WEEK 10

Ôn Tập – Final review

Loop

```
loopy:
        # a in %rdi, n in %esi
        movl    $0, %ecx
        movl    $0, %edx
        testl   %esi, %esi
        jle     .L3
.L6:
        movslq  %edx,%rax
        movl    (%rdi,%rax,4), %eax
        cmpl    %eax, %ecx
        cmovl   %eax, %ecx
        addl    $1, %edx
        cmpl    %ecx, %esi
        jg      .L6
.L3:
        movl    %ecx, %eax
        ret
```

Fill the blank of corresponding C code.

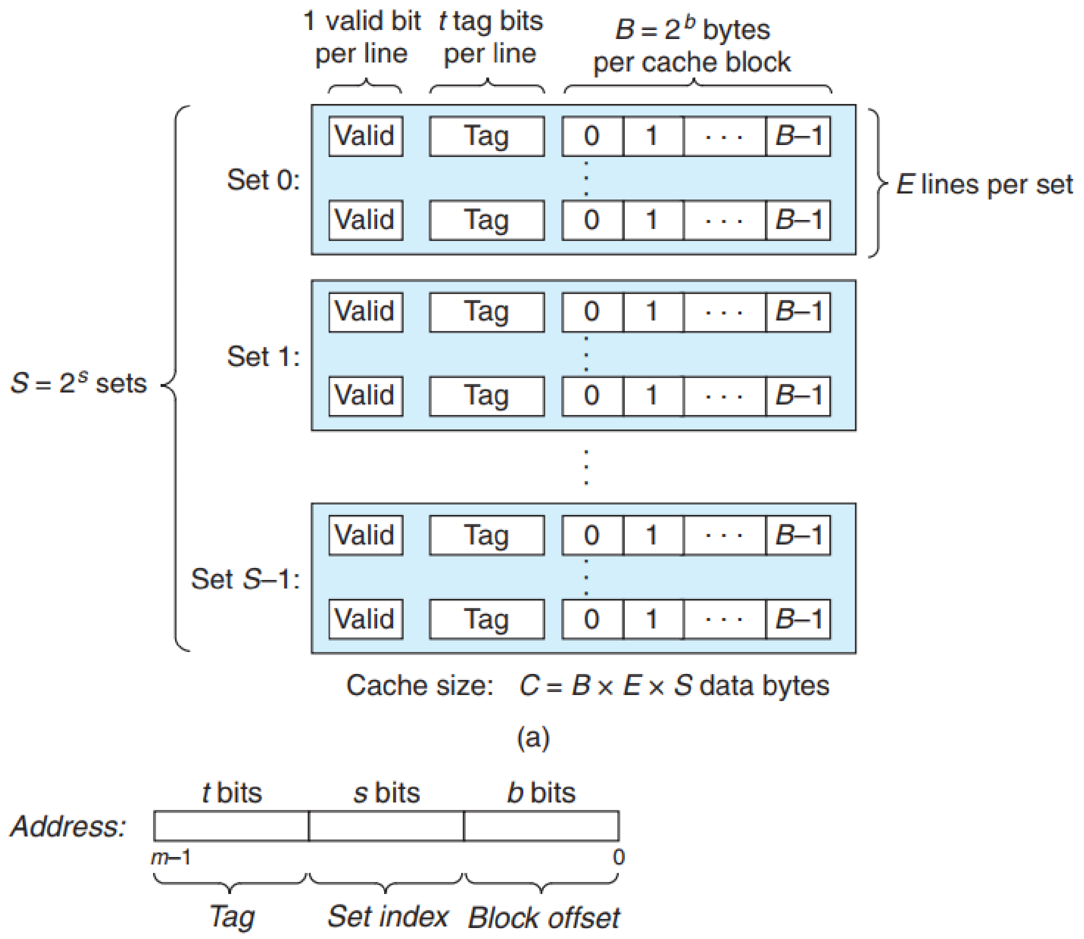
```
int loopy(int a[], int n)
{
    int i;
    int x = __0__;

    for(i = __0__; __x<n__; __i++__) {

        if (__x<a[i]__)

            x = __a[i]__;
    }
    return x;
}
```

Cache



Parameter	Description
Fundamental parameters	
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits
Derived quantities	
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of *set index bits*
$b = \log_2(B)$	Number of *block offset bits*
$t = m - (s + b)$	Number of *tag bits*
$C = B \times E \times S$	Cache size (bytes), not including overhead such as the valid and tag bits

cache

The following table gives the parameters for a number of different caches. Your task is to fill in the missing fields in the table. Recall that is the number of physical address bits, C is the cache size (number of data bytes), B is the block size in bytes, E is the associativity, S is the number of cache sets, t is the number of tag bits, s is the number of set index bits, and b is the number of block offset bits.

m	C	B	E	S	t	s	b
32	2048	8	1	**256**	21	8	3
32	2048	**4**	**4**	128	23	7	2
32	1024	2	8	64	**25**	6	1
32	1024	**32**	2	16	23	4	**5**

Cache hit - miss

Suppose we have a system with the following properties:

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not to 4-byte words).
- Addresses are 12 bits wide.
- The cache is two-way set associative (E = 2), with a 4-byte block size (B = 4) and four sets (S = 4).

The contents of the cache are as follows, with all addresses, tags, and values given in hexadecimal notation:

Set index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	00	1	40	41	42	43
	83	1	FE	97	CC	D0
1	00	1	44	45	46	47
	83	0	---	---	---	---
2	00	1	48	49	4A	4B
	40	0	---	---	---	---
3	FF	1	9A	C0	03	FF
	00	0	---	---	---	---

CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CO	CO
11	10	9	8	7	6	5	4	3	2	1	0

Operation	Address	Hit	Read (or unknow)
Read	0x834	No	Unknow
Write	0x836	Yes	Unknow
Read	0xFFD	Yes	0xC0

Xác định giá trị

4-way set associative cache

Index	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3
0	F0	1	ED 32 0A A2	8A	1	BF 80 1D FC	14	1	EF 09 86 2A	BC	0	25 44 6F 1A
1	BC	0	03 3E CD 38	A0	0	16 7B ED 5A	BC	1	8E 4C DF 18	E4	1	FB B7 12 02
2	BC	1	54 9E 1E FA	B6	1	DC 81 B2 14	00	0	B6 1F 7B 44	74	0	10 F5 B8 2E
3	BE	0	2F 7E 3D A8	C0	1	27 95 A4 74	C4	0	07 11 6B D8	BC	0	C7 B7 AF C2
4	7E	1	32 21 1C 2C	8A	1	22 C2 DC 34	BC	1	BA DD 37 D8	DC	0	E7 A2 39 BA
5	98	0	A9 76 2B EE	54	0	BC 91 D5 92	98	1	80 BA 9B F6	BC	1	48 16 81 0A
6	38	0	5D 4D F7 DA	BC	1	69 C2 8C 74	8A	1	A8 CE 7F DA	38	1	FA 93 EB 48
7	8A	1	04 2A 32 6A	9E	0	B1 86 56 0E	CC	1	96 30 47 F2	BC	1	F8 1D 42 30

CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO
12	11	10	9	8	7	6	5	4	3	2	1	0

13bit Addr

Read 0x071A

Block offset (CO)	**0x2**
Index (CI)	**0x6**
Cache tag (CT)	**0x38**
Cache hit? (Y/N)	**Y**
Cache byte return	**0xEB**

Xác định địa chỉ

4-way set associative cache												
Index	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3
0	F0	1	ED 32 0A A2	8A	1	BF 80 1D FC	14	1	EF 09 86 2A	BC	0	25 44 6F 1A
1	BC	0	03 3E CD 38	A0	0	16 7B ED 5A	BC	1	8E 4C DF 18	E4	1	FB B7 12 02
2	BC	1	54 9E 1E FA	B6	1	DC 81 B2 14	00	0	B6 1F 7B 44	74	0	10 F5 B8 2E
3	BE	0	2F 7E 3D A8	C0	1	27 95 A4 74	C4	0	07 11 6B D8	BC	0	C7 B7 AF C2
4	7E	1	32 21 1C 2C	8A	1	22 C2 DC 34	BC	1	BA DD 37 D8	DC	0	E7 A2 39 BA
5	98	0	A9 76 2B EE	54	0	BC 91 D5 92	98	1	80 BA 9B F6	BC	1	48 16 81 0A
6	38	0	5D 4D F7 DA	BC	1	69 C2 8C 74	8A	1	A8 CE 7F DA	38	1	FA 93 EB 48
7	8A	1	04 2A 32 6A	9E	0	B1 86 56 0E	CC	1	96 30 47 F2	BC	1	F8 1D 42 30

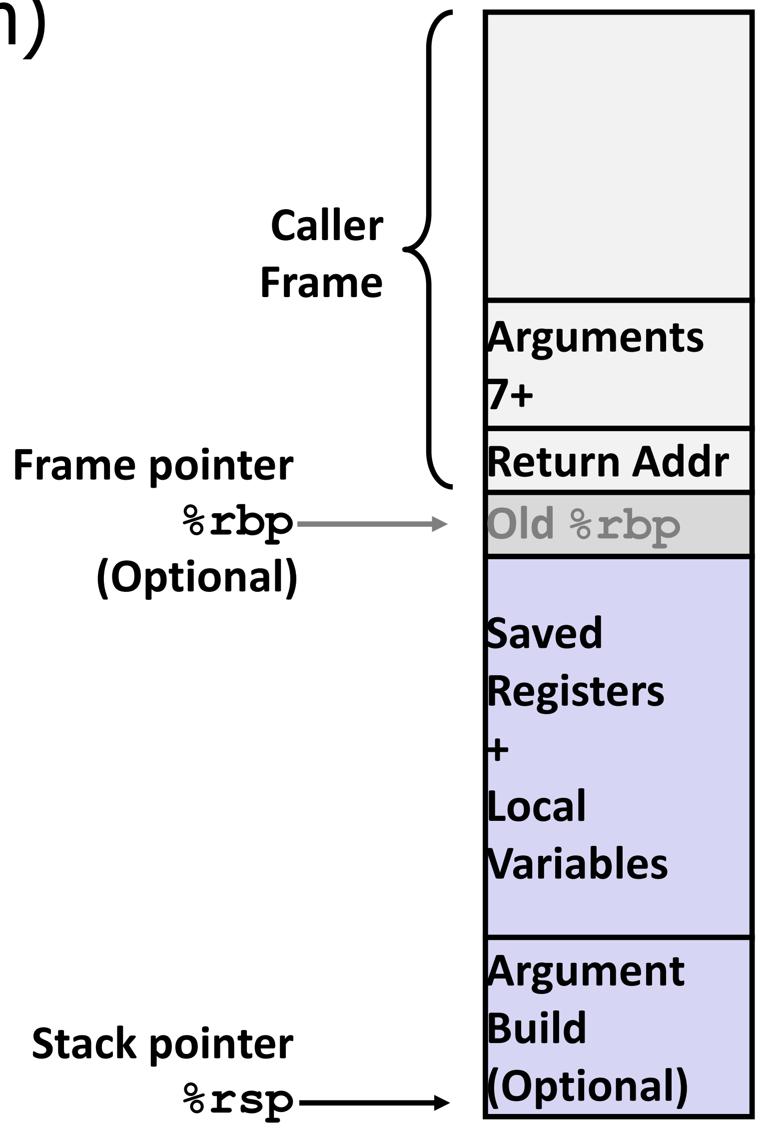
Read 0x16E8

Block offset (CO)	**0x0**
Index (CI)	**0x2**
Cache tag (CT)	**0xB7**
Cache hit? (Y/N)	**N**
Cache byte return	**---**

x86-64/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)

- Caller Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



Example: `incr`

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

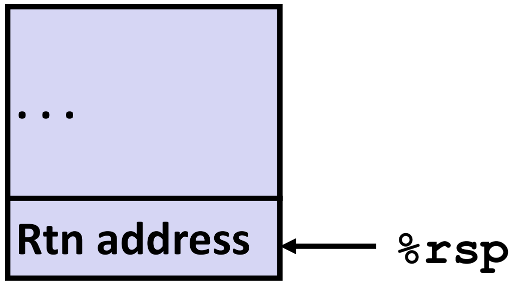
Register	Use(s)
`%rdi`	Argument `p`
`%rsi`	Argument `val`, `y`
`%rax`	`x`, Return value

Example: Calling `incr` #1

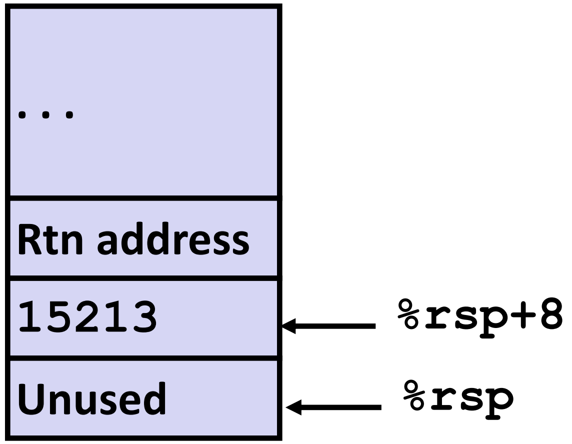
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack Structure



Resulting Stack Structure

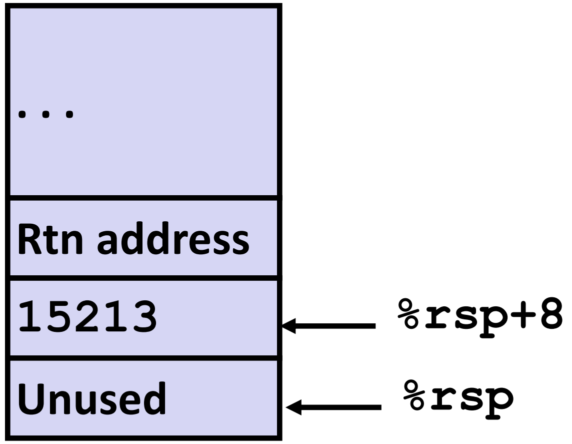


Example: Calling `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq     $16, %rsp
  movq     $15213, 8(%rsp)
  movl     $3000, %esi
  leaq     8(%rsp), %rdi
  call     incr
  addq     8(%rsp), %rax
  addq     $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure

...	
Rtn address	
18213	← `%rsp+8`
Unused	← `%rsp`

Register	Use(s)
`%rdi`	`&v1`
`%rsi`	`3000`

Example: Calling `incr` #4

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure

. . .
Rtn address
18213 ← `%rsp+8`
Unused ← `%rsp`

Register	Use(s)
`%rax`	Return value

Updated Stack Structure

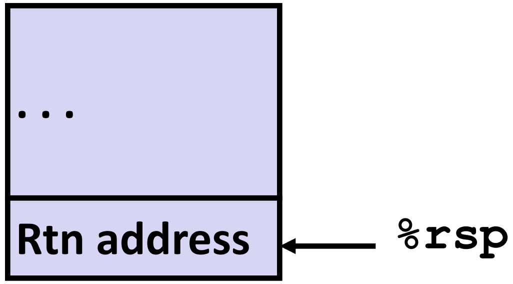
. . .
Rtn address ← `%rsp`

Example: Calling `incr` #5

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

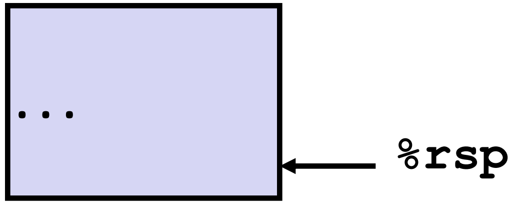
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Updated Stack Structure



Register	Use(s)
`%rax`	Return value

Final Stack Structure



```
    Disassembly of last(long u, long v)
    u in %rdi, v in %rsi
0000000000400540 <last>:
  400540:  48 89 f8          mov    %rdi,%rax          L1: u
  400543:  48 0f af c6       imul   %rsi,%rax          L2: u*v
  400547:  c3                retq                      L3: Return

    Disassembly of last(long x)
    x in %rdi
0000000000400548 <first>:
  400548:  48 8d 77 01       lea    0x1(%rdi),%rsi     F1: x+1
  40054c:  48 83 ef 01       sub    $0x1,%rdi          F2: x-1
  400550:  e8 eb ff ff ff    callq  400540 <last>      F3: Call last(x-1,x+1)
  400555:  f3 c3             repz retq                 F4: Return
    .
    .
  400560:  e8 e3 ff ff ff    callq  400548 <first>     M1: Call first(10)
  400565:  48 89 c2          mov    %rax,%rdx          M2: Resume
```

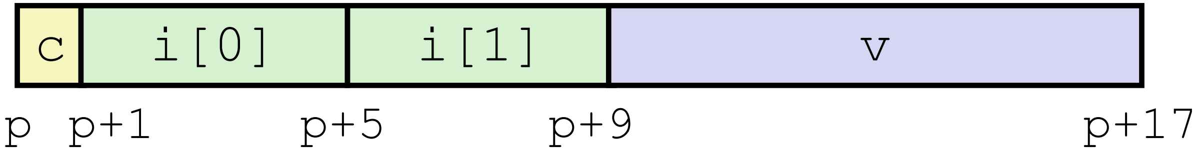
Instruction			State values (at beginning)					
Label	PC	Instruction	%rdi	%rsi	%rax	%rsp	*%rsp	Description
M1	0x400560	callq	10	—	—	0x7fffffffe820	—	Call first(10)
F1								
F2								
F3								
L1								
L2								
L3								
F4								
M2								

Starting with the calling of first(10) by main, fill in the following table to trace instruction execution through to the point where the program returns back to main

Instruction			State values (at beginning)					
Label	PC	Instruction	%rdi	%rsi	%rax	%rsp	*%rsp	Description
M1	0x400560	callq	10	—	—	0x7fffffffe820	—	Call first(10)
F1	0x400548	lea	10	—	—	0x7fffffffe818	0x400565	Entry of first
F2	0x40054c	sub	10	11	—	0x7fffffffe818	0x400565	
F3	0x400550	callq	9	11	—	0x7fffffffe818	0x400565	Call last(9, 11)
L1	0x400540	mov	9	11	—	0x7fffffffe810	0x400555	Entry of last
L2	0x400543	imul	9	11	9	0x7fffffffe810	0x400555	
L3	0x400547	retq	9	11	99	0x7fffffffe810	0x400555	Return 99 from last
F4	0x400555	repz repq	9	11	99	0x7fffffffe818	0x400565	Return 99 from first
M2	0x400565	mov	9	11	99	0x7fffffffe820	—	Resume main

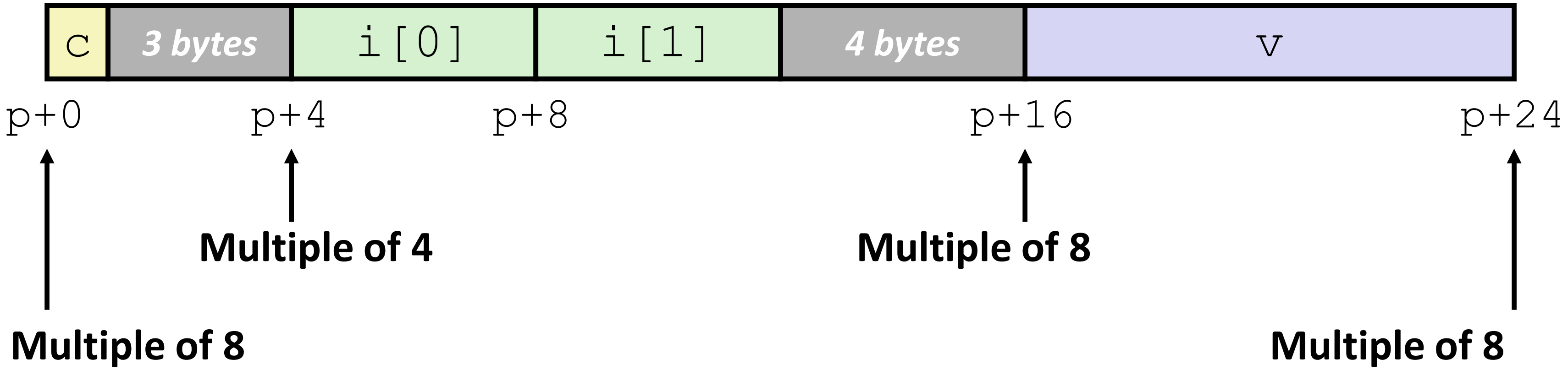
Structures & Alignment

- **Unaligned Data**




```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- **Aligned Data**
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***



Alignment Principles

- Aligned Data
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***
  - Required on some machines; advised on x86-64
- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory trickier when datum spans 2 pages
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

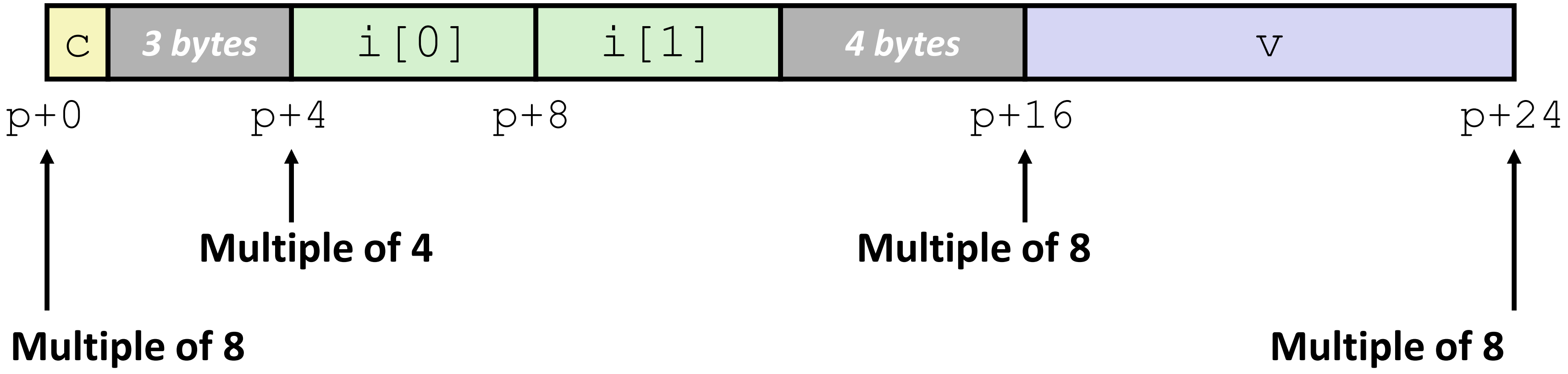
Specific Cases of Alignment (x86-64)

- 1 byte: **`char`**, …
  - no restrictions on address
- 2 bytes: **`short`**, …
  - lowest 1 bit of address must be 0_2
- 4 bytes: **`int`**, **`float`**, …
  - lowest 2 bits of address must be 00_2
- 8 bytes: **`double`**, `long`, **`char *`**, …
  - lowest 3 bits of address must be 000_2
- 16 bytes: **`long double`** (GCC on Linux)
  - lowest 4 bits of address must be 0000_2

Satisfying Alignment with Structures

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement **K**
    - **K** = Largest alignment of any element
  - Initial address & structure length must be multiples of **K**
- Example:
  - K = 8, due to **`double`** element

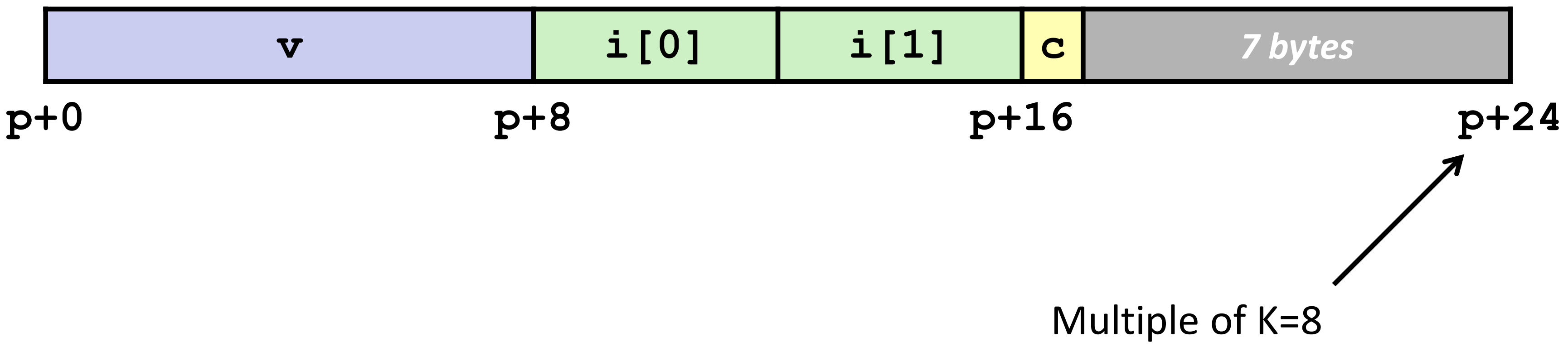
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Meeting Overall Alignment Requirement

- For largest alignment requirement K
- Overall structure must be multiple of K

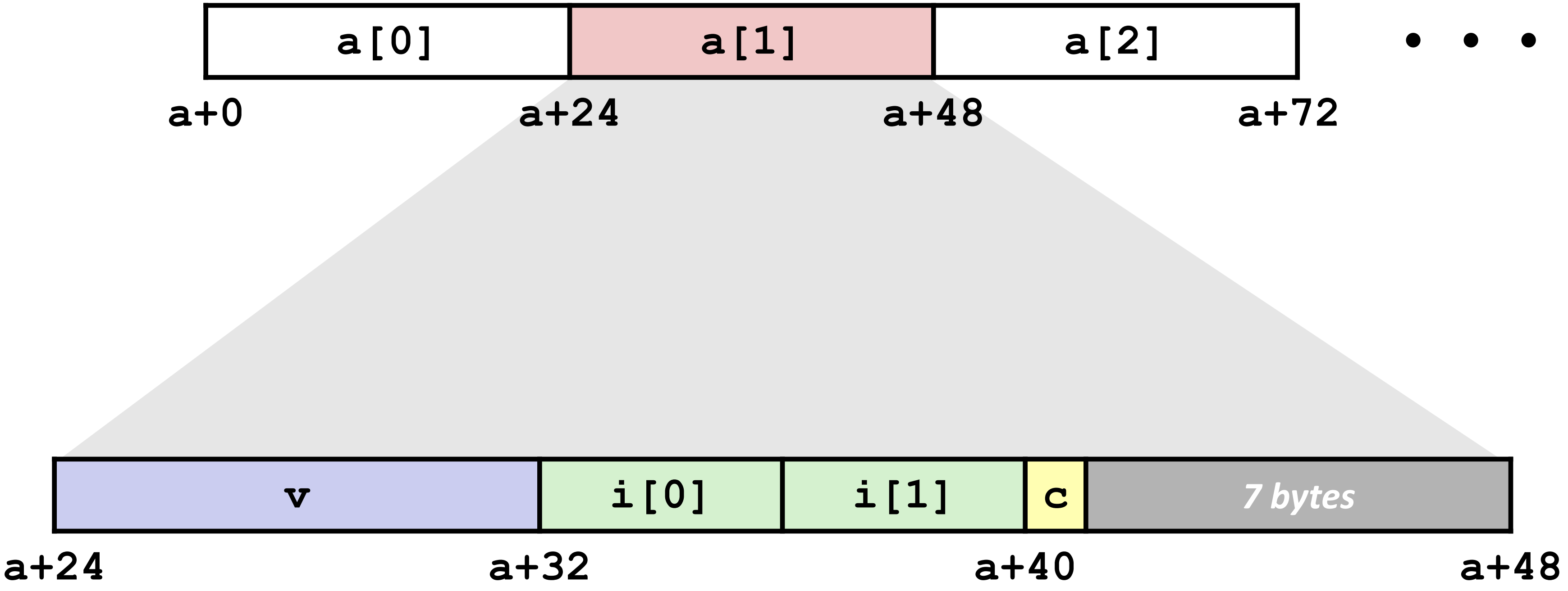
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Struct data alignment

Show how the struct would appear on a 64-bit machine (primitives of size *k* are *k*-byte aligned). Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks or x's to indicate bytes that are allocated in the struct but are not used.

```
struct {
   char a[9];  // 9 bytes
   short b[3]; // 6 bytes
   float c;    // 4 bytes
   char d;     // 1 byte
   int e;      // 4 bytes
   char *f;    // 8 bytes (pointer size on 64-bit machine)
   short g;    // 2 bytes
} foo;
```

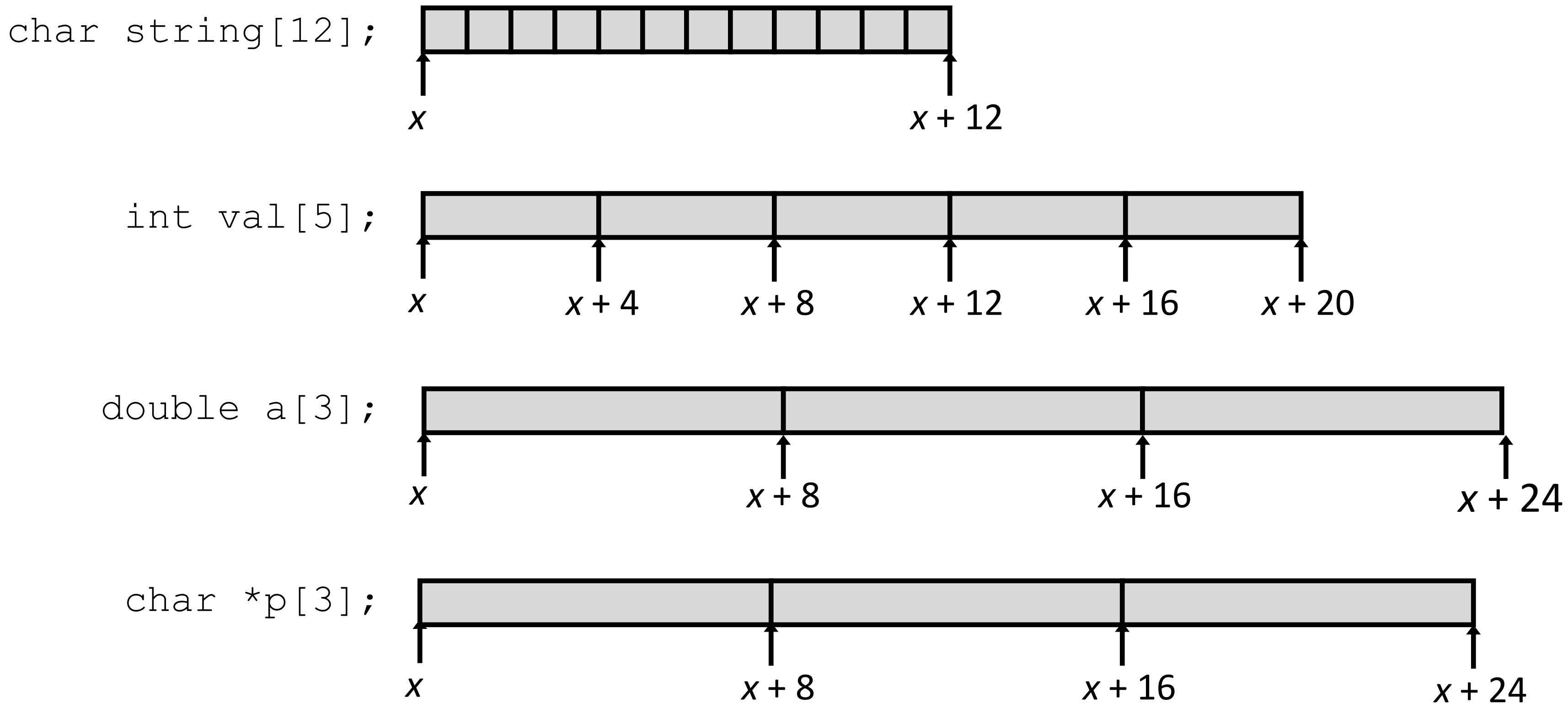
a									x	b						c				d	x	x	x	e				x	x	x	x	f								g		x	x	x	x	x	x
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47

Array Allocation

Basic Principle

T **A[*L*]**;

- Array of data type *T* and length *L*
- Contiguously allocated region of *L* * **sizeof**(*T*) bytes in memory



Array find M?

```c
void transpose(long A[M][M]) {
    long i, j;
    for (i = 0; i < M; i++)
        for (j = 0; j < i; j++) {
            long t = A[i][j];
            A[i][j] = A[j][i];
            A[j][i] = t;
        }
}
```

```
1  .L6:
2    movq    (%rdx), %rcx
3    movq    (%rax), %rsi
4    movq    %rsi, (%rdx)
5    movq    %rcx, (%rax)
6    addq    $8, %rdx
7    addq    $120, %rax
8    cmpq    %rdi, %rax
9    jne     .L6
```

- %rax và %rdx được dùng như con trỏ, ta thấy dòng 6, %rdx tăng lên mỗi lần 8, vậy %rdx chính là A[i][j].
- Thanh ghi còn lại %rax chính là A[j][i]
- Ta thấy %rax tăng mỗi lần là 120, vậy nên M =120/8=15.

Array

Consider the following source code, where M and N are constants declared with `#define`:

```
long P[M][N];
long Q[N][M];

long sum_element(long i, long j) {
    return P[i][j] + Q[j][i];
}
```

```
    long sum_element(long i, long j)
    i in %rdi, j in %rsi
1   sum_element:
2     leaq    0(,%rdi,8), %rdx
3     subq    %rdi, %rdx
4     addq    %rsi, %rdx
5     leaq    (%rsi,%rsi,4), %rax
6     addq    %rax, %rdi
7     movq    Q(,%rdi,8), %rax
8     addq    P(,%rdx,8), %rax
9     ret
```

Use your reverse engineering skills to determine the values of M and N based on this assembly code.

Fomular for calculating the column sum of Array

&A[i][j]=xA+L(C.i+j)

2. 8i
3. 7i
4. 7i+j
5. 5j
6. 5j+i
7.Q+8(5j+i)
8. P+8(7i+j)

N=7
M=5

Array

```c
int array1[H][J];
int array2[J][H];

int copy_array(int x, int y)
{
    array2[y][x] = array1[x][y];
    return 1;
}
```

J=?

H=?

```
# On entry:
#   %edi = x
#   %esi = y
#
copy_array:
   movslq %edi, %rdi
   movslq %esi, %rsi
   movq %rdi, %rax
   leaq (%rsi, %rsi, 2), %rdx
   salq $5, %rax
   subq %rdi, %rax
   leaq (%rdi, %rdx, 2), %rdx
   addq %rsi, %rax
   movl array1(%rax, 4), %eax
   movl %eax, array2(%rdx, 4)
   movl $1, %eax
   ret
```