COMP201

Computer Systems & Programming

Lecture #19 – Data and Stack Frames

Aykut Erdem // Koç University // Fall 2022

KOÇ UNIVERSITY

Good news, everyone!

- Assignment 4 is out
  (due Dec 25)

Recap

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions

Recap: Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – `%rip` must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.

- **Pass Data** – we must pass any parameters and receive any return value.

- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.

Recap: Instruction Pointer

- Machine code instructions live in main memory, just like stack and heap data.

- `%rip` is a register that stores a number (an address) of the next instruction to execute. It marks our place in the program's instructions.

- To advance to the next instruction, special hardware adds the size of the current instruction in bytes.

- `jmp` instructions work by adjusting `%rip` by a specified amount.

Recap: `%rsp`

- **`%rsp`** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).

Main Memory
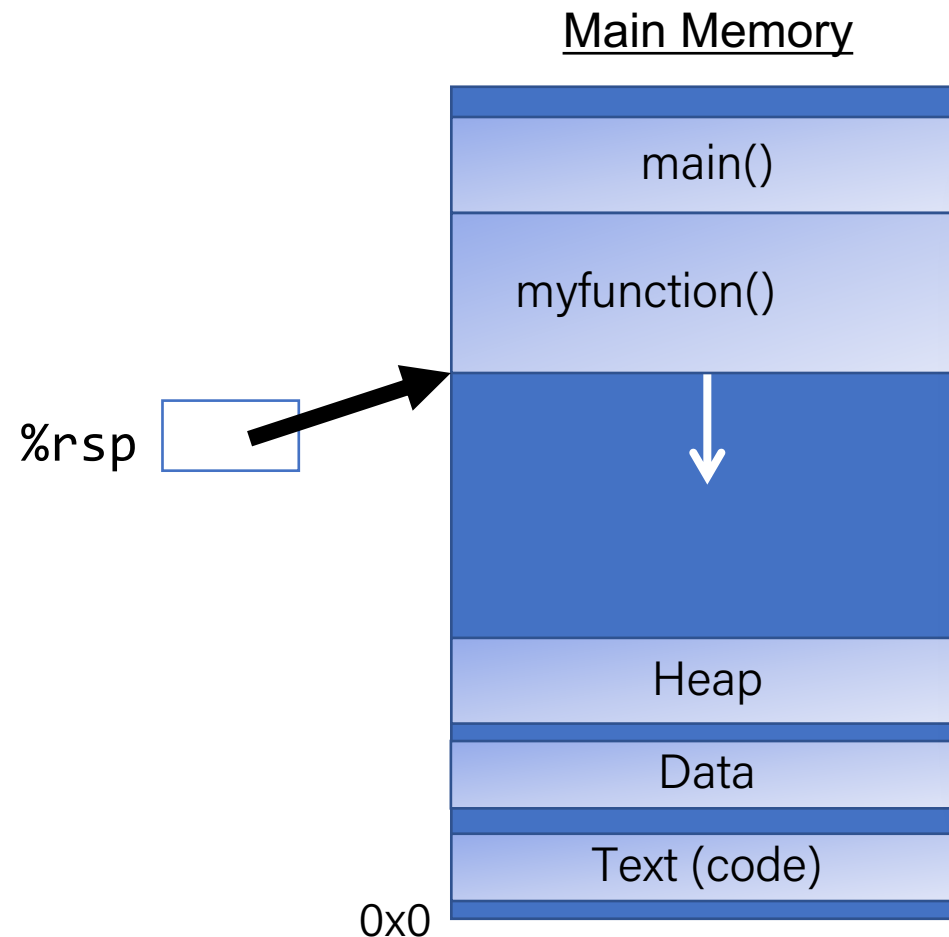
Key idea: `%rsp` must point to the same place before a function is called and after that function returns, since stack frames go away when a function finishes.

Recap: push and pop

Instruction	Effect
pushq S	R[%rsp] ← R[%rsp] – 8; M[R[%rsp]] ← S

Instruction	Effect
popq D	D ← M[R[%rsp]] R[%rsp] ← R[%rsp] + 8;

- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

- The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting **%rsp** accordingly.

- **Note:** this _does not_ remove/clear out the data! It just increments **%rsp** to indicate the next push can overwrite that location.

Recap: Call And Return

The **call** instruction pushes the address of the instruction immediately following the **call** instruction onto the stack and sets **%rip** to point to the beginning of the specified function's instructions.

<div align="center">

call Label

call *Operand

</div>

The **ret** instruction pops this instruction address from the stack and stores it in **%rip**.

<div align="center">

ret

</div>

The stored **%rip** value for a function is called its **return address.** It is the address of the instruction at which to resume the function's execution. (not to be confused with **return value**, which is the value returned from a function).

Recap: Local Storage

- So far, we've often seen local variables stored directly in registers, rather than on the stack as we'd expect. This is for optimization reasons.

- There are **three** common reasons that local data must be in memory:
  - We've run out of registers
  - The '**&**' operator is used on it, so we must generate an address for it
  - They are arrays or structs (need to use address arithmetic)

Recap: Register Restrictions

Caller-Owned (Callee Saved)

- Callee must *save* the existing value and *restore* it when done.

- Caller can store values and assume they will be preserved across function calls.

Callee-Owned (Caller Saved)

- Callee does not need to save the existing value.

- Caller's values could be overwritten by a callee! The caller may consider saving values elsewhere before calling functions.

63		31		15	7	0	
%rax		%eax		%ax		%al	Return value
%rbx		%ebx		%bx		%bl	Callee saved
%rcx		%ecx		%cx		%cl	4th argument
%rdx		%edx		%dx		%dl	3rd argument
%rsi		%esi		%si		%sil	2nd argument
%rdi		%edi		%di		%dil	1st argument
%rbp		%ebp		%bp		%bpl	Callee saved
%rsp		%esp		%sp		%spl	Stack pointer
%r8		%r8d		%r8w		%r8b	5th argument
%r9		%r9d		%r9w		%r9b	6th argument
%r10		%r10d		%r10w		%r10b	Caller saved
%r11		%r11d		%r11w		%r11b	Caller saved
%r12		%r12d		%r12w		%r12b	Callee saved
%r13		%r13d		%r13w		%r13b	Callee saved
%r14		%r14d		%r14w		%r14b	Callee saved
%r15		%r15d		%r15w		%r15b	Callee saved

Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

Recap: Caller-Owned Registers

main

calls
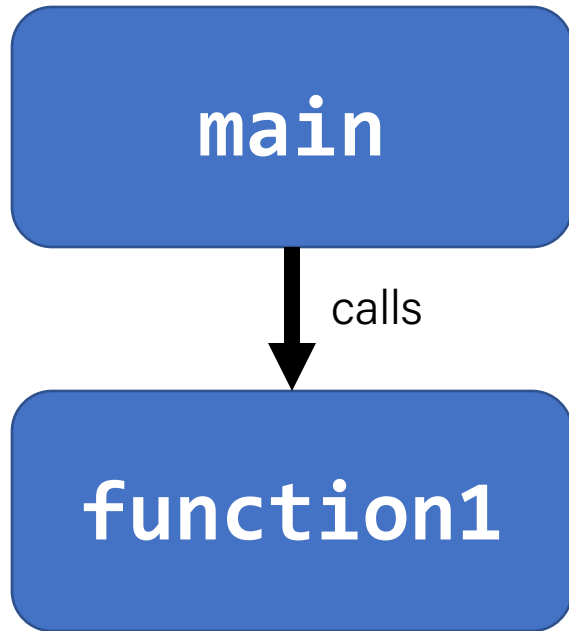
function1

```
function1:
    push %rbp
    push %rbx
    ...
    pop %rbx
    pop %rbp
    retq
```
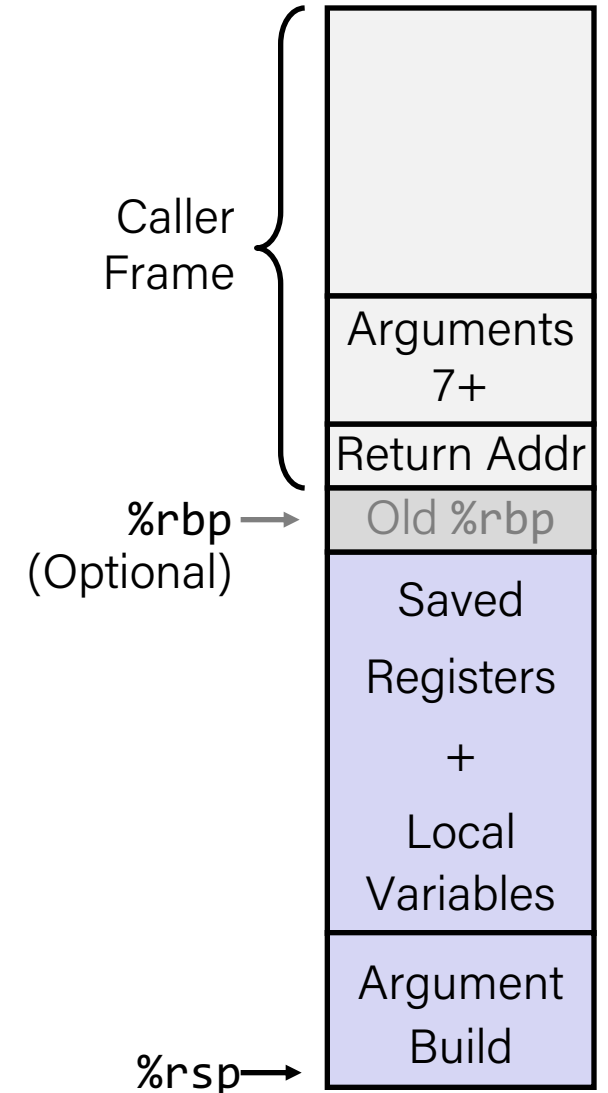
Recap: Callee-Owned Registers
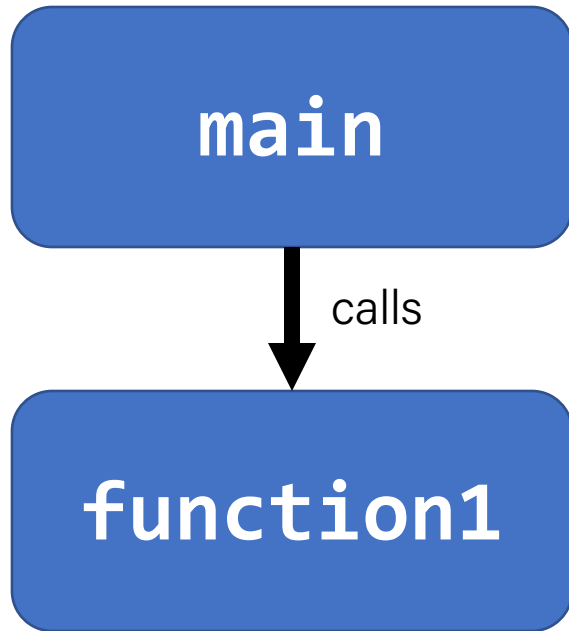

```
main:
    ...
    push %r10
    push %r11
    callq function1
    pop %r11
    pop %r10
    ...
```

Recap: x86-64 Procedure Summary

- Important Points
  - Stack is the right data structure for procedure call/return
    - If P calls Q, then Q returns before P
- Recursion (& mutual recursion) handled by normal calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in `%rax`
- Pointers are addresses of values
  - On stack or global

Caller Frame

Arguments 7+

Return Addr

%rbp → (Optional) Old %rbp

Saved Registers + Local Variables

Argument Build

%rsp →

Plan for Today

- Arrays

- Structures

- Floating Point

Disclaimer: Slides for this lecture were borrowed from

—Randal E. Bryant and David R. O'Hallaroni's CMU 15-213 class

Lecture Plan

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
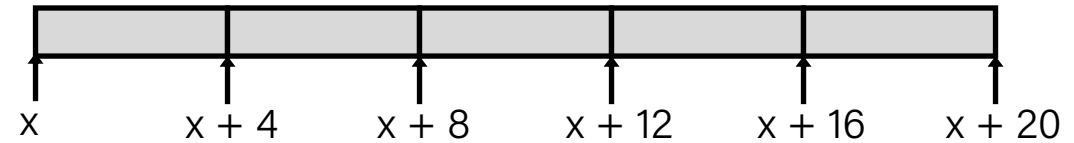- Floating Point

Array Allocation

Basic Principle

T A[L];

- Array of data type T and length L

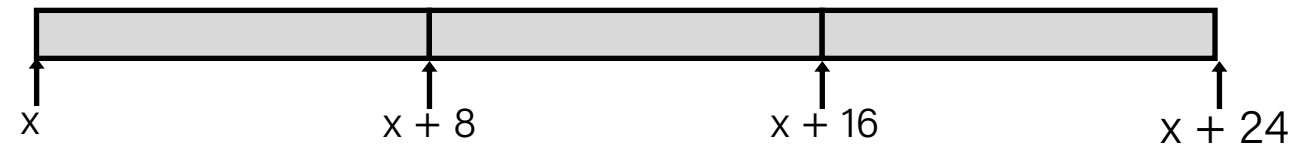- Contiguously allocated region of L*sizeof(T) bytes in memory

char string[12];

x   x + 12

int val[5];

x   x + 4   x + 8   x + 12   x + 16   x + 20

double a[3];

x   x + 8   x + 16   x + 24

char *p[3];

x   x + 8   x + 16   x + 24
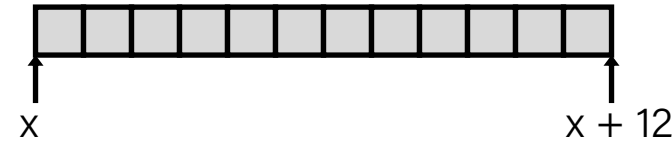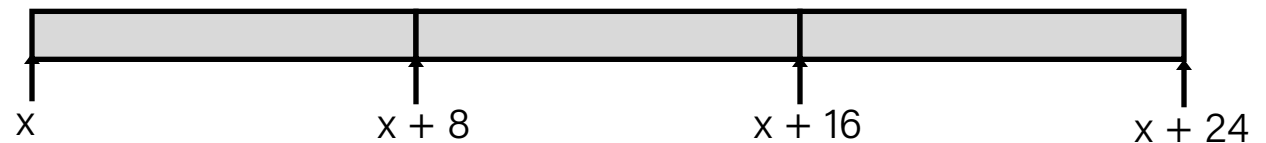
Array Access

- **Basic Principle**

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

  - Array of data type *T* and length *L*
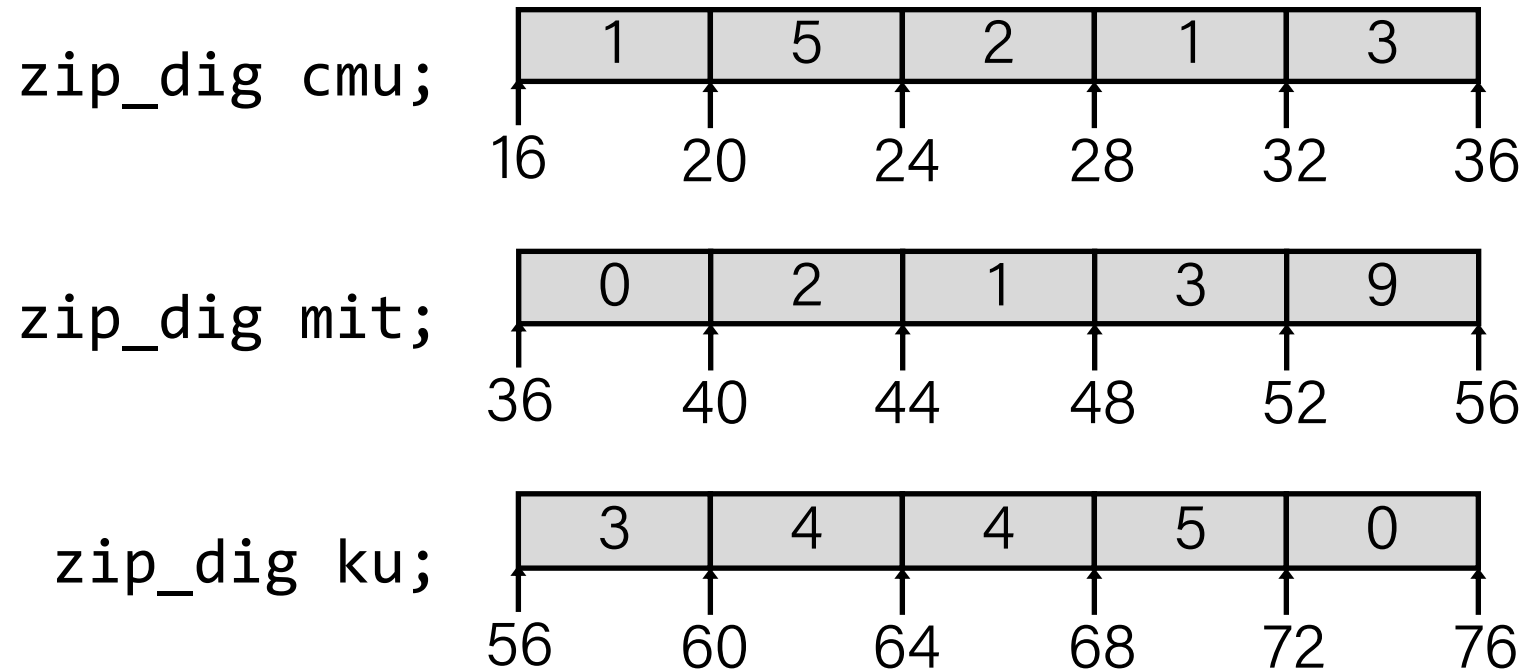 - Identifier **A** can be used as a pointer to array element 0: Type *T**

Reference	Type	Value
val[4]	int	3
val	int *	x
val+1	int *	x + 4
&val[2]	int *	x + 8
val[5]	int	??
*(val+1)	int	5
val + i	int *	x + 4 i

int val[5];

1	5	2	1	3

x   x + 4   x + 8   x + 12   x + 16   x + 20
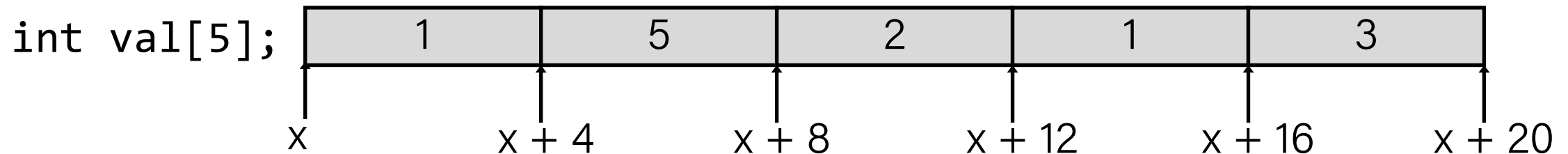
Array Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = {1,5,2,1,3};
zip_dig mit = {0,2,1,3,9};
zip_dig ku =  {3,4,4,5,0};
```

zip_dig cmu;

1	5	2	1	3

16   20   24   28   32   36

zip_dig mit;

0	2	1	3	9

36   40   44   48   52   56

zip_dig ku;

3	4	4	5	0

56   60   64   68   72   76

- Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

18

Array Accessing Example

```
int get_digit
  (zip_dig z, int digit)
{
  return z[digit];
}
```

```
zip_dig ku;
```

3	4	4	5	0

16   20   24   28   32   36

```
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

- Register **%rdi** contains starting address of array
- Register **%rsi** contains array index
- Desired digit at **%rdi + 4*%rsi**
- Use memory reference **(%rdi,%rsi,4)**

Array Loop Example

```c
void zincr(zip_dig z) {
  size_t i;
  for (i=0; i<ZLEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax           #   i = 0
  jmp     .L3                 #   goto middle
.L4:                         # loop:
  addl    $1, (%rdi,%rax,4)  #   z[i]++
  addq    $1, %rax           #   i++
.L3:                         # middle
  cmpq    $4, %rax           #   i:4
  jbe     .L4                #   if <=, goto loop
  rep; ret
```

Multidimensional (Nested) Arrays

Declaration

T `A[`R`][`C`];`

- 2D array of data type T
- R rows, C columns
- Type T element requires K bytes

Array Size

- $R * C * K$ bytes

Arrangement

- Row-Major Ordering

$$\begin{bmatrix} A[0][0] & \cdots & A[0][C-1] \\ & \vdots & \\ A[R-1][0] & \cdots & A[R-1][C-1] \end{bmatrix}$$

`int A[R][C];`

A [0] [0]	• • •	A [0] [C-1]	A [1] [0]	• • •	A [1] [C-1]	• • •	A [R-1] [0]	• • •	A [R-1] [C-1]

←——————————————— 4*R*C Bytes ———————————————→
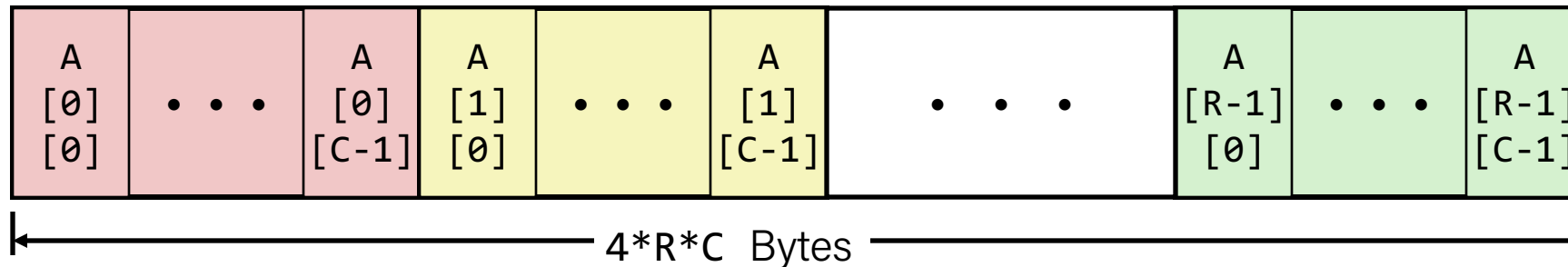
Nested Array Example

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

`zip_dig`
`pgh[4];`

1	5	2	0	6	1	5	2	1	3	1	5	2	1	7	1	5	2	2	1

76   96   116   136   156
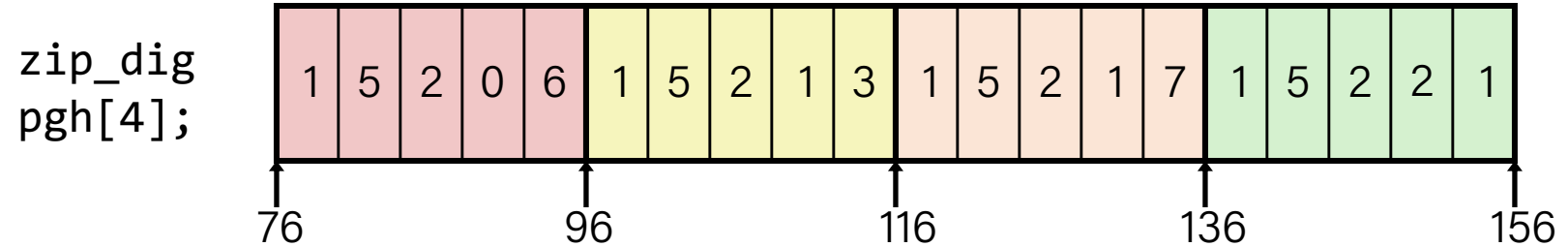
- "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  - Variable `pgh`: array of 4 elements, allocated contiguously
  - Each element is an array of 5 `int`'s, allocated contiguously
- "Row-Major" ordering of all elements in memory

Nested Array Row Access

Row Vectors

- **A[i]** is array of *C* elements

- Each element of type *T* requires *K* bytes

- Starting address: **A +** $i * (C * K)$

```
int A[R][C];
```
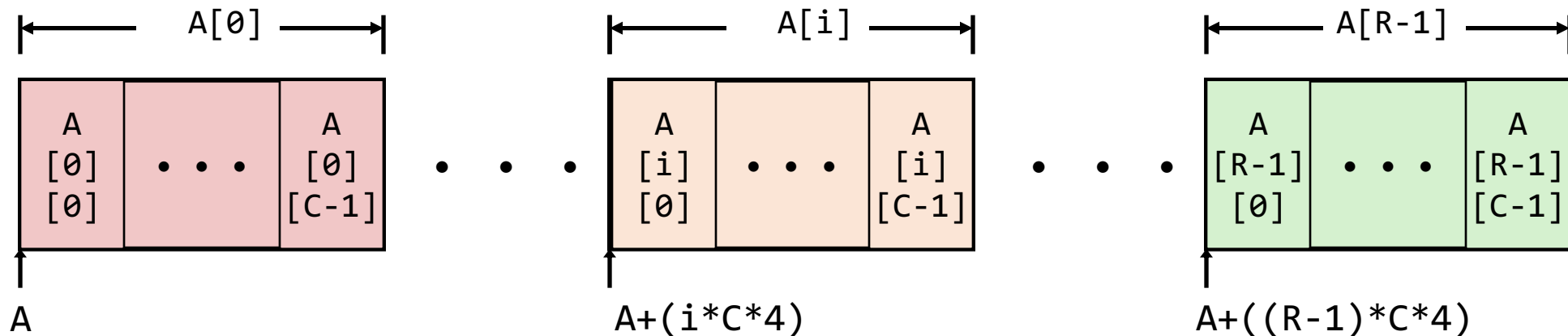
Nested Array Row Access Code
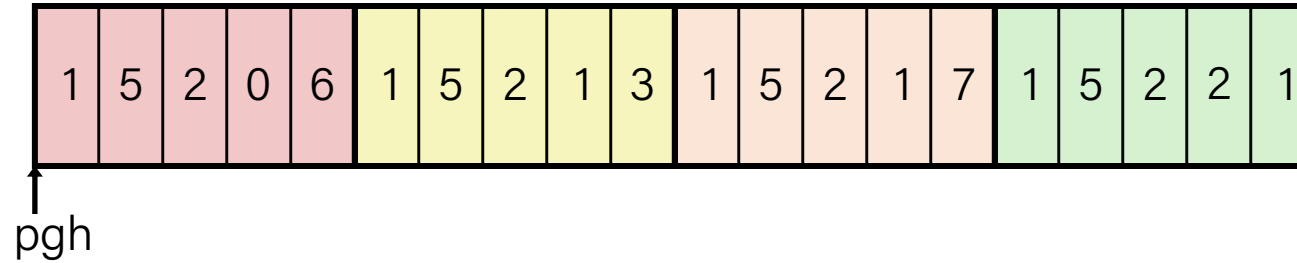

```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

```
# %rdi = index
leaq (%rdi,%rdi,4),%rax    # 5 * index
leaq pgh(,%rax,4),%rax     # pgh + (20 * index)
```

Row Vector
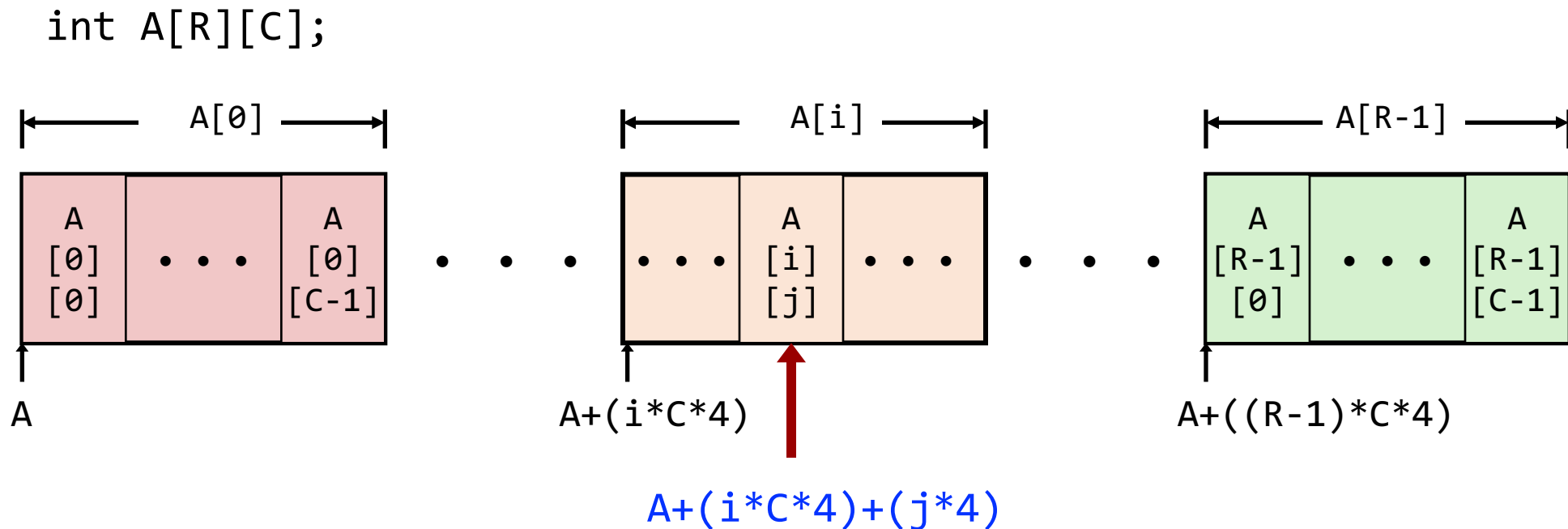- `pgh[index]` is array of 5 `int`'s
- Starting address `pgh+20*index`

Machine Code
- Computes and returns address
- Compute as
  `pgh+4*(index+4*index)`

Nested Array Element Access

Array Elements

- `A[i][j]` is element of type *T*, which requires *K* bytes
- Address $A + i * (C * K) + j * K = A + (i * C + j) * K$

```
int A[R][C];
```


$$A+(i*C*4)+(j*4)$$

Nested Array Element Access Code

1	5	2	0	6	1	5	2	1	3	1	5	2	1	7	1	5	2	2	1

pgh
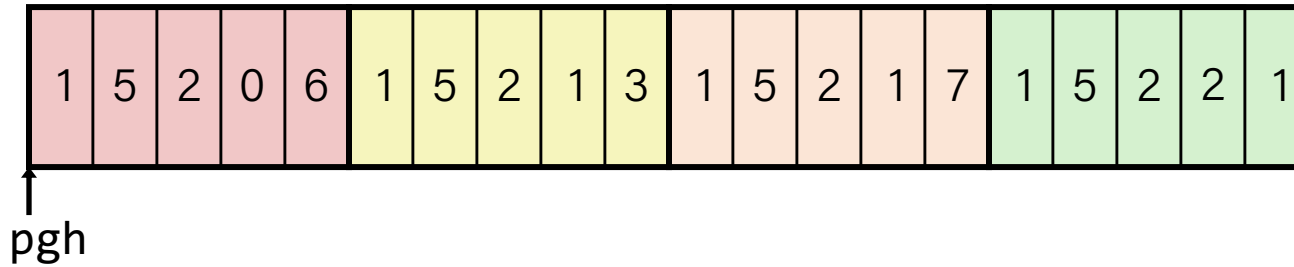
```
int get_pgh_digit
  (int index, int dig)
{
  return pgh[index][dig];
}
```

```
leaq  (%rdi,%rdi,4), %rax   # 5*index
addl  %rax, %rsi            # 5*index+dig
movl  pgh(,%rsi,4), %eax    # M[pgh + 4*(5*index+dig)]
```

Array Elements

- `pgh[index][dig]` is `int`
- Address: `pgh + 20*index + 4*dig`
  `= pgh + 4*(5*index + dig)`

Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ku  = { 3, 4, 4, 5, 0 };
```
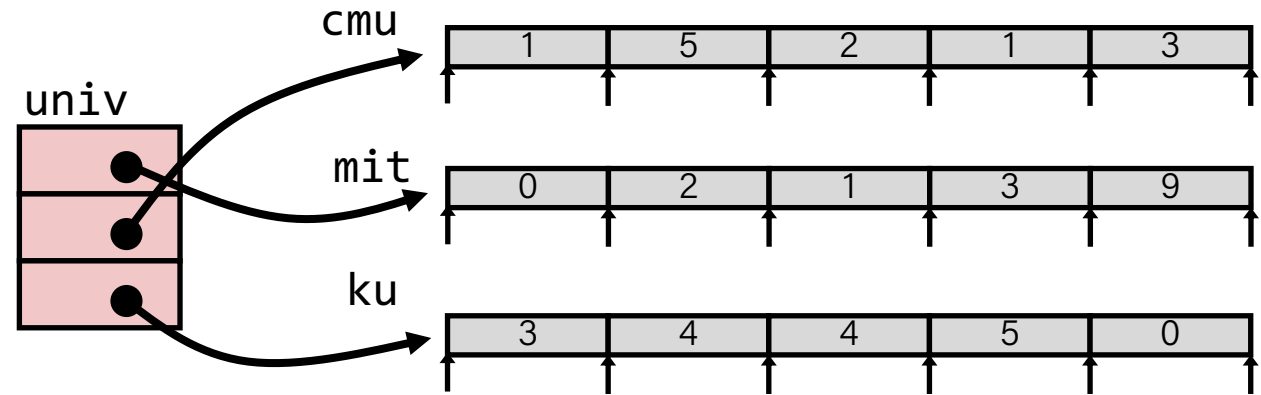
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ku};
```

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
    - 8 bytes
- Each pointer points to array of `int`'s
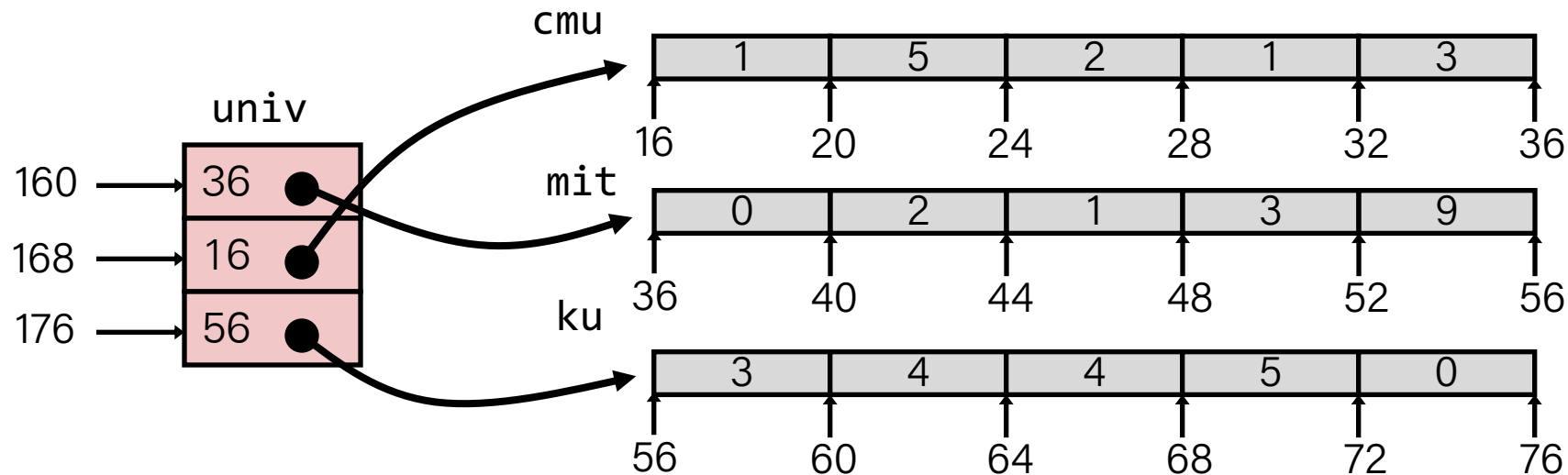
Element Access in Multi-Level Array

```
int get_univ_digit
  (size_t index, size_t digit)
{
  return univ[index][digit];
}
```



```
salq    $2, %rsi              # 4*digit
addq    univ(,%rdi,8), %rsi  # p = univ[index] + 4*digit
movl    (%rsi), %eax         # return *p
ret
```
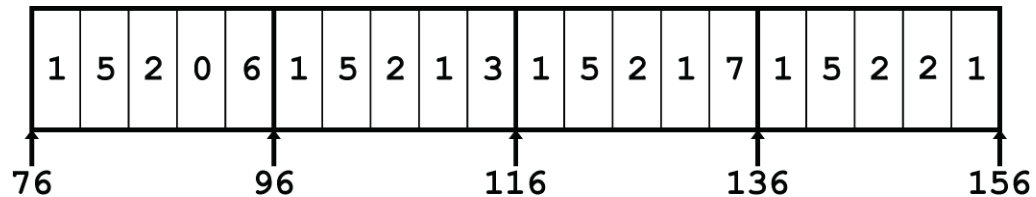
Computation

- Element access `Mem[Mem[univ+8*index]+4*digit]`
- Must do two memory reads
  - First get pointer to row array
  - Then access element within array
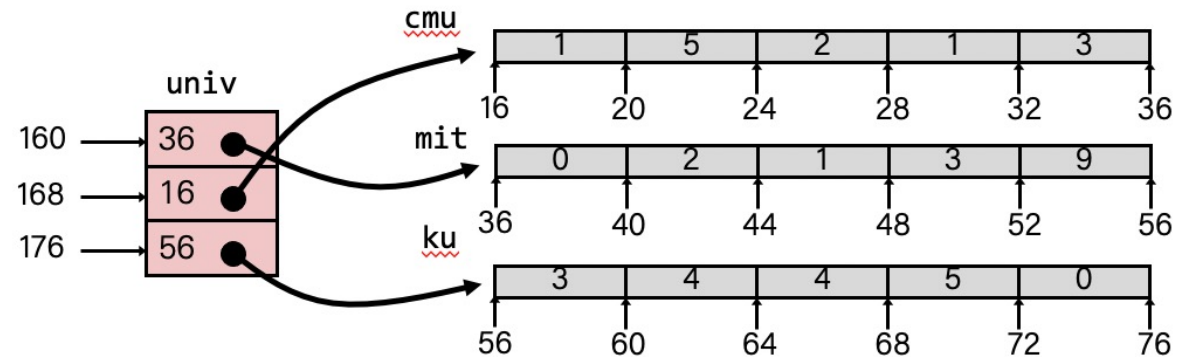
Array Element Accesses

Nested array

```
int get_pgh_digit
   (size_t index, size_t digit)
{
   return pgh[index][digit];
}
```


Multi-level array

```
int get_univ_digit
   (size_t index, size_t digit)
{
   return univ[index][digit];
}
```


- <mark>Accesses looks similar in C, but address computations very different:</mark>

```
Mem[pgh+20*index+4*digit]            Mem[Mem[univ+8*index]+4*digit]
```

N × N Matrix Code

Fixed dimensions

- Know value of N at compile time

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele(fix_matrix a,
            size_t i, size_t j) {
  return a[i][j];
}
```

Variable dimensions, explicit indexing

- Traditional way to implement dynamic arrays

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele(size_t n, int *a,
            size_t i, size_t j) {
  return a[IDX(n,i,j)];
}
```

Variable dimensions, implicit indexing

- Now supported by gcc

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n],
            size_t i, size_t j) {
  return a[i][j];
}
```

16 × 16 Matrix Access

```c
/* Get element a[i][j] */
int fix_ele(fix_matrix a, size_t i, size_t j) {
  return a[i][j];
}
```

```
# a in %rdi, i in %rsi, j in %rdx
salq    $6, %rsi              # 64*i
addq    %rsi, %rdi           # a + 64*i
movl    (%rdi,%rdx,4), %eax  # M[a + 64*i + 4*j]
ret
```

Array Elements
- Address A $+ i * (C * K) + j * K$
- $C = 16, K = 4$

n × n Matrix Access

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n], size_t i, size_t j) {
  return a[i][j];
}
```

```
# n in %rdi, a in %rsi, i in %rdx, j in %rcx
imulq   %rdx, %rdi           # n*i
leaq    (%rsi,%rdi,4), %rax  # a + 4*n*i
movl    (%rax,%rcx,4), %eax  # a + 4*n*i + 4*j
ret
```

Array Elements

- Address $A + i * (C * K) + j * K$
- $C = 16, K = 4$
- Must perform integer multiplication

Practice 1: Reverse Engineering

```c
#define M ??
#define N ??

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

```
# long sum_elem(long i, long j)
# i in %rdi, j in %rsi
1 sum_element:
2    leaq   0(,%rdi,8), %rdx        Compute 8*i
3    subq   %rdi, %rdx              Compute 7*i
4    addq   %rsi, %rdx              Compute 7*i+j
5    leaq   (%rsi,%rsi,4), %rax     Compute 5*j
6    addq   %rax, %rdi              Compute i+5*j
7    movq   Q(,%rdi,8), %rax        Retrieve M[Q+8*(5*j+i)]
8    add    P(,%rdx,8), %rax        Add M[P+8*(7*i+j)]
9    ret
```

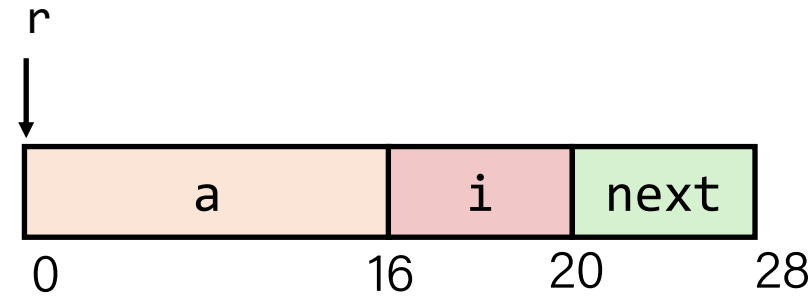**What is the value of M and N?**
M = 5 and N = 7

Lecture Plan

- Arrays

- Structures
  - Allocation
  - Access
  - Alignment
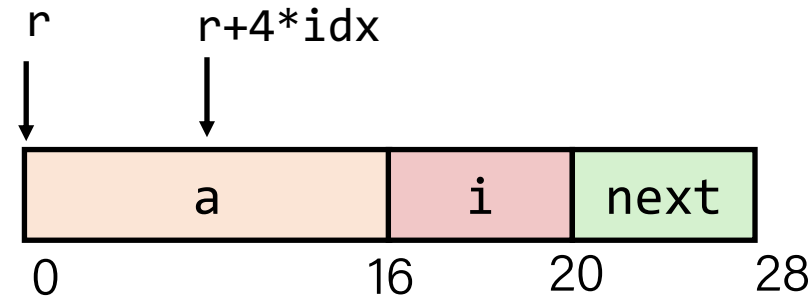
- Floating Point

Structure Representation

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```

r

	a		i	next
0		16	20	28

- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```


Generating Pointer to Array Element

- Offset of each structure member determined at compile time
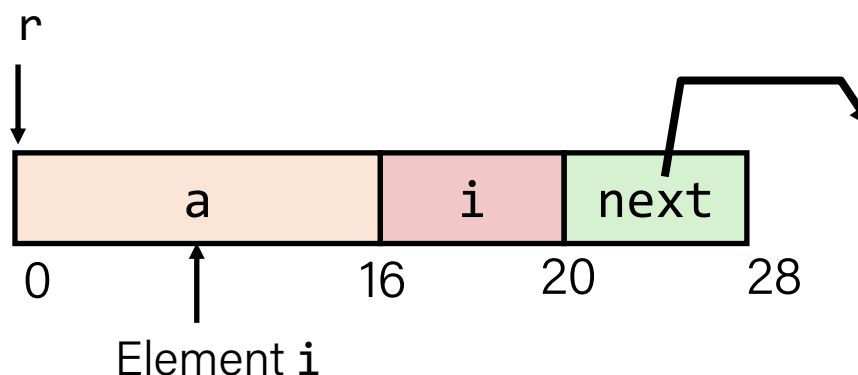
- Compute as **r + 4*idx**

```
int *get_ap
 (struct rec *r, size_t idx)
{
    return &r->a[idx];
}
```

```
# r in %rdi, idx in %rsi
leaq  (%rdi,%rsi,4), %rax
ret
```

Following Linked List

```c
struct rec {
    int a[4];
    int i;
    struct rec *next;
};

void set_val (struct rec *r, int val) {
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```


r

| a | i | next |

0   16   20   28

Element i

Register	Value
%rdi	r
%esi	val

```
.L11:                              # loop:
  movslq  16(%rdi), %rax           #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)      #   M[r+4*i] = val
  movq    20(%rdi), %rdi           #   r = M[r+20]
  testq   %rdi, %rdi               #   Test r
  jne     .L11                     #   if !=0 goto loop
```
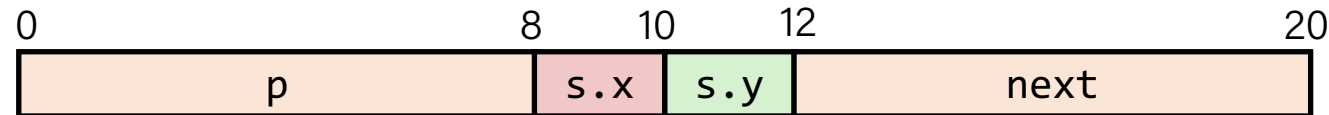
Practice 2: Reverse Engineering

Fill in the blanks by inspecting the assembly code generated by gcc.

```
struct test {
    short *p;
    struct {
        short x;
        short y;
    } s;
    struct test *next;
};

void st_init(struct test *st) {
    st->s.y  = st->s.x   ;
    st->p    = &(st->s.y);
    st->next = st        ;
}
```
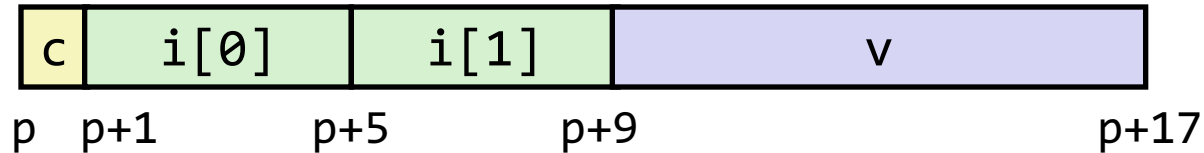
```
# void st_init(struct test *st)
# st in %rdi
1 st_init:
2     movl 8(%rdi), %eax     Get st->s.x
3     movl %eax, 10(%rdi)    Save in st->s.y
4     leaq 10(%rdi), %rax    Compute &(st->s.y)
5     movq %rax, (%rdi)      Store in st->p
6     movq %rdi, 12(%rdi)    Store st in st->next
7     ret
```
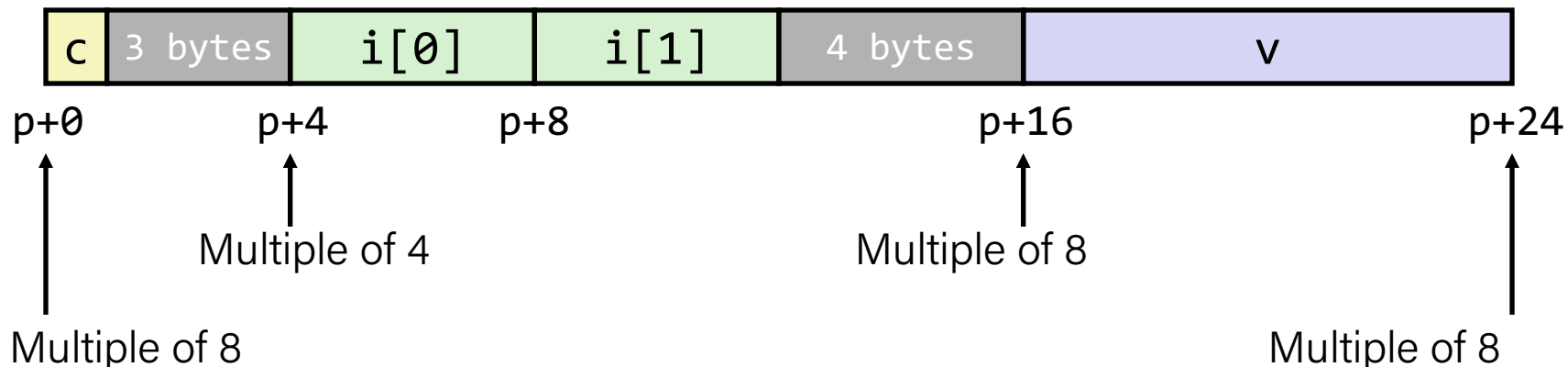
0		8	10	12		20
	p		s.x	s.y	next	

Structures & Alignment

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

Unaligned Data

| c | i[0] | i[1] | v |

p   p+1   p+5   p+9   p+17

Aligned Data

- Primitive data type requires K bytes

- Address must be multiple of K

| c | 3 bytes | i[0] | i[1] | 4 bytes | v |

p+0   p+4   p+8   p+16   p+24

Multiple of 4   Multiple of 8

Multiple of 8   Multiple of 8

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:

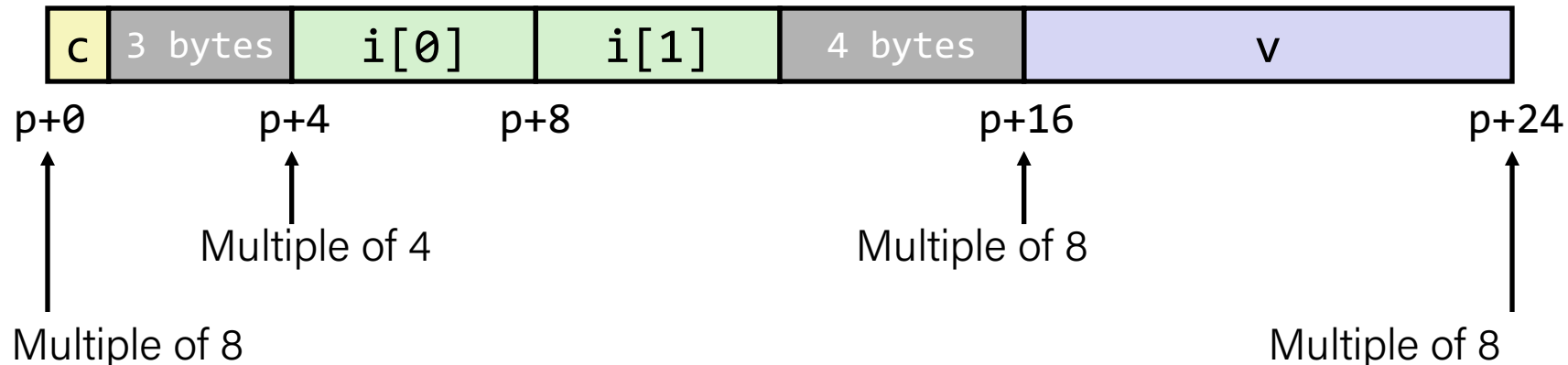- Must satisfy each element's alignment requirement

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;
```

| c | 3 bytes | i[0] | i[1] | 4 bytes | v |

p+0   p+4   p+8   p+16   p+24

Multiple of 8 (p+0)

Multiple of 4 (p+4)

Multiple of 8 (p+16)

Multiple of 8 (p+24)
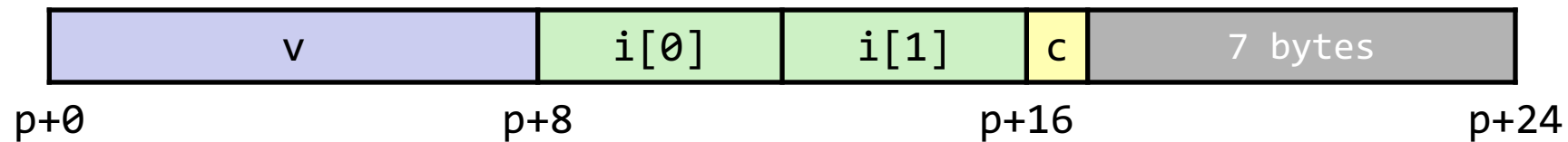
Meeting Overall Alignment Requirement

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

```
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```
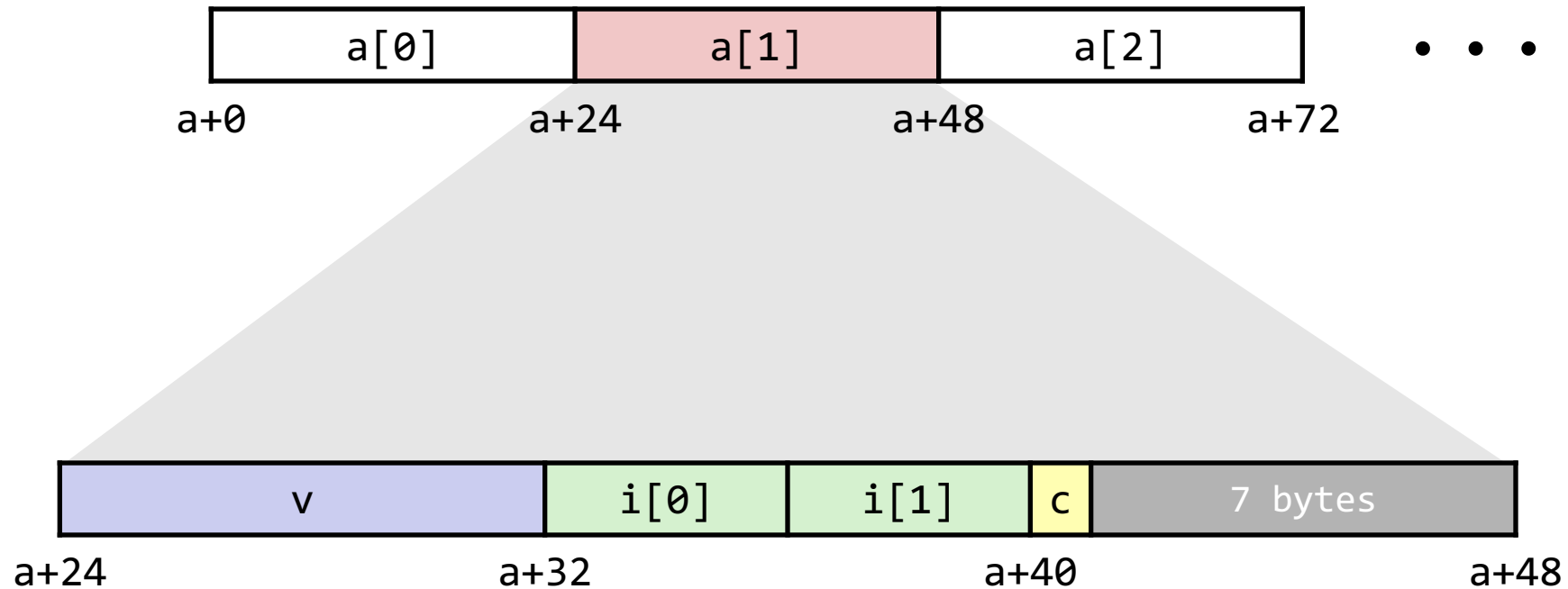
v	i[0]	i[1]	c	7 bytes

p+0   p+8   p+16   p+24

Multiple of K=8

Arrays of Structures

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

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

Accessing Array Elements

```
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```

- Compute array offset 12*idx
  - `sizeof(S3)`, including alignment spacers
- Element `j` is at offset 8 within structure
- Assembler gives offset `a+8` (resolved during linking)

a[0]	• • •	a[idx]	• • •

a+0   a+12   a+12*idx

i	2 bytes	v	j	2 bytes

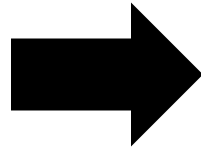a+12*idx   a+12*idx+8
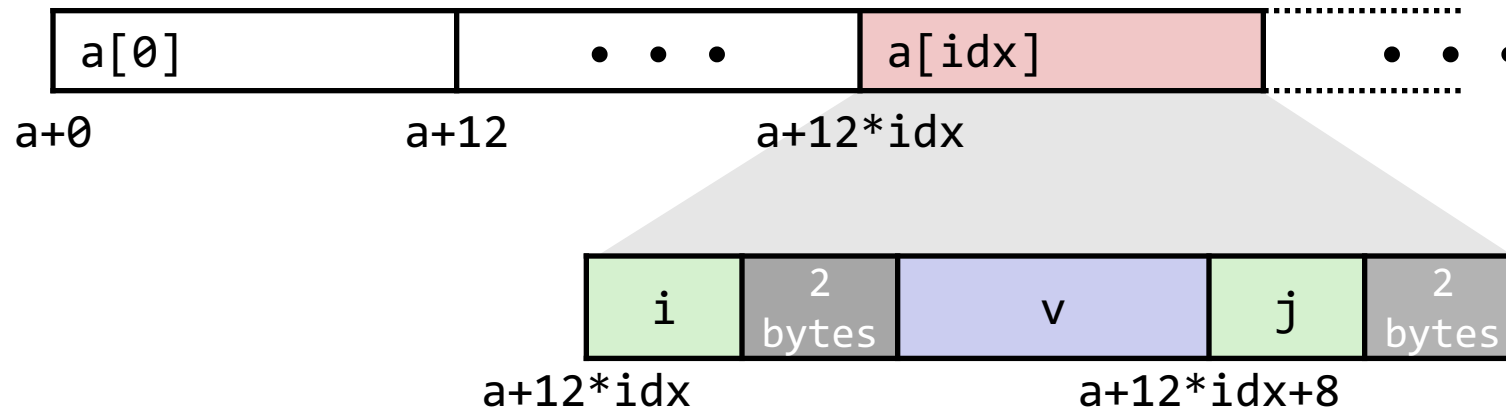
```
short get_j(int idx) {
    return a[idx].j;
}
```

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

Saving Space
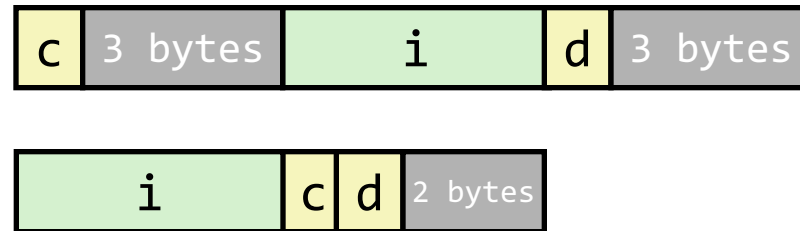
- Put large data types first

```
struct S4 {
    char c;
    int i;
    char d;
} *p;
```

➡️

```
struct S5 {
    int i;
    char c;
    char d;
} *p;
```

- Effect (K=4)

c	3 bytes	i	d	3 bytes

i	c	d	2 bytes

Practice 3: Alignment

Determine the offset of each field, the total size of the structure, and its alignment requirement for x86-64.

```
struct mystruct {
    int *a;
    float b;
    char c;
    short d;
    float e;
    double f;
    int g;
    char *h;
};
```

Field	*a	b	c	d	e	f	g	*h	Total	Alignment
Size	8	4	1	2	4	8	4	8		
Offset	0	8	12	14	16	24	32	40	48	8

No extra padding needed to satisfy alignment requirement

Rearranged structure with minimum wasted space:

Field	*a	f	h	b	e	g	d	c	Total	Alignment
Size	8	8	8	4	4	4	2	1		
Offset	0	8	16	24	28	32	36	38	40	8

1 bytes padded to satisfy alignment requirement

Lecture Plan

- Arrays
- Structures
- Floating Point

Background

- History
  - x87 FP
    - Legacy, very ugly
  - Streaming SIMD Extensions (SSE) FP
    - SIMD: single instruction, multiple data
    - Special case use of vector instructions
  - AVX FP
    - Newest version
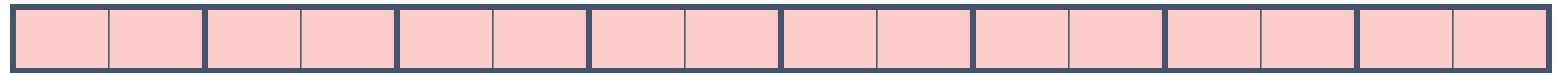   - Similar to SSE
    - Documented in book

Programming with SSE3

XMM Registers
- 16 total, each 16 bytes
- 16 single-byte integers

- 8 16-bit integers

- 4 32-bit integers

- 4 single-precision floats

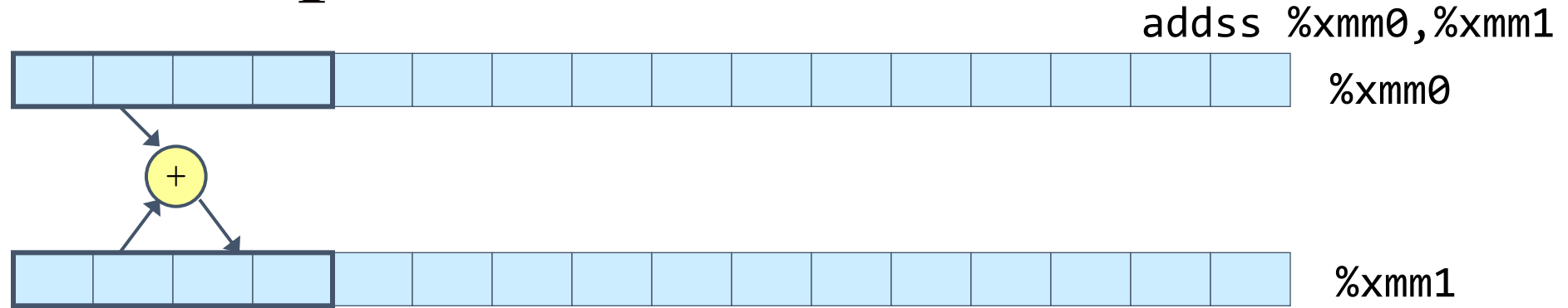- 2 double-precision floats

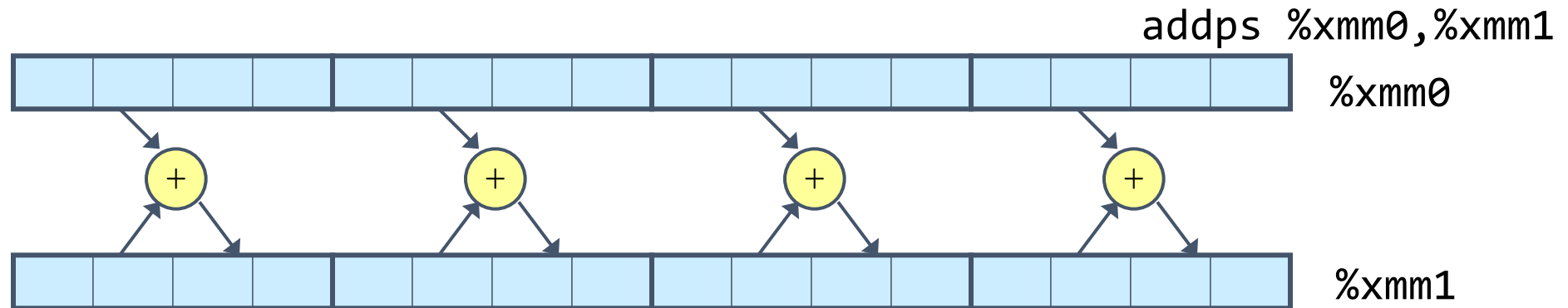- 1 single-precision float

- 1 double-precision float
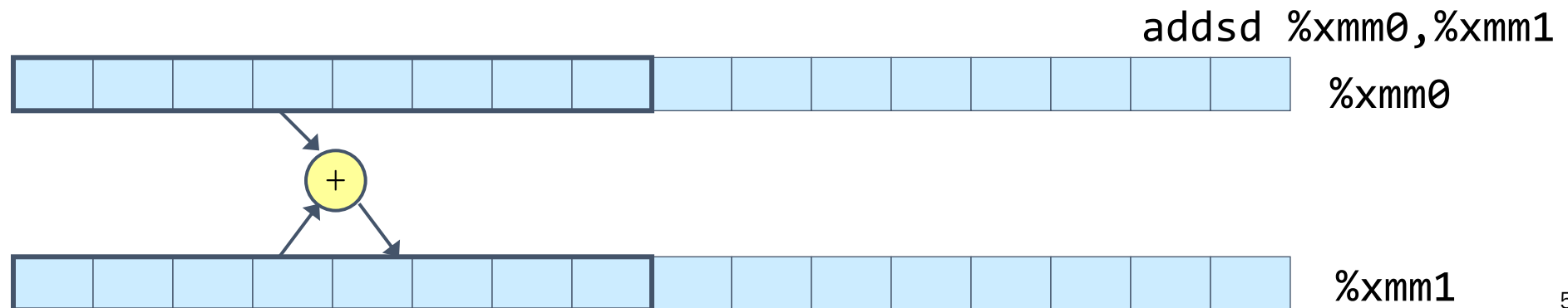
Scalar & SIMD Operations

- Scalar Operations: Single Precision

- SIMD Operations: Single Precision

- Scalar Operations: Double Precision

`addss %xmm0,%xmm1`

`%xmm0`

`%xmm1`

`addps %xmm0,%xmm1`

`%xmm0`

`%xmm1`

`addsd %xmm0,%xmm1`

`%xmm0`

`%xmm1`

FP Basics

- Arguments passed in **%xmm0**, **%xmm1**, …
- Result returned in **%xmm0**
- All XMM registers caller-saved

```
float fadd(float x, float y) {
    return x + y;
}
```

```
# x in %xmm0, y in %xmm1
addss   %xmm1, %xmm0
ret
```

```
double dadd(double x, double y) {
    return x + y;
}
```

```
# x in %xmm0, y in %xmm1
addsd   %xmm1, %xmm0
ret
```

FP Memory Referencing

- Integer (and pointer) arguments passed in regular registers

- FP values passed in XMM registers

- Different mov instructions to move between XMM registers, and between memory and XMM registers

```
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```

```
# p in %rdi, v in %xmm0
movapd  %xmm0, %xmm1    # Copy v
movsd   (%rdi), %xmm0   # x = *p
addsd   %xmm0, %xmm1    # t = x + v
movsd   %xmm1, (%rdi)   # *p = t
ret
```

Other Aspects of FP Code

- Lots of instructions
  - Different operations, different formats, …

- Floating-point comparisons
  - Instructions `ucomiss` and `ucomisd`
  - Set condition codes CF, ZF, and PF

- Using constant values
  - Set XMM0 register to 0 with instruction `xorpd %xmm0, %xmm0`
  - Others loaded from memory

Recap

- Arrays

- Structures

- Floating Point

That's it for assembly!

Next time: *security vulnerabilities, memory hierarchy*