0000 0000 0000 11	01 1111 111	0 0000
A[2045]	0xDFE8	0000 0000 0000 0000 11	01 1111 111	0 1000
A[2046]	0xDFF0	0000 0000 0000 0000 11	01 1111 111	1 0000
A[2047]	0xDFF8	0000 0000 0000 0000 11	01 1111 111	1 1000



VSR – Memory Addresses

		Tag : 18b	Index: 9b	Offset: 5b
				Index
A[0]	0xA000	0000 0000 0000 0000 10	10 0000 000	0 0000
A[1]	0xA008	0000 0000 0000 0000 10	10 0000 000	0 1000
A[2]	0xA010	0000 0000 0000 0000 10	10 0000 000	1 0000
A[3]	0xA018	0000 0000 0000 0000 10	10 0000 000	1 1000
A[4]	0xA020	0000 0000 0000 0000 10	10 0000 001	0 0000
	...			
A[2043]	0xDFD8	0000 0000 0000 0000 11	01 1111 110	1 1000
A[2044]	0xDFE0	0000 0000 0000 0000 11	01 1111 111	0 0000
A[2045]	0xDFE8	0000 0000 0000 0000 11	01 1111 111	0 1000
A[2046]	0xDFF0	0000 0000 0000 0000 11	01 1111 111	1 0000
A[2047]	0xDFF8	0000 0000 0000 0000 11	01 1111 111	1 1000

VSR – Memory Addresses

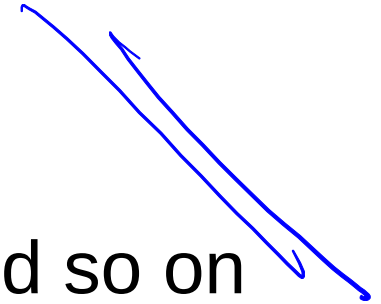
		Tag : 18b	Index: 9b	Offset: 5b	Index
A[0]	0xA000	0000 0000 0000 0000 10	10 0000 000	0 0000	256
A[1]	0xA008	0000 0000 0000 0000 10	10 0000 000	0 1000	256
A[2]	0xA010	0000 0000 0000 0000 10	10 0000 000	1 0000	256
A[3]	0xA018	0000 0000 0000 0000 10	10 0000 000	1 1000	256
A[4]	0xA020	0000 0000 0000 0000 10	10 0000 001	0 0000	257
...					
A[2043]	0xDFD8	0000 0000 0000 0000 11	01 1111 110	1 1000	254
A[2044]	0xDFE0	0000 0000 0000 0000 11	01 1111 111	0 0000	255
A[2045]	0xDFE8	0000 0000 0000 0000 11	01 1111 111	0 1000	255
A[2046]	0xDFF0	0000 0000 0000 0000 11	01 1111 111	1 0000	255
A[2047]	0xDFF8	0000 0000 0000 0000 11	01 1111 111	1 1000	255

VSR – Memory References

- How many elements from array A fit in 1 cache block? Block Size = 32B.

VSR – Memory References

- How many elements from array A fit in 1 cache block? Block Size = 32B.
  - 4 consecutive array elements
  - A[0] – A[3] have index of 256
  - A[4] – A[7] have index of 257 and so on



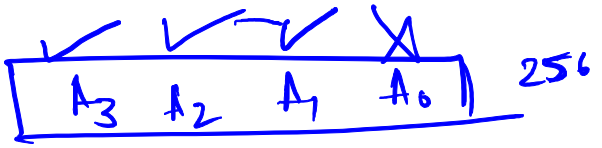
Cache Hits/Misses

Load	Addr	Index	H/M
load A[0]	0xA000	256	M
load A[1]	0xA008	256	
A[2]	0xA010	256	
A[3]	0xA018	256	
A[4]	0xA020	257	
	...	...	
A[2043]	0xDFD8	254	
A[2044]	0xDFE0	255	
A[2045]	0xDFE8	255	
A[2046]	0xDFF0	255	
A[2047]	0xDFF8	255	

Cache Hits/Misses

Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
A[1]	0xA008	256	H	
A[2]	0xA010	256	H	
A[3]	0xA018	256	H	
A[4]	0xA020	257	M	
...		...		
A[2043]	0xDFD8	254		
A[2044]	0xDFE0	255		
A[2045]	0xDFE8	255		
A[2046]	0xDFF0	255		
A[2047]	0xDFF8	255		

Cold Start Miss. Cache is empty at start. The first access to an address is always a miss. Also called Compulsory Miss.



Cache Hits/Misses

Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
A[1]	0xA008	256	H	
A[2]	0xA010	256	H	
A[3]	0xA018	256	H	
A[4]	0xA020	257		
...		...		
A[2043]	0xDFD8	254		
A[2044]	0xDFE0	255		
A[2045]	0xDFE8	255		
A[2046]	0xDFF0	255		
A[2047]	0xDFF8	255		

Cold Start Miss. Cache is empty at start. The first access to an address is always a miss. Also called Compulsory Miss.

Cache Hits/Misses

Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
A[1]	0xA008	256	H	
A[2]	0xA010	256	H	
A[3]	0xA018	256	H	
A[4]	0xA020	257	M	Cold Start
...		...		
A[2043]	0xDFD8	254	H	
A[2044]	0xDFE0	255	M	Cold Start
A[2045]	0xDFE8	255	H	
A[2046]	0xDFF0	255	H	
A[2047]	0xDFF8	255	H	

2048 - 2051

M, H, H, H

Cold Start Miss. Cache is empty at start. The first access to an address is always a miss. Also called Compulsory Miss.

Cache Hits/Misses

Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
A[1]	0xA008	256	H	
A[2]	0xA010	256	H	
A[3]	0xA018	256	H	
A[4]	0xA020	257	M	Cold Start
...		...		
A[2043]	0xDFD8	254	H	
A[2044]	0xDFE0	255	M	Cold Start
A[2045]	0xDFE8	255	H	
A[2046]	0xDFF0	255	H	
A[2047]	0xDFF8	255	H	

Cold Start Miss. Cache is empty at start. The first access to an address is always a miss. Also called Compulsory Miss.

Hit Ratio = **75%**

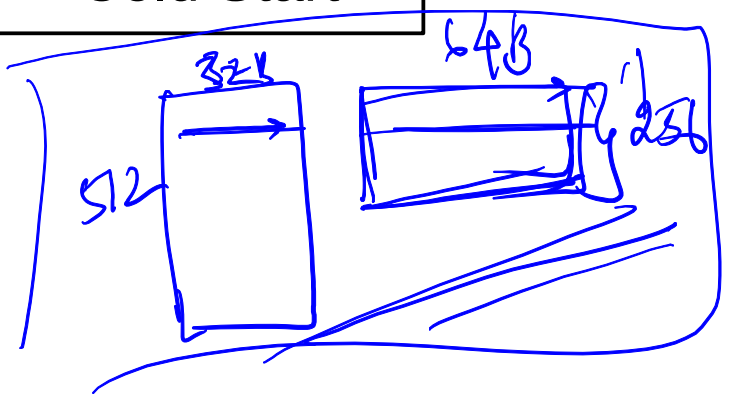
Cache Hits/Misses

Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
A[1]	0xA008	256	H	
A[2]	0xA010	256	H	
A[3]	0xA018	256	H	
A[4]	0xA020	257	M	Cold Start
...	...			
A[2043]	0xDFD8	254	H	
A[2044]	0xDFE0	255	M	Cold Start
A[2045]	0xDFE8	255	H	
A[2046]	0xDFF0	255	H	
A[2047]	0xDFF8	255	H	

Cold Start Miss. Cache is empty at start. The first access to an address is always a miss. Also called Compulsory Miss.

Hit Ratio = **75%**

Spatial Locality of Reference!



Cache Hits/Misses

Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
A[1]	0xA008	256	H	
A[2]	0xA010	256	H	
A[3]	0xA018	256	H	
A[4]	0xA020	257	M	Cold Start
...		...		
A[2043]	0xDFD8	254	H	
A[2044]	0xDFE0	255	M	Cold Start
A[2045]	0xDFE8	255	H	
A[2046]	0xDFF0	255	H	
A[2047]	0xDFF8	255	H	

Cold Start Miss. Cache is empty at start. The first access to an address is always a miss. Also called Compulsory Miss.

Hit Ratio = **75%**

Spatial Locality of Reference!

How to improve the Hit Ratio?

Cache Hits/Misses

Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
A[1]	0xA008	256	H	
A[2]	0xA010	256	H	
A[3]	0xA018	256	H	
A[4]	0xA020	257	M	Cold Start
...		...		
A[2043]	0xDFD8	254	H	
A[2044]	0xDFE0	255	M	Cold Start
A[2045]	0xDFE8	255	H	
A[2046]	0xDFF0	255	H	
A[2047]	0xDFF8	255	H	

Cold Start Miss. Cache is empty at start. The first access to an address is always a miss. Also called Compulsory Miss.

Hit Ratio = **75%**

Spatial Locality of Reference!

How to improve the Hit Ratio?

If the loop was preceded by a loop that accessed all array elements, the hit ratio of our loop would be 100%, 25% due to temporal locality and 75% due to spatial locality

VSR - v2

```
double A[4096], sum=0.0;
for (i=0; i<4096, i++) sum = sum +A[i];
```

VSR - v2

```
double A[4096], sum=0.0;
for (i=0; i<4096, i++) sum = sum +A[i];
```

- Array Size: 2048 vs 4096
  - A[4096] does not fit into the cache.

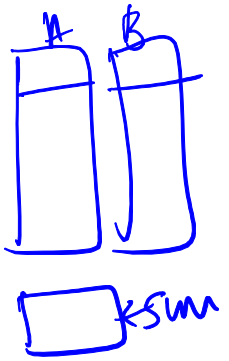
VSR - v2

```
double A[4096], sum=0.0;
for (i=0; i<4096, i++) sum = sum +A[i];
```

- Array Size: 2048 vs 4096
  - A[4096] does not fit into the cache.
- Hit Ratio?

Vector Dot Product

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



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, - - -

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], ...

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], ...
- Base addresses of A and B are 0xA000 and 0xE000

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], ...
- Base addresses of A and B are 0xA000 and 0xE000
- Size of array elements is 8B – 4 consecutive array elements fit into each cache block

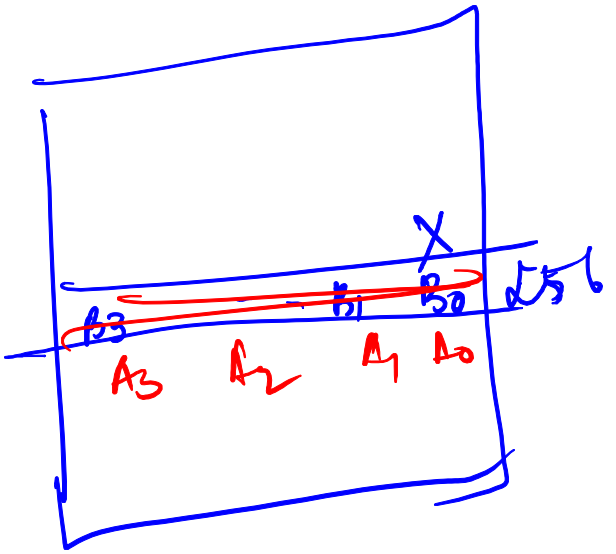
Cache Hits/Misses

Load	Addr	Index	H/M
A[0]	0xA000	256	M — cs.
B[0]	0xE000	256	→ M — cs
A[1]	0xA008	256	M
B[1]	0xE008	256	M
A[2]	0xA010		
B[2]	0xE010		
A[3]	0xA018		
B[3]	0xE018		
...	...	...	

load

load

bad



HR = 0 %

MR = 100 %

Cache Hits/Misses

Load	Addr	Index	H/M
A[0]	0xA000	256	
B[0]	0xE000	256	
A[1]	0xA008	256	
B[1]	0xE008	256	
A[2]	0xA010	256	
B[2]	0xE010	256	
A[3]	0xA018	256	
B[3]	0xE018	256	
...	...	...	

Cache Hits/Misses

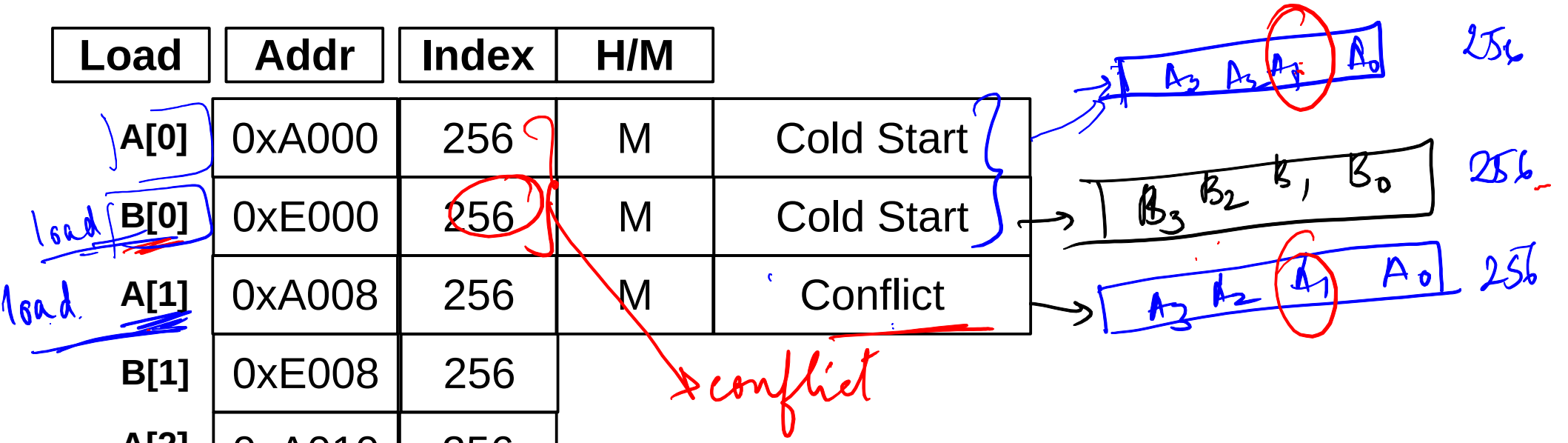
Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
B[0]	0xE000	256		
A[1]	0xA008	256		
B[1]	0xE008	256		
A[2]	0xA010	256		
B[2]	0xE010	256		
A[3]	0xA018	256		
B[3]	0xE018	256		
	...	...	...	

Cache Hits/Misses

Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
B[0]	0xE000	256	M	Cold Start
A[1]	0xA008	256		
B[1]	0xE008	256		
A[2]	0xA010	256		
B[2]	0xE010	256		
A[3]	0xA018	256		
B[3]	0xE018	256		
...	...	...		

Cache Hits/Misses

Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
B[0]	0xE000	256	M	Cold Start
A[1]	0xA008	256	M	Conflict
B[1]	0xE008	256		
A[2]	0xA010	256		
B[2]	0xE010	256		
A[3]	0xA018	256		
B[3]	0xE018	256		
...	...	...		



Cache Hits/Misses

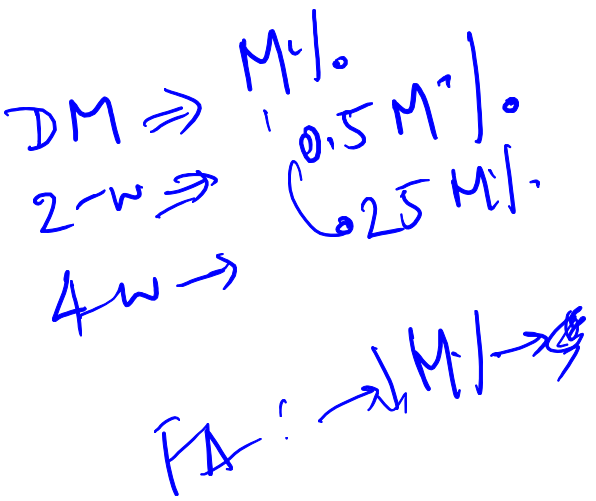
Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
B[0]	0xE000	256	M	Cold Start
A[1]	0xA008	256	M	Conflict
B[1]	0xE008	256	M	Conflict
A[2]	0xA010	256		
B[2]	0xE010	256		
A[3]	0xA018	256		
B[3]	0xE018	256		
...	...	...		

Conflict Miss. A non-ideal replacement policy causes this miss.

Cache Hits/Misses

Conflict Miss.
A non-ideal replacement policy causes this miss.

Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
B[0]	0xE000	256	M	Cold Start
A[1]	0xA008	256	M	Conflict
B[1]	0xE008	256	M	Conflict
A[2]	0xA010	256	M	Conflict
B[2]	0xE010	256	M	Conflict
A[3]	0xA018	256	M	Conflict
B[3]	0xE018	256	M	Conflict
...	...	...	...	

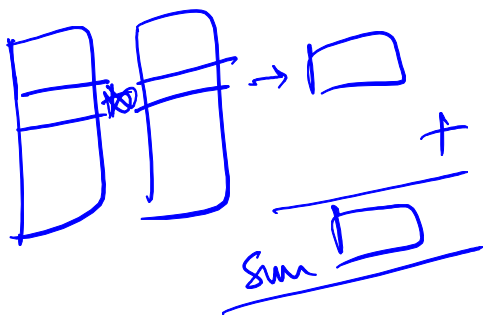


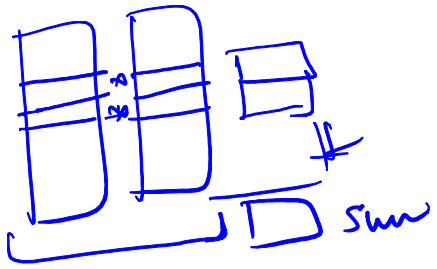
Cache Hits/Misses

Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
B[0]	0xE000	256	M	Cold Start
A[1]	0xA008	256	M	Conflict
B[1]	0xE008	256	M	Conflict
A[2]	0xA010	256	M	Conflict
B[2]	0xE010	256	M	Conflict
A[3]	0xA018	256	M	Conflict
B[3]	0xE018	256	M	Conflict
...	...	...		

Conflict Miss.
A non-ideal replacement policy causes this miss.

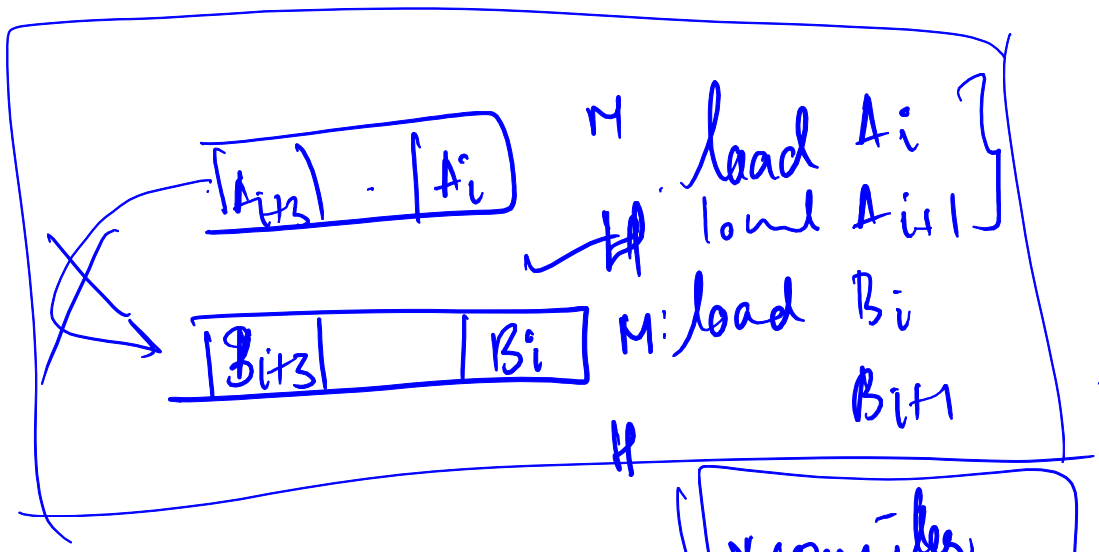
Hit Ratio = **0% !**





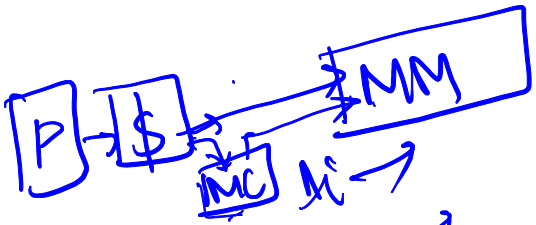

```
for  i+=2;
{
sum += A[i] * B[i];
sum += A[i+1] * B[i+1];
}
```

load A_i, load B_i, load A_{i+1}, load B_{i+1},



M load A_i, H load A_{i+1}, H load A_{i+2} / load A_{i+3}

load B_i, B_{i+1}, B_{i+2} / B_{i+3}



* compiler
* LSQ

load
* EA
* Queue
* D Macces
* Reg File update / RoB update

* mem controller

Cache Hits/Misses

Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
B[0]	0xE000	256	M	Cold Start
A[1]	0xA008	256	M	Conflict
B[1]	0xE008	256	M	Conflict
A[2]	0xA010	256	M	Conflict
B[2]	0xE010	256	M	Conflict
A[3]	0xA018	256	M	Conflict
B[3]	0xE018	256	M	Conflict
	...	...	...	

Conflict Miss.
A non-ideal replacement policy causes this miss.

Hit Ratio = **0% !**

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

Cache Hits/Misses

Load	Addr	Index	H/M	
A[0]	0xA000	256	M	Cold Start
B[0]	0xE000	256	M	Cold Start
A[1]	0xA008	256	M	Conflict
B[1]	0xE008	256	M	Conflict
A[2]	0xA010	256	M	Conflict
B[2]	0xE010	256	M	Conflict
A[3]	0xA018	256	M	Conflict
B[3]	0xE018	256	M	Conflict
	...	...	...	

Conflict Miss.
A non-ideal replacement policy causes this miss.

Hit Ratio = **0% !**

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

Hit ratio would be better if the base address of B is such that these cache indices differ

VDP with Packing

- Assume: Compiler assigns addresses as it finds variable declarations

VDP with Packing

- Assume: Compiler assigns addresses as it finds variable declarations
- To shift base address of B enough to make cache index of B[0] different from that of A[0]
  -

VDP with Packing

- Assume: Compiler assigns addresses as it finds variable declarations
- 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];

VDP with Packing

- Assume: Compiler assigns addresses as it finds variable declarations
- 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;

VDP with Packing

- Assume: Compiler assigns addresses as it finds variable declarations
- 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

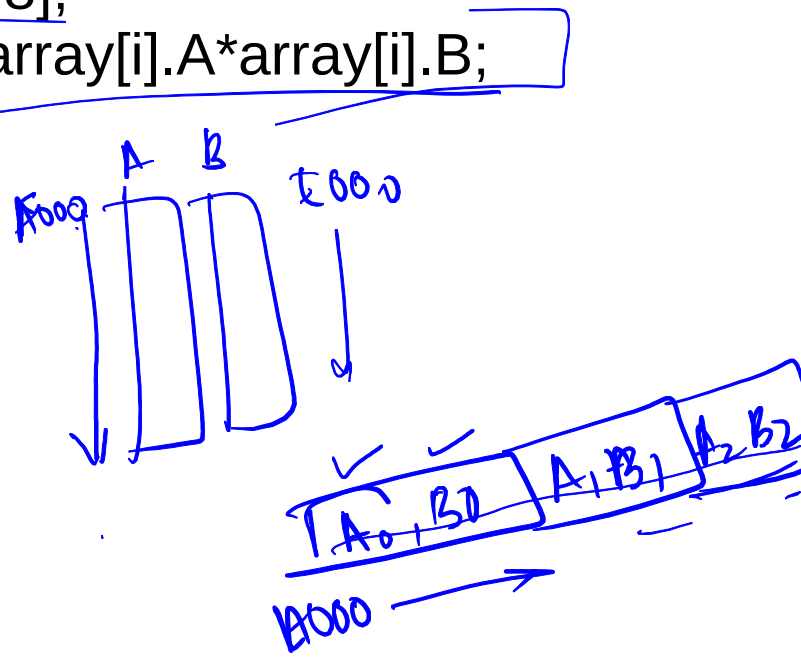
VDP with Packing

- Assume: Compiler assigns addresses as it finds variable declarations
- 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 = 75%

VDP with Array Merging

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

- Hit Ratio?



VDP with Array Merging

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

- Hit Ratio?
  - 75%

DAXPY

- Double precision Y = aX + Y, where X and Y are vectors and a is a scalar

```
double X[2048], Y[2048], a;
for (i=0; i<2048;i++) Y[i] = a*X[i]+Y[i];
```

DAXPY

- Double precision Y = aX + Y, where X and Y are vectors and a is a scalar

```
double X[2048], Y[2048], a;
for (i=0; i<2048;i++) Y[i] = a*X[i]+Y[i];
```

- Reference sequence:
  - load X[0], load Y[0], store Y[0],
  - load X[1], load Y[1], store Y[1], ... etc

DAXPY

- Double precision Y = aX + Y, where X and Y are vectors and a is a scalar

```
double X[2048], Y[2048], a;
for (i=0; i<2048;i++) Y[i] = a*X[i]+Y[i];
```

- Reference sequence:
  - load X[0], load Y[0], store Y[0],
  - load X[1], load Y[1], store Y[1], ... etc
- Hit Ratio?

DAXPY

- Double precision Y = aX + Y, where X and Y are vectors and a is a scalar

```
double X[2048], Y[2048], a;
for (i=0; i<2048;i++) Y[i] = a*X[i]+Y[i];
```

- Reference sequence:
  - load X[0], load Y[0], store Y[0],
  - load X[1], load Y[1], store Y[1], ... etc
- Hit Ratio?
  - The behaviour of loads is as before. Stores?

DAXPY

- Double precision Y = aX + Y, where X and Y are vectors and a is a scalar

```
double X[2048], Y[2048], a;
for (i=0; i<2048;i++) Y[i] = a*X[i]+Y[i];
```

- Reference sequence:
  - load X[0], load Y[0], store Y[0],
  - load X[1], load Y[1], store Y[1], ... etc
- Hit Ratio?
  - The behaviour of loads is as before. Stores?
  - Assuming that base addresses of X and Y don’t conflict in cache, hit ratio of 83.3%

Matrix Sum

```
double A[1024][1024], B[1024][1024];
for (j=0;j<1024;j++)
   for (i=0;i<1024;i++)
      B[i][j] = A[i][j] + B[i][j];
```

- Reference Sequence:
  - –

Matrix Sum

```
double A[1024][1024], B[1024][1024];
for (j=0;j<1024;j++)
   for (i=0;i<1024;i++)
      B[i][j] = A[i][j] + B[i][j];
```

- Reference Sequence:
  - load A[0,0] load B[0,0] store B[0,0]
  - load A[1,0] load B[1,0] store B[1,0] ...
-

Matrix Sum

```
double A[1024][1024], B[1024][1024];
for (j=0;j<1024;j++)
   for (i=0;i<1024;i++)
      B[i][j] = A[i][j] + B[i][j];
```

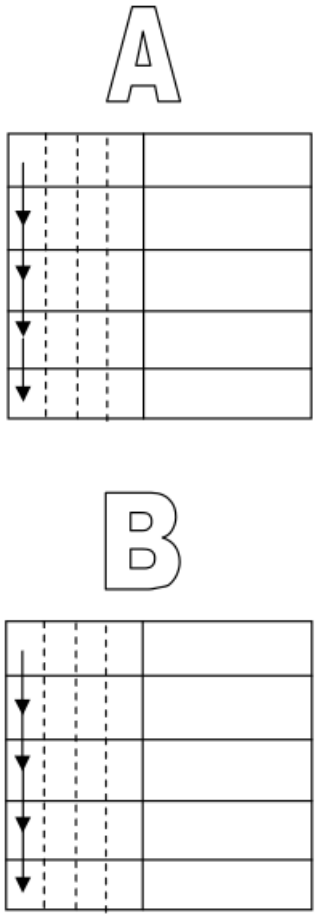
- Reference Sequence:
  - load A[0,0] load B[0,0] store B[0,0]
  - load A[1,0] load B[1,0] store B[1,0] ...
- Question: In what order are the elements of a multidimensional array stored in memory?
  -

Matrix Sum

```
double A[1024][1024], B[1024][1024];
for (j=0;j<1024;j++)
   for (i=0;i<1024;i++)
      B[i][j] = A[i][j] + B[i][j];
```

- Reference Sequence:
  - load A[0,0] load B[0,0] store B[0,0]
  - load A[1,0] load B[1,0] store B[1,0] ...
- Question: In what order are the elements of a multidimensional array stored in memory?
  - Row major/Column Major
  - C/FORTRAN

Matrix Sum



- load A[0,0] load B[0,0] store B[0,0]
- load A[1,0] load B[1,0] store B[1,0] ...

Matrix Sum

- load A[0,0] load B[0,0] store B[0,0]
- load A[1,0] load B[1,0] store B[1,0] ...
- Reference order is different from storage order!
  - Storage in Row-major, but refs are in Column-major order

Matrix Sum

- load A[0,0] load B[0,0] store B[0,0]
- load A[1,0] load B[1,0] store B[1,0] ...
- Reference order is different from storage order!
  - Storage in Row-major, but refs are in Column-major order
- The loop will show no spatial locality !

Matrix Sum

- Assume: Packing has been done to eliminate conflict misses due to base addresses

Matrix Sum

- Assume: Packing has been done to eliminate conflict misses due to base addresses
- Miss(cold), Miss(cold), Hit for each array element

Matrix Sum

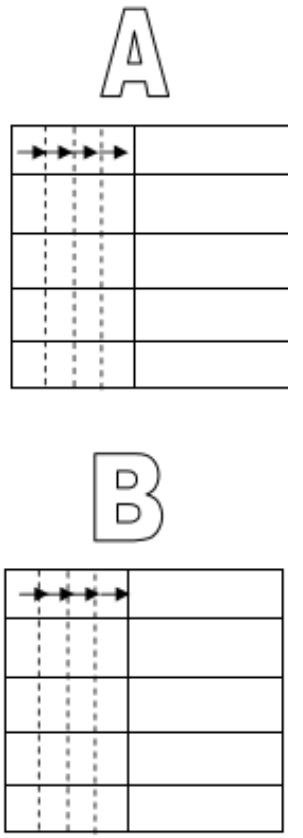
- Assume: Packing has been done to eliminate conflict misses due to base addresses
- Miss(cold), Miss(cold), Hit for each array element
- Hit ratio = ?

Matrix Sum

- Assume: Packing has been done to eliminate conflict misses due to base addresses
- Miss(cold), Miss(cold), Hit for each array element
- Hit ratio = ?
- Question: Will A[0,1] be in the cache when required?

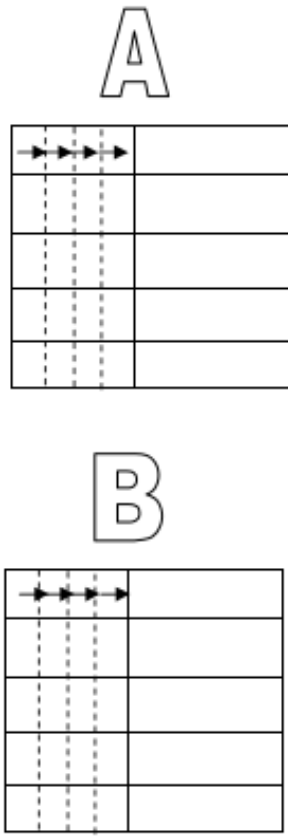
Matrix Sum – Loop Interchange

```
double A[1024][1024], B[1024][1024];
for (i=0; i<1024; i++)
   for (j=0; j<1024; j++)
      B[i][j] = A[i][j] + B[i][j];
```



Matrix Sum – Loop Interchange

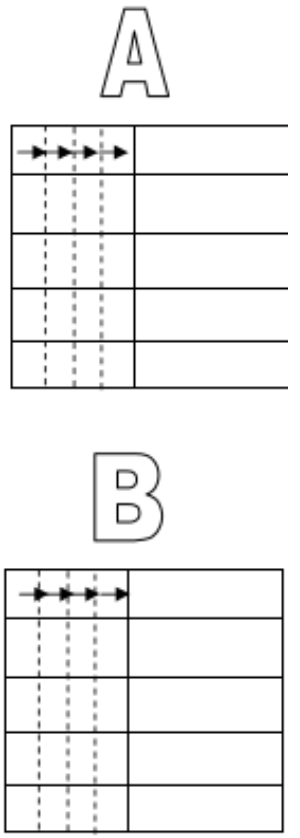
```
double A[1024][1024], B[1024][1024];
for (i=0; i<1024; i++)
   for (j=0; j<1024; j++)
      B[i][j] = A[i][j] + B[i][j];
```



- Reference Sequence:

Matrix Sum – Loop Interchange

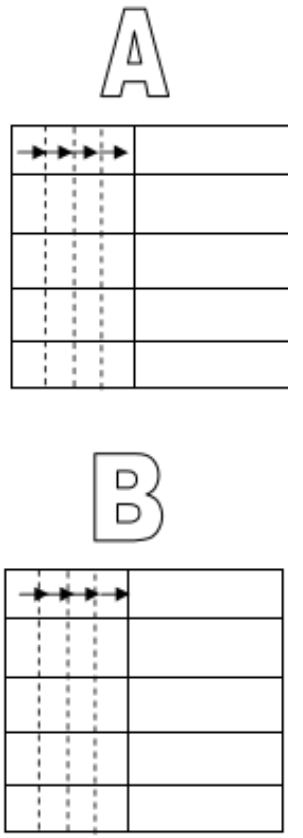
```
double A[1024][1024], B[1024][1024];
for (i=0; i<1024; i++)
   for (j=0; j<1024; j++)
      B[i][j] = A[i][j] + B[i][j];
```



- Reference Sequence:
  - load A[0,0] load B[0,0] store B[0,0]
  - load A[0,1] load B[0,1] store B[0,1] ...
-

Matrix Sum – Loop Interchange

```
double A[1024][1024], B[1024][1024];
for (i=0; i<1024; i++)
   for (j=0; j<1024; j++)
      B[i][j] = A[i][j] + B[i][j];
```



- Reference Sequence:
  - load A[0,0] load B[0,0] store B[0,0]
  - load A[0,1] load B[0,1] store B[0,1] ...
- Hit ratio = ?

Is Loop Interchange Always Safe?

```
for (j=1; j<2048; j++)
   for (i=1; i<2048; i++)
      A[i][j] = A[i+1][j-1] + A[i][j-1];
```

Is Loop Interchange Always Safe?

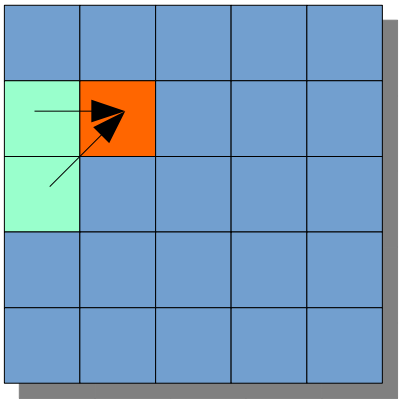
```
for (j=1; j<2048; j++)
   for (i=1; i<2048; i++)
      A[i][j] = A[i+1][j-1] + A[i][j-1];
```

A[1][1] = A[2][0] + A[1][0]
A[2][1] = A[3][0] + A[2][0]
A[3][1] = A[4][0] + A[3][0]
...

Is Loop Interchange Always Safe?

```
for (j=1; j<2048; j++)
   for (i=1; i<2048; i++)
      A[i][j] = A[i+1][j-1] + A[i][j-1];
```

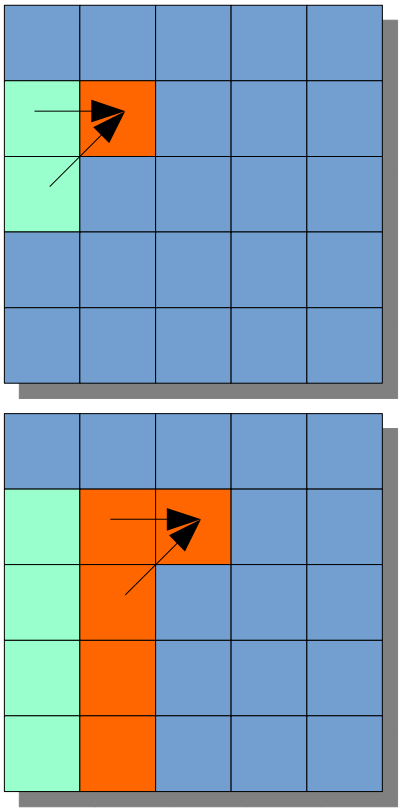
A[1][1] = A[2][0] + A[1][0]
A[2][1] = A[3][0] + A[2][0]
A[3][1] = A[4][0] + A[3][0]
...



Is Loop Interchange Always Safe?

```
for (j=1; j<2048; j++)
   for (i=1; i<2048; i++)
      A[i][j] = A[i+1][j-1] + A[i][j-1];
```

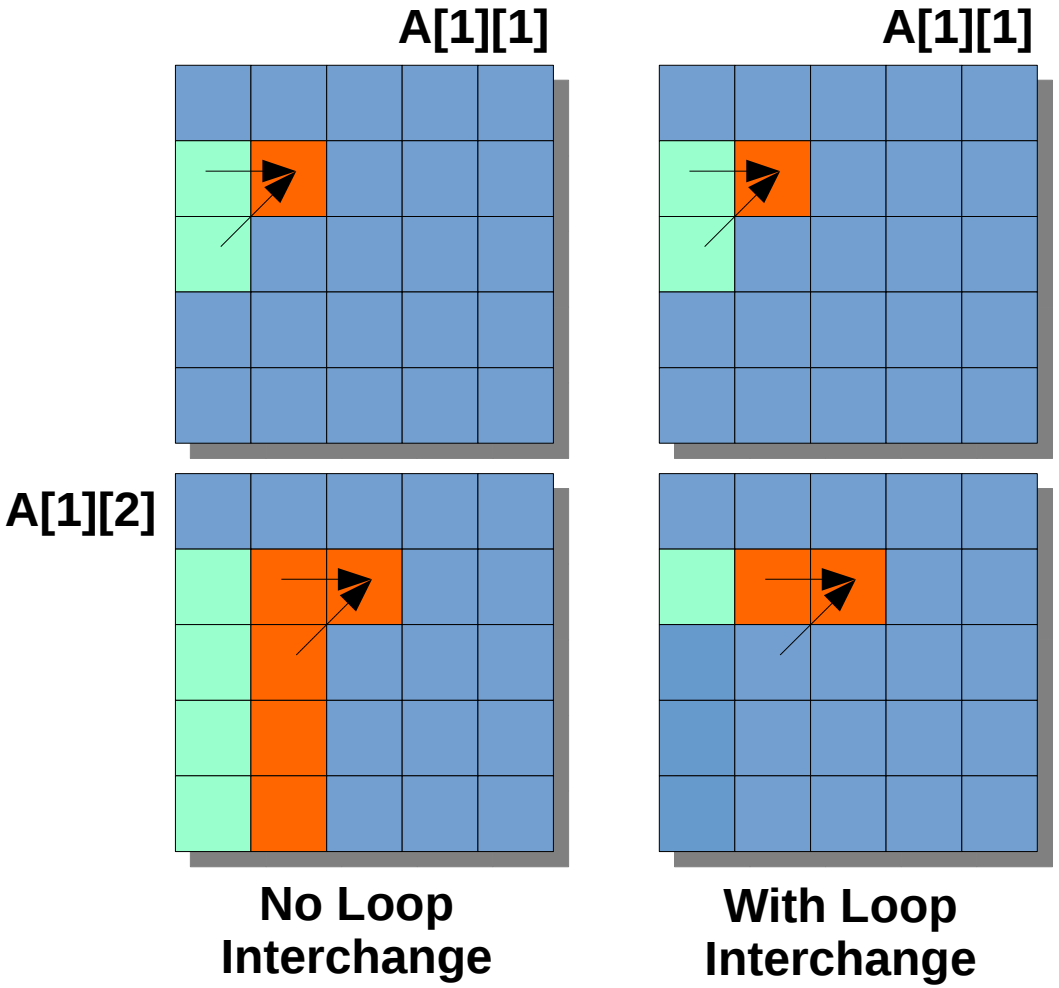
A[1][1] = A[2][0] + A[1][0]
A[2][1] = A[3][0] + A[2][0]
A[3][1] = A[4][0] + A[3][0]
...
A[1][2] = A[2][1] + A[1][1]
A[2][2] = A[3][1] + A[2][1]
A[3][2] = A[4][1] + A[3][1]
...



Is Loop Interchange Always Safe?

```
for (i=1; i<2048; i++)
   for (j=1; j<2048; j++)
      A[i][j] = A[i+1][j-1] + A[i][j-1];
```

A[1][1] = A[2][0] + A[1][0]
A[1][2] = A[2][1] + A[1][1]
A[1][3] = A[2][2] + A[1][2]
…
…



Is Loop Interchange Always Safe?

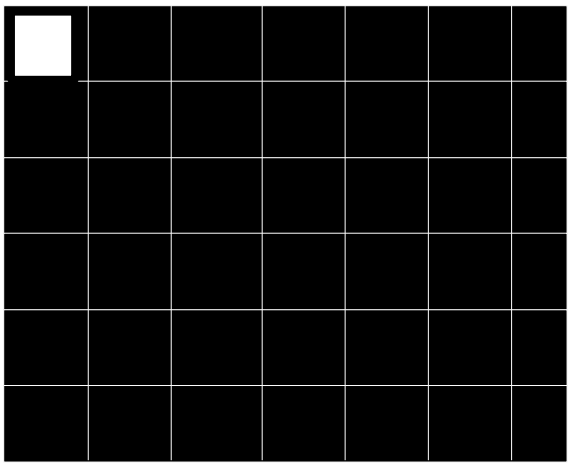
```
for (i=2047; i>1; i--)
   for (j=1; j<2048; j++)
      A[i][j] = A[i+1][j-1] + A[i][j-1];
```

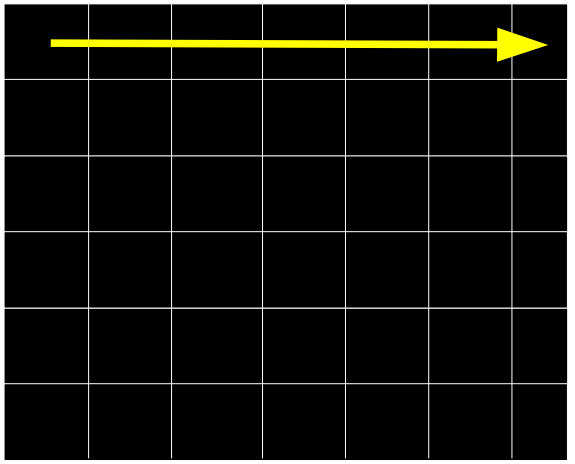
Matrix Multiplication

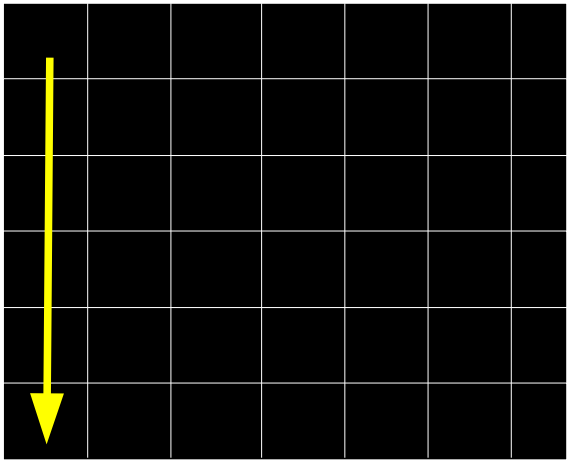
```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      X[i][j] += Y[i][k] * Z[k][j];
```

Matrix Multiplication

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      X[i][j] += Y[i][k] * Z[k][j];
```







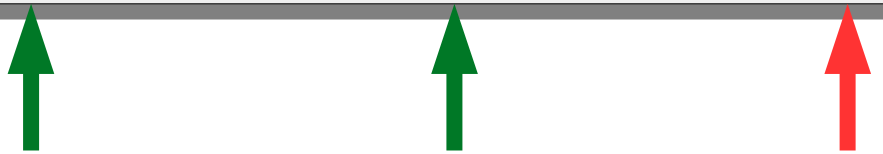
Matrix Multiplication

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      X[i][j] += Y[i][k] * Z[k][j];
```

Y[0,0], Z[0,0], Y[0,1], Z[1,0], Y[0,2], Z[2,0], ... X[0,0]
Y[0,0], Z[0,1], Y[0,1], Z[1,1], Y[0,2], Z[2,1], ... X[0,1]
...
...
Y[1,0], Z[0,0], Y[1,1], Z[1,0], Y[1,2], Z[2,0], ... X[1,0]

Matrix Multiplication

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      X[i][j] += Y[i][k] * Z[k][j];
```



Y[0,0], Z[0,0], Y[0,1], Z[1,0], Y[0,2], Z[2,0], ... X[0,0]
Y[0,0], Z[0,1], Y[0,1], Z[1,1], Y[0,2], Z[2,1], ... X[0,1]
...
...
Y[1,0], Z[0,0], Y[1,1], Z[1,0], Y[1,2], Z[2,0], ... X[1,0]

MM with Loop Interchange

```
double X[N][N], Y[N][N], Z[N][N];
  for (k=0;k<N;k++)
     for (j=0;j<N;j++)
        for (i=0;i<N;i++)
           X[i][j] += Y[i][k] * Z[k][j];
```

Z[k][j] can be loaded into a register once for each (k,j), reducing the number of memory references.

MM with Loop Interchange

```
double X[N][N], Y[N][N], Z[N][N];
  for (k=0;k<N;k++)
     for (j=0;j<N;j++)
        for (i=0;i<N;i++)
           X[i][j] += Y[i][k] * Z[k][j];
```

Z[k][j] can be loaded into a register once for each (k,j), reducing the number of memory references.

MM with Loop Unrolling

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      X[i][j] += Y[i][k] * Z[k][j];
```

MM with Loop Unrolling

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      X[i][j] += Y[i][k] * Z[k][j];
```

Unroll the **k** loop

MM with Loop Unrolling

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      X[i][j] += Y[i][k] * Z[k][j];
```

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k+=2)
```

Unroll the **k** loop

MM with Loop Unrolling

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      X[i][j] += Y[i][k] * Z[k][j];
```

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k+=2) {
      X[i][j] += Y[i][k] * Z[k][j];
      X[i][j] += Y[i][k+1] * Z[k+1][j];
     }
```

MM with Loop Unrolling

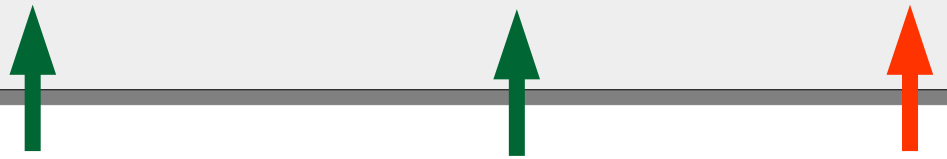
```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      X[i][j] += Y[i][k] * Z[k][j];
```

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k+=2) {
      X[i][j] += Y[i][k] * Z[k][j] +
                 Y[i][k+1] * Z[k+1][j];
     }
```

MM with Loop Unrolling

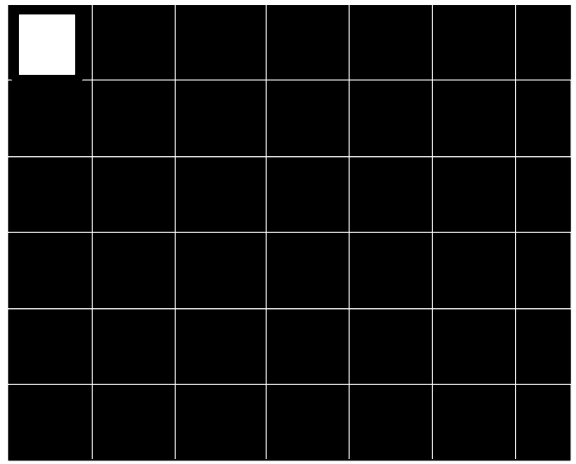
```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      X[i][j] += Y[i][k] * Z[k][j];
```

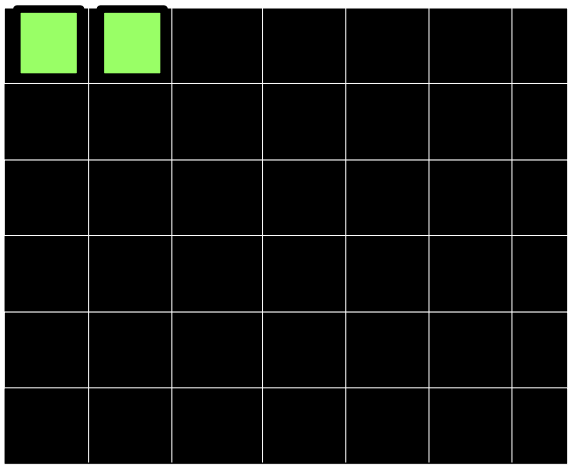
```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k+=2) {
      X[i][j] += Y[i][k] * Z[k][j] +
                 Y[i][k+1] * Z[k+1][j];
     }
```

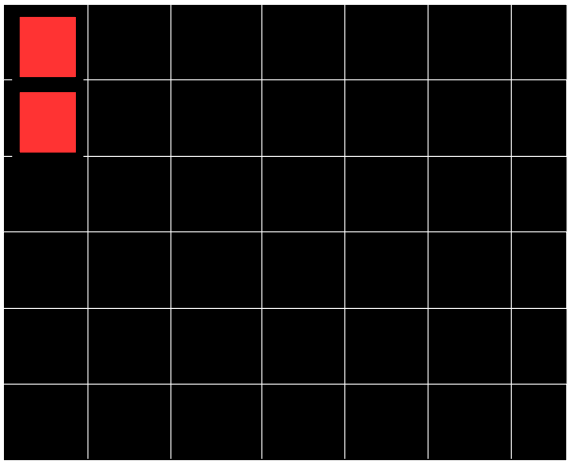


MM with Loop Unrolling

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k+=2) {
      X[i][j] += Y[i][k] * Z[k][j] +
                 Y[i][k+1] * Z[k+1][j];
     }
```



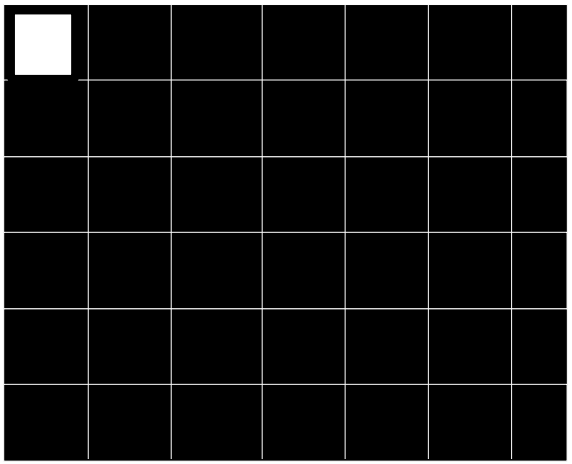




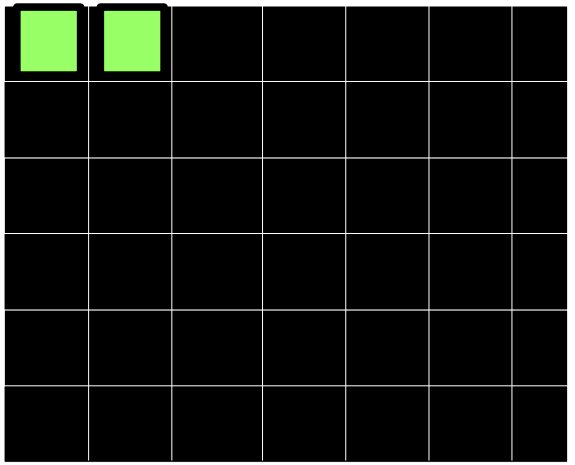
MM with Loop Unrolling

```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k+=2) {
      X[i][j] += Y[i][k] * Z[k][j] +
                 Y[i][k+1] * Z[k+1][j];
     }
```

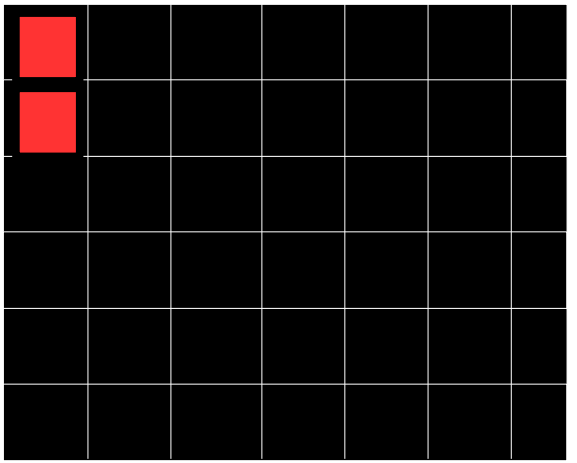
Unroll the **j** loop



X



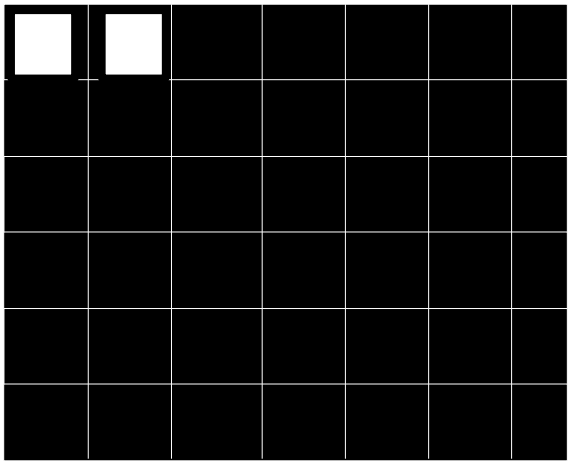
Y



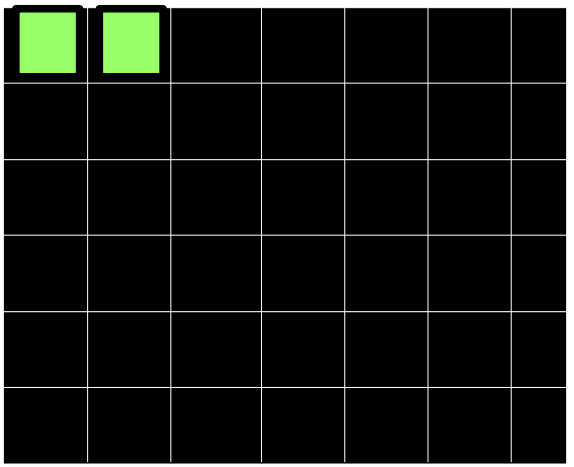
Z

MM with Loop Unrolling

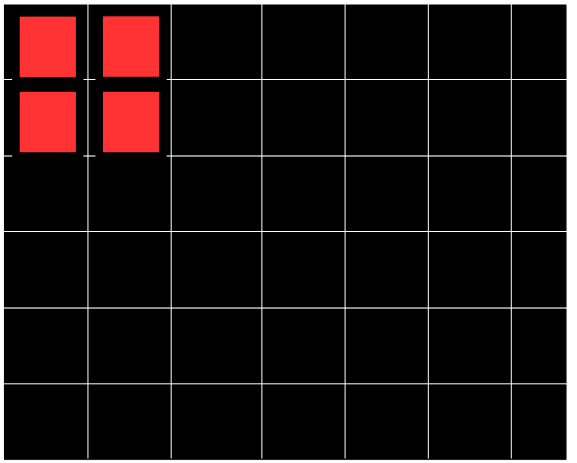
```
double X[N][N], Y[N][N], Z[N][N];
for (i=0;i<N;i++)
 for (j=0;j<N;j++)
  for (k=0;k<N;k+=2) {
   X[i][j] += Y[i][k] * Z[k][j] + Y[i][k+1] * Z[k+1][j];
   X[i][j+1]+=Y[i][k]*Z[k][j+1]+ Y[i][k+1]* Z[k+1][j+1];
  }
```



X

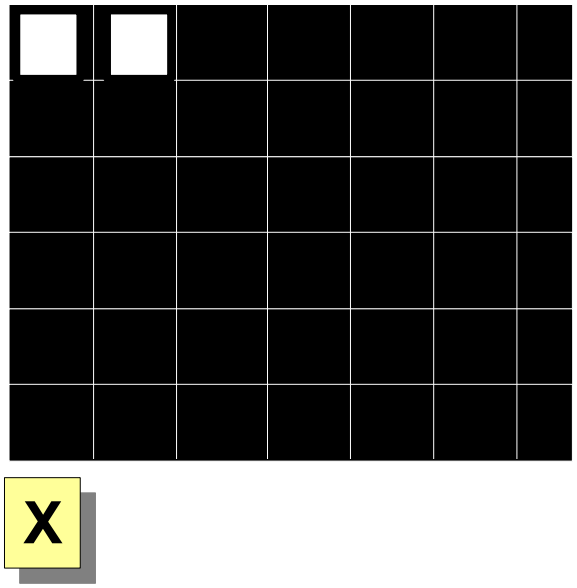


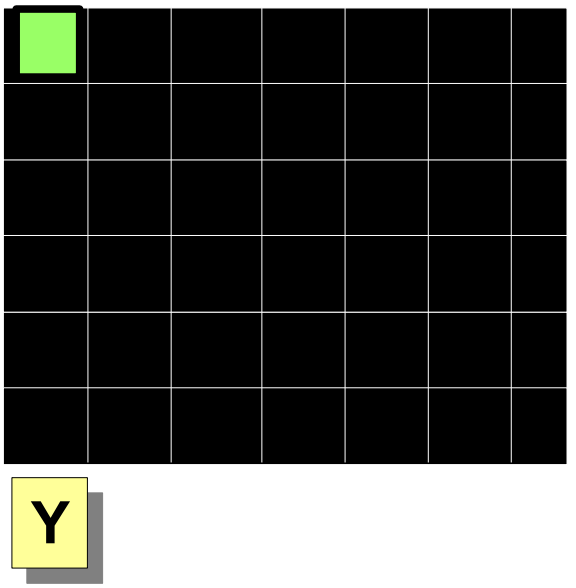
Y

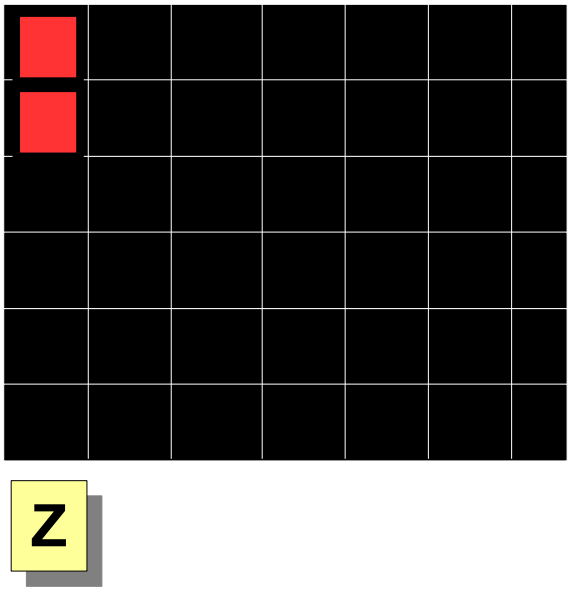


Z

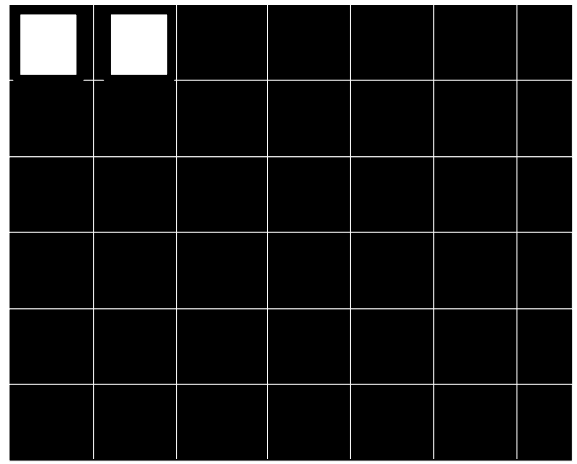
Blocking / Tiling



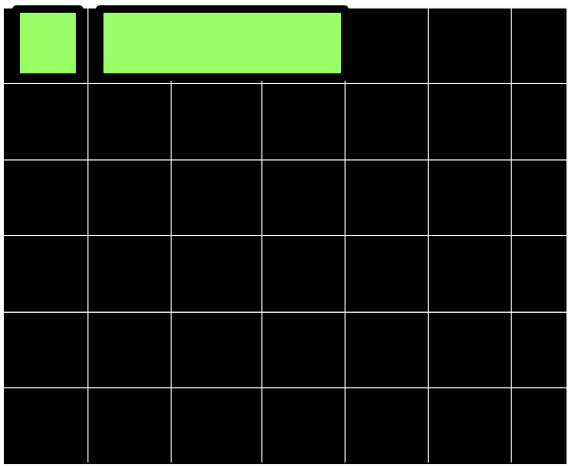




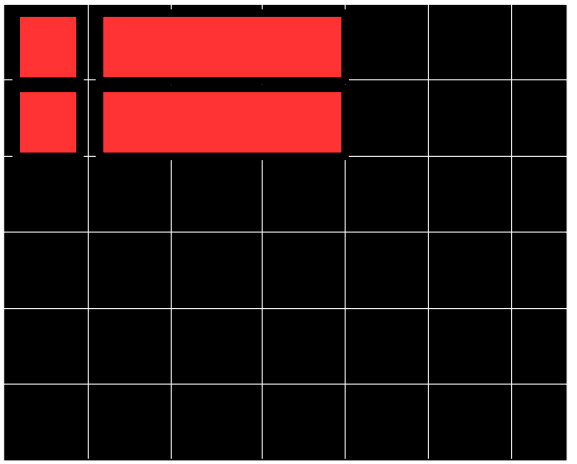
Blocking / Tiling



X

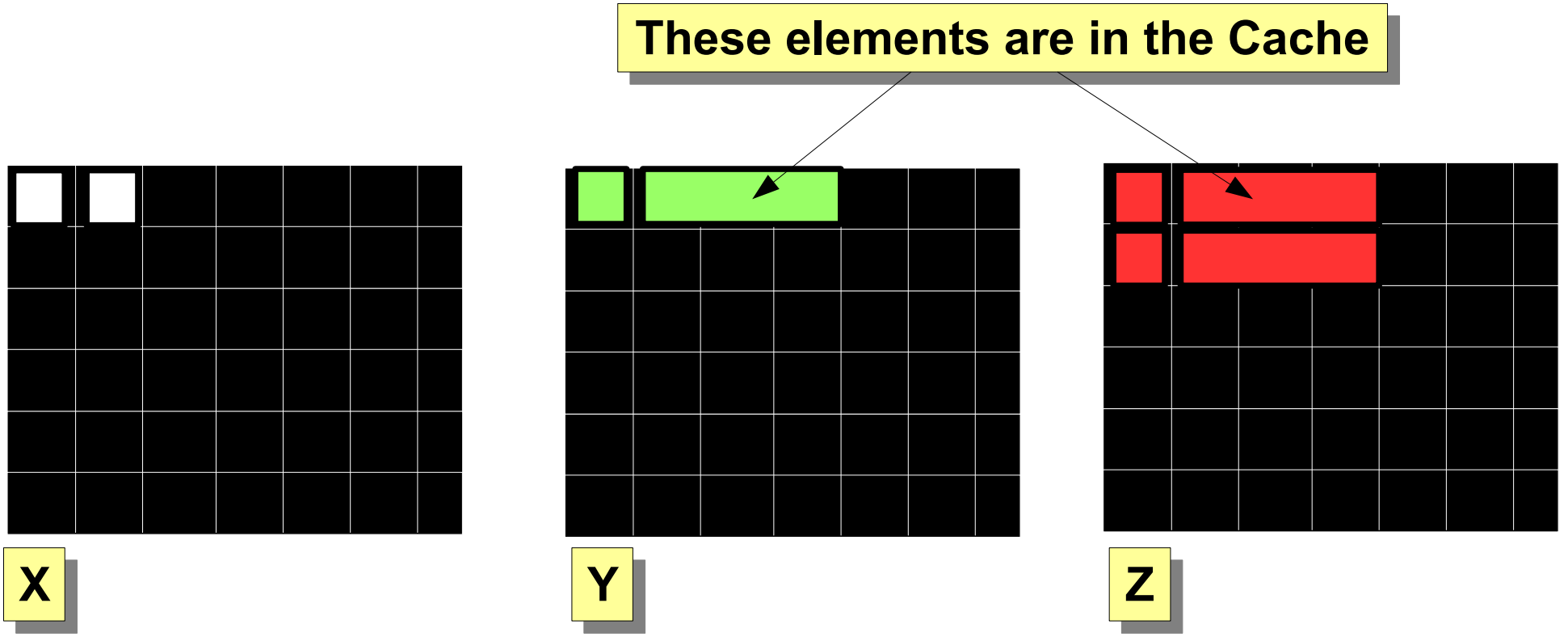


Y

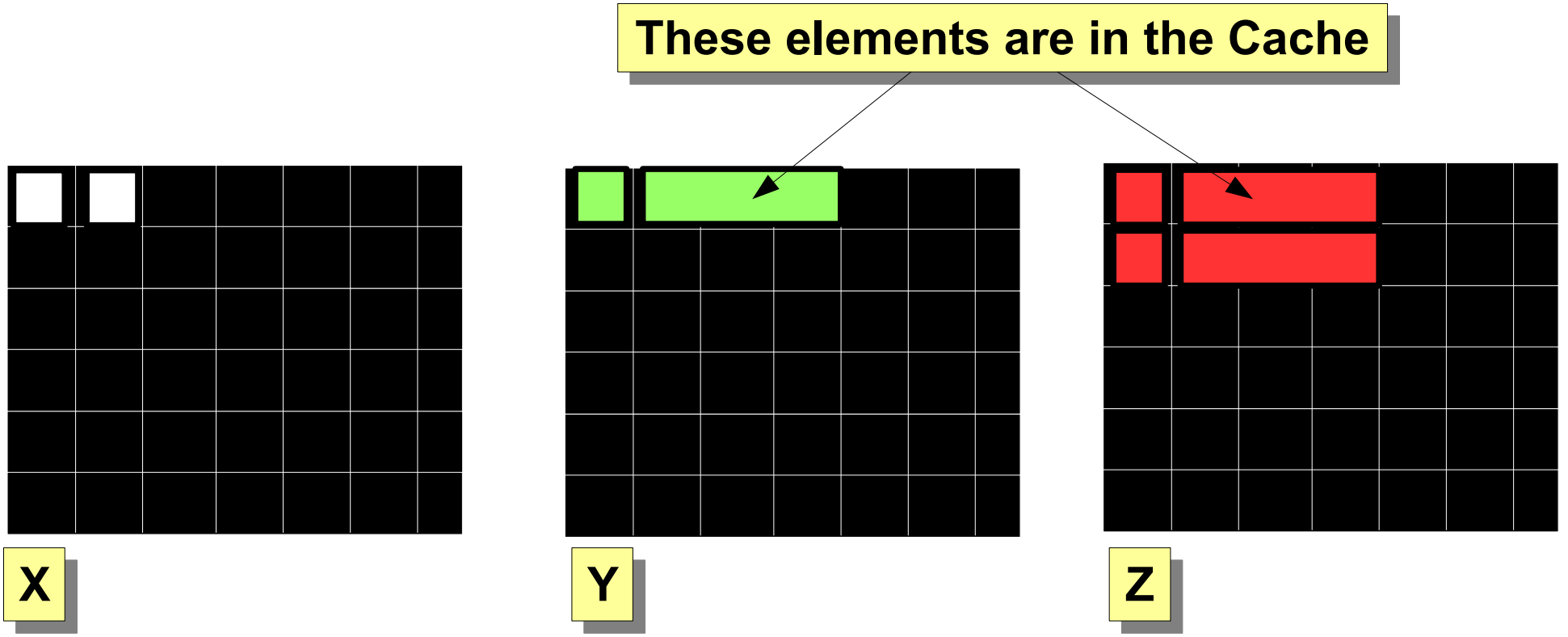


Z

Blocking / Tiling



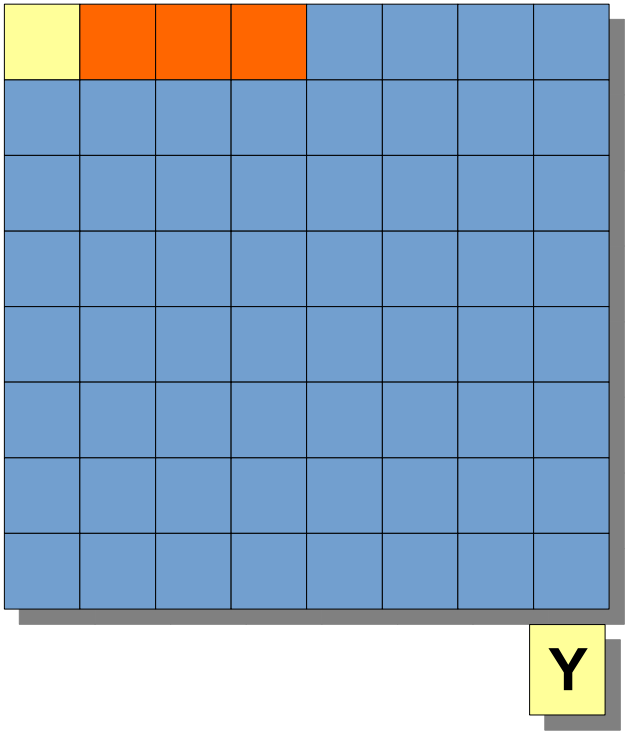
Blocking / Tiling

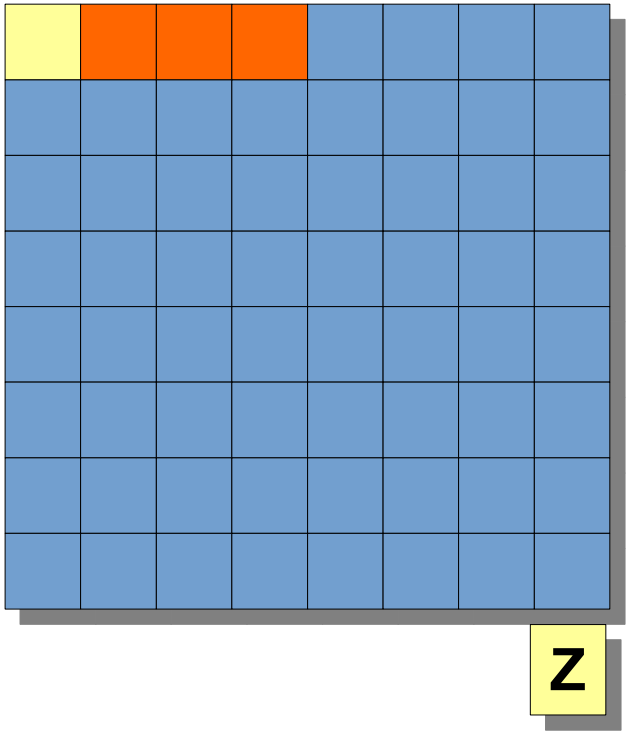


Idea of Blocking

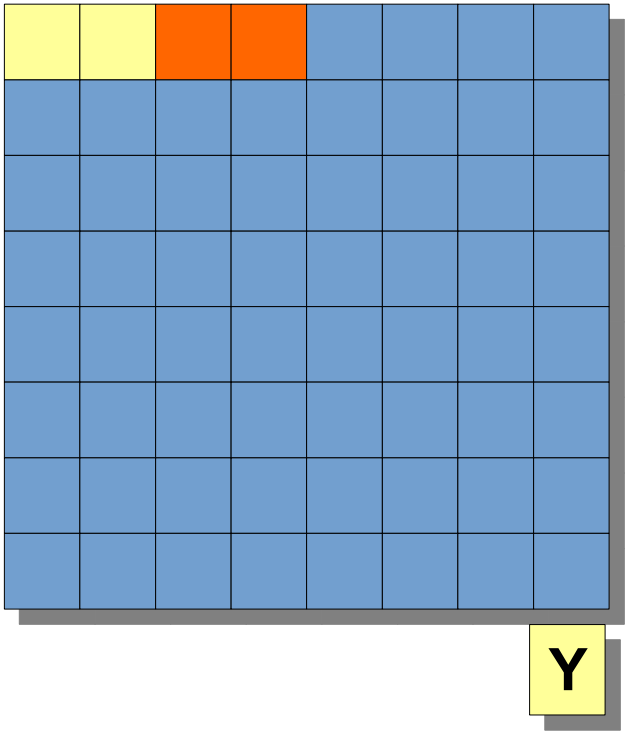
Make full use of elements of when they are brought into the cache

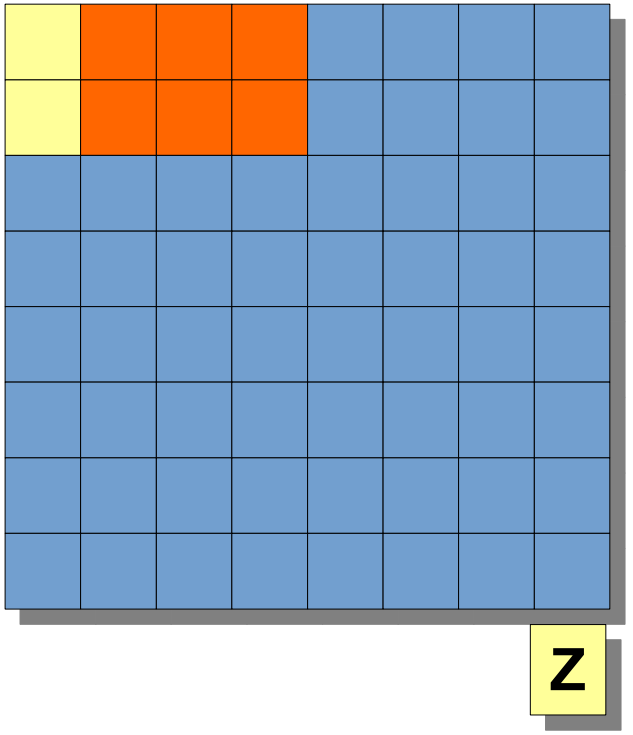
Blocking / Tiling



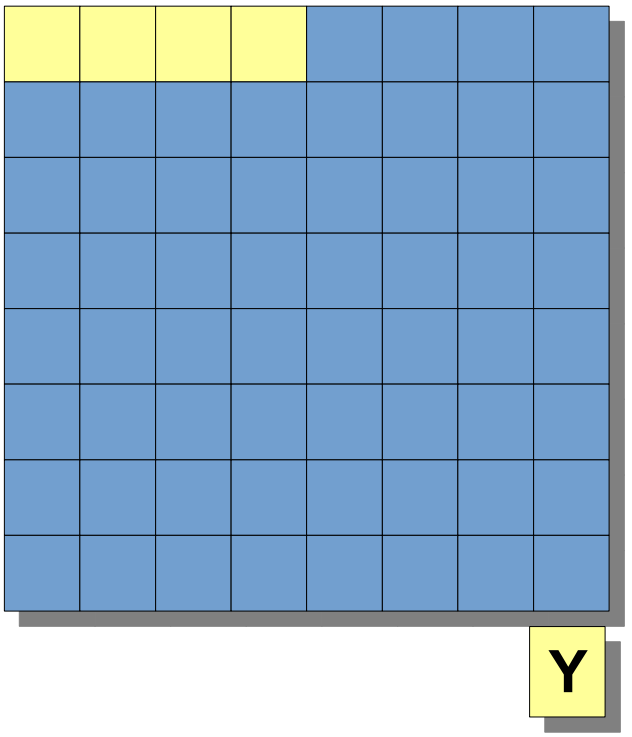


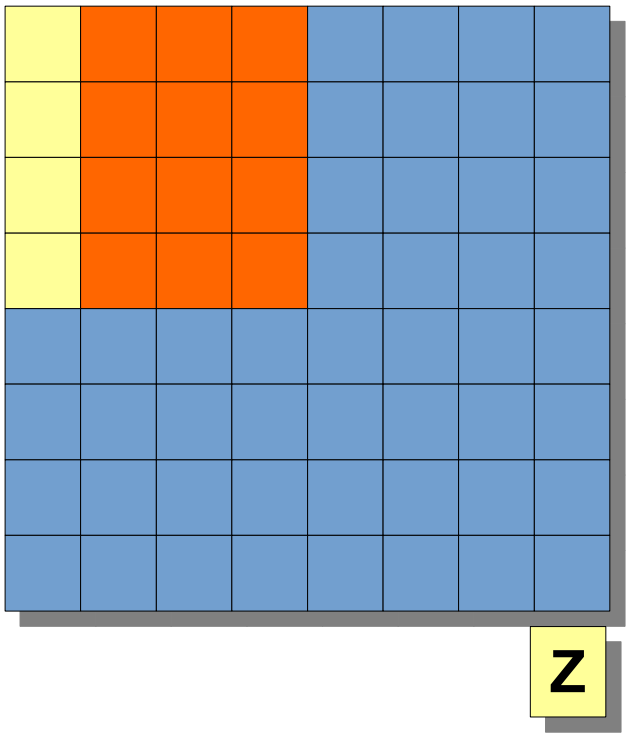
Blocking / Tiling



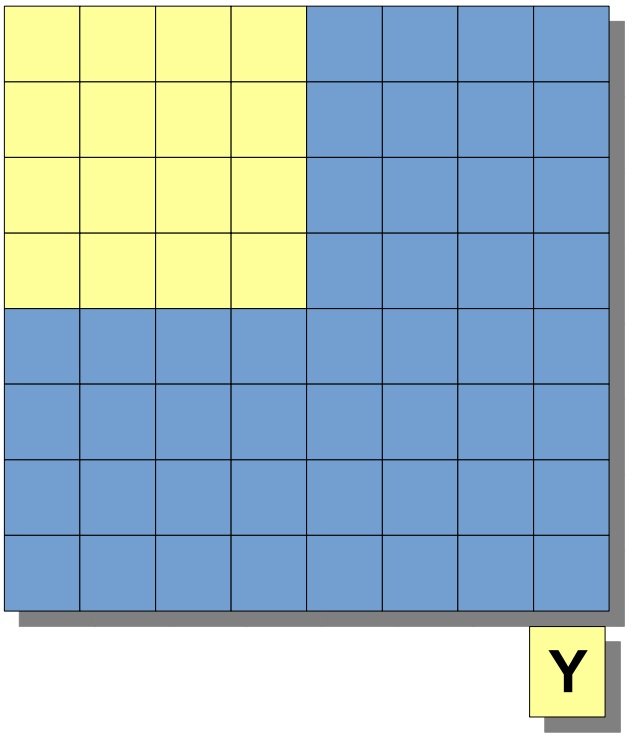


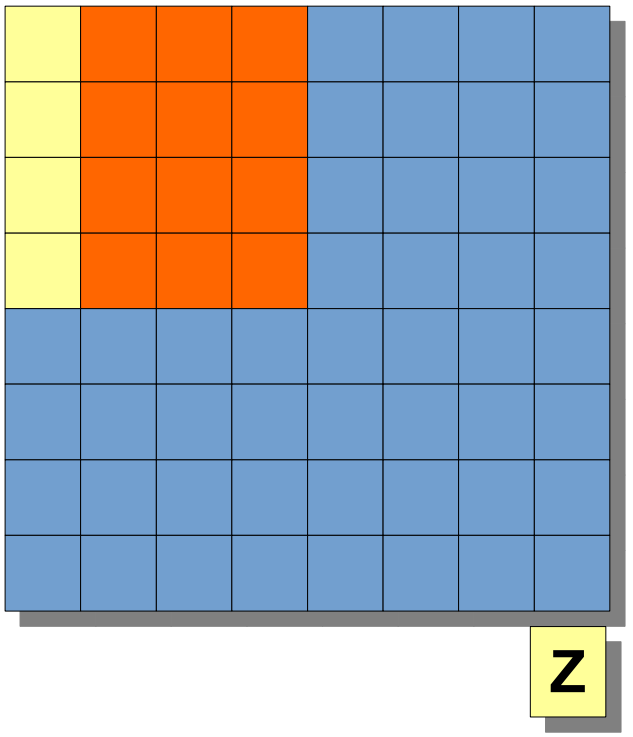
Blocking / Tiling





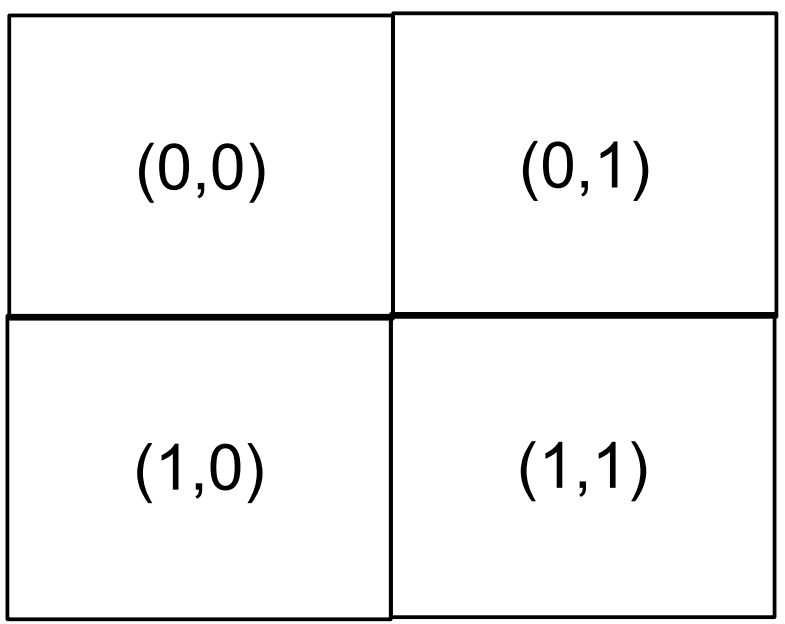
Blocking / Tiling



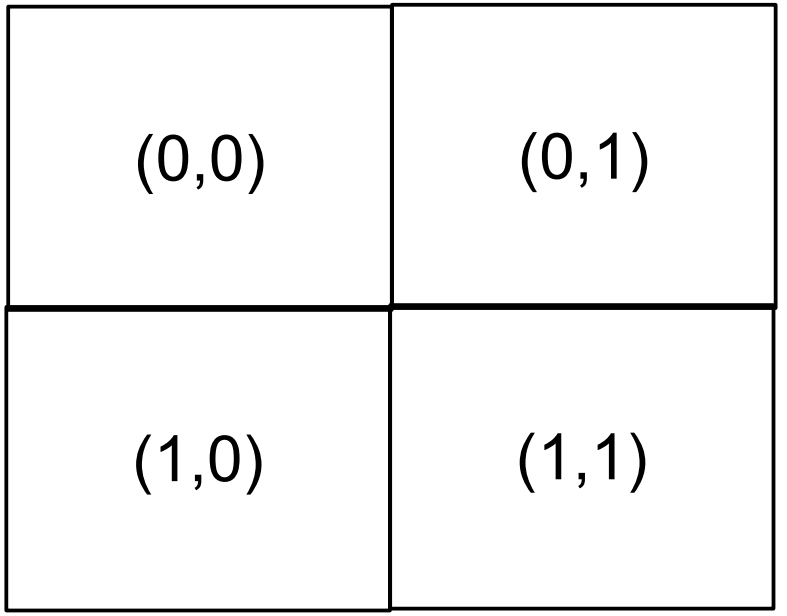


Blocking

- Make full use of the elements of Z when they are brought into the cache



Y



Z

X	Y x Z
0,0	0,0 x 0,0 + 0,1 x 1,0
1,0	1,0 x 0,0 + 1,1 x 1,0

Blocking

```
double X[N][N], Y[N][N], Z[N][N];

for (J=0; J<N; J+=B)
for (K=0; K<N; K+=B)

for (i=0; i<N; i++)
  for (j=J; j<min(J+B,N); j++)
    for (k=K,r=0; k<min(K+B,N); k++)
       r += Y[i][k] * Z[k][j];
    X[i][j] += r;
```

Slide Contents

- Cache Aware Programming – NPTEL Course on High Performance Computing, Prof. Matthew Jacob, IISc.

M2 – Outline

- Memory Hierarchy
- Cache Blocking – Cache Aware Programming
- SRAM, DRAM
- Virtual Memory
- Virtual Machines
- Non-volatile Memory, Persistent NVM