CSO-Recitation 15

CSCI-UA 0201-007

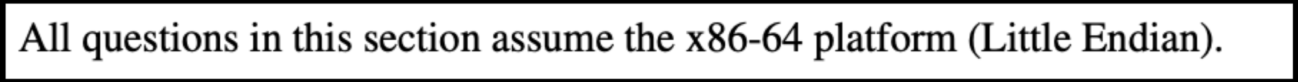
R15: Assessment 13 & Mock Exam

Today’s Topics

- Mock Exam
- Assessment 13
  - Review pipelined CPU

Mock Exam

1 Basic C and X86 64 machine instructions (28 points)



All questions in this section assume the x86-64 platform (Little Endian).

1-A

Given a binary sequence (00000110)2, what is its decimal value?

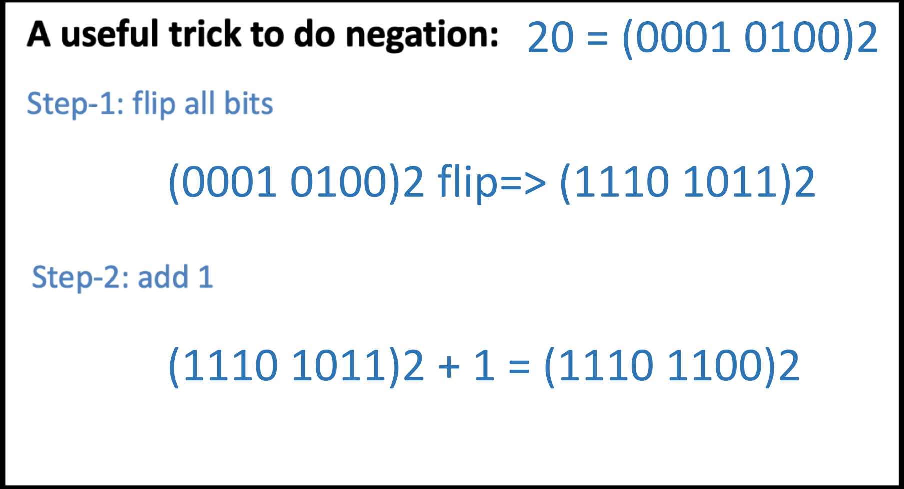
1. [✓ 6]
2. 5
3. 8
4. 10
5. None of the above

(00000110)2
= 1 * 2^2 + 1 * 2^1
= 6

1-B

Given a signed char −20, what is its binary representation?

1. (1001 0100)2
2. (1110 1011)2
3. [✓ (1110 1100)2]
4. (0001 0100)2
5. None of the above



A useful trick to do negation: 20 = (0001 0100)2

Step-1: flip all bits

(0001 0100)2 flip=> (1110 1011)2

Step-2: add 1

(1110 1011)2 + 1 = (1110 1100)2

1-C

Suppose %eax contains signed int 255. After successfully executing movl %eax, (%ecx), what is the byte value stored at the address given by %ecx?

1. 0xff
2. 0x00
3. 0xf0
4. 0x0f
5. None of the above

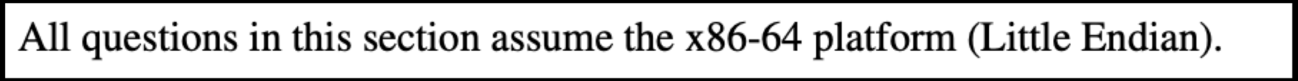
%eax & movl: 4 bytes
255 => 0x00 00 00 ff

%ecx

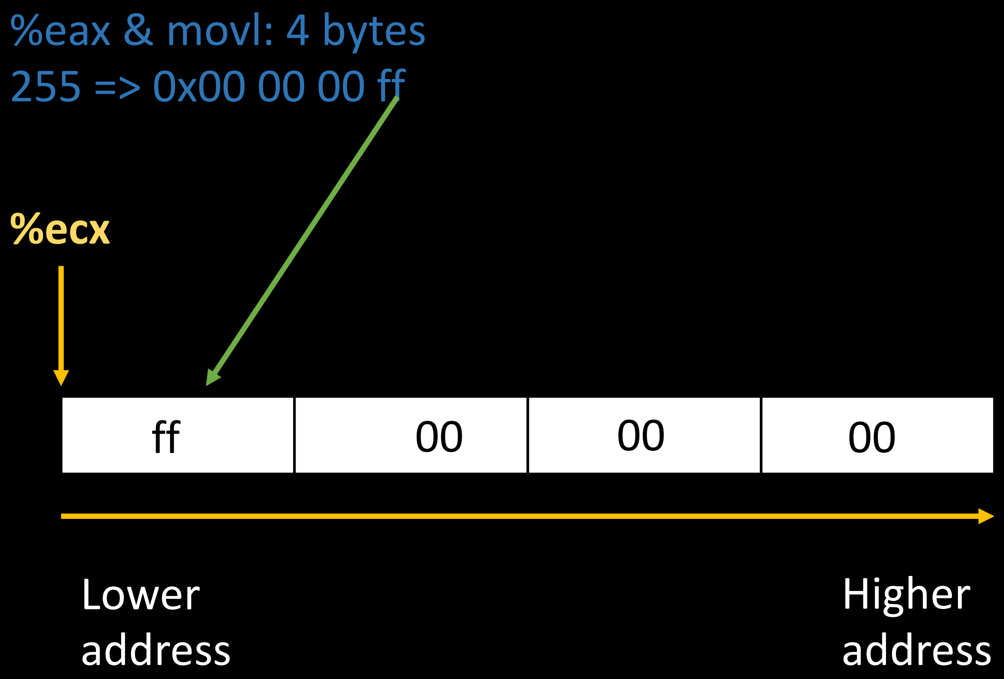
1-C

Suppose %eax contains signed int 255. After successfully executing movl %eax, (%ecx), what is the byte value stored at the address given by %ecx?

1. 0xff
2. 0x00
3. 0xf0
4. 0x0f
5. None of the above



All questions in this section assume the x86-64 platform (Little Endian).



%eax & movl: 4 bytes

255 => 0x00 00 00 ff

%ecx

ff	00	00	00

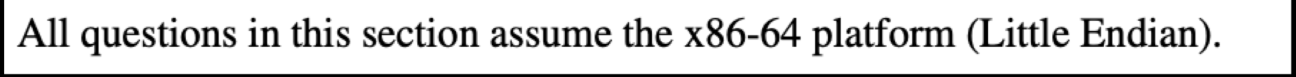
Lower address

Higher address

1-C

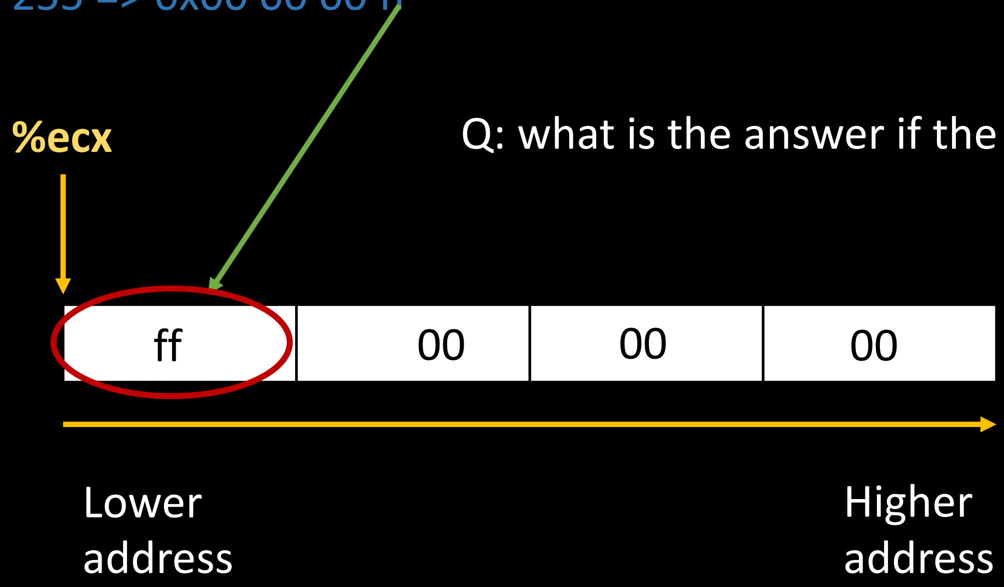
Suppose %eax contains signed int 255. After successfully executing movl %eax, (%ecx), what is the byte value stored at the address given by %ecx?

1. [✓ 0xff]
2. 0x00
3. 0xf0
4. 0x0f
5. None of the above



All questions in this section assume the x86-64 platform (Little Endian).

%eax & movl: 4 bytes
255 => 0x00 00 00 ff

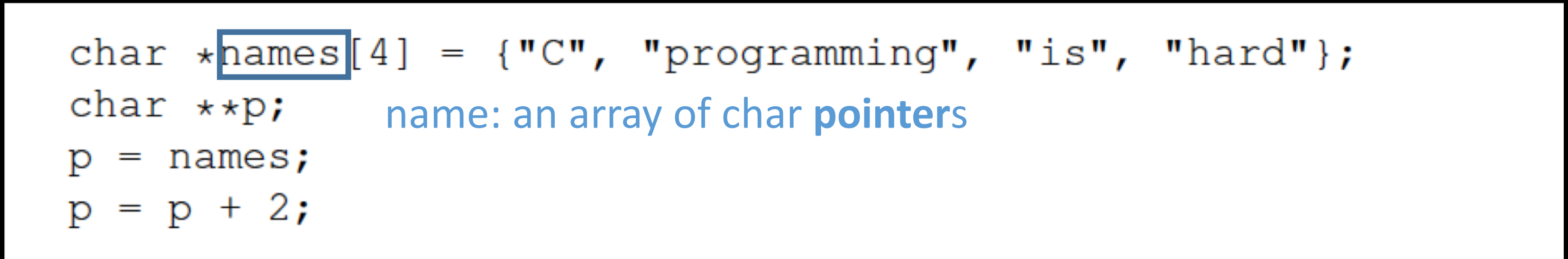


ff	00	00	00

Q: what is the answer if the machine is Big Endian?

1-D

Consider the following code snippet,

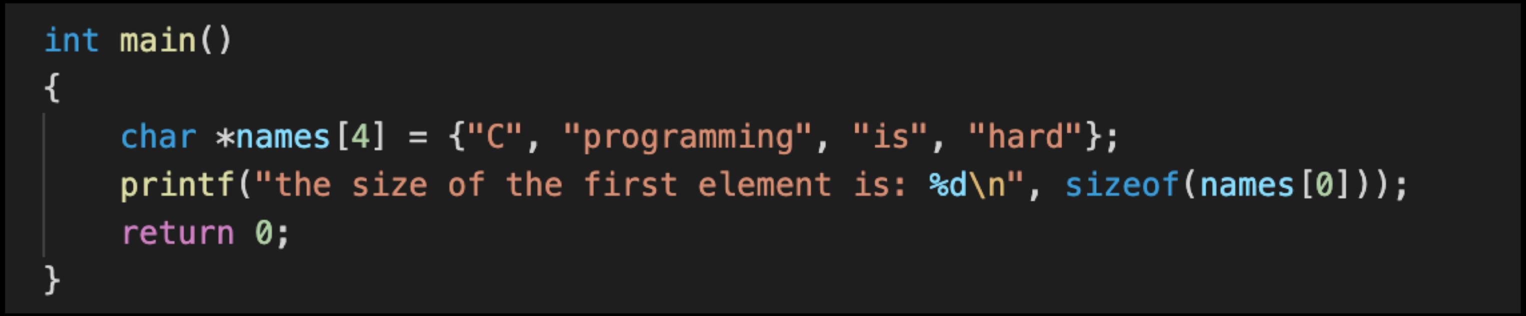


```
char *names[4] = {"C", "programming", "is", "hard"};
char **p;
p = names;
p = p + 2;
```

name: an array of char **pointers**

After executing the above, what is the value of p[1][1]?

Q1: what is the size of name[0]?



```
int main()
{
    char *names[4] = {"C", "programming", "is", "hard"};
    printf("the size of the first element is: %d\n", sizeof(names[0]));
    return 0;
}
```

1-D

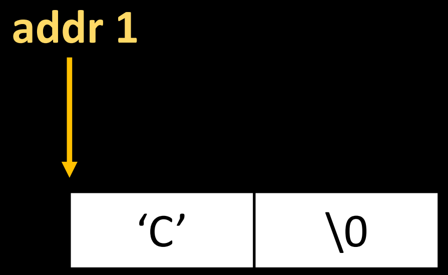
Consider the following code snippet,

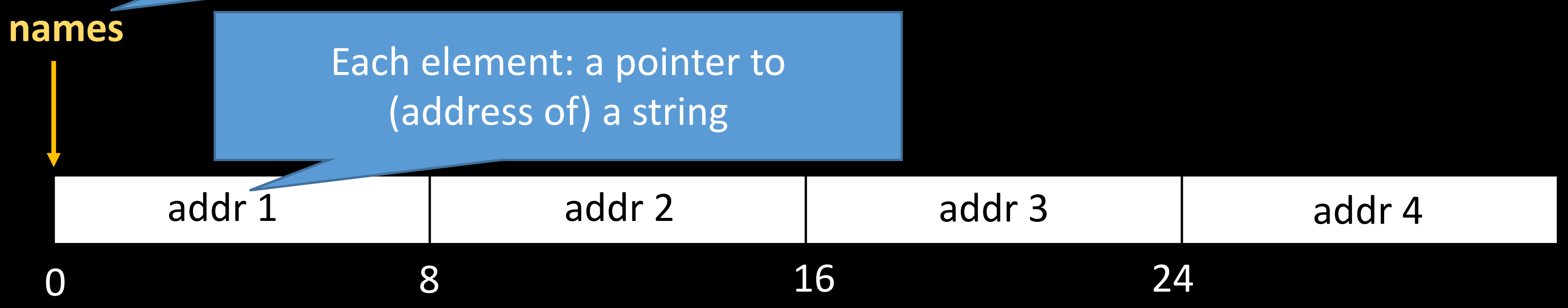
```
char *names[4] = {"C", "programming", "is", "hard"};
char **p;
p = names;
p = p + 2;
```

name: an array of char **pointers**

Af   what is the value of p[1][1]?

The name of an array ⇔ a pointer

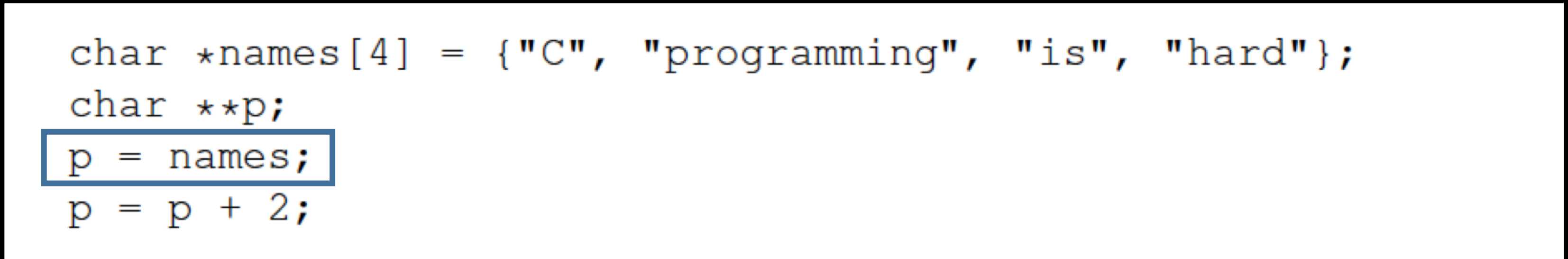




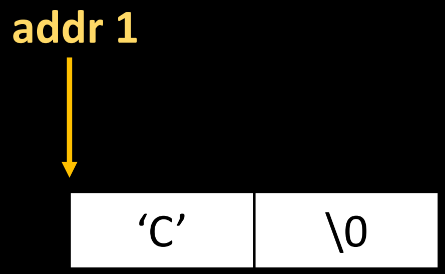
Each element: a pointer to (address of) a string

1-D

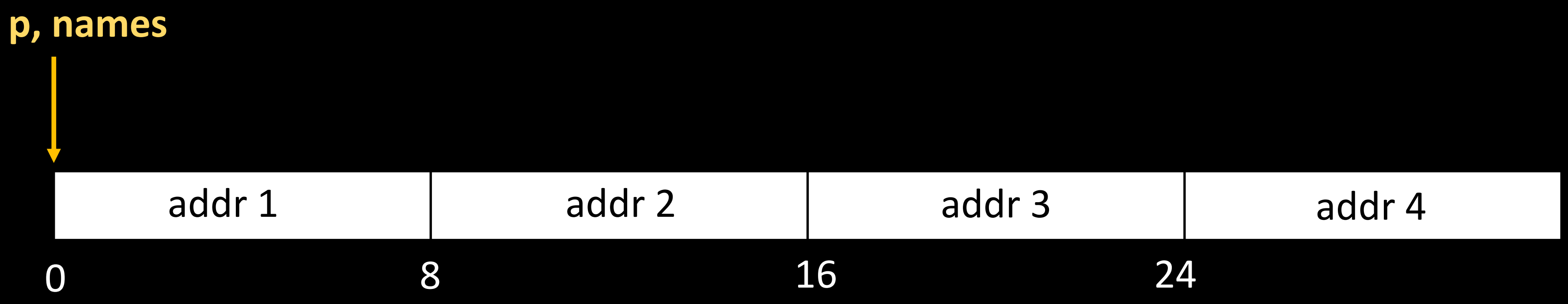
Consider the following code snippet,



```
char *names[4] = {"C", "programming", "is", "hard"};
char **p;
p = names;
p = p + 2;
```

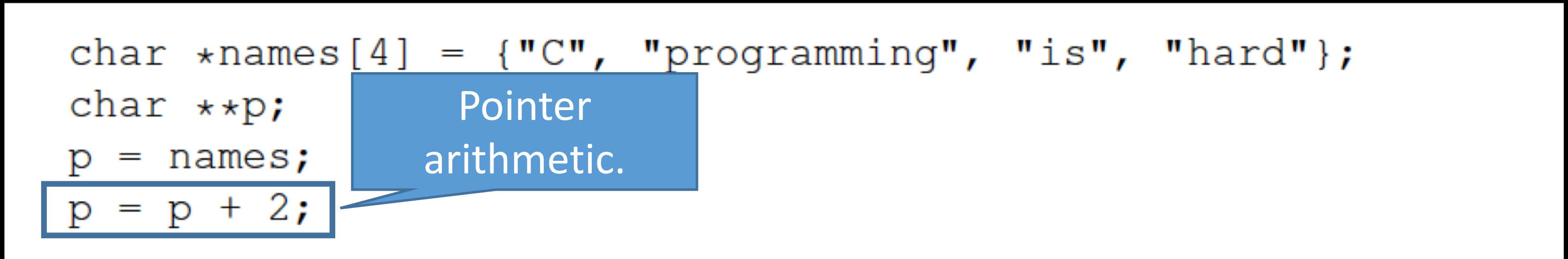


After executing the above, what is the value of p[1][1]?



1-D

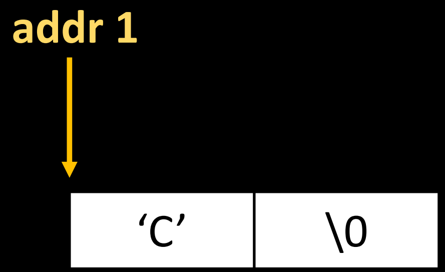
Consider the following code snippet,

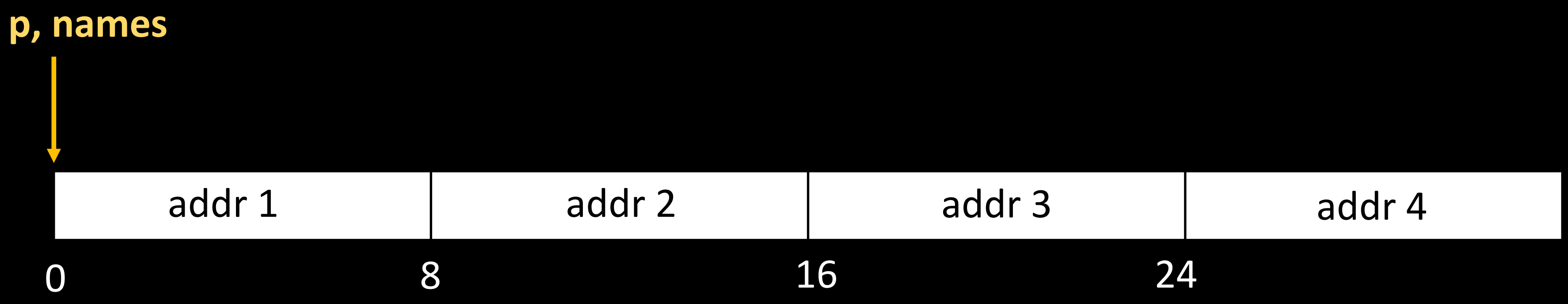


```
char *names[4] = {"C", "programming", "is", "hard"};
char **p;
p = names;
p = p + 2;
```

Pointer arithmetic.

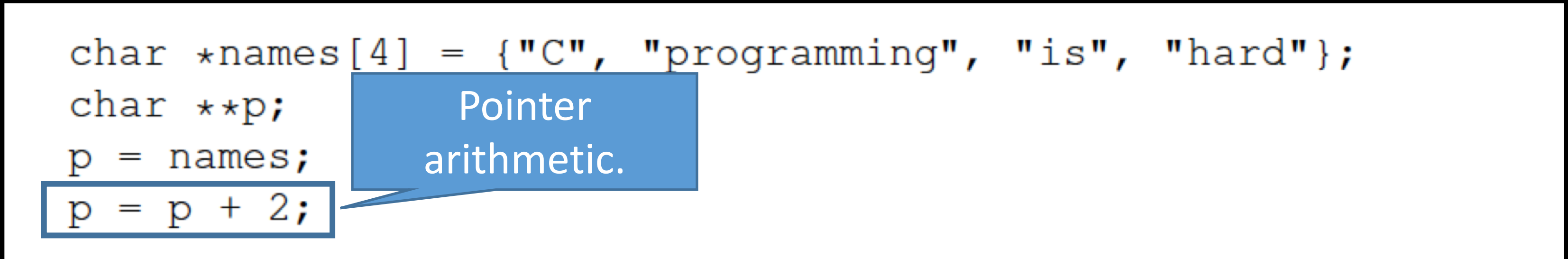
After executing the above, what is the value of p[1][1]?





1-D

Consider the following code snippet,



```
char *names[4] = {"C", "programming", "is", "hard"};
char **p;
p = names;
p = p + 2;
```

Pointer arithmetic.

After executing the above, what is the value of p[1][1]?

p: a pointer to **char ***

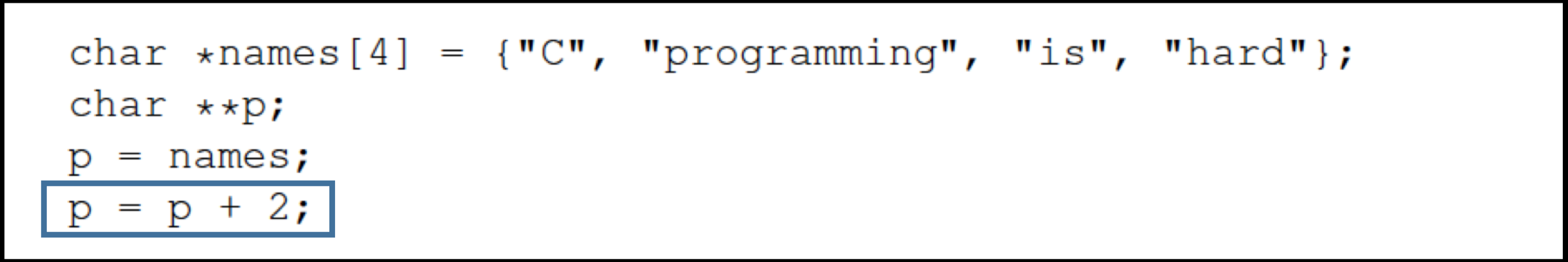
Pointer arithmetic: value of (p+2) = value of p + 2 * sizeof(char *)

char * type: pointer to a char => 8 bytes

value of (p+2) = value of p + 16

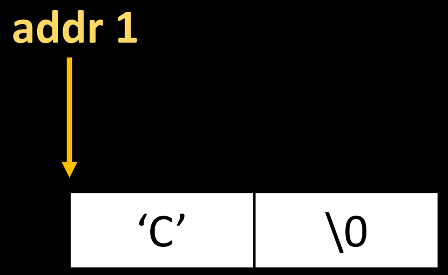
1-D

Consider the following code snippet,

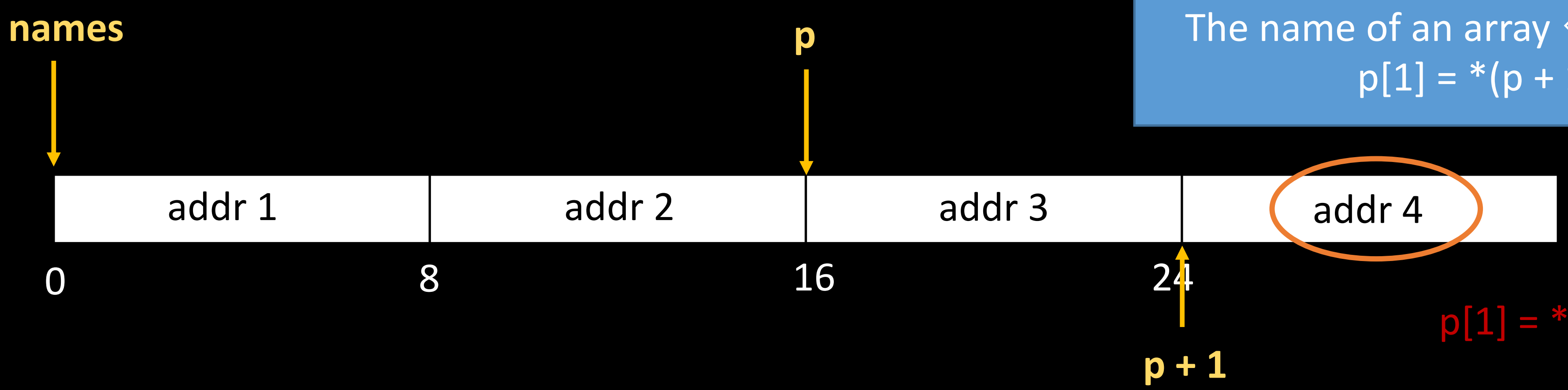


```
char *names[4] = {"C", "programming", "is", "hard"};
char **p;
p = names;
p = p + 2;
```

After executing the above, what is the value of p[1][1]?



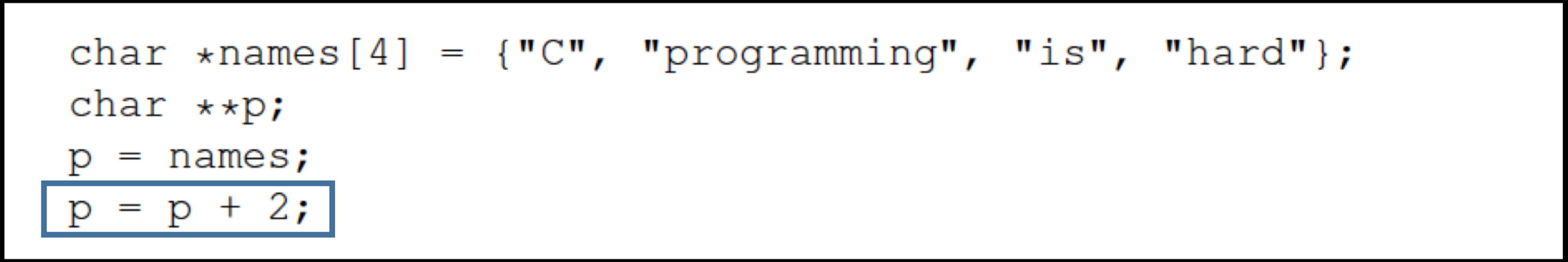
The name of an array ⇔ a pointer
p[1] = *(p + 1)



p[1] = *(p+1) = addr4

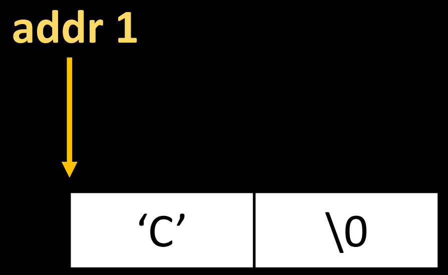
1-D

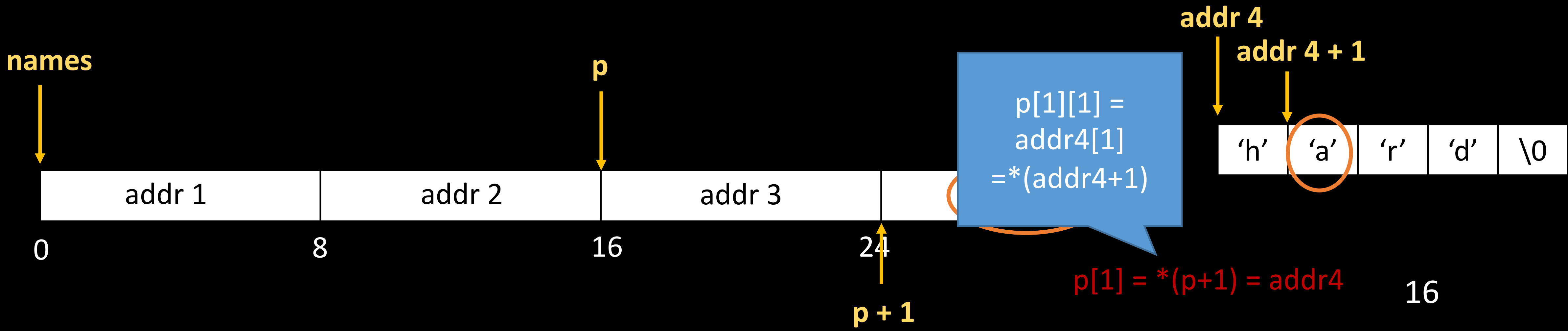
Consider the following code snippet,



```
char *names[4] = {"C", "programming", "is", "hard"};
char **p;
p = names;
p = p + 2;
```

After executing the above, what is the value of p[1][1]?



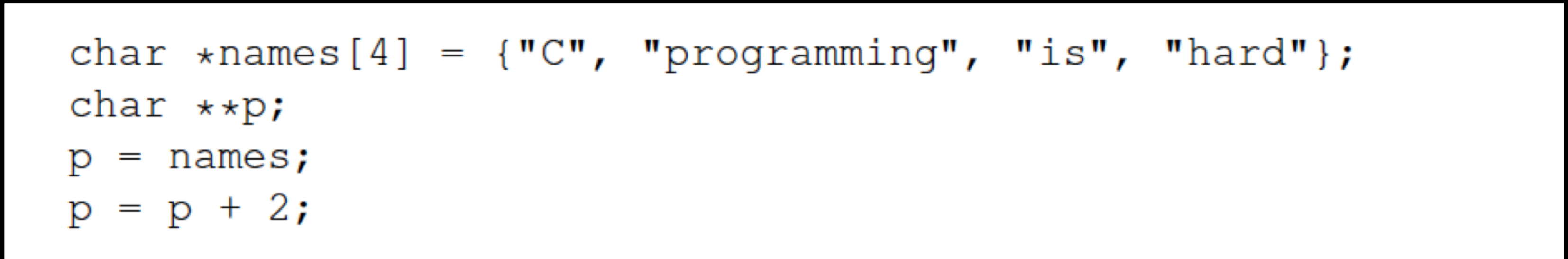


p[1][1] = addr4[1] =*(addr4+1)

p[1] = *(p+1) = addr4

1-D

Consider the following code snippet,



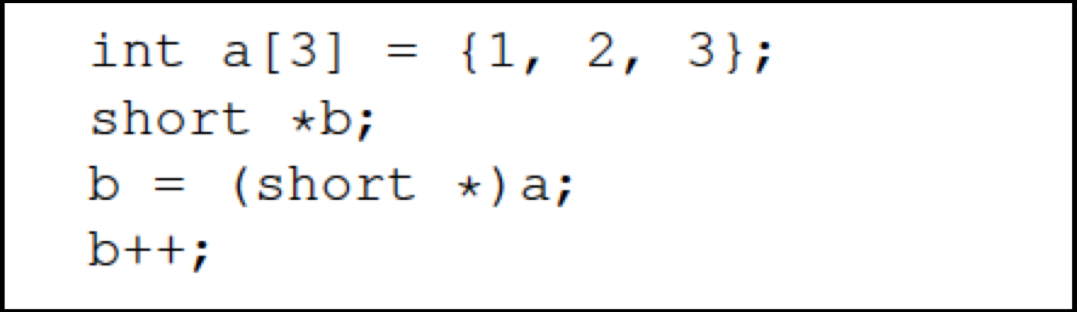
```
char *names[4] = {"C", "programming", "is", "hard"};
char **p;
p = names;
p = p + 2;
```

After executing the above, what is the value of p[1][1]?

‘a’

1-E

Consider the following code snippet,



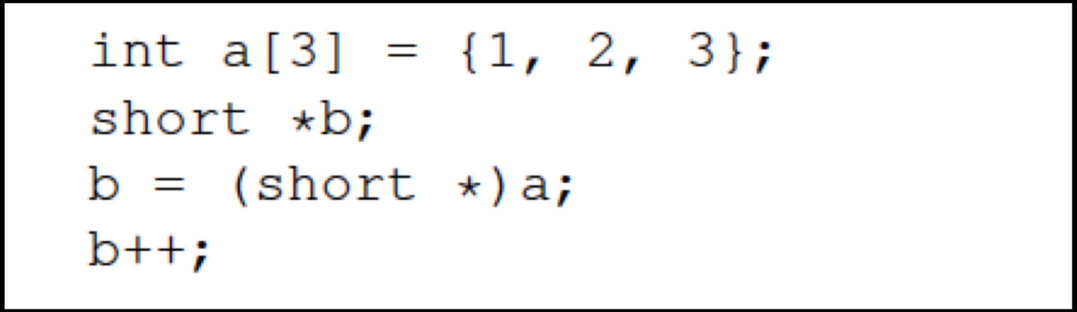
```
int a[3] = {1, 2, 3};
short *b;
b = (short *)a;
b++;
```

After executing the above, what is the value of *b?

Size of int: 4 bytes
Size of short: 2 bytes

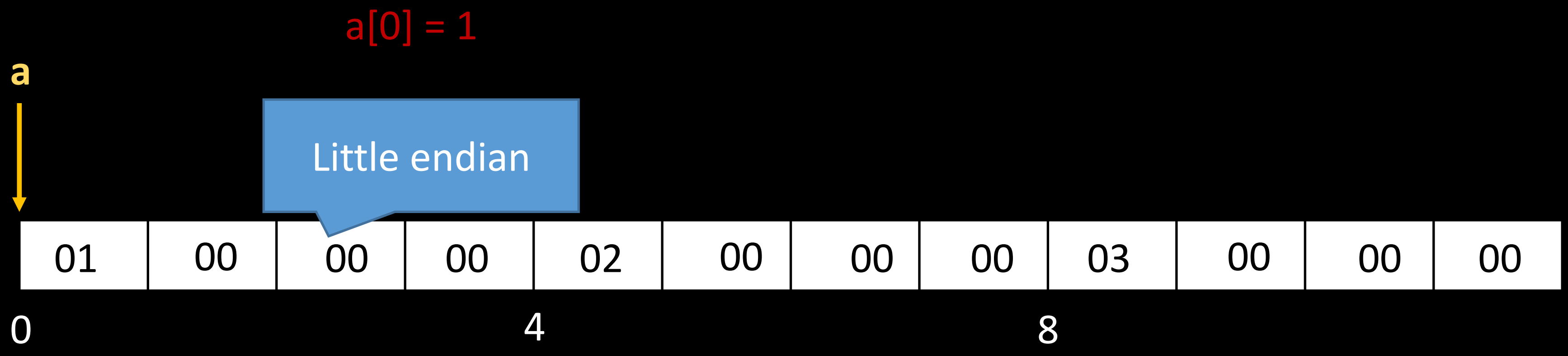
1-E

Consider the following code snippet,



```
int a[3] = {1, 2, 3};
short *b;
b = (short *)a;
b++;
```

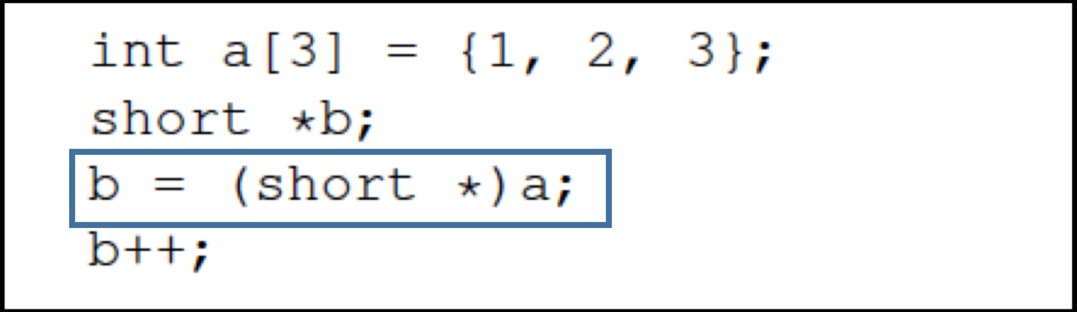
After executing the above, what is the value of *b?



01	00	00	00	02	00	00	00	03	00	00	00

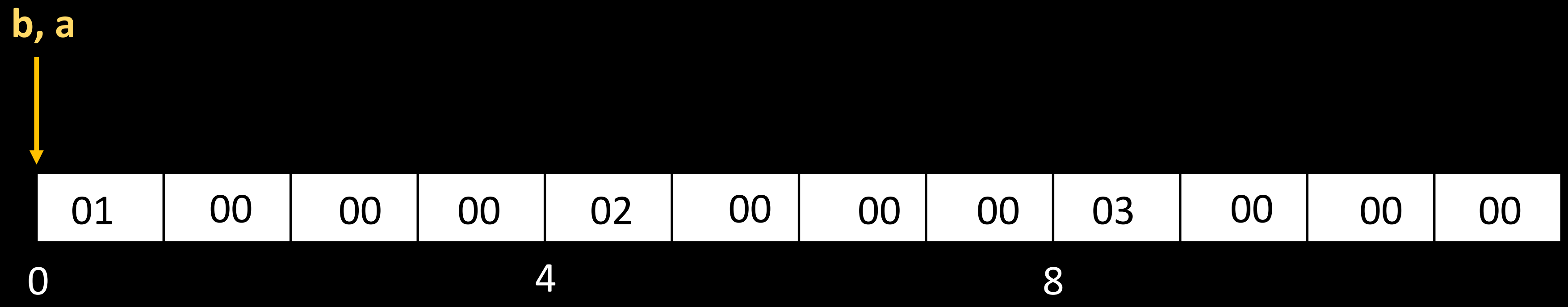
1-E

Consider the following code snippet,



```
int a[3] = {1, 2, 3};
short *b;
b = (short *)a;
b++;
```

After executing the above, what is the value of *b?

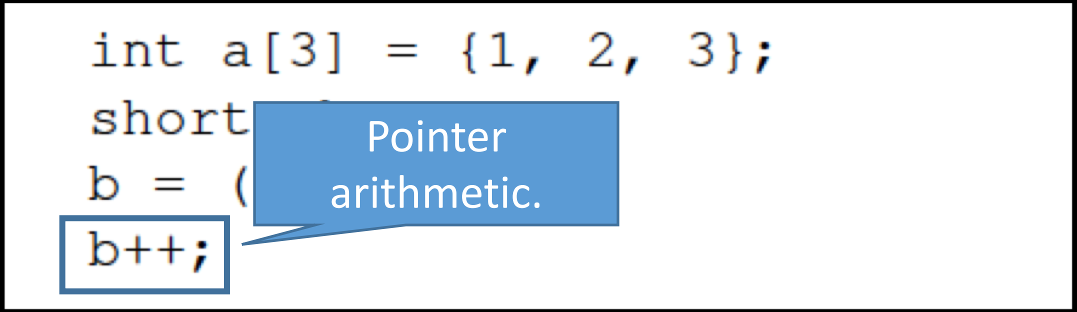


b, a

01	00	00	00	02	00	00	00	03	00	00	00
0				4				8			

1-E

Consider the following code snippet,



```
int a[3] = {1, 2, 3};
short
b = (
b++;
```

Pointer arithmetic.

After executing the above, what is the value of *b?

b: a pointer to **short**

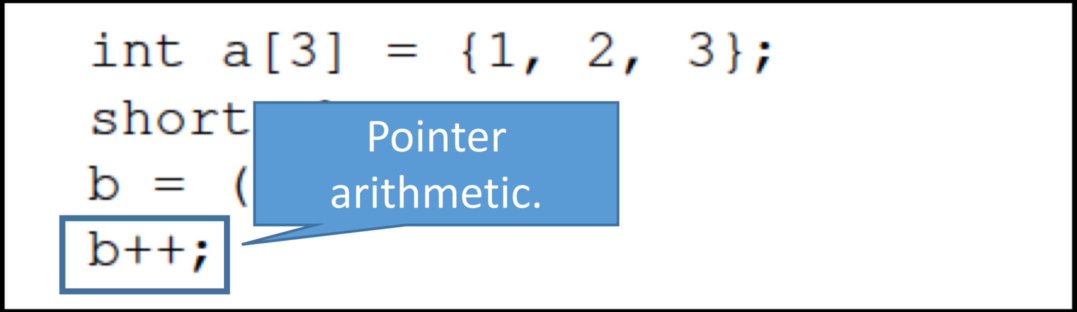
Pointer arithmetic: value of (b+1) = value of b + 1 * sizeof(short)

short type: 2 bytes

value of (b+1) = value of b + 2

1-E

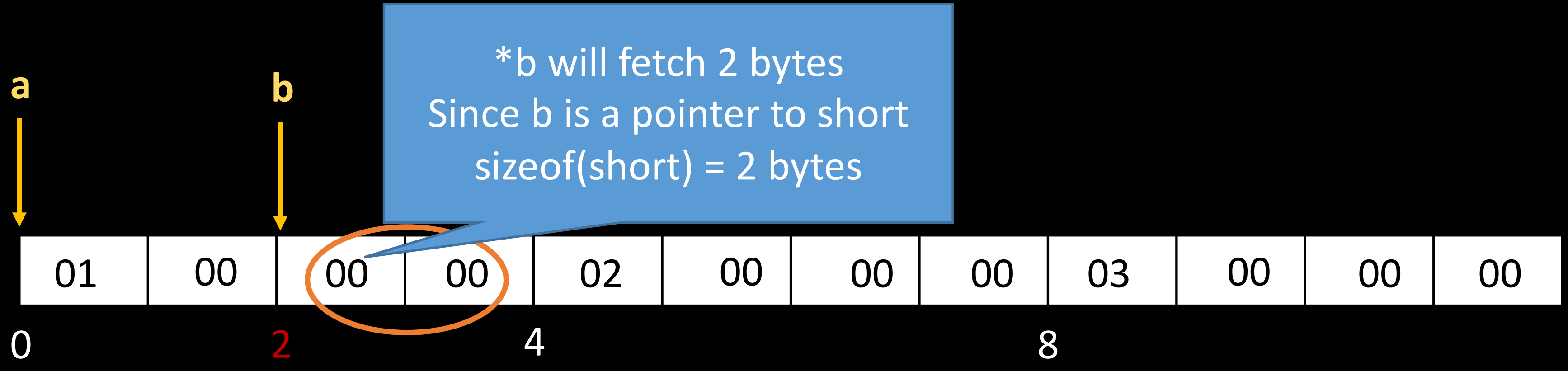
Consider the following code snippet,



```
int a[3] = {1, 2, 3};
short
b = (
b++;
```

Pointer arithmetic.

After executing the above, what is the value of *b?

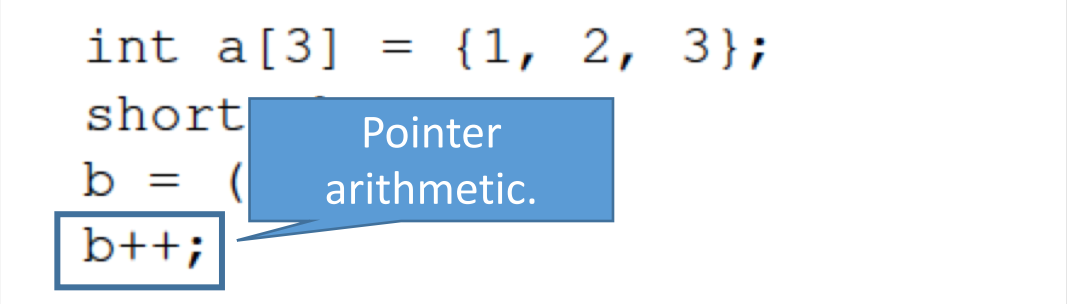


*b will fetch 2 bytes
Since b is a pointer to short
sizeof(short) = 2 bytes

01	00	00	00	02	00	00	00	03	00	00	00
0		2		4				8			

1-E

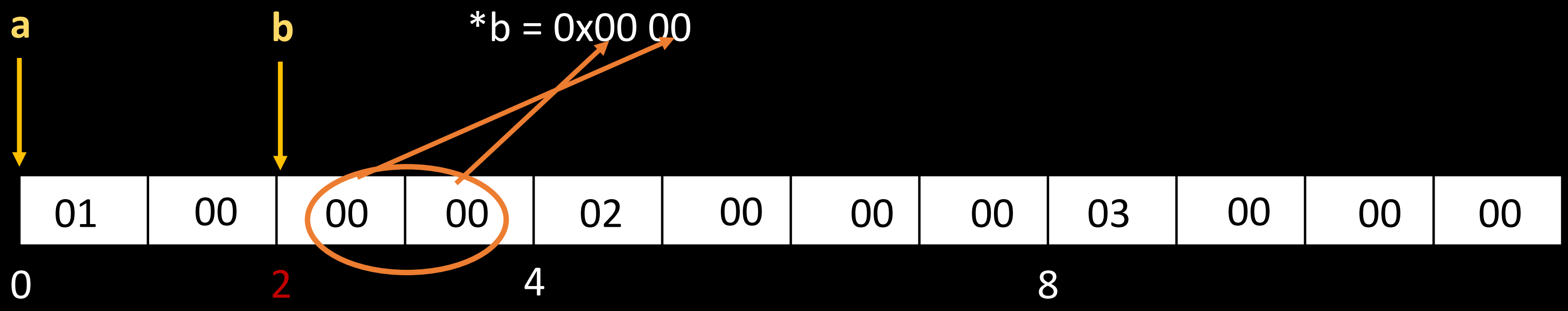
Consider the following code snippet,



```
int a[3] = {1, 2, 3};
short
b = (
b++;
```

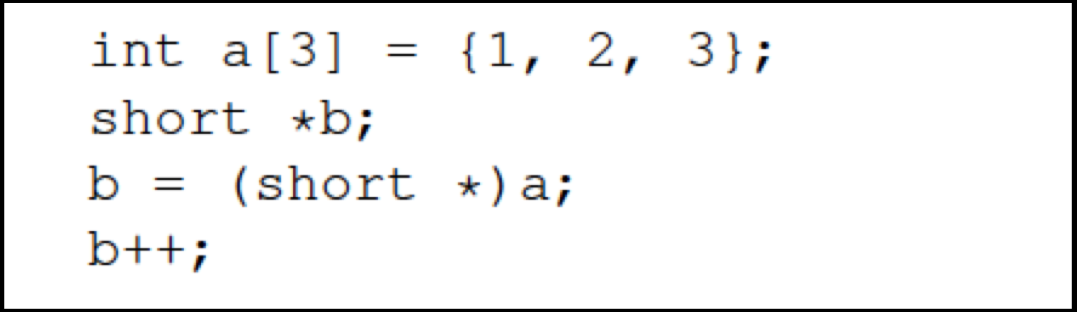
Pointer arithmetic.

After executing the above, what is the value of *b?



1-E

Consider the following code snippet,



```
int a[3] = {1, 2, 3};
short *b;
b = (short *)a;
b++;
```

After executing the above, what is the value of *b?

0

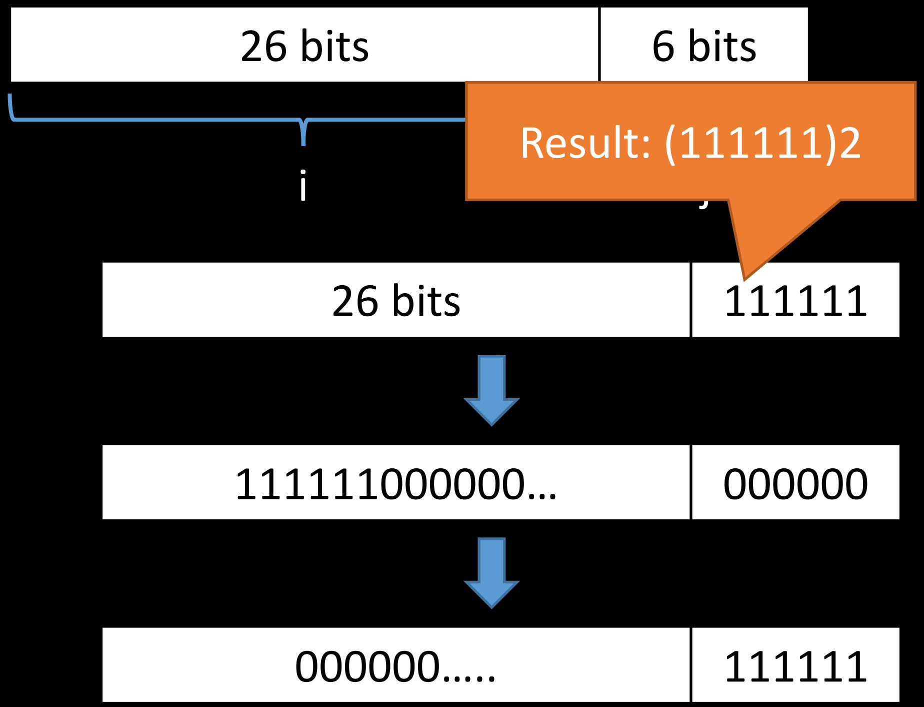
1-F

Which following expressions compute the remainder of x modulo 64 (x is of type unsigned int)?

1. [✓ x % 64]
2. x / 64
3. x >> 6
4. [✓ x & 0x0000003f]
5. [✓ (x << 26) >> 26]
6. None of the above

0x3F: 00111111

x = i * 64 + j, 0<= j < 64
x: 32bits, 64 = 2^6



26 bits | 6 bits

i

j

Result: (111111)2

26 bits | 111111

111111000000… | 000000

000000….. | 111111

1-G

Which of the follo e w.r.t. malloc?

First try to search in the current heap to see whether there is an available free chunk.

✗ 1. Every call to malloc results in the memory all (e.g. sbrk) to request memory from OS.

If there is no free chunks, it will make a syscall (sbrk) to ask for more space in the heap.

✗ 2. malloc returns emory allocator does not have any free c

Implicit-list: potentially traverse all chunks (both free and allocated)
Explicit-list: only traverse free chunks.

[✓ 3.] When using the implicit-list design, malloc tends to traverse more chunks than when using the explicit-list design.

4. None of the above.

2 Memory hierarchy
(25 points + bonus 10 points)

2-A

What is the latency to access L1 cache and main memory, respectively? Do not forget to write down the time unit. (We consider the answer correct if it's within a factor of 10)

L1 cache: SRAM 0.5-2.5ns
Main memory: DRAM 50-70ns

2-B

The rest of the questions assume a cache whose total size is 1KB and each cache line/block is 64-byte.

How many cacheline/blocks does the cache contain?

(total size)/(size of each line) = 1KB/64 bytes = 16

2-C

The rest of the question otal size is 1KB and each cache line/block is

1 cacheline per set
16 cachelines => 16 sets

Suppose the cache is a direct mapped cache. Given a 32-bit address 0xab345f78, which cacheline/block contains the cached data for this address?

Used to index which byte **within a specific cacheline**.
2^(# of bits) = how many bytes in total in a cacheline

Tag	index	Byte offset
		# of bits = log2(64) = 6

2-C

The rest of the question otal size is 1KB and each cache line/block is

1 cacheline per set
16 cachelines => 16 sets

Suppose the cache is a direct mapped cache. Given a 32-bit address 0xab345f78, which cacheline/block contains the cached data for this address?

Used to index which set caches this data.
2^(# of bits) = how many sets in total

Tag	index	Byte offset
	# of bits = log2(16) = 4	# of bits = log2(64) = 6

2-C

The rest of the question otal size is 1KB and each cache line/block is

1 cacheline per set
16 cachelines => 16 sets

Suppose the cache is a direct mapped cache. Given a 32-bit address 0xab345f78, which cacheline/block contains the cached data for this address?

Used to distinguish different addresses which are mapped to the same set.
# of bits = total bits – index - byte offset

Tag	index	Byte offset
# of bits = 32 – 4 – 6 = 22	# of bits = log2(16) = 4	# of bits = log2(64) = 6

2-C

The rest of the questions assume a cache whose total size is 1KB and each cache line/block is 64-byte.

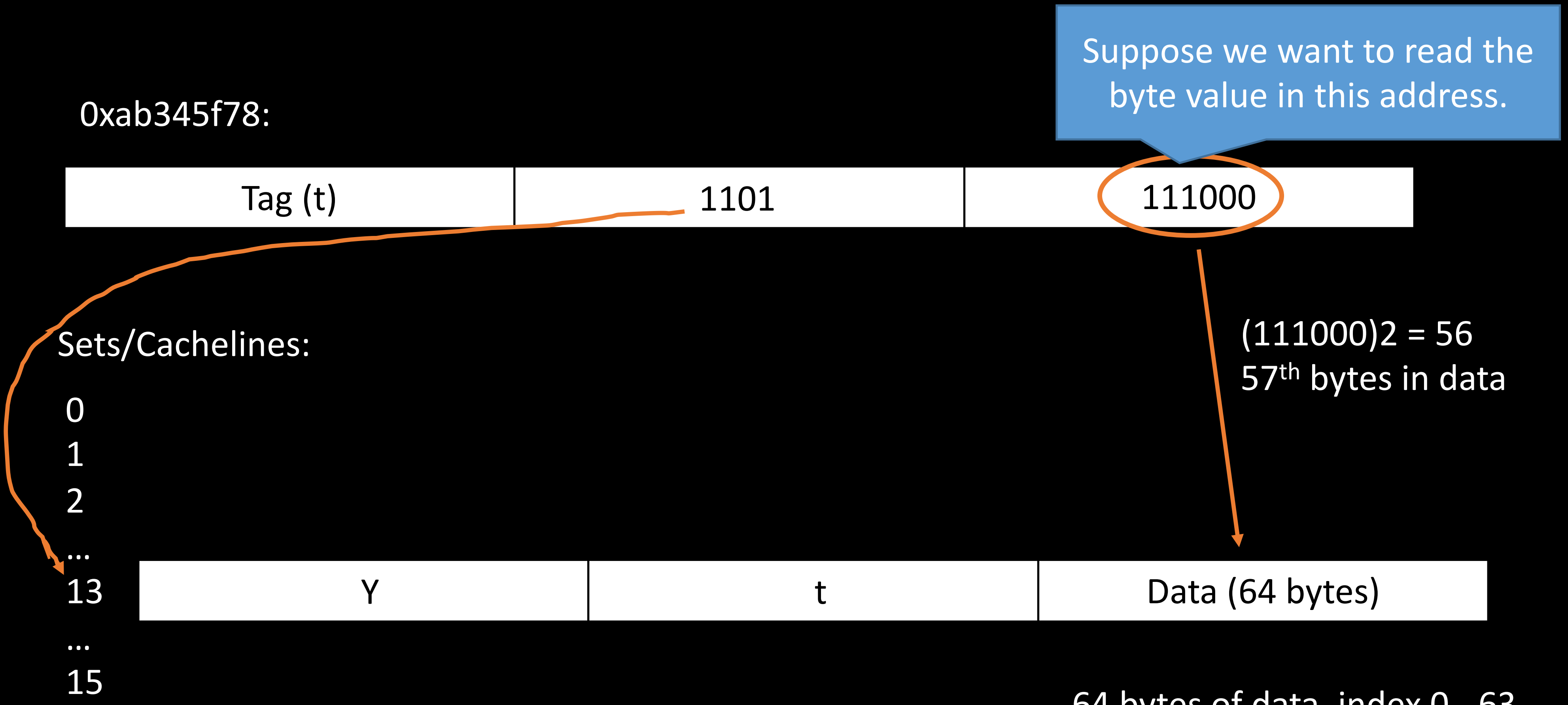
Suppose the cache is a direct mapped cache. Given a 32-bit address 0xab345f78, which cacheline/block contains the cached data for this address?

13

0xab345f78:

Tag	1101	111000
# of bits = 32 – 4 – 6 = 22	# of bits = log2(16) = 4	# of bits = log2(64) = 6

2-C



Suppose we want to read the byte value in this address.

0xab345f78:

Tag (t)	1101	111000

(111000)2 = 56
57th bytes in data

Sets/Cachelines:

0
1
2
…
13

Y	t	Data (64 bytes)

…
15

16 sets, index 0 - 15
can be represented by 4 bits (0000 – 1111)
=> The size of index part

64 bytes of data, index 0 - 63
can be represented by 6 bits (000000 – 111111)
=> The size of byte offset part

2-D

How many cacheline-aligned memory addresses can be mapped to the same cache location as 0xab345f78 (assuming 32-bit address space)?

2^22

0xab345f78:

Tag	1101	Byte offset

Can be arbitrary value.

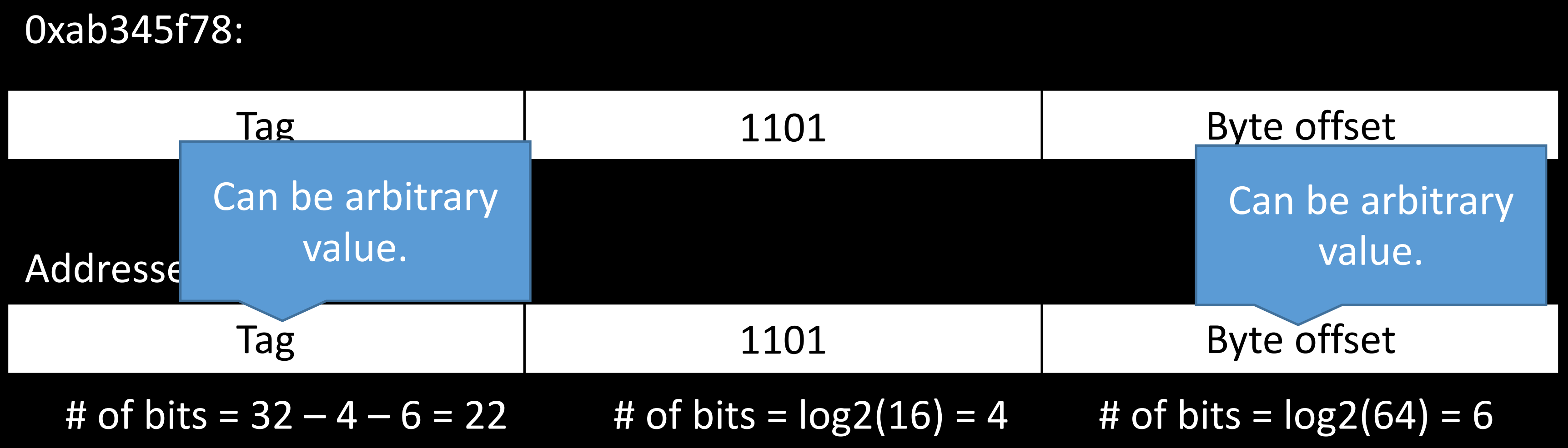
Addresse

Tag	1101	000000
# of bits = 32 – 4 – 6 = 22	# of bits = log2(16) = 4	# of bits = log2(64) = 6

2-D: exercise

How many ~~cacheline-aligned~~ memory addresses can be mapped to the same cache location as 0xab345f78 (assuming 32-bit address space)?

2^28



0xab345f78:

Tag	1101	Byte offset

Addresse

Tag	1101	Byte offset
# of bits = 32 – 4 – 6 = 22	# of bits = log2(16) = 4	# of bits = log2(64) = 6

Can be arbitrary value.

Can be arbitrary value.

2-E

If your answer in D. is bigger than 1, how can we determine which of the memory addresses is actually stored at a given cache location?

Compare the tag.

Tag	index	Byte offset

2-F

Suppose the cache is a 2-way associative cache, i.e. the cache is organized into sets each of which contains 2 cachelines. How many sets does the cache contain?

# of cachelines = 16 (Q2-A)
# of sets = # of cachelines / # of cachelines per set = 16/2 = 8

2-G

For the 2-way assocative cache, given a 32-bit address 0xab345f78, which set may contain the cached data for this address?

5

0xab345f78:

Tag	101	111000
# of bits = total number of bits - index – byte offset = 32 – 3 – 6 = 23	# of bits = log2(# of sets) = log2(8) = 3	# of bits = log2(# of bytes per cacheline) = log2(64) = 6

2-H

For the 2-way associactive cache, how many cacheline-aligned memory addresses can be mapped to the same set as 0xab345f78

2^23

0xab345f78:

Tag	101	Byte offset

Can be arbitrary value.

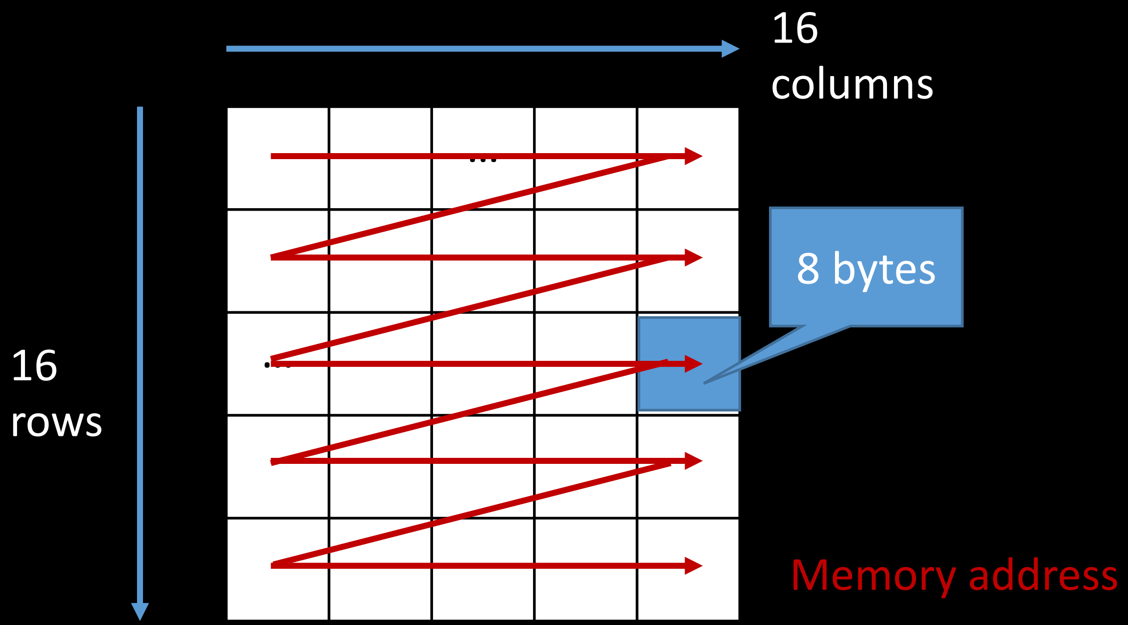
Addresse

Tag	101	000000
# of bits = 32 – 3 – 6 = 23	# of bits = log2(8) = 3	# of bits = log2(64) = 6

2-I (bonus)

Suppose ROWS=16, COLS=16. Out of ROWS*COLS total 8-byte memory accesses, how many of them result in cache miss (assuming the cache is direct-mapped and it's empty before the start of the loop)?

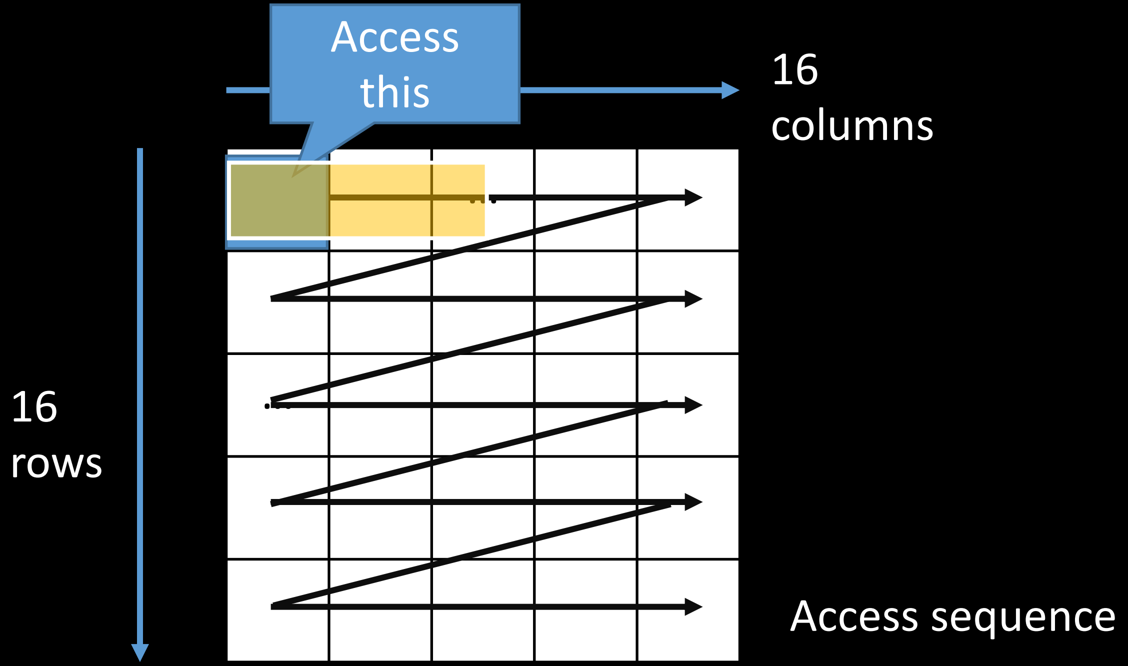
Note: Here we suppose the start address of this array is cacheline-aligned.



2-I (bonus)

Suppose ROWS=16, COLS=16. Out of ROWS*COLS total 8-byte memory accesses, how many of them result in cache miss (assuming the cache is direct-mapped and it's empty before the start of the loop)?

Note: Here we suppose the start address of this array is cacheline-aligned.

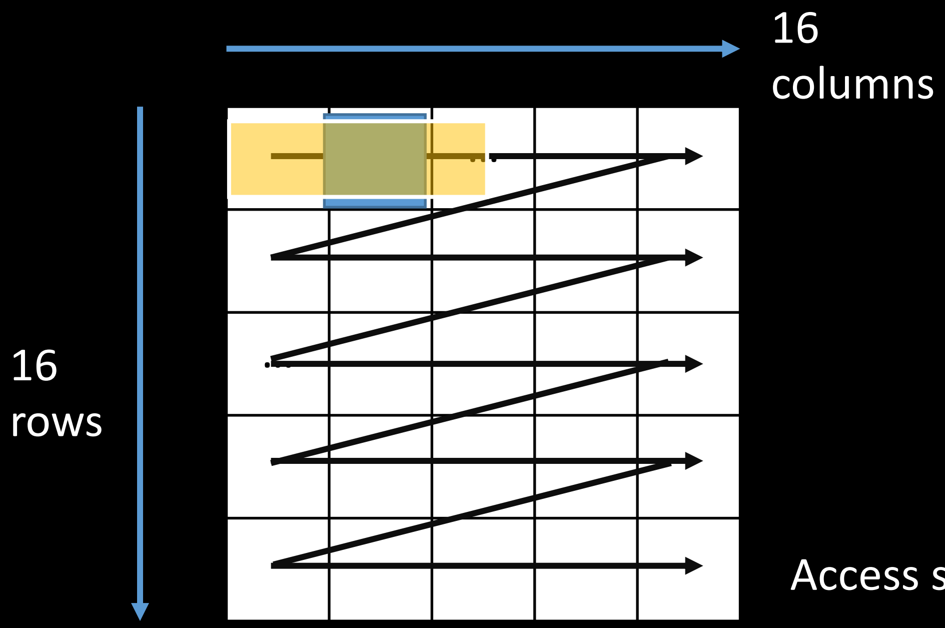


Cache miss!
Add into cache.

2-I (bonus)

Suppose ROWS=16, COLS=16. Out of ROWS*COLS total 8-byte memory accesses, how many of them result in cache miss (assuming the cache is direct-mapped and it's empty before the start of the loop)?

Note: Here we suppose the start address of this array is cacheline-aligned.



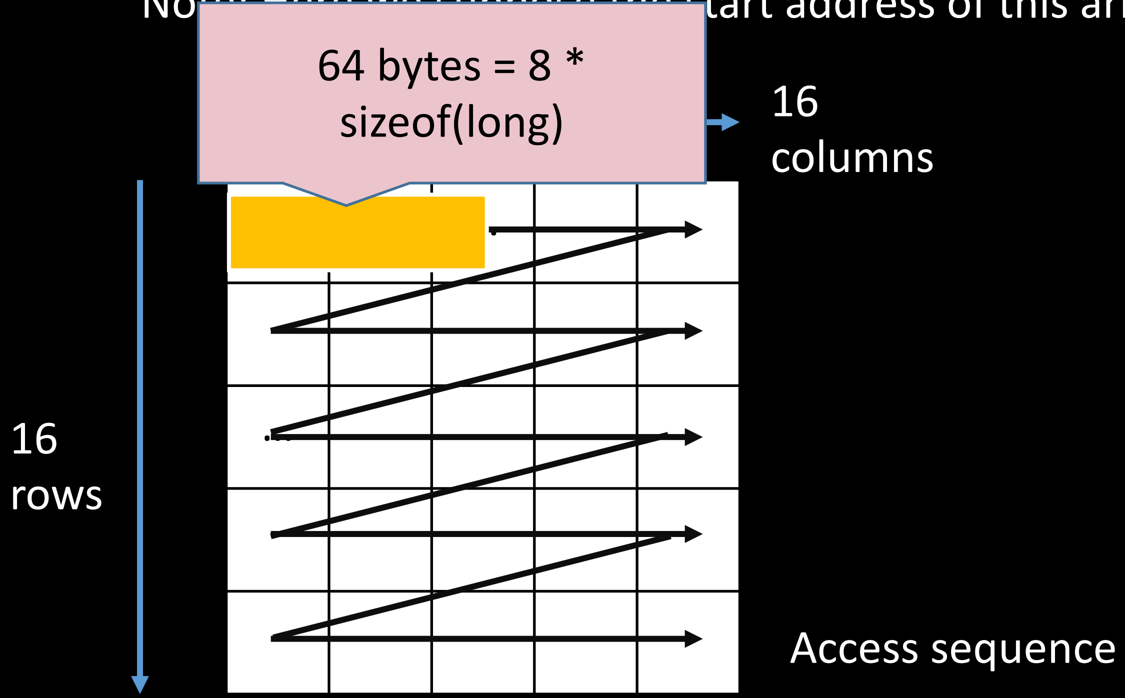
Cache hit!

Access sequence

2-I (bonus)

Suppose ROWS=16, COLS=16. Out of ROWS*COLS total 8-byte memory accesses, how many of them result in cache miss (assuming the cache is direct-mapped and it's empty before the start of the loop)?

Note: Here we suppose the start address of this array is cacheline-aligned.

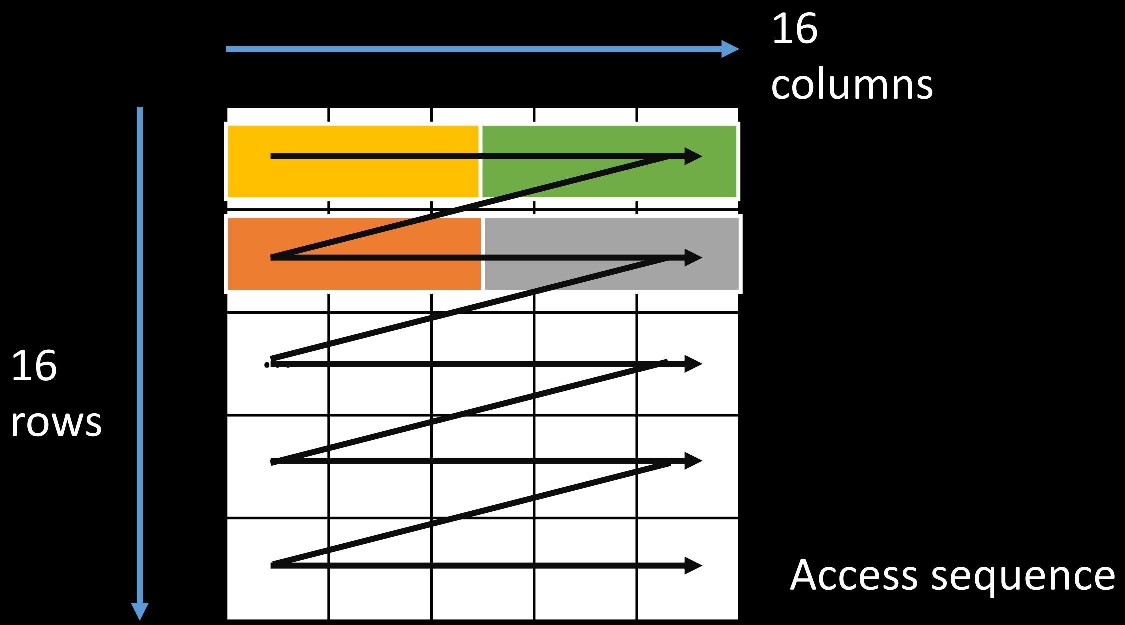


- First access: cache miss.
- Add 64 bytes into cacheline.
- Therefore the following 7 accesses are cache hit.

2-I (bonus)

Suppose ROWS=16, COLS=16. Out of ROWS*COLS total 8-byte memory accesses, how many of them result in cache miss (assuming the cache is direct-mapped and it’s empty before the start of the loop)?

Note: Here we suppose the start address of this array is cacheline-aligned.

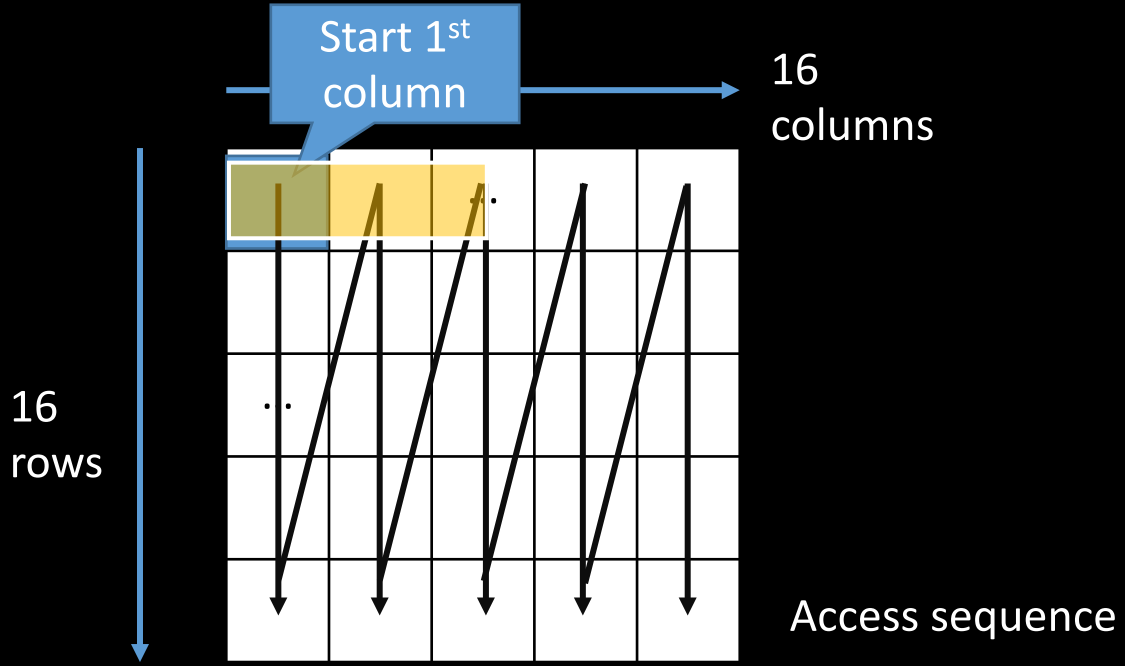


- Same to the following accesses.
- Cache miss rate = 1/8
- Cache miss times = 16 * 16 * 1/8 = 32

2-J (bonus)

Suppose ROWS=16, COLS=16. Out of ROWS*COLS total 8-byte memory accesses, how many of them result in cache miss (assuming the cache is direct-mapped and it's empty before the start of the loop)?

Note: Here we suppose the start address of this array is cacheline-aligned.

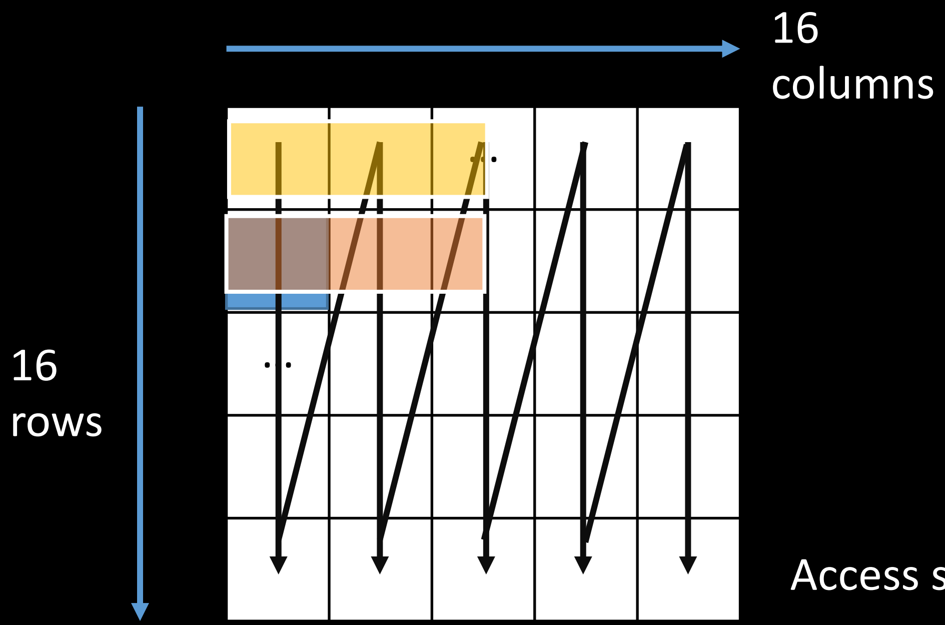


Cache miss!
Add into cache.

2-J (bonus)

Suppose ROWS=16, COLS=16. Out of ROWS*COLS total 8-byte memory accesses, how many of them result in cache miss (assuming the cache is direct-mapped and it’s empty before the start of the loop)?

Note: Here we suppose the start address of this array is cacheline-aligned.



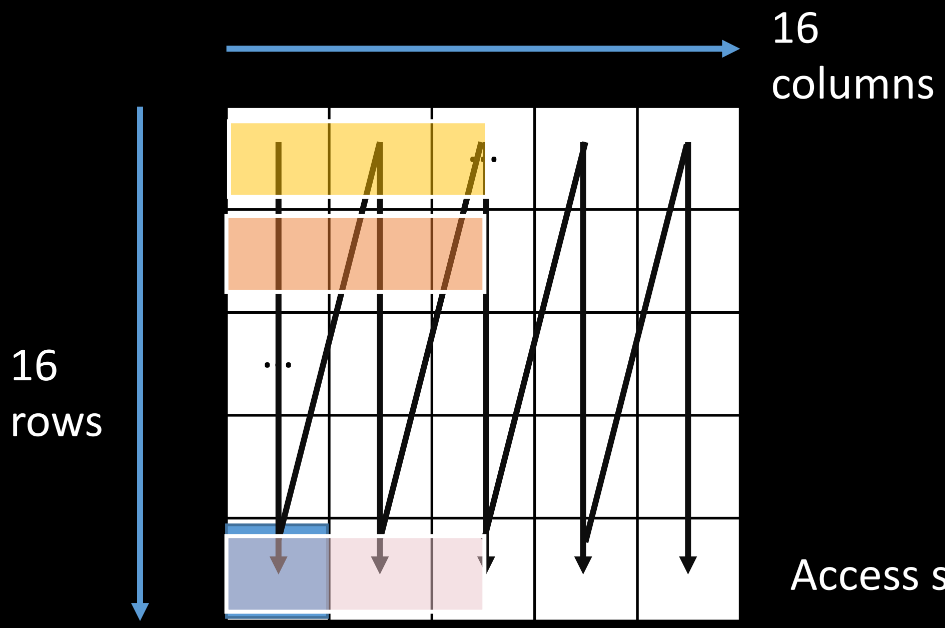
Cache miss!
Add into cache.

Access sequence

2-J (bonus)

Suppose ROWS=16, COLS=16. Out of ROWS*COLS total 8-byte memory accesses, how many of them result in cache miss (assuming the cache is direct-mapped and it's empty before the start of the loop)?

Note: Here we suppose the start address of this array is cacheline-aligned.



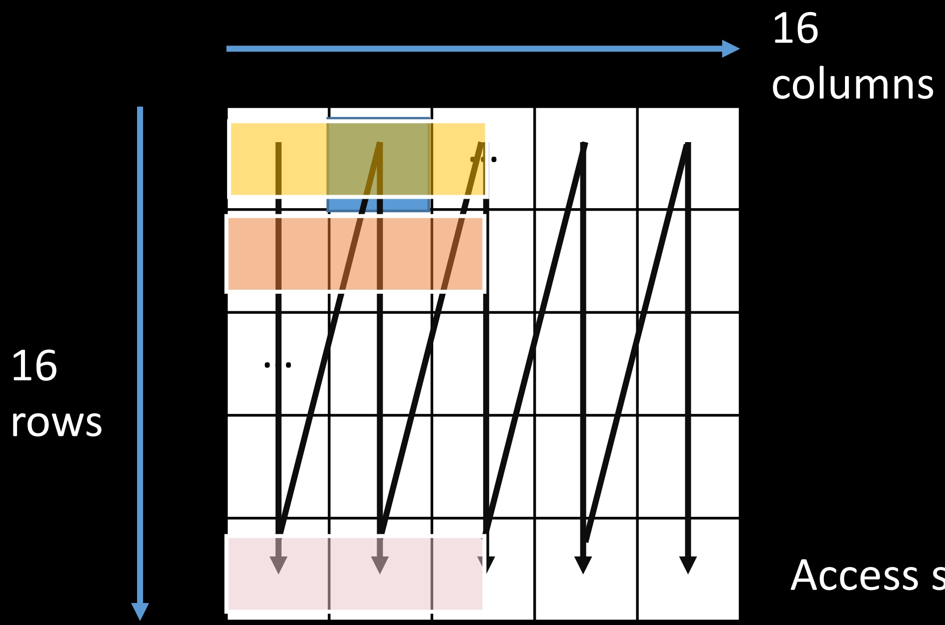
Cache miss!
Add into cache.

Access sequence

2-J (bonus)

Suppose ROWS=16, COLS=16. Out of ROWS*COLS total 8-byte memory accesses, how many of them result in cache miss (assuming the cache is direct-mapped and it's empty before the start of the loop)?

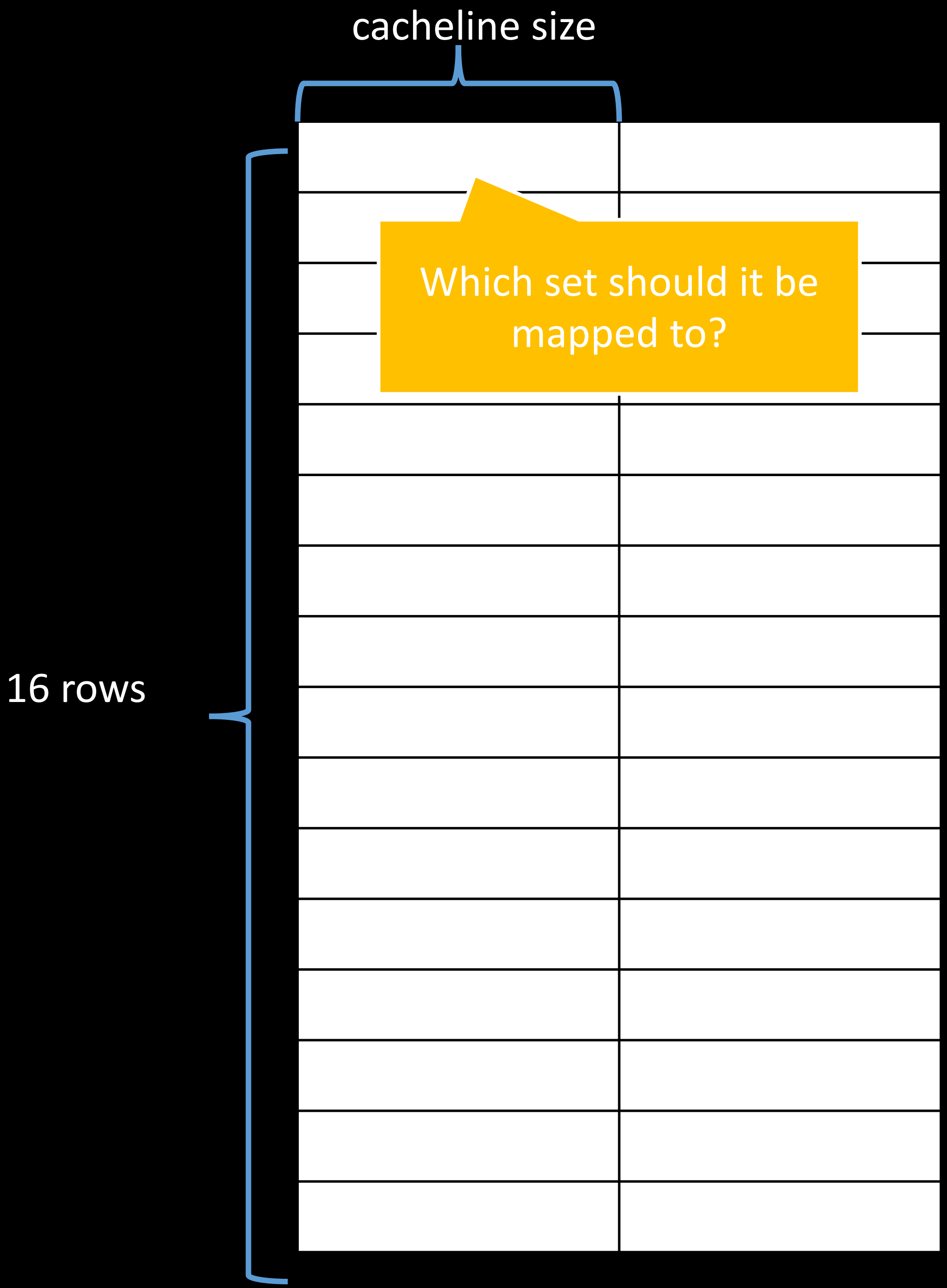
Note: Here we suppose the start address of this array is cacheline-aligned.



Cache hit!

- ***Is this correct?***
- ***No!***

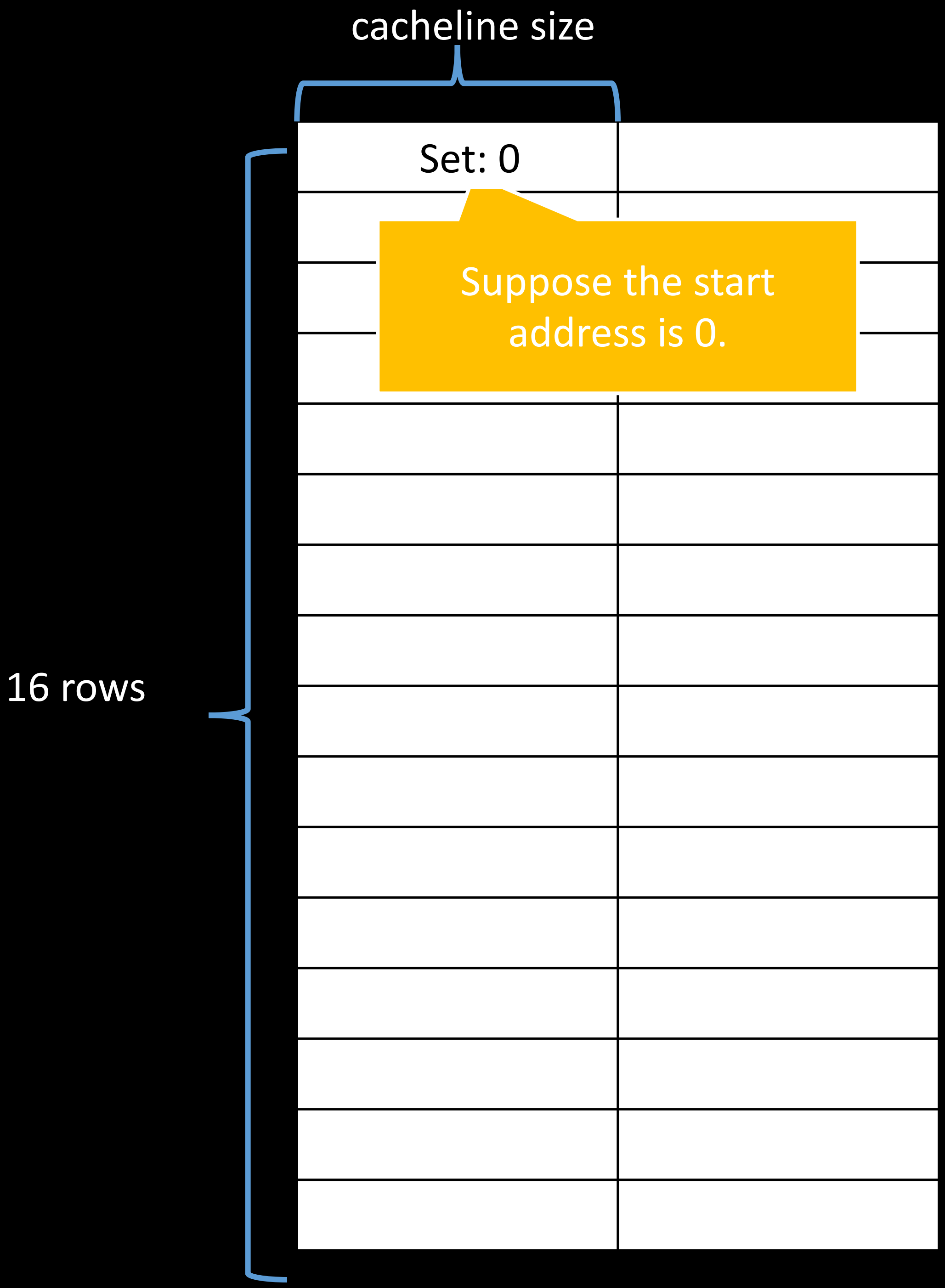
Access sequence



Which set should it be mapped to?

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size



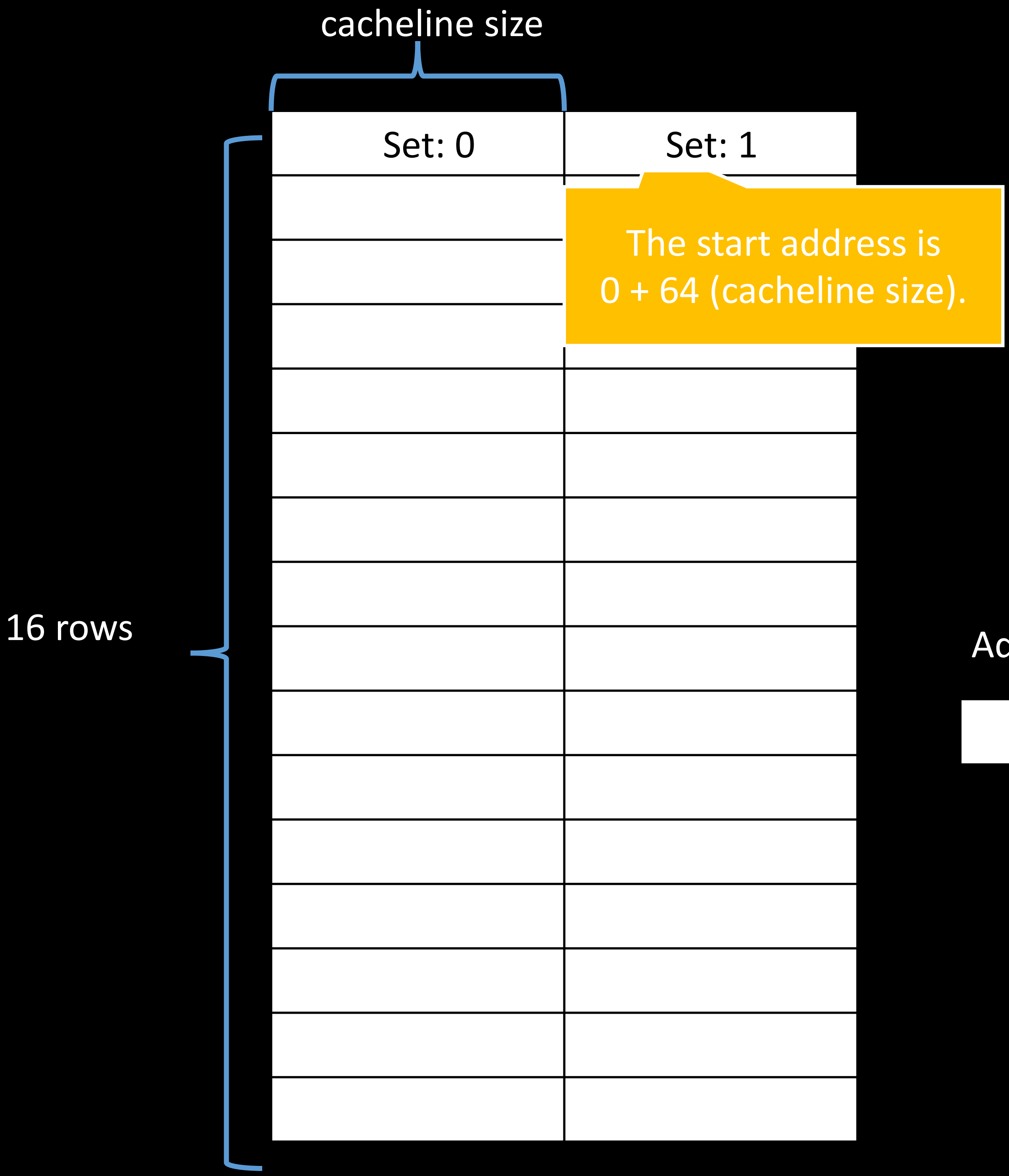
Suppose the start address is 0.

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size

Address: 0

Tag (22 bits)	0000	Byte offset (6 bits)



cacheline size

Set: 0

Set: 1

The start address is
0 + 64 (cacheline size).

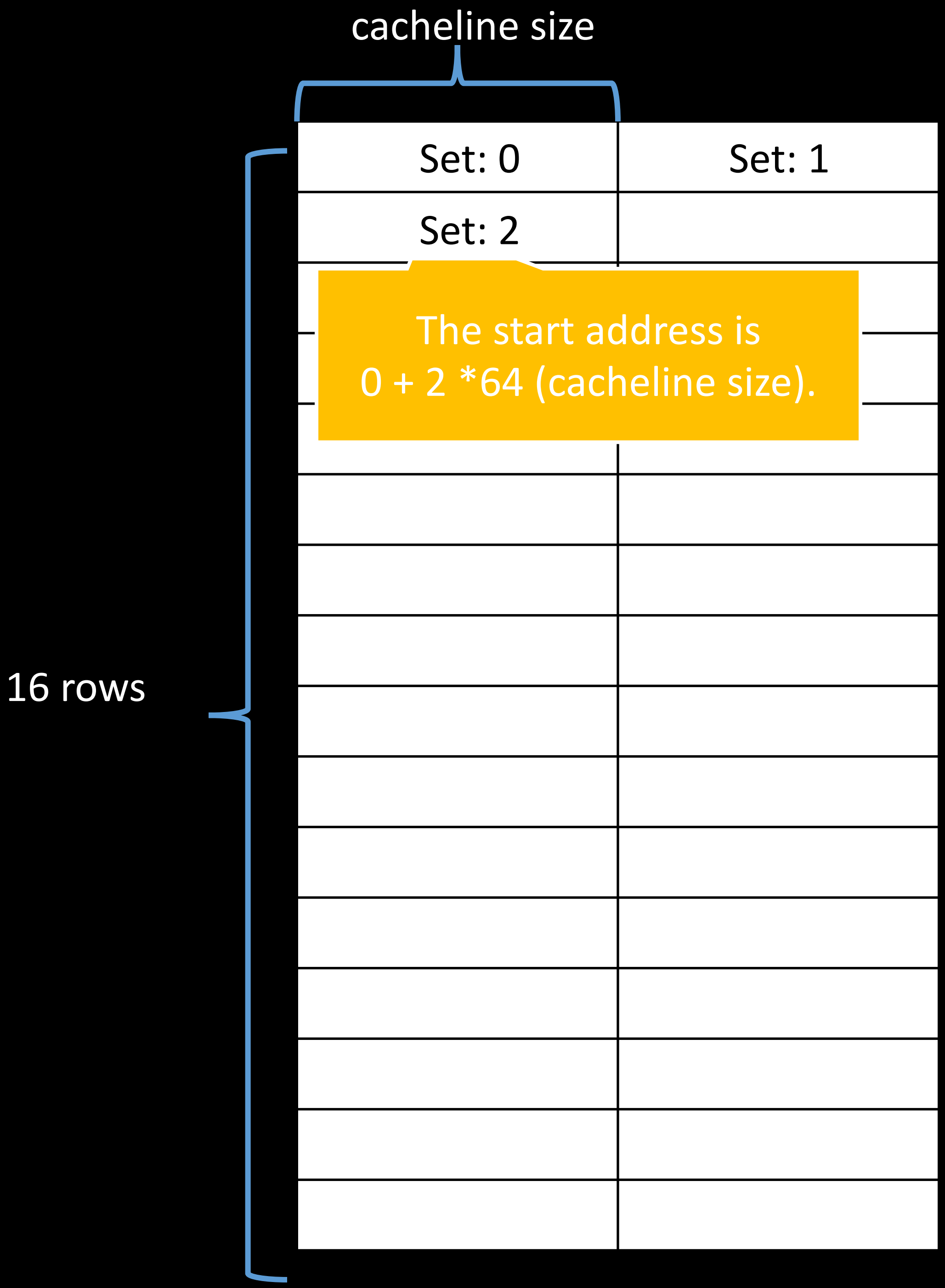
16 rows

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size

Address: 64 = 1 * 2^6

Tag (22 bits)	0001	Byte offset (6 bits)



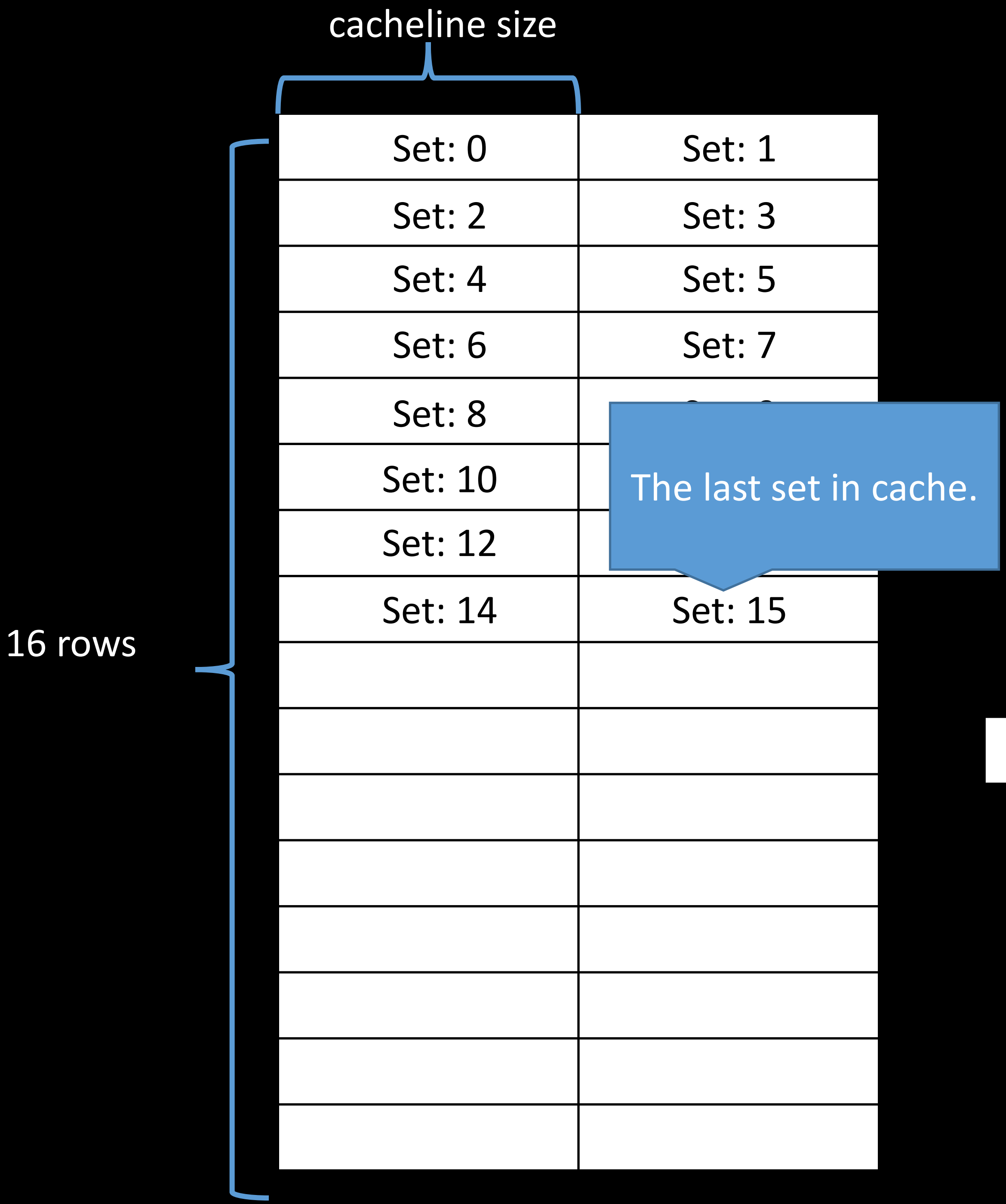
The start address is 0 + 2 *64 (cacheline size).

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size

Address: 64 = 2 * 2^6

Tag (22 bits)	0010	Byte offset (6 bits)

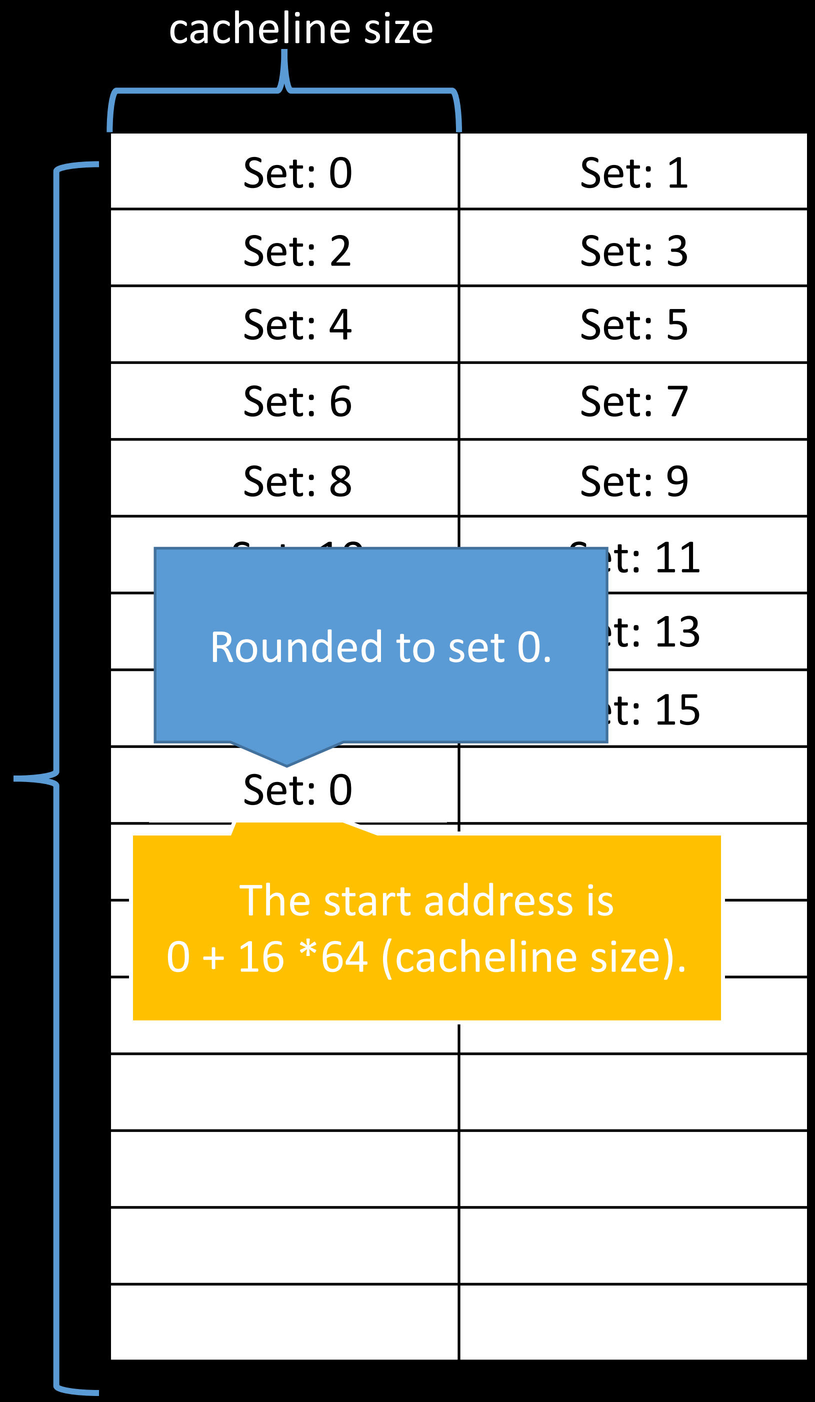


Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	
Set: 10	
Set: 12	
Set: 14	Set: 15

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size

Tag (22 bits)	Index (4 bits)	Byte offset (6 bits)



cacheline size

16 rows

Set: 0 Set: 1

Set: 2 Set: 3

Set: 4 Set: 5

Set: 6 Set: 7

Set: 8 Set: 9

Set: 11

Rounded to set 0.

t: 13

t: 15

Set: 0

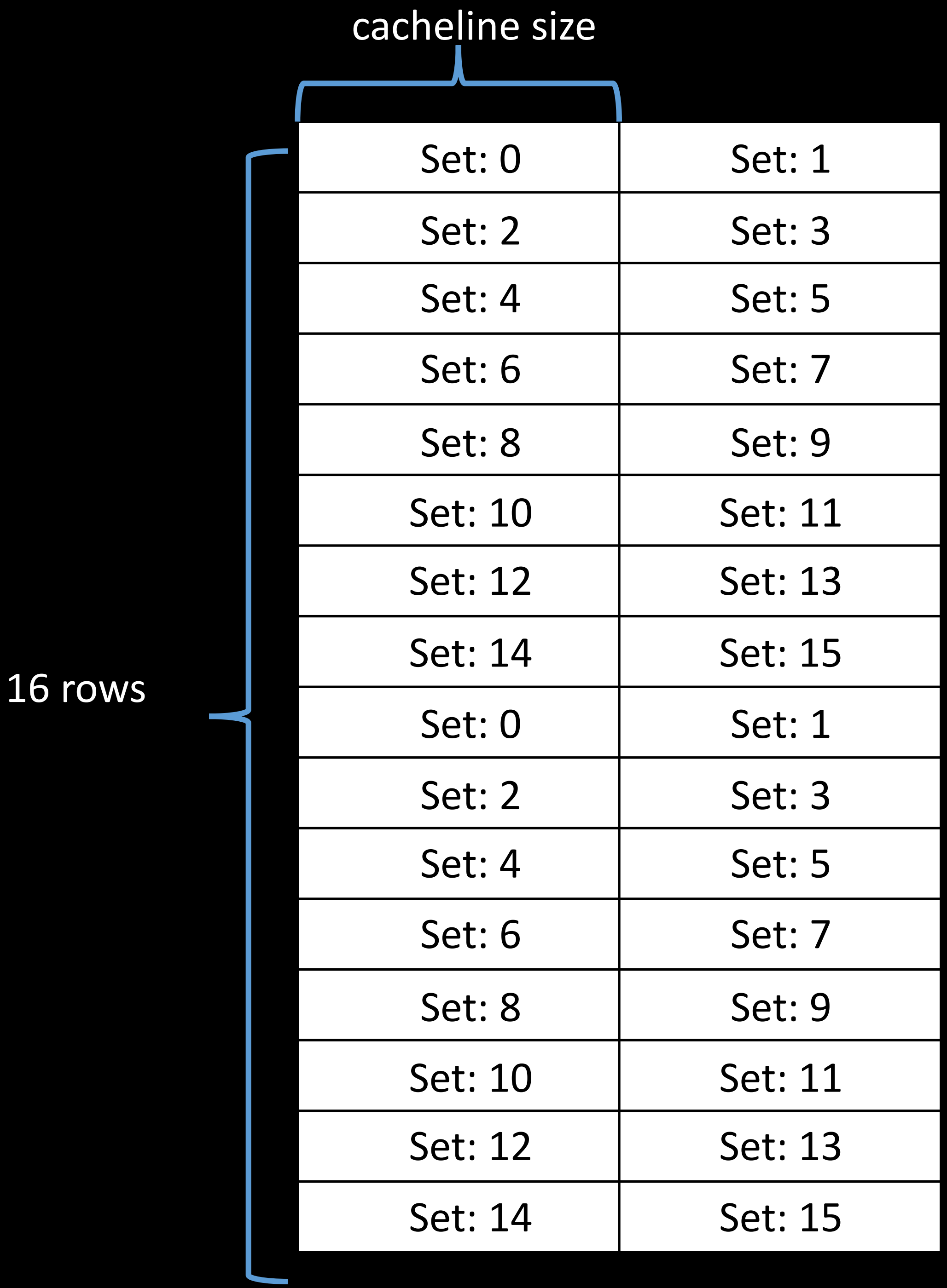
The start address is
0 + 16 *64 (cacheline size).

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size

Address: 64 = 16 * 2^6

Tag (22 bits)	0000	Byte offset (6 bits)

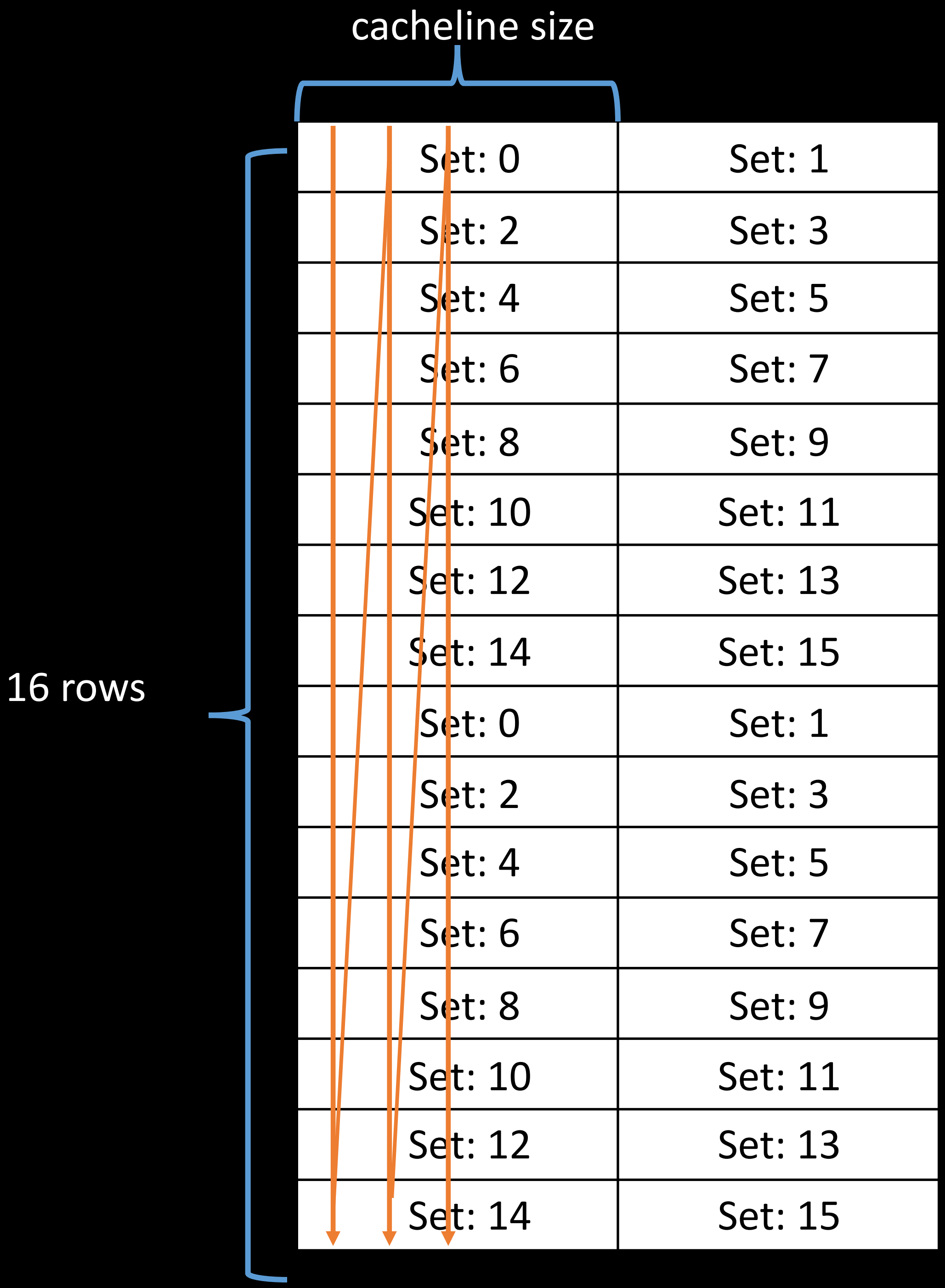


Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15
Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size

Tag (22 bits)	Index (4 bits)	Byte offset (6 bits)



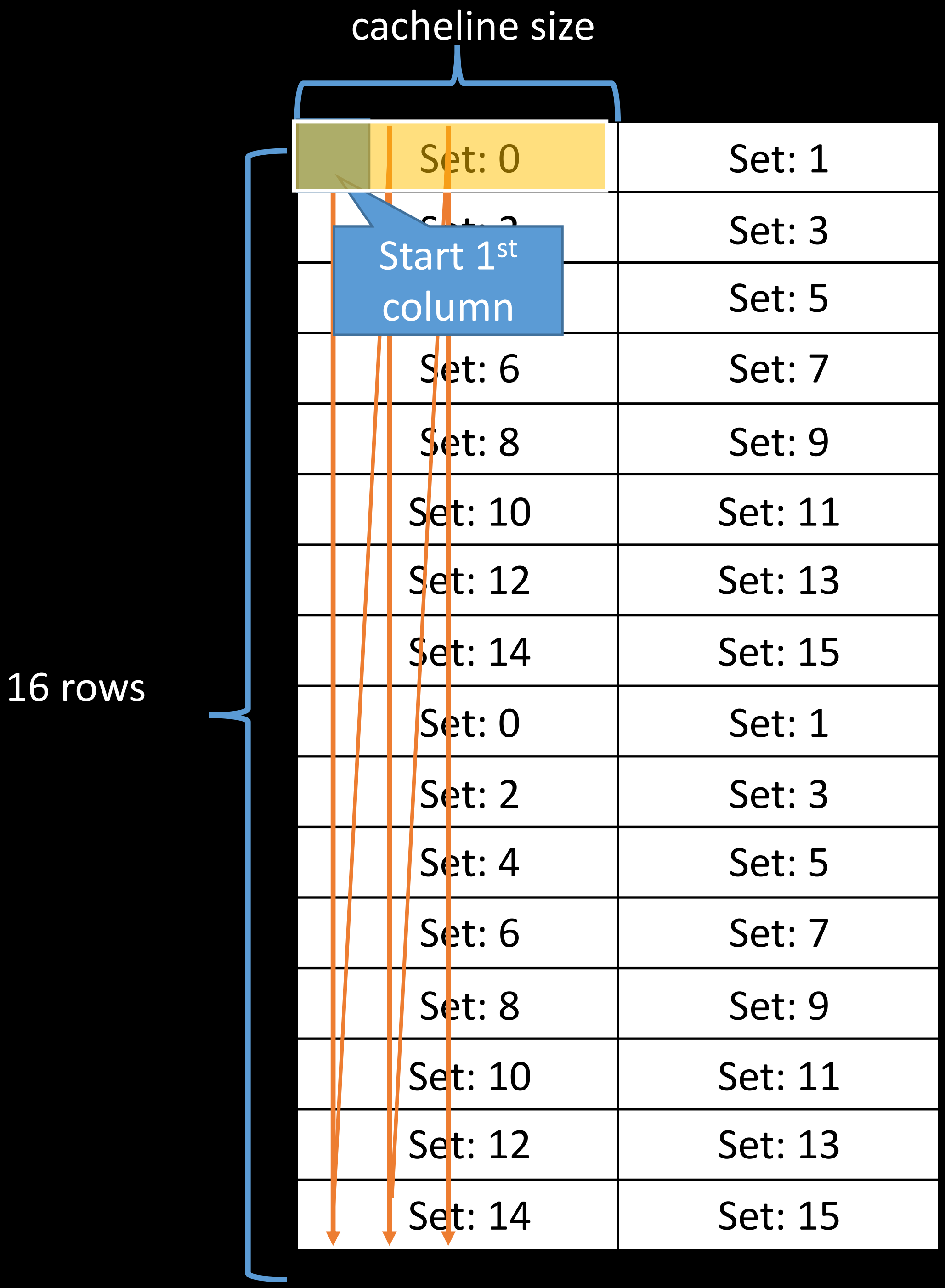
cacheline size

16 rows

Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15
Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size

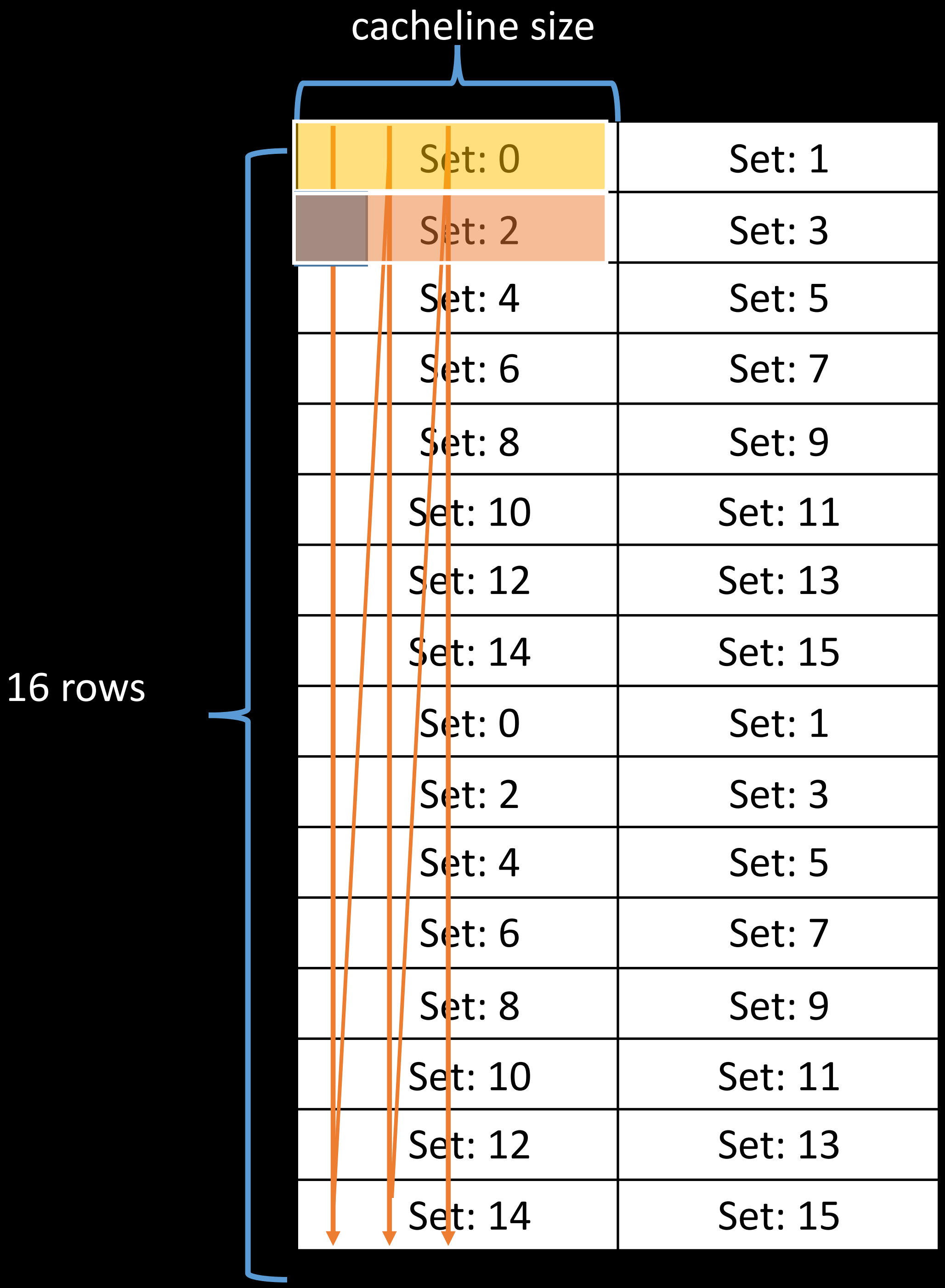


Set: 0	Set: 1
Set: 2	Set: 3
[illegible]	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15
Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size

Cache miss!
Add into cache set 0.

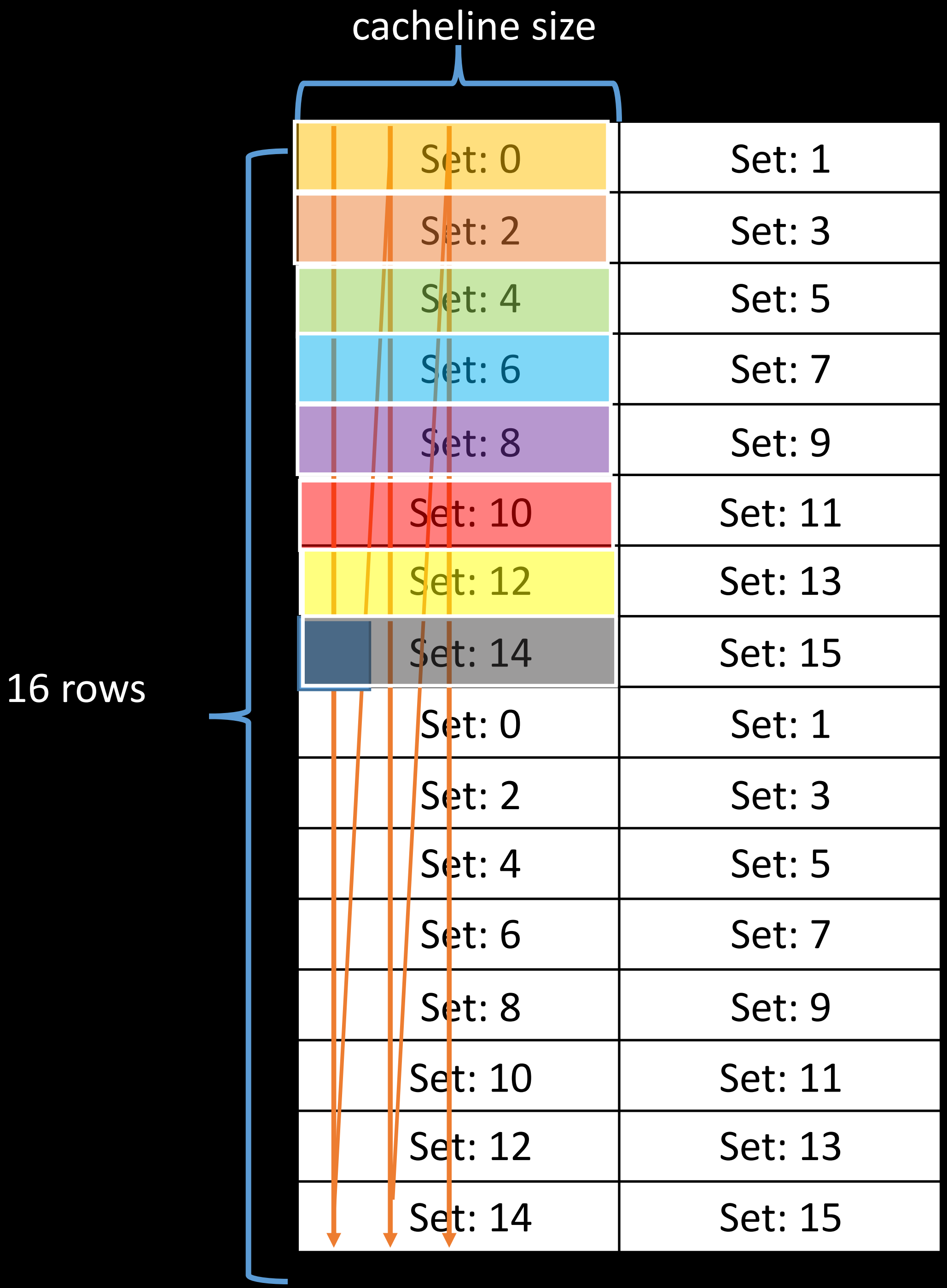


Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15
Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size

Cache miss!
Add into cache set 2.

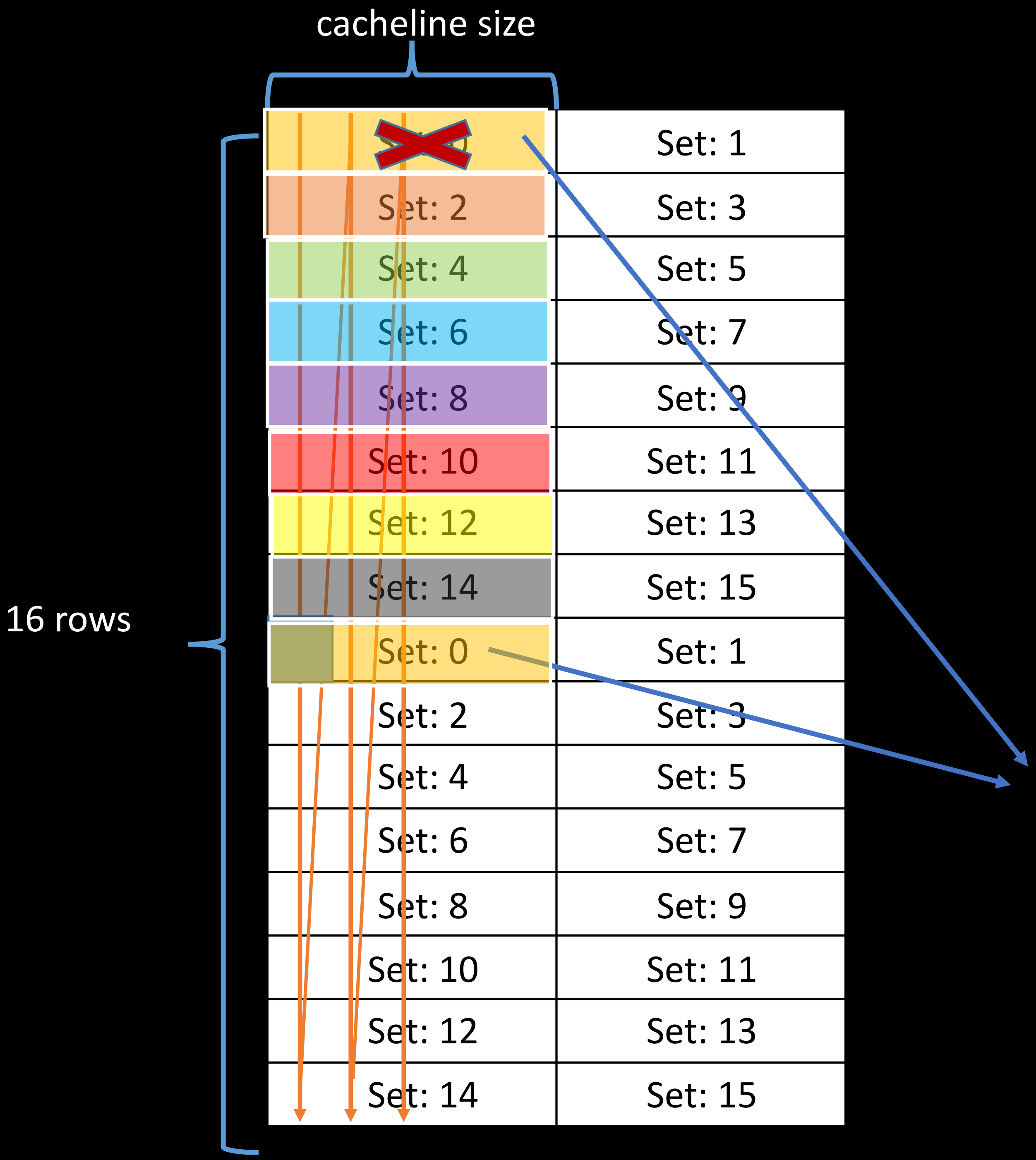


Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15
Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size

Cache miss!
Add into cache set 14.



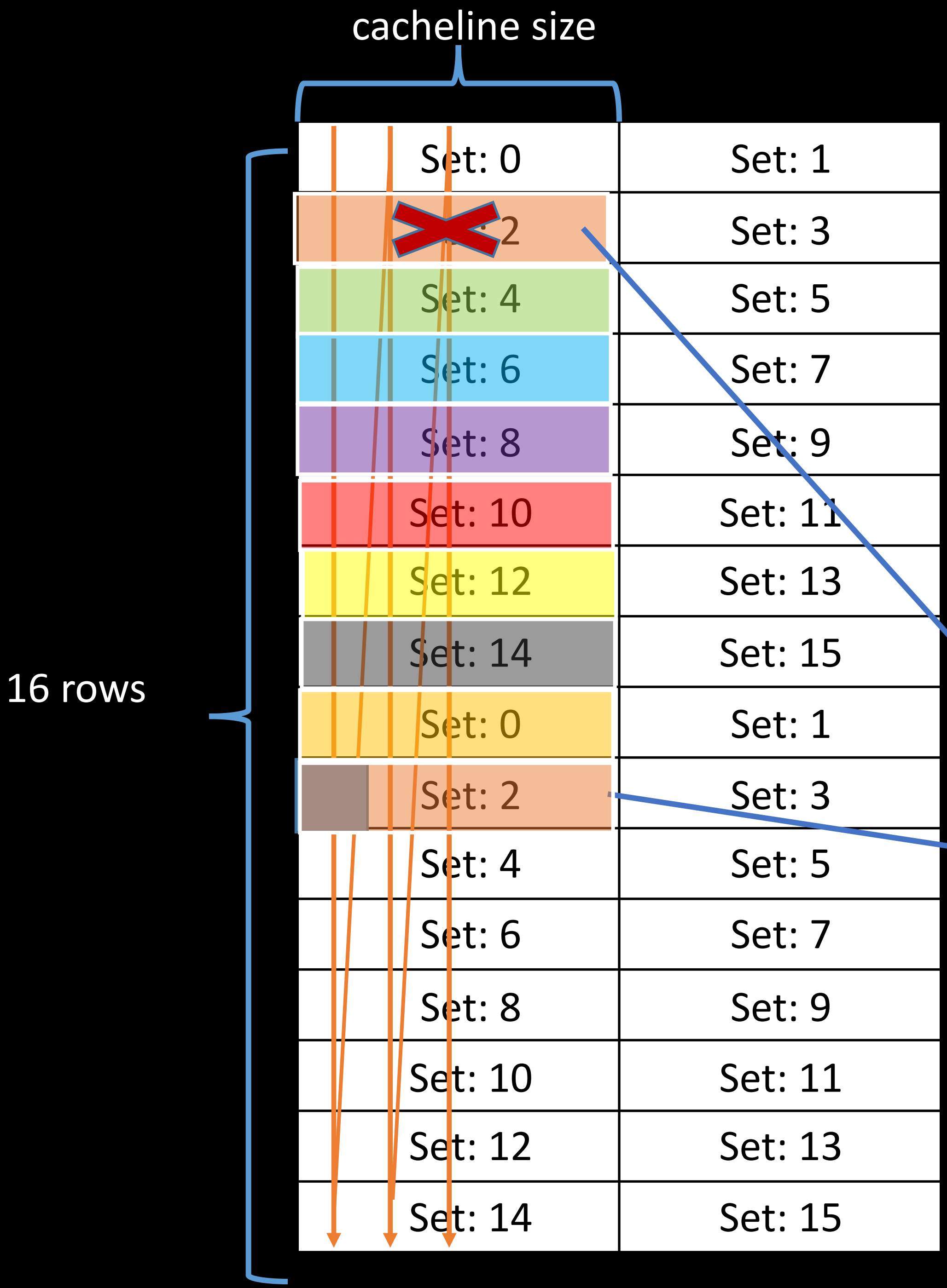
~~Set: 0~~	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15
Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size

Cache miss!
Add into cache set 0.

Mapped to the same set!
And only one cacheline per set!
=> replacement



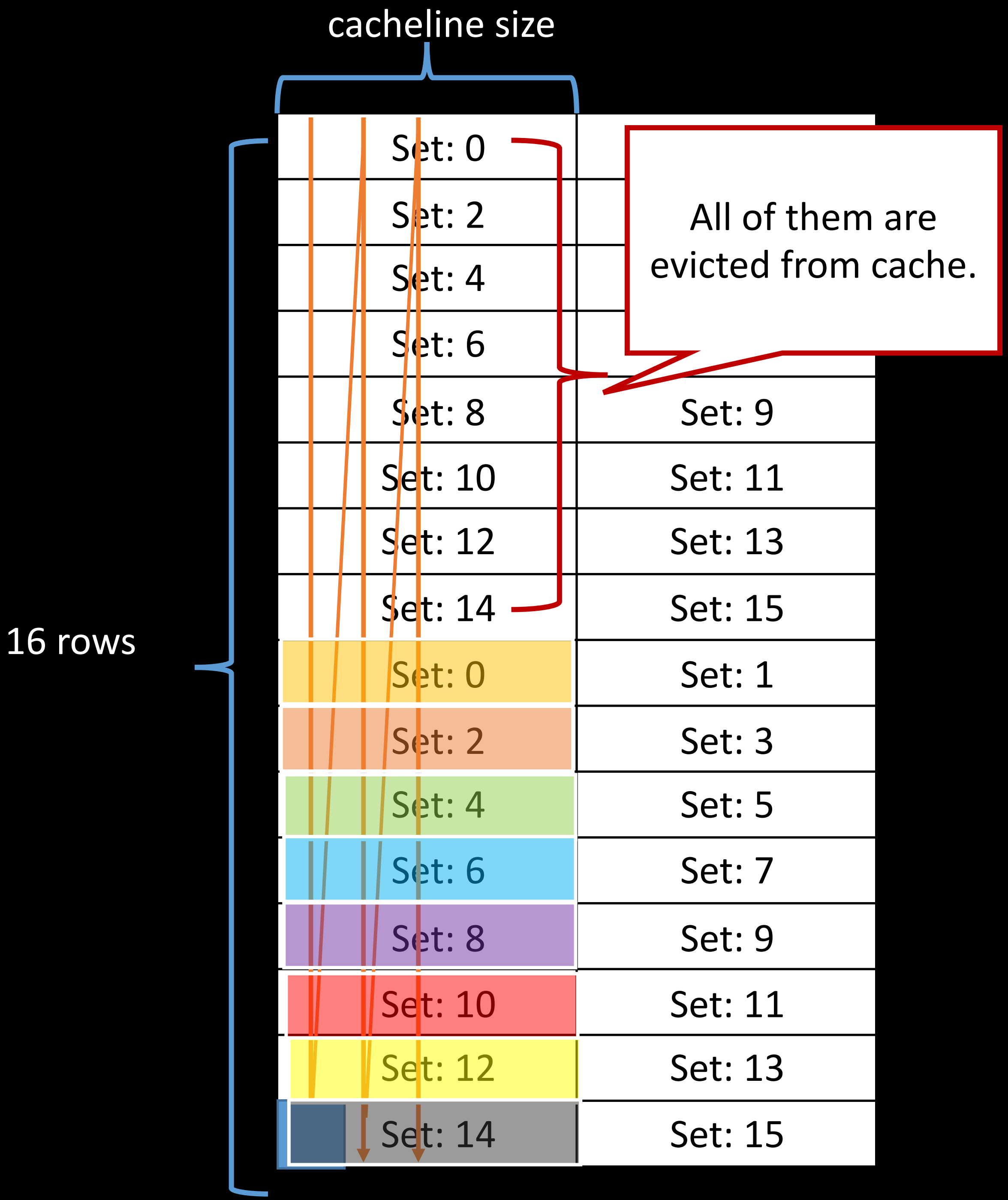
Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15
Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size

Cache miss!
Add into cache set 2.

Mapped to the same set!
And only one cacheline per set!
=> replacement



cacheline size

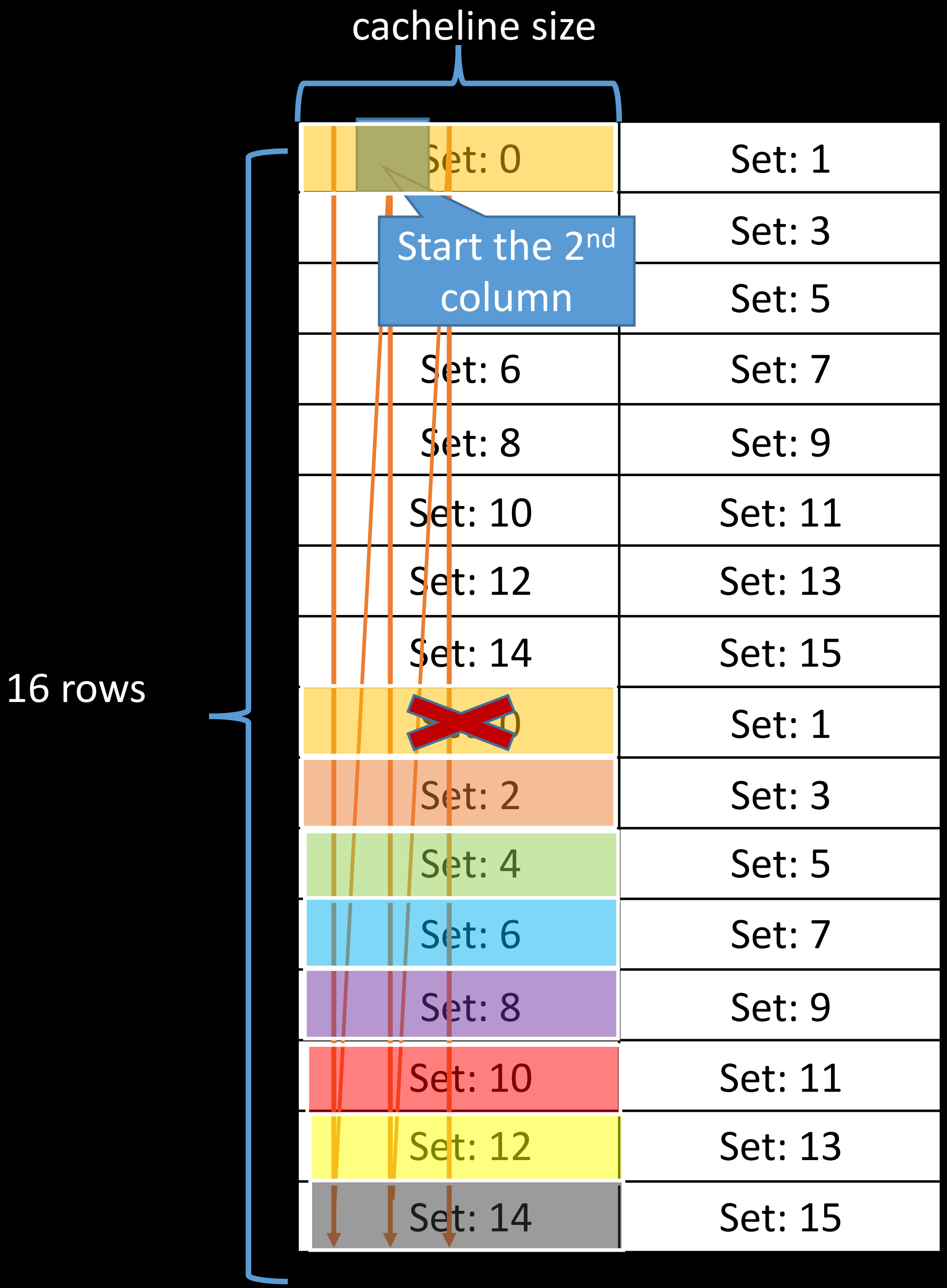
16 rows

All of them are evicted from cache.

Set: 0	
Set: 2	
Set: 4	
Set: 6	
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15
Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size



cacheline size

16 rows

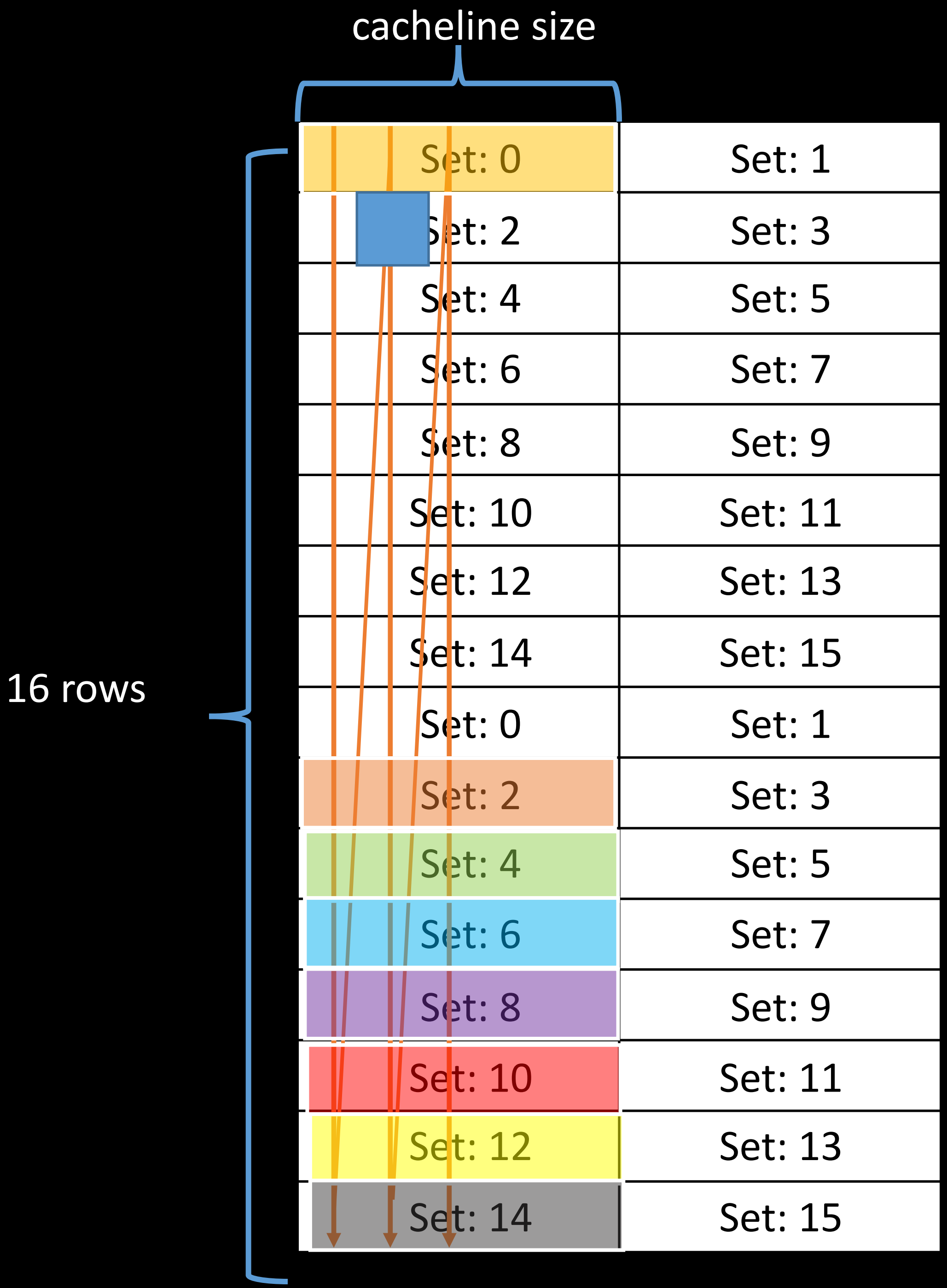
Start the 2nd column

Set: 0	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15
~~Set: 0~~	Set: 1
Set: 2	Set: 3
Set: 4	Set: 5
Set: 6	Set: 7
Set: 8	Set: 9
Set: 10	Set: 11
Set: 12	Set: 13
Set: 14	Set: 15

- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size

Cache miss!
Add into cache set 0.
Set conflict => replacement



- ***Direct mapped: one cacheline per set***
- ***# number of sets/cachelines in the cache: 16 (Q2-B)***
- ***Index: 0000 – 1111 (0 – 15)***

- Each row = 16 * sizeof(long) = 2 cacheline size

Same for all the following accesses.

- Next time you try to access the same slot,
- It has already been evicted from the cache
- Cache miss!

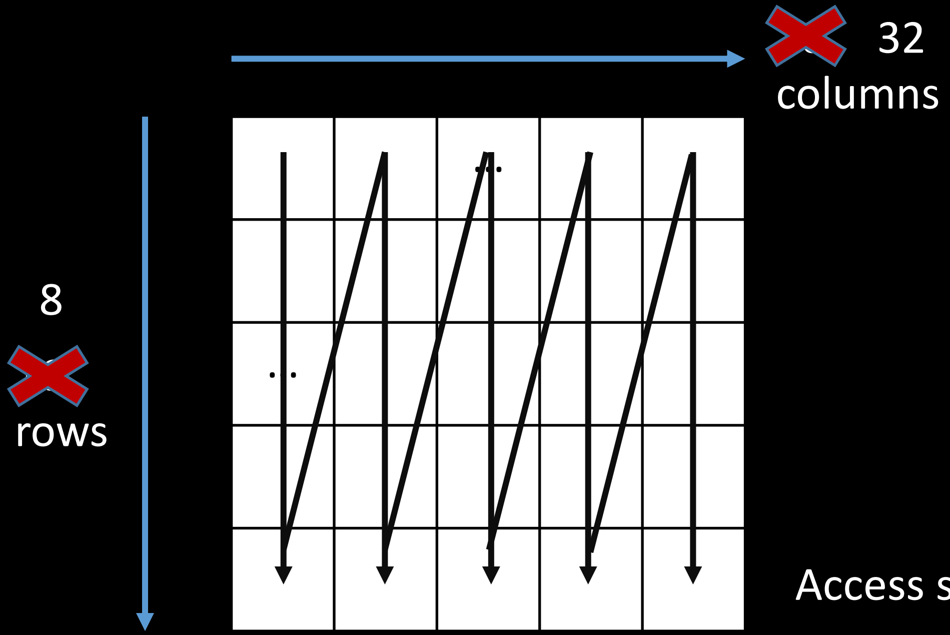
Cache miss rate: 100%

Cache miss times: 16*16 = 256

2-J (bonus) exercise1

Suppose ROWS=8, COLS=32. Out of ROWS*COLS total 8-byte memory accesses, how many of them result in cache miss (assuming the cache is direct-mapped and it’s empty before the start of the loop)?

Note: Here we suppose the start address of this array is cacheline-aligned.

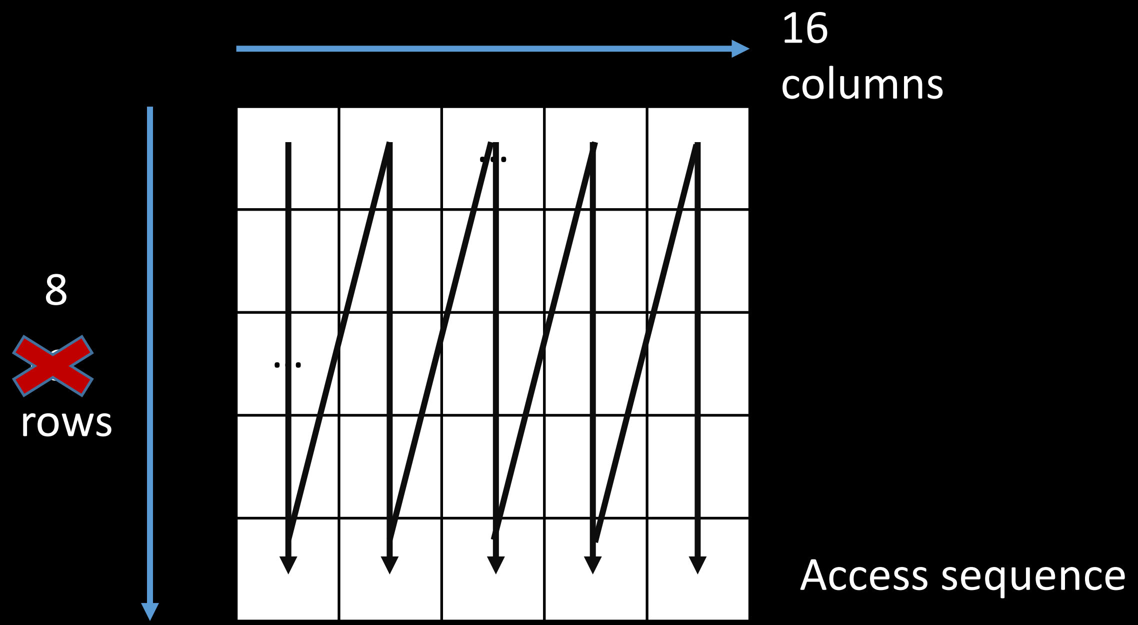


Access sequence

2-J (bonus) exercise2

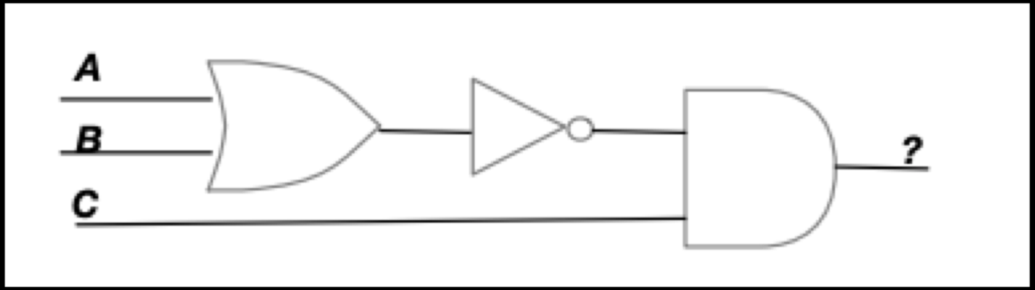
Suppose ROWS=8, COLS=16. Out of ROWS*COLS total 8-byte memory accesses, how many of them result in cache miss (assuming the cache is direct-mapped and it's empty before the start of the loop)?

Note: Here we suppose the start address of this array is cacheline-aligned.



3 Logic design (25 points)

3-A



What is the boolean expression that corresponds to what the following logic circuit outputs? (Please write the expression with the most direct correspondence to this circuit without simplification.)

- (bar(A+B))·C

3-B

Please simplify the boolean expression $(A+B)\cdot(A+B)$ according to the boolean logic laws in Appendix 6. Write down the detailed steps where each step applies only one law and name that law

- Covered in Q2 of recitation 13

Q2 Simplify boolean expression

- Simplify boolean expression (A+B) ·($\bar{A}$ + $\bar{B}$).
- You may write `*` for ·, and write `barA` for $\bar{A}$ (or `barB` or $\bar{B}$)
- (A+B)*(barA+barB)
- =(A+B)*barA + (A+B)*barB Distribution law
- =barA*A + barA*B + barB*A + barB*B Distribution law
- =0+barA*B+barB*A+0 Inverse law
- =barA*B+barB*A Basic law

3-C

- If we are to use a single logic gate to implement the simplified expression in B. Which gate to use?

1. AND
2. OR
3. NOR
4. NAND
5. [✓ XOR]
6. None of the above

3-Question C. and D.

- Question C. and D. ask you to implement a combinatorial circuit to compute a 3-input parity function. A parity function takes multiple 1-bit input signals and computes a 1-bit parity bit as the output. The parity bit is the sum of all input signals modulo 2. For example, if the 3 input signals are 110, the parity output is (1+1+0)%2 = 0. If the 3 input signals are 111, the parity output is (1+1+1)%2 = 1.

3-C

- Please write the truth table for the 3-input parity function.

A	B	C	Output
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

3-D

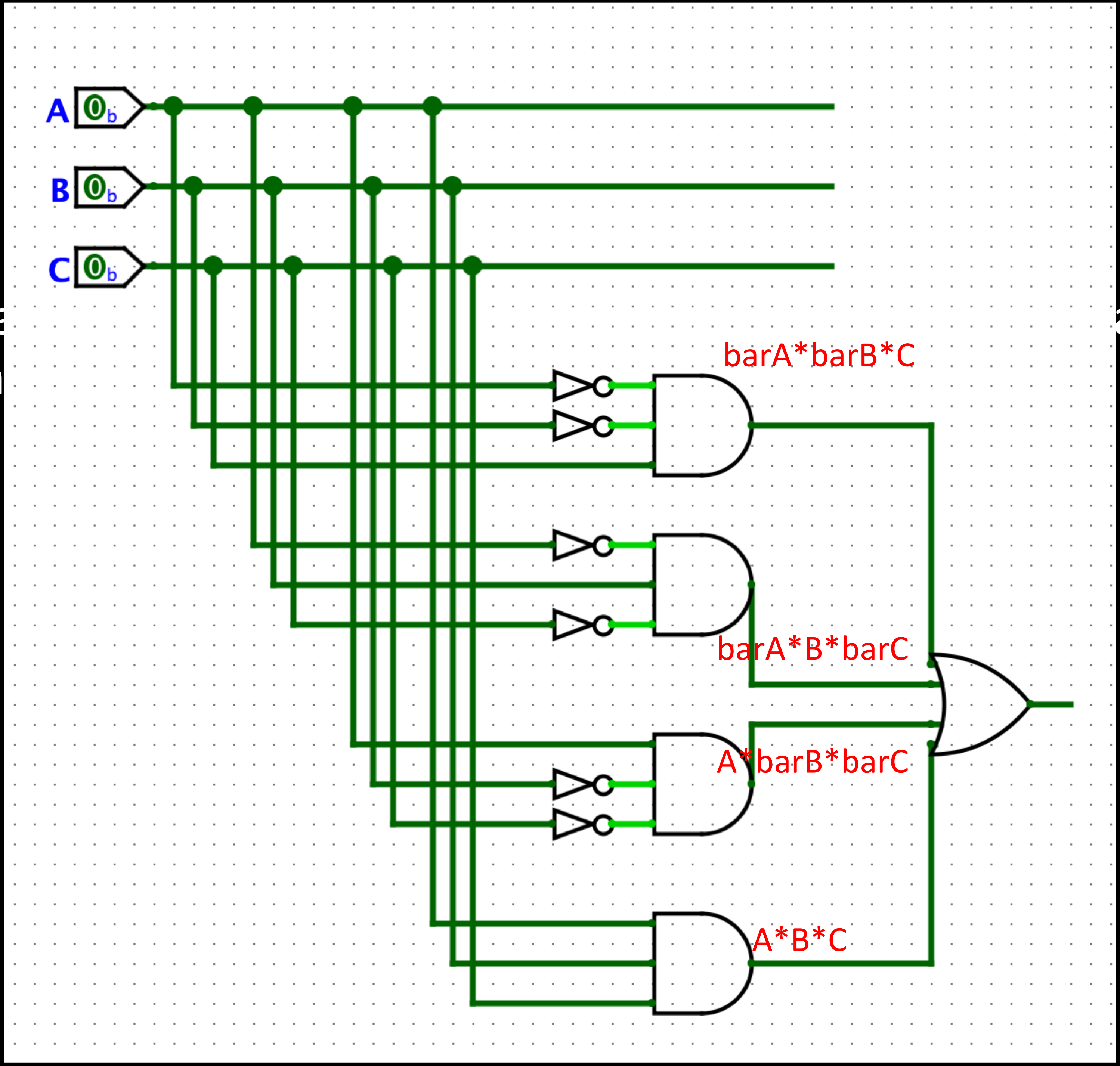
- Please draw the combinatorial circuit that implements the above parity function.

Follow the rule of "sum of products"

A	B	C	Output	Products
0	0	0	0	
0	0	1	1	barA*barB*C
0	1	0	1	barA*B*barC
0	1	1	0	
1	0	0	1	A*barB*barC
1	0	1	0	
1	1	0	0	
1	1	1	1	A*B*C

3-D

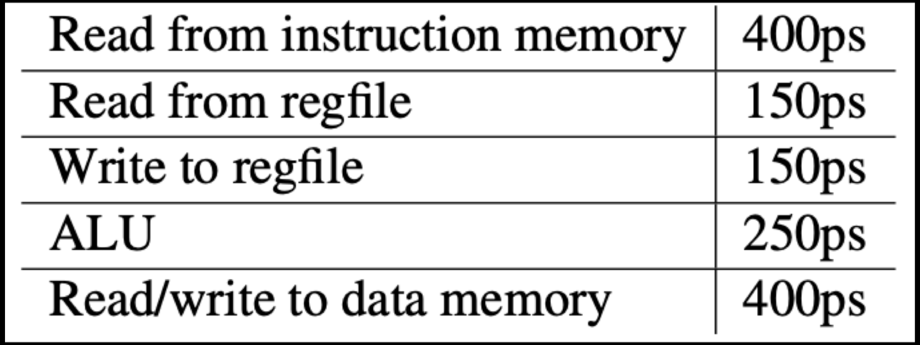
- Please dra above parity fun



4. CPU design

- Figure 1 shows a basic single-cycle CPU design for RISC-V. The design handles 3 types of instructions: arithmetic instructions involving two source registers and one destination register, memory load instructions and store instructions. Note that the design does not implement conditional branches. Suppose the major logic blocks in Figure 1 have the following latencies (If a logic block’s latency is not specified, it is small enough to be ignored.)

4. CPU design



Read from instruction memory	400ps
Read from regfile	150ps
Write to regfile	150ps
ALU	250ps
Read/write to data memory	400ps

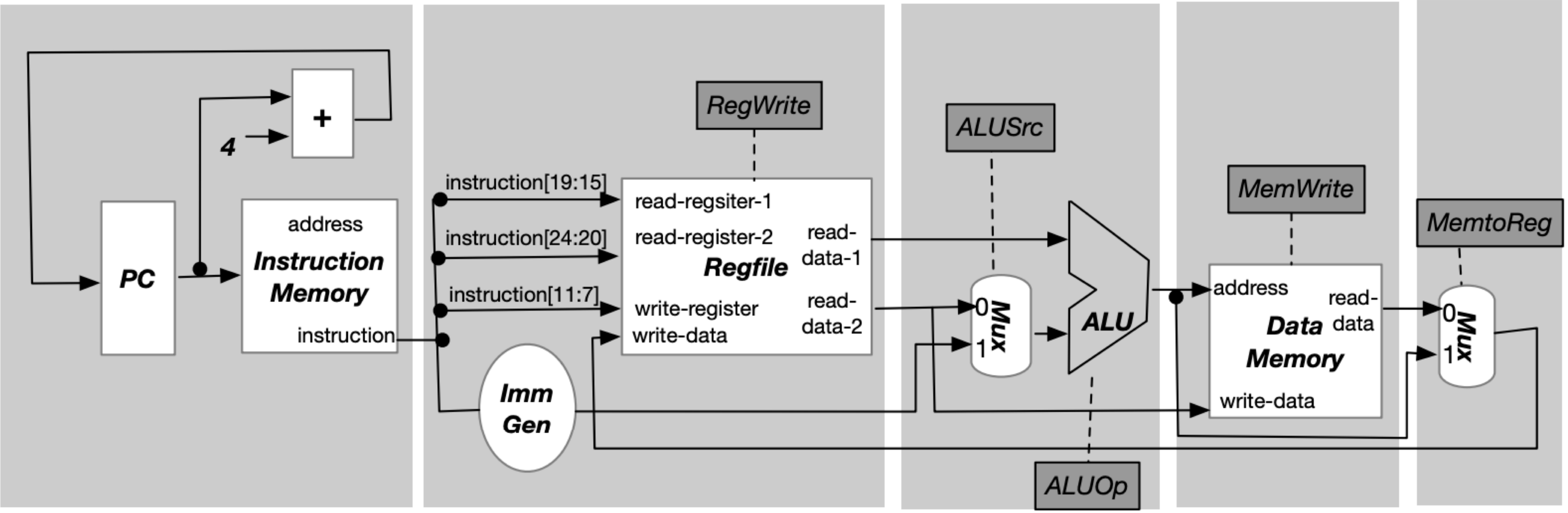
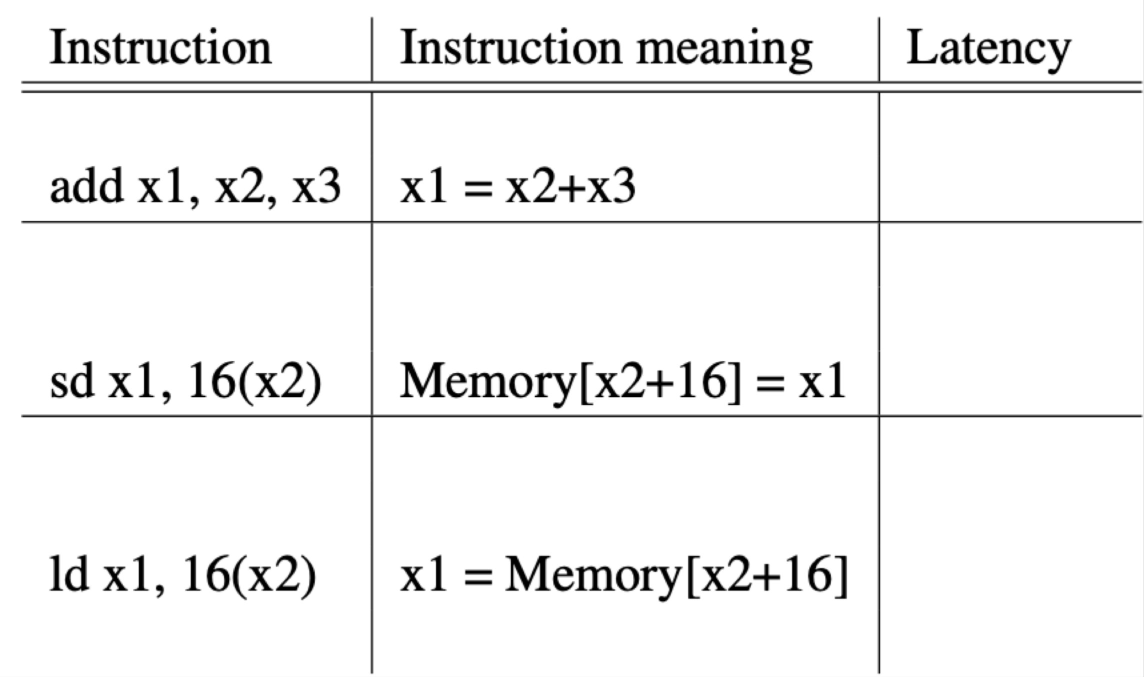


Figure 1: Basic RISC-V CPU design

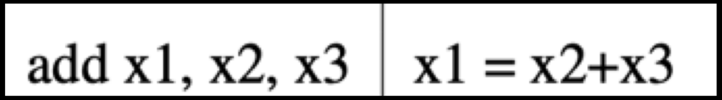
4-A

- What is the minimum latency to execute each of the following instructions in the single-cycle design:

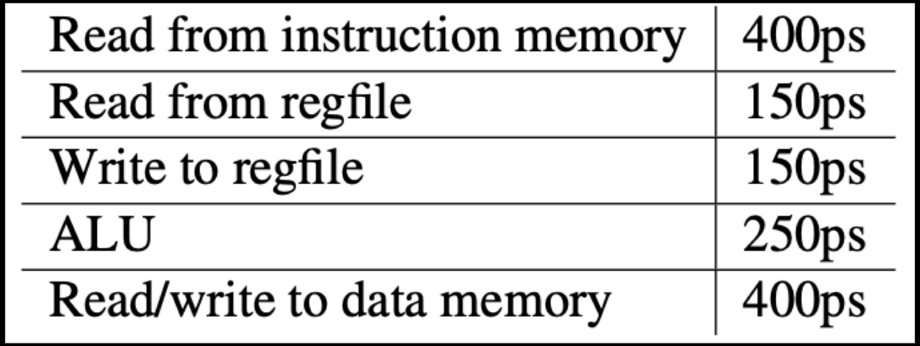


Instruction	Instruction meaning	Latency
add x1, x2, x3	x1 = x2+x3	
sd x1, 16(x2)	Memory[x2+16] = x1	
ld x1, 16(x2)	x1 = Memory[x2+16]	

4-A



add x1, x2, x3	x1 = x2+x3



Read from instruction memory	400ps
Read from regfile	150ps
Write to regfile	150ps
ALU	250ps
Read/write to data memory	400ps

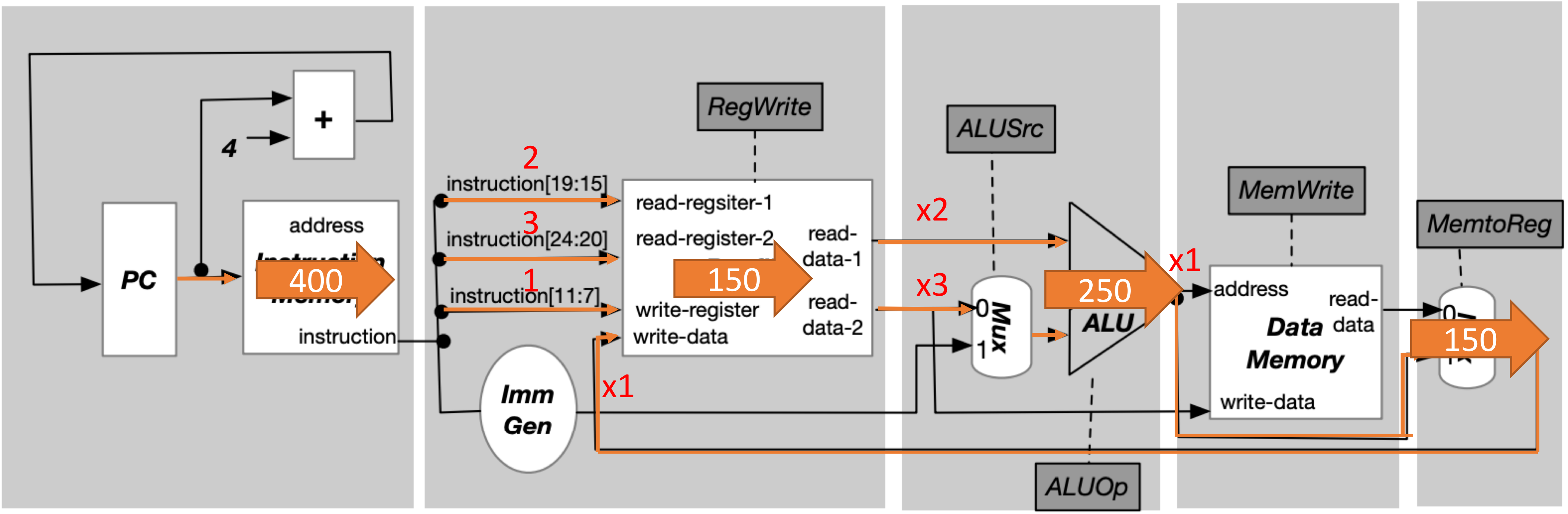


Figure 1: Basic RISC-V CPU design

400+150+250+150=950

4-A

sd x1, 16(x2)	Memory[x2+16] = x1

Read from instruction memory	400ps
Read from regfile	150ps
Write to regfile	150ps
ALU	250ps
Read/write to data memory	400ps

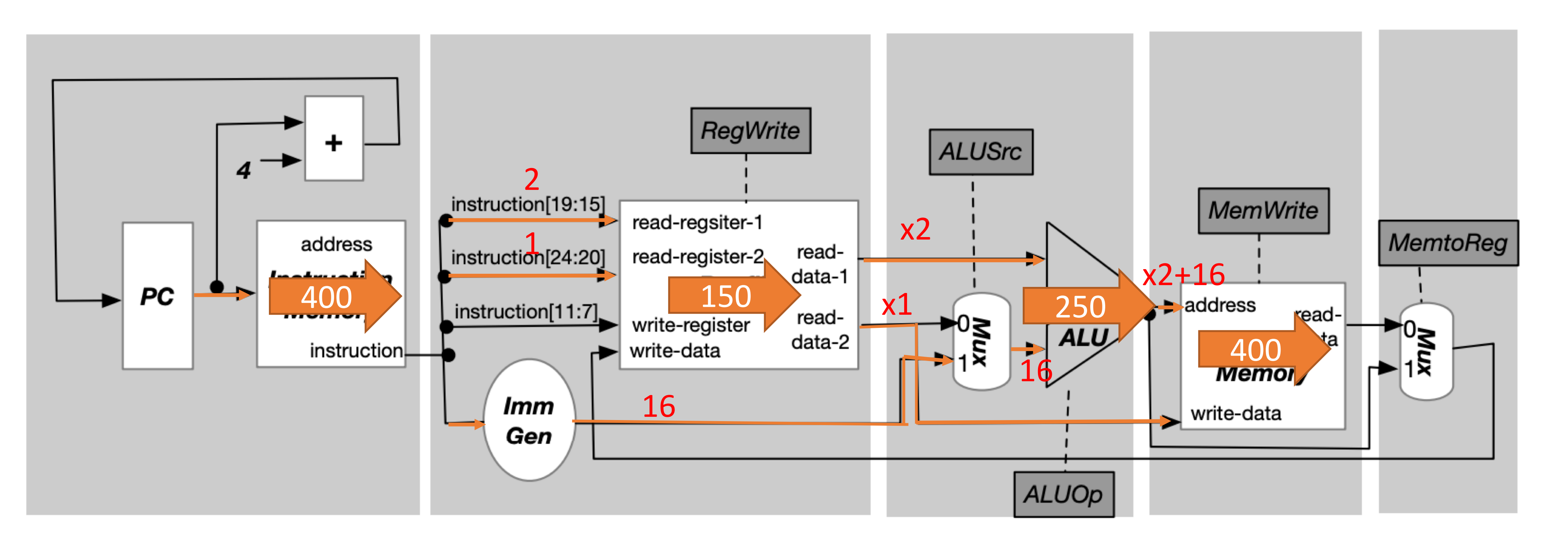


Figure 1: Basic RISC-V CPU design

400+150+250+400=1200

4-A

ld x1, 16(x2)	x1 = Memory[x2+16]

Read from instruction memory	400ps
Read from regfile	150ps
Write to regfile	150ps
ALU	250ps
Read/write to data memory	400ps

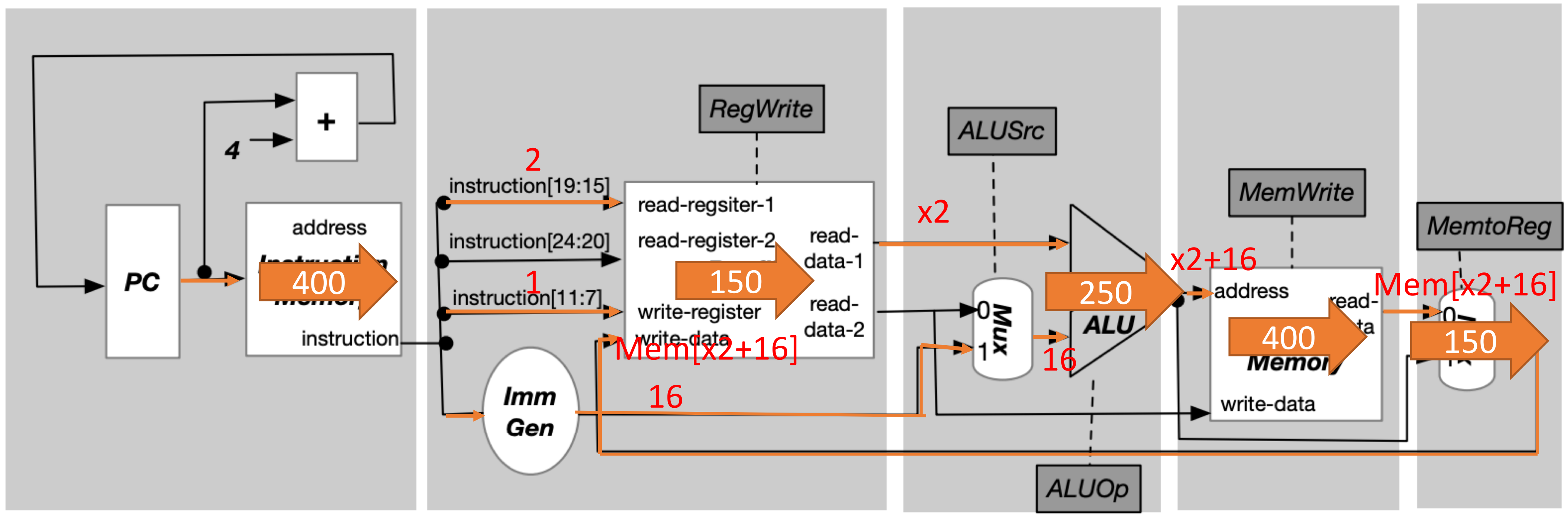


Figure 1: Basic RISC-V CPU design

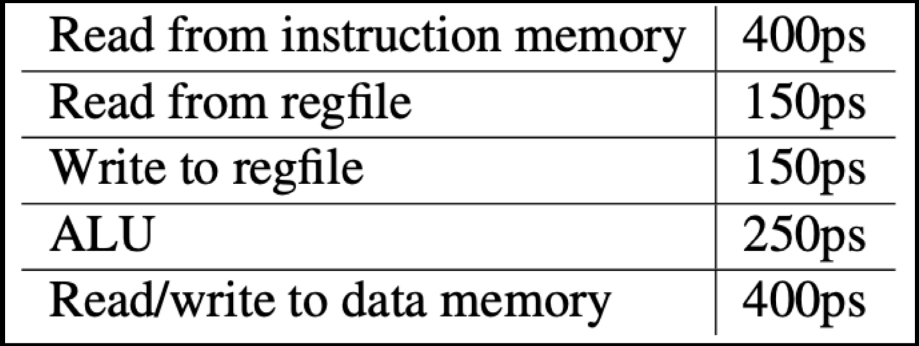
400+150+250+400+150=1350

4-B

What is fastest clock cycle time that can be used for the single-cycle CPU design?

- The time sufficient for all instructions to finish
- = time of slowest instruction
- 1350

4-C



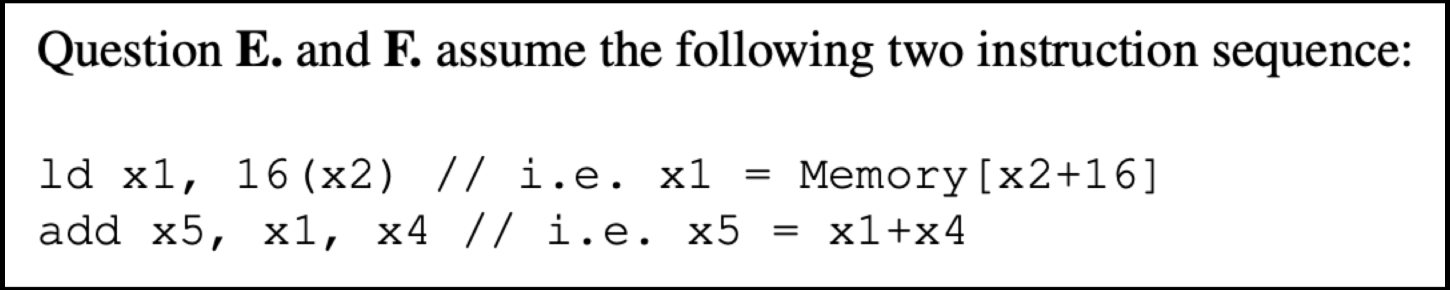
Read from instruction memory	400ps
Read from regfile	150ps
Write to regfile	150ps
ALU	250ps
Read/write to data memory	400ps

Suppose we change the CPU in Figure 1 into 5-stage pipelined design: IF(instruction fetch), ID(instruction decode and register read), EX(execute operation or calculate address), MEM(memory access), WB(write back to regfile). The five stages of the pipeline are marked as the large grey rectangles in Figure 1. What is the fastest clock cycle time that can be used for the 5-stage design?

- 400ps

pipeline clock cycle=max_stage_time

4-E

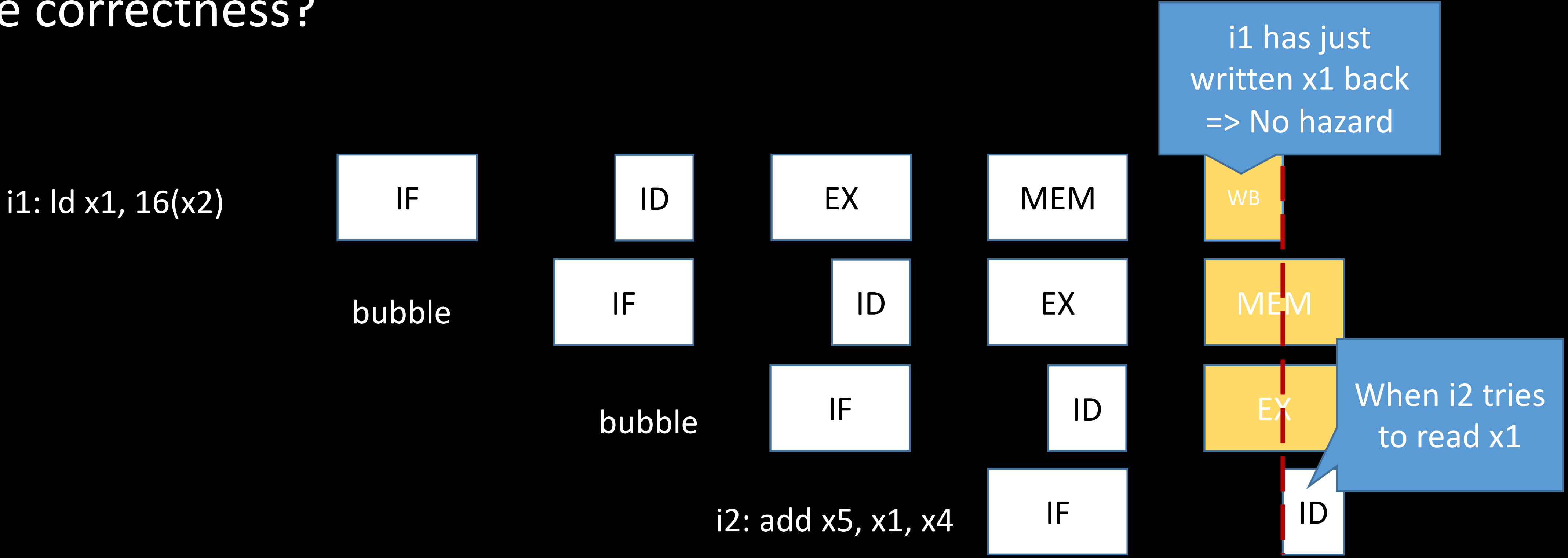


Question **E.** and **F.** assume the following two instruction sequence:

```
ld x1, 16(x2)  // i.e. x1 = Memory[x2+16]
add x5, x1, x4 // i.e. x5 = x1+x4
```

Suppose there is no forwarding between pipeline stages, how many bubbles (aka nop) must be inserted between the two instructions to ensure correctness?

- 2

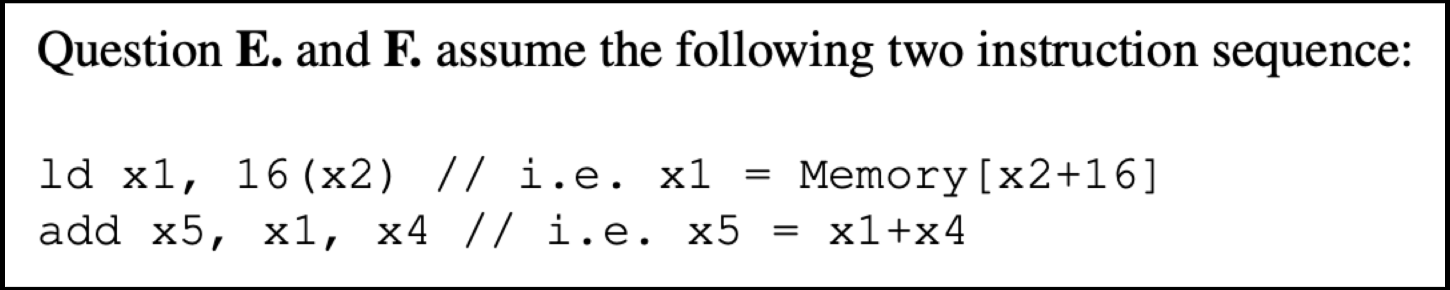


i1: ld x1, 16(x2)	IF	ID	EX	MEM	WB
bubble		IF	ID	EX	MEM
bubble			IF	ID	EX
i2: add x5, x1, x4				IF	ID

i1 has just written x1 back => No hazard

When i2 tries to read x1

4-F

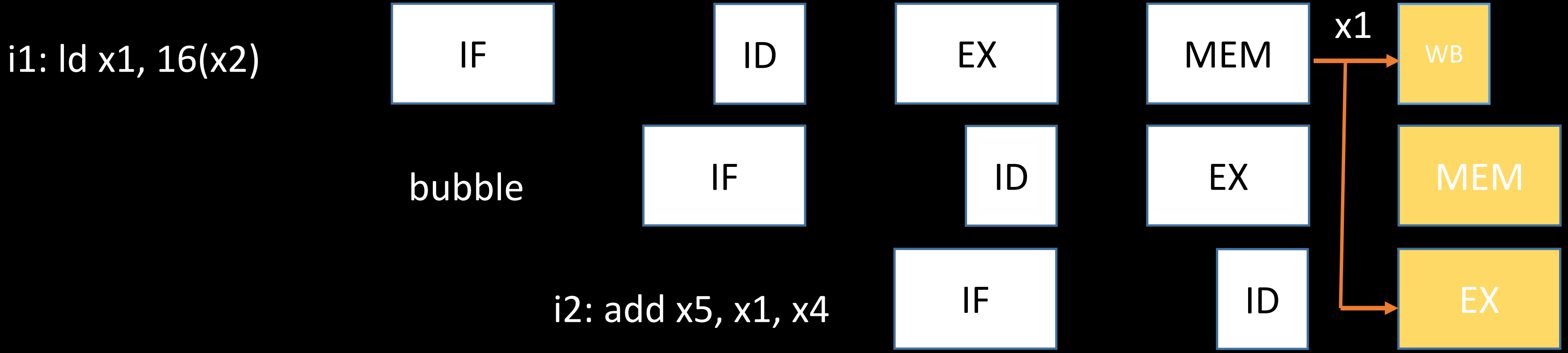


Question **E.** and **F.** assume the following two instruction sequence:

```
ld x1, 16(x2) // i.e. x1 = Memory[x2+16]
add x5, x1, x4 // i.e. x5 = x1+x4
```

Suppose the result of MEM stage is forwarded to the EX stage, how many bubbles (aka nop) must be inserted between the two instructions to ensure correctness?

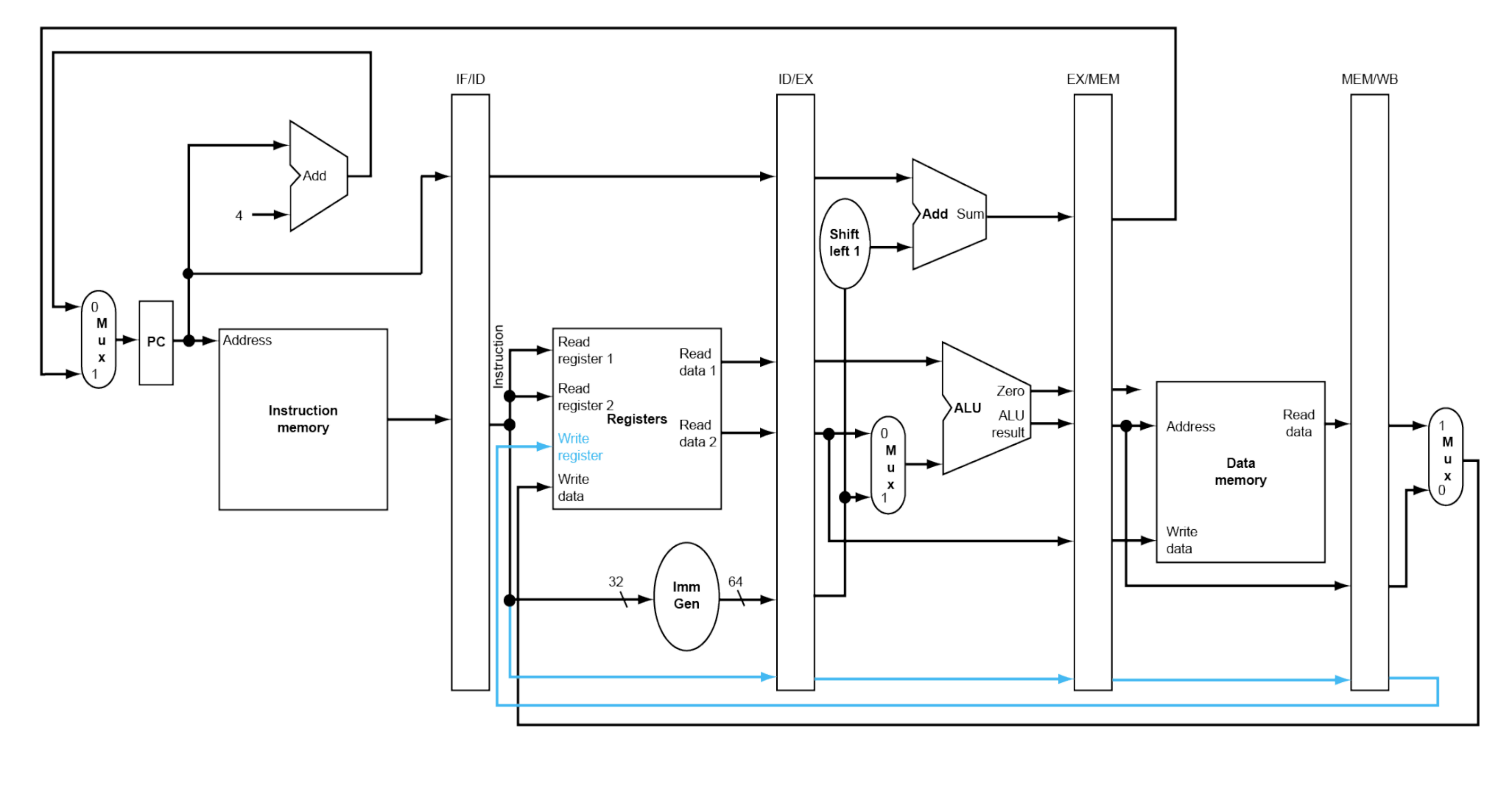
- 1



Assessment 13

Q1 5-stage pipeline

- This question assumes the 5-stage pipelined design of RISC-V (slide 21 of https://nyu-cso.github.io/notes/arch-cpu-pipeline.pdf), also reproduced here:



Q1.1

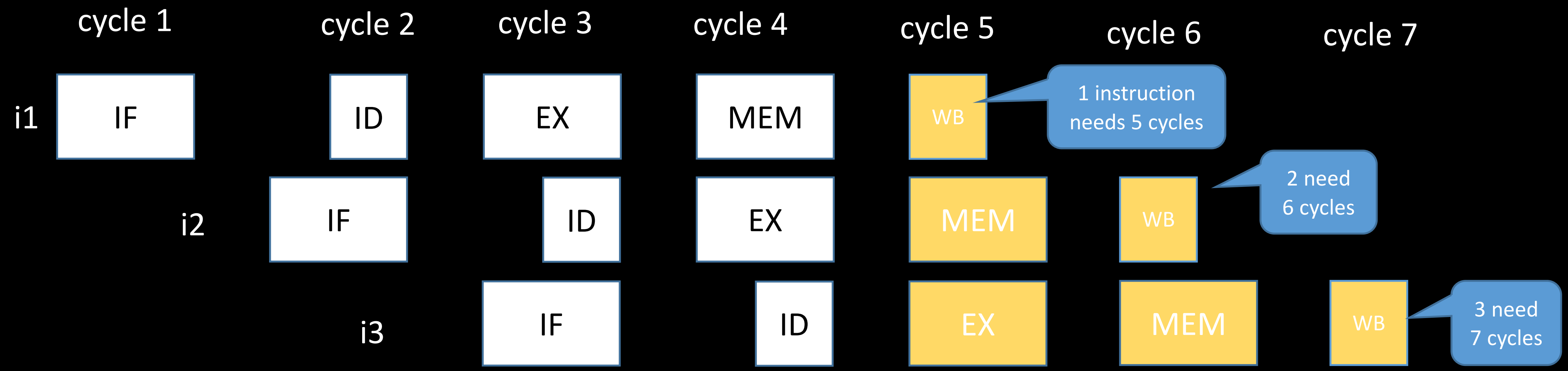
- Consider the following sequence of RISC-V instructions.

```
add x2, x1, x3 //x2=x1+x3
and x5, x1, x3 //x5 = x1 & x3
or x10, x11, x9 //x10 = x11 | x9
add x14, x1, x1 //x14 = x1+x1
add x15, x11, x12 //x15 = x11+x12
```

- How many clock cycles are needed to execute the above instruction sequence?

Q1.1

- For n instructions with no hazards, #cycles = n+#stages-1 = n+4



	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6	cycle 7
i1	IF	ID	EX	MEM	WB		
i2		IF	ID	EX	MEM	WB	
i3			IF	ID	EX	MEM	WB

1 instruction needs 5 cycles

2 need 6 cycles

3 need 7 cycles

Q1.1 has no hazards; it has 5(n) instructions, so answer is 5+4=9

- More generally (considering hazards), #cycles = n+#stages-1+#bubbles

Q1.2

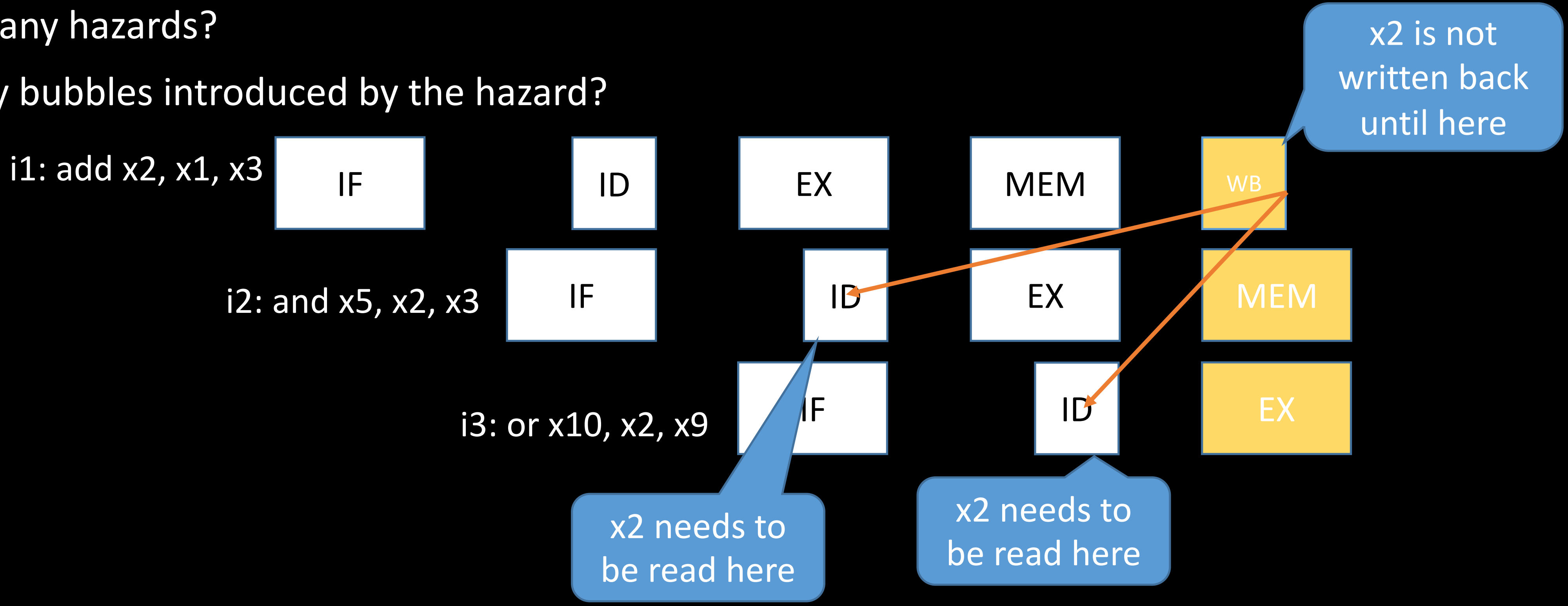
- Consider a different sequence of instructions:
- How many clock cycles are needed to execute the above instruction sequence? We assume the pipeline does not perform forwarding but only relies on stalls (aka inserting bubbles) to handle hazard. We also assume that one can read the data written to the register in the same cycle.

```
add x2, x1, x3 //x2=x1+x3
and x5, x2, x3 //x5 = x2 & x3
or x10, x2, x9 //x10 = x2 | x9
add x14, x2, x2 //x14 = x2+x2
add x15, x2, x12 //x15 = x2+x12
```

Q1.2

```
add x2, x1, x3 //x2=x1+x3
and x5, x2, x3 //x5 = x2 & x3
or x10, x2, x9 //x10 = x2 | x9
add x14, x2, x2 //x14 = x2+x2
add x15, x2, x12 //x15 = x2+x12
```

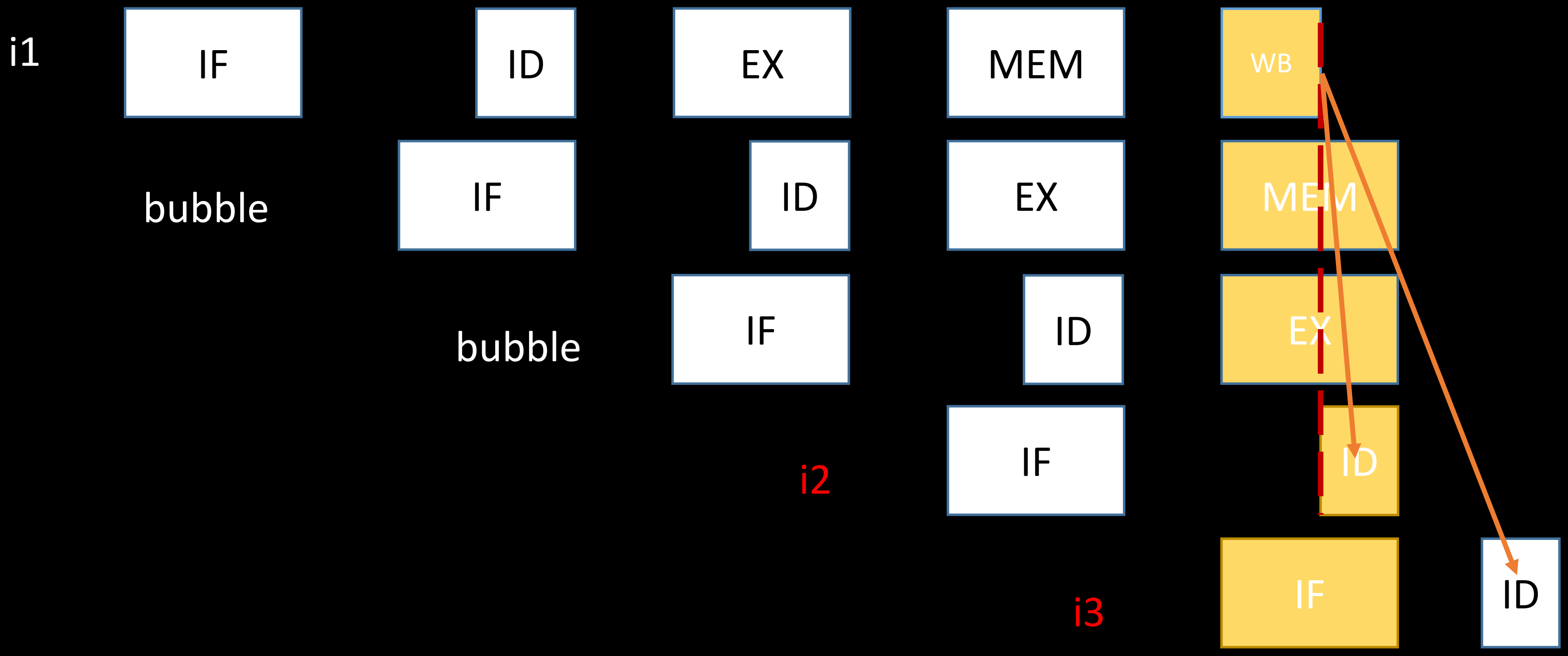
- Answer= n+#stages-1+#bubbles
- Are there any hazards?
- How many bubbles introduced by the hazard?



Q1.2

```
add x2, x1, x3 //x2=x1+x3
and x5, x2, x3 //x5 = x2 & x3
or x10, x2, x9 //x10 = x2 | x9
add x14, x2, x2 //x14 = x2+x2
add x15, x2, x12 //x15 = x2+x12
```

- Answer= n+#stages-1+#bubbles
- Are there any hazards?
- How many bubbles introduced by the hazard?
  - 2
- No hazards afterwards
- answer=9+2=11



Q1.3

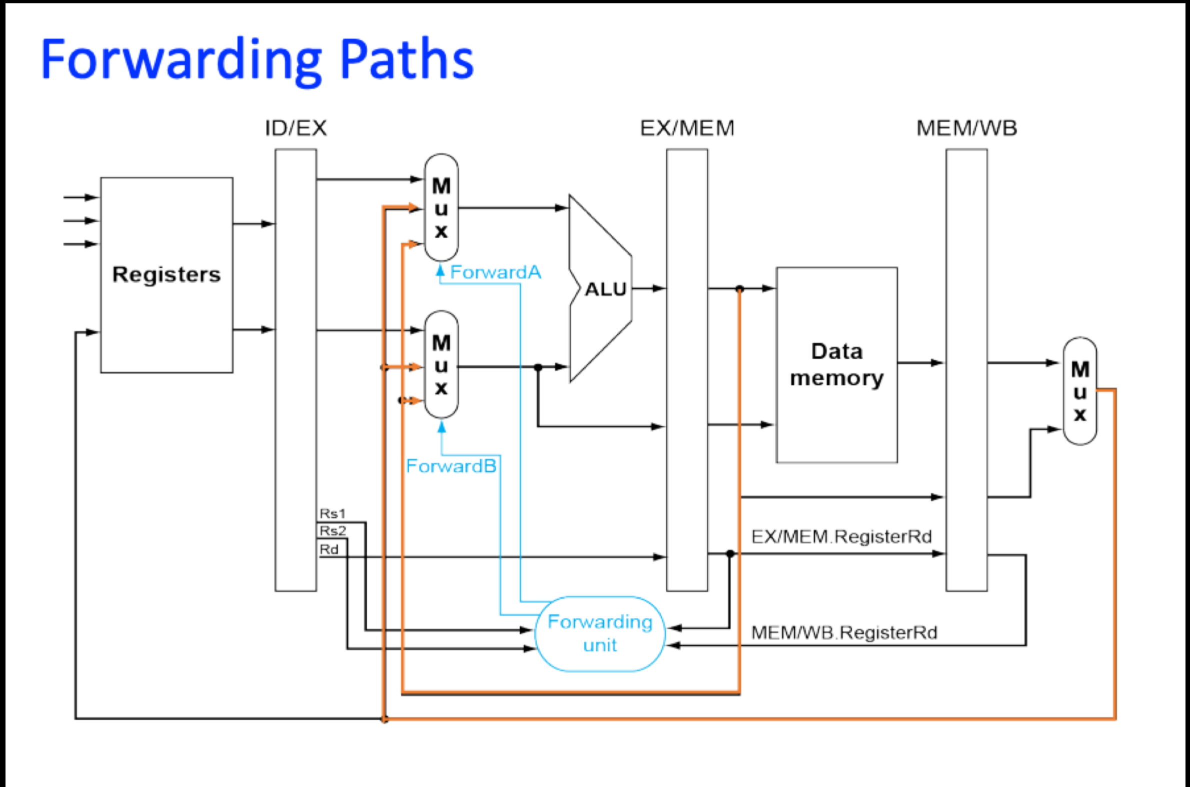
```
ld x2, 100(x1) //x2=Memory[100+x1]
and x5, x2, x3 //x5 = x2 & x3
or x10, x2, x9 //x10 = x2 | x9
add x14, x2, x2 //x14 = x2+x2
add x15, x2, x12 //x15 = x2+x12
```

- Consider yet another sequence of instructions:
- How many clock cycles are needed to execute the above instruction sequence? Like before, we assume the pipeline does not perform forwarding but only relies on stalls (aka inserting bubbles) to handle hazard. We also assume that one can read the data written to the register in the same cycle.
- same as previous question: need to postpone i2’s ID until i1’s WB’s cycle
- 11

Q1.4

```
add x2, x1, x3 //x2=x1+x3
and x5, x2, x3 //x5 = x2 & x3
or x10, x2, x9 //x10 = x2 | x9
add x14, x2, x2 //x14 = x2+x2
add x15, x2, x12 //x15 = x2+x12
```

- Suppose the CPU performs forwarding, as done on slide 34 of https://nyu-cso.github.io/notes/arch-cpu-pipeline.pdf
- How many clock cycles are needed to execute the instruction sequence in Q1.2?

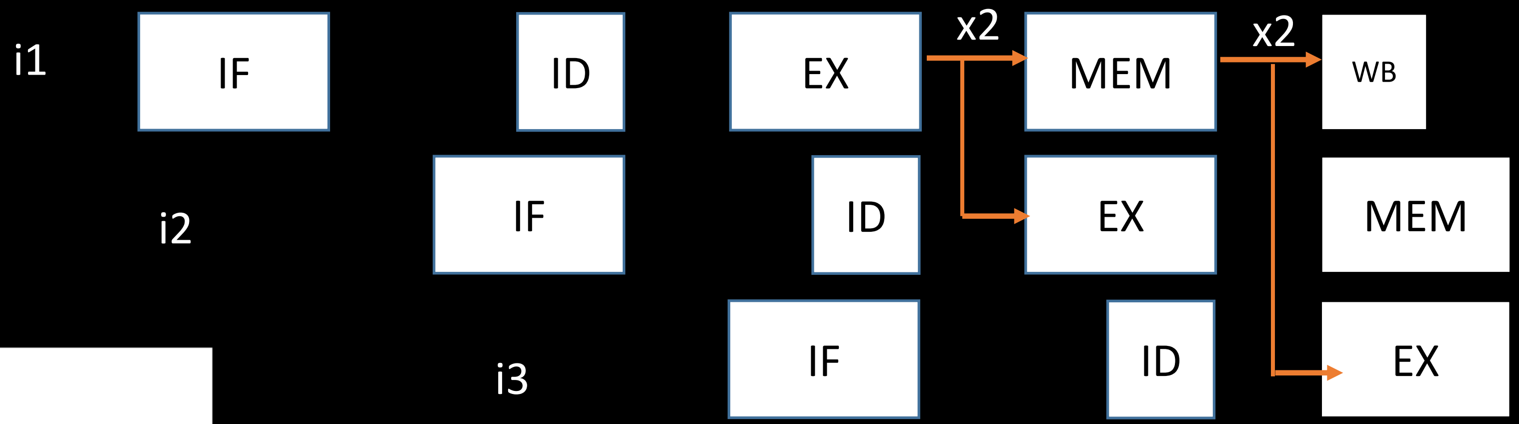


Forwarding Paths

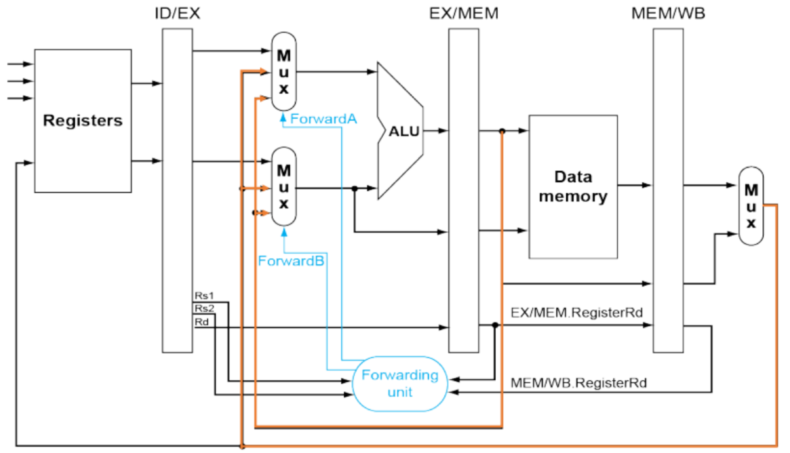
Q1.4

```
add x2, x1, x3 //x2=x1+x3
and x5, x2, x3 //x5 = x2 & x3
or x10, x2, x9 //x10 = x2 | x9
add x14, x2, x2 //x14 = x2+x2
add x15, x2, x12 //x15 = x2+x12
```

- Answer= n+#stages-1+#bubbles



Forwarding Paths

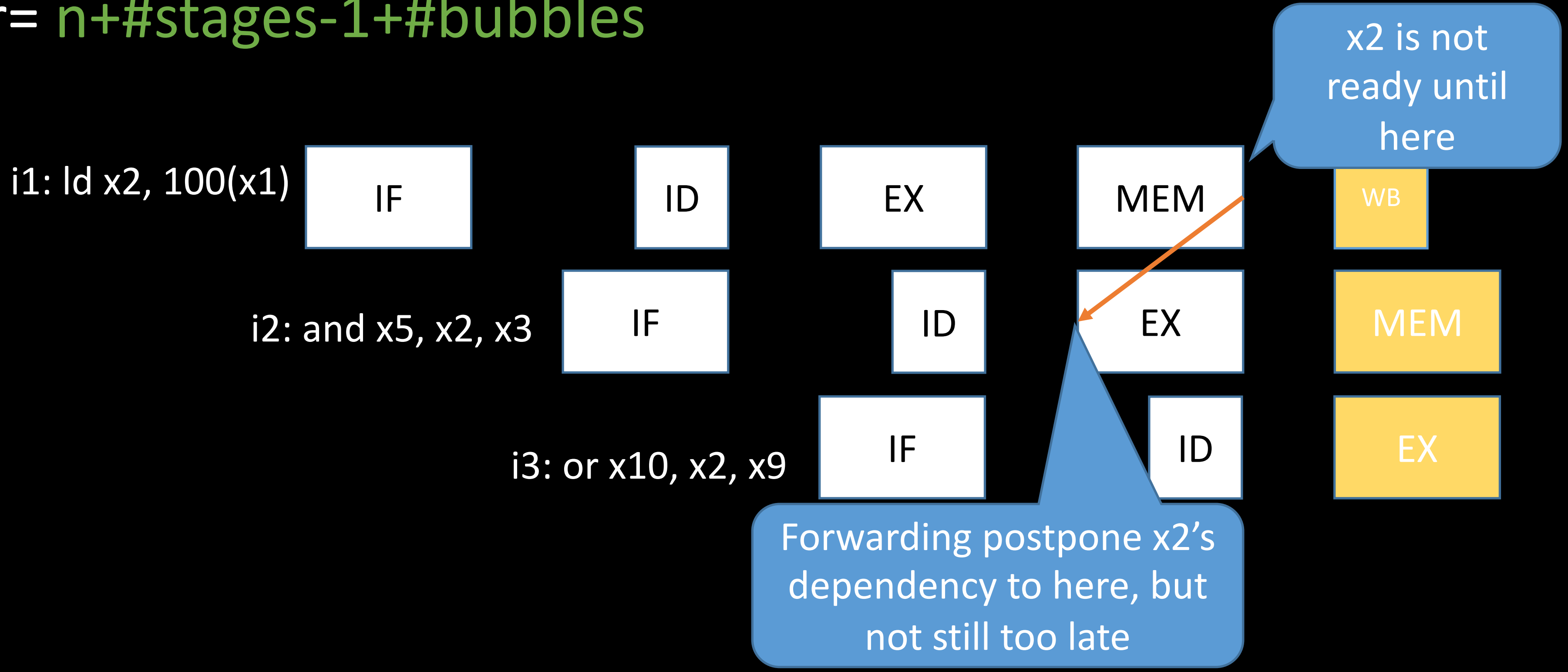


No bubbles
Answer=n+4=9

Q1.5

```
ld x2, 100(x1) //x2=Memory[100+x1]
and x5, x2, x3 //x5 = x2 & x3
or x10, x2, x9 //x10 = x2 | x9
add x14, x2, x2 //x14 = x2+x2
add x15, x2, x12 //x15 = x2+x12
```

- Suppose the CPU performs forwarding, how many clock cycles are needed to execute the instruction sequence in Q1.3?
- Answer= n+#stages-1+#bubbles

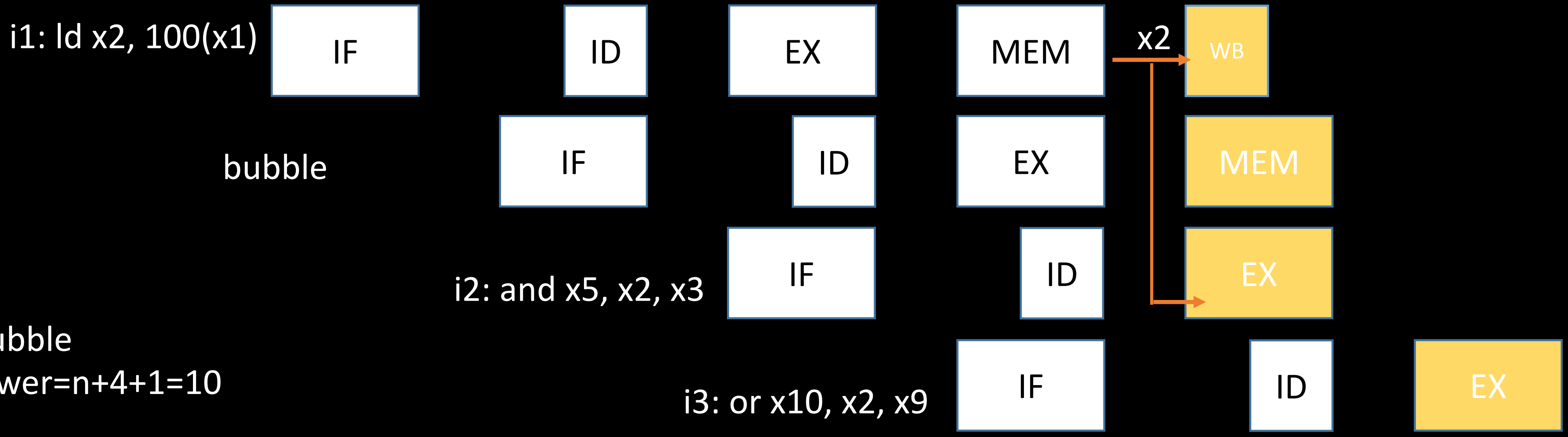


i1: ld x2, 100(x1)
IF
ID
EX
MEM
WB
i2: and x5, x2, x3
IF
ID
EX
MEM
i3: or x10, x2, x9
IF
ID
EX
x2 is not ready until here
Forwarding postpone x2's dependency to here, but not still too late

Q1.5

```
ld x2, 100(x1) //x2=Memory[100+x1]
and x5, x2, x3 //x5 = x2 & x3
or x10, x2, x9 //x10 = x2 | x9
add x14, x2, x2 //x14 = x2+x2
add x15, x2, x12 //x15 = x2+x12
```

- Suppose the CPU performs forwarding, how many clock cycles are needed to execute the instruction sequence in Q1.3?
- Answer= n+#stages-1+#bubbles

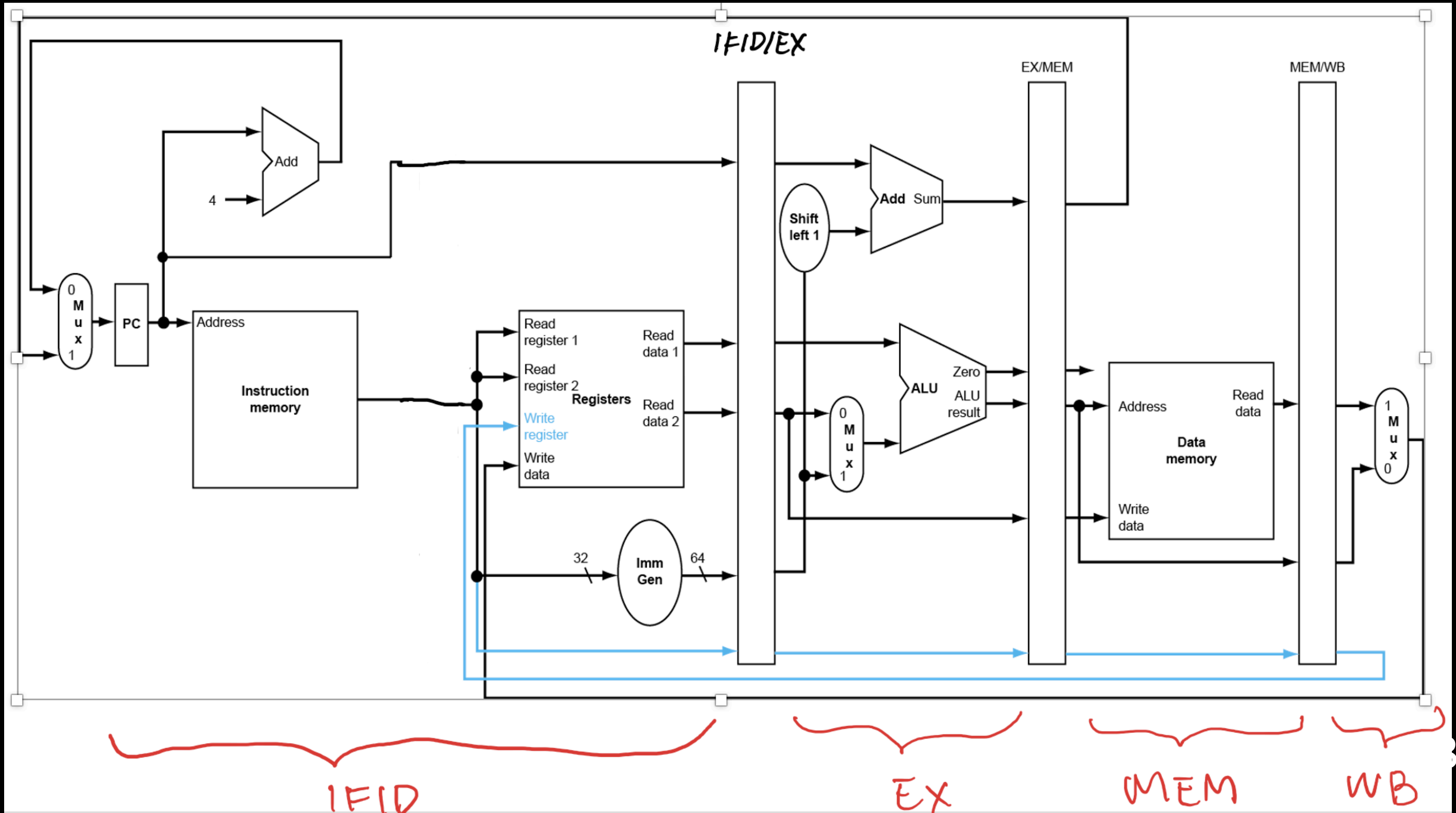


i1: ld x2, 100(x1)	IF	ID	EX	MEM	WB	
bubble		IF	ID	EX	MEM	
i2: and x5, x2, x3			IF	ID	EX	
i3: or x10, x2, x9				IF	ID	EX

1 bubble
Answer=n+4+1=10

Q2 4-stage pipeline

- Ben Bitdiddle decides to make a 4-stage pipelined CPU by merging the IF and ID stage of the 5-stage pipeline into a combined IFID stage, as shown below.



Q2.1

```
add x2, x1, x3 //x2=x1+x3
and x5, x1, x3 //x5 = x1 & x3
or x10, x11, x9 //x10 = x11 | x9
add x14, x1, x1 //x14 = x1+x1
add x15, x11, x12 //x15 = x11+x12
```

- For the instruction sequence in Q1.1, how many clock cycles are needed to execute them all? (We assume Ben sets the clock frequency appropriately so that each stage can be completed in one clock cycle)
- 8

  - #cycles = n+#stages-1+#bubbles
  - #bubbles=0
  - #stages=4
  - answer=8

Q2.2

	IF	ID	EX	MEM	WB	max
old	x	x	x	x	x	x
new	2x					2x

Assuming in the original 5-stage design, each stage takes exactly the **same** amount of time to execute. Which of the following statements are true for Ben's 4-stage pipeline design?

A. Under the new 4-stage design, the clock period can remain the same as that in the 5-stage design.

clock period=max_stage_time

[✓ B.] Under the new 4-stage design, the clock period is twice as long as that in the 5-stage design.

[✓ C.] It takes longer time to execute the instruction sequence of Q1.1 under the 4-stage design than the original 5-stage design.

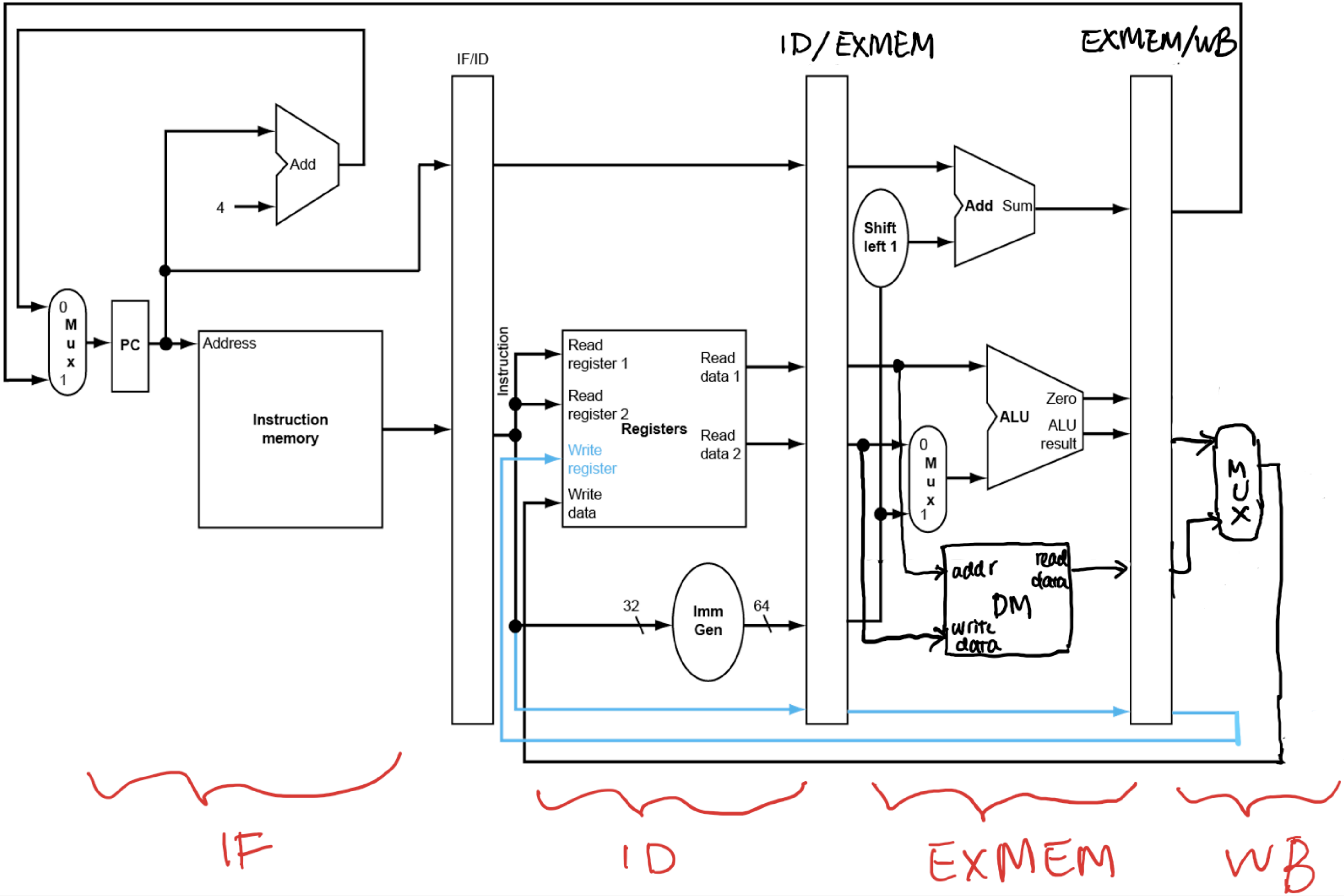
D. It takes shorter time to execute the instruction sequence o stage design than the original 5-stage design.

E. It takes the same amount of time to execute the instructio under the 4-stage design than the original 5-stage design.

	cycles	clock period	time
old	9	x	9x
new	8	2x	16x

Q3 Another 4-stage design

- Ben Bitdiddle tries another a 4-stage pipelined design. This time, he combines EX and MEM stage into a single stage EXMEM, as shown below.
- To make this new design possible, Ben modifies the RISC-V instruction set so that the load and store instructions no longer take an offset. Rather, the member address must be calculated explicitly through an arithmetic instruction and stored in some register before load and store.


IF/ID
ID/EXMEM
EXMEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
32
Imm Gen
64
Shift left 1
Add
Sum
ALU
Zero
ALU result
0
M
u
x
1
addr
read data
DM
write data
MUX
IF
ID
EXMEM
WB

Q3.1

- In the instruction sequence of Q1.3, the first instruction is ld x2, 100(x1), which is not supported by Ben's new ISA. Please write an equivalent RISC-V instruction sequence that's supported by Ben's ISA while achieving the same desired effect.
- addi x1, x1, 100
- ld x2, x1

Q3.2

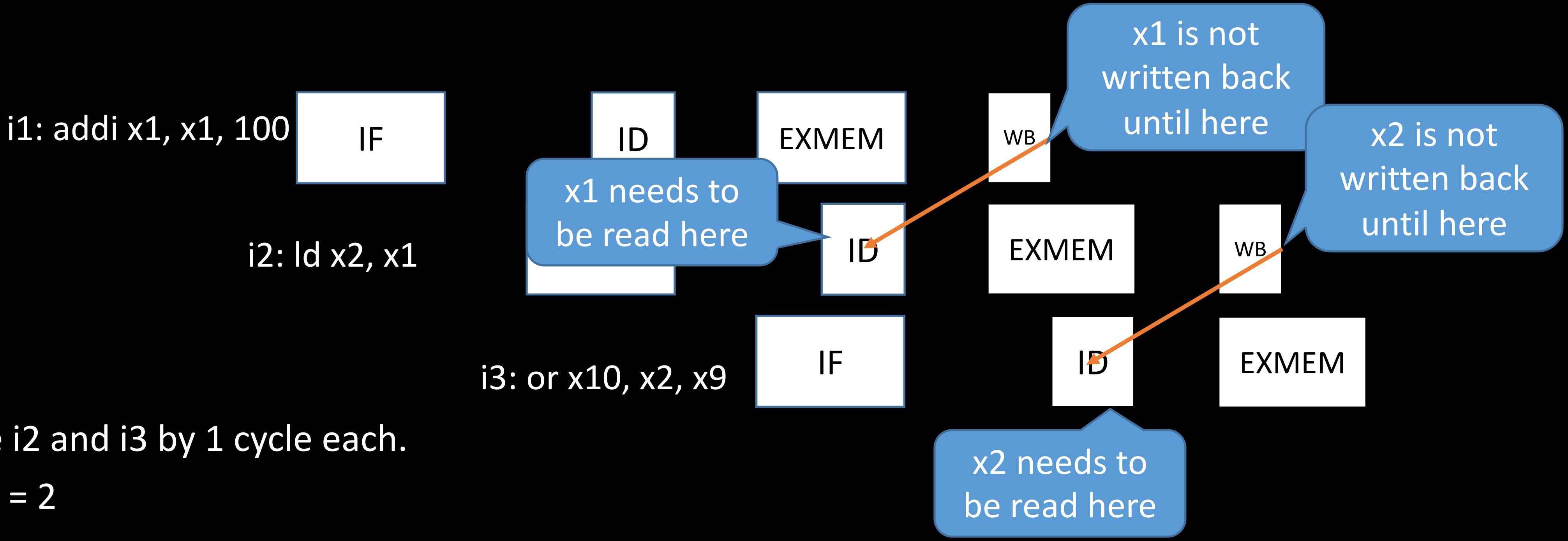
```
addi x1, x1, 100
ld x2, x1
and x5, x2, x3 //x5 = x2 & x3
or x10, x2, x9 //x10 = x2 | x9
add x14, x2, x2 //x14 = x2+x2
add x15, x2, x12 //x15 = x2+x12
```

- How many clock cycles are needed to execute the instruction sequence of Q1.3 under Ben's new 4-stage CPU design?
- We assume no forwarding.
- We assume the instruction ld x2, 100(x1) is substituted with your answer in Q3.1.

Q3.2

```
addi x1, x1, 100
ld x2, x1
and x5, x2, x3 //x5 = x2 & x3
or x10, x2, x9 //x10 = x2 | x9
add x14, x2, x2 //x14 = x2+x2
add x15, x2, x12 //x15 = x2+x12
```

- Answer= n+#stages-1+#bubbles
- Are there any hazards?



x1 is not written back until here

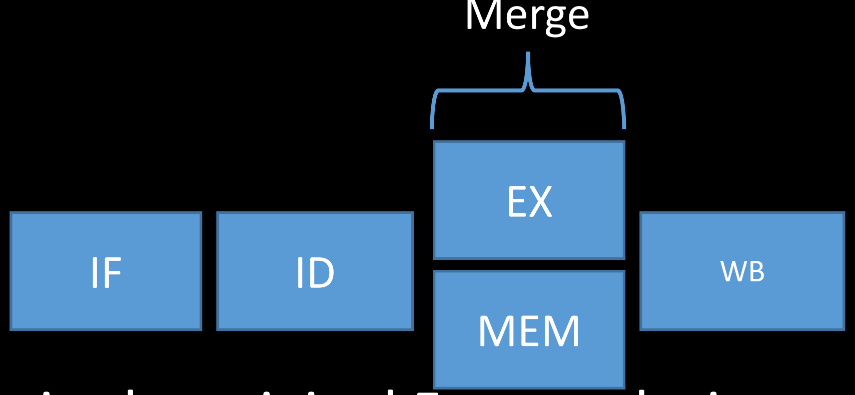
x1 needs to be read here

x2 is not written back until here

x2 needs to be read here

- Postpone i2 and i3 by 1 cycle each.
- #bubbles = 2
- Answer=6+4-1+2=11

Q3.3



	IF	ID	EX	MEM	WB	max
old	x	x	x	x	x	x
new			max(x,x)=x			

Since only one of EX or MEM used for each instruction

Assuming in the original 5-stage design, each stage takes exactly the **sa** amount of time to execute. Which of the following statements are true abou stage pipeline design?

[✓ A.] Under this 4-stage design, the clock period can remain the same as that in the 5-stage design.

B. Under this 4-stage design, the clock period is twice as long as that in the 5-stage design.

C. It takes longer time to execute the instruction sequence of **Q1.1** under this 4-stage design than the original 5-stage design.

[✓ D.] It takes shorter time to execute the instruction sequence o stage design than the original 5-stage design.

E. It takes the same amount of time to execute the instructio of **Q1.1** under this 4-stage design than the original 5-stage design.

	cycles	clock period	time
old	9	x	9x
new	8	x	8x