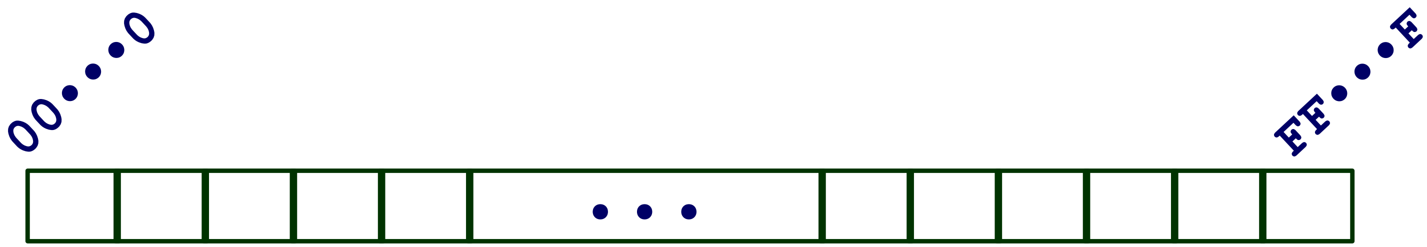
Machine-Level Programming IV: Data

CS2011: Introduction to Computer Systems
Lecture 9 (3.8, 3.9, 3.11)

Machine-Level Programming IV: Data

- **Partial recap: Integers**
  - Word size
  - Addresses
- **One-Dimensional Arrays**
- **Structs**
  - Alignment
  - Arrays of Structs
- **Multi-Dimensional Arrays**
  - Nested (Arrays of Arrays)
  - (Arrays of) Pointers to Arrays
- **Floating Point Machine Code**

Byte-Oriented Memory Organization



- **Programs refer to data by address**
  - Imagine all of RAM as an enormous array of bytes
  - An address is an index into that array
    - A pointer variable stores an address
- **System provides a private *address space* to each "process"**
  - A process is an instance of a program, being executed
  - An address space is one of those enormous arrays of bytes
  - Each program can see only its own code and data within its enormous array
  - We'll come back to this later ("virtual memory")

Machine Words

Any given computer has a "Word Size"

- Nominal size of integer-valued data
  - and of addresses

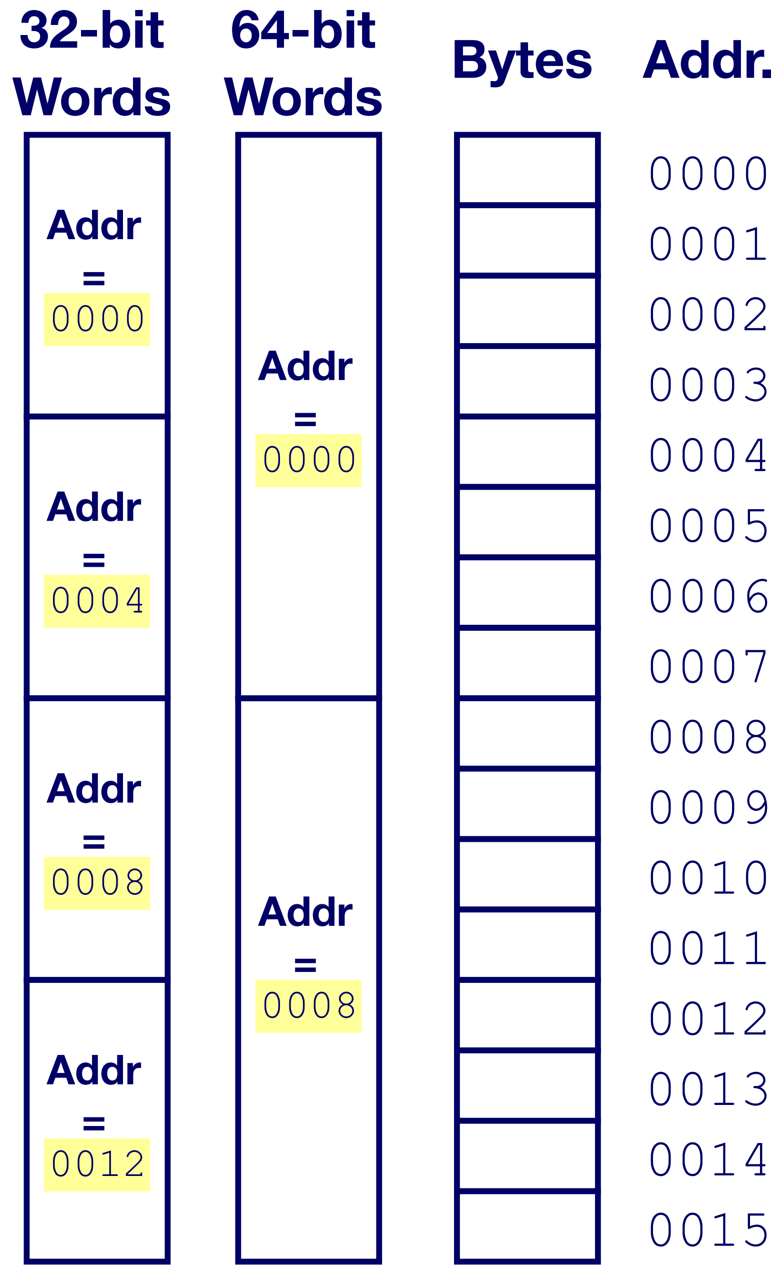
- Until recently, most machines used 32 bits (4 bytes) as word size
  - Limits addresses to 4GB (2^{32} bytes)

- Increasingly, machines have 64-bit word size
  - Potentially, could have 16 EB (exabytes) of addressable memory
  - That's 18.4×10^{18} bytes

- Machines still support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes

Addresses *Always* Specify Byte Locations

- Address of a word is address of the first byte in the word
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



Machine-Level Programming IV: Data

- Partial recap: Integers
  - Word size
  - Addresses
- **One-Dimensional Arrays**
- Structs
  - Alignment
  - Arrays of Structs
- Multi-Dimensional Arrays
  - Nested (Arrays of Arrays)
  - (Arrays of) Pointers to Arrays
- Floating Point Machine Code

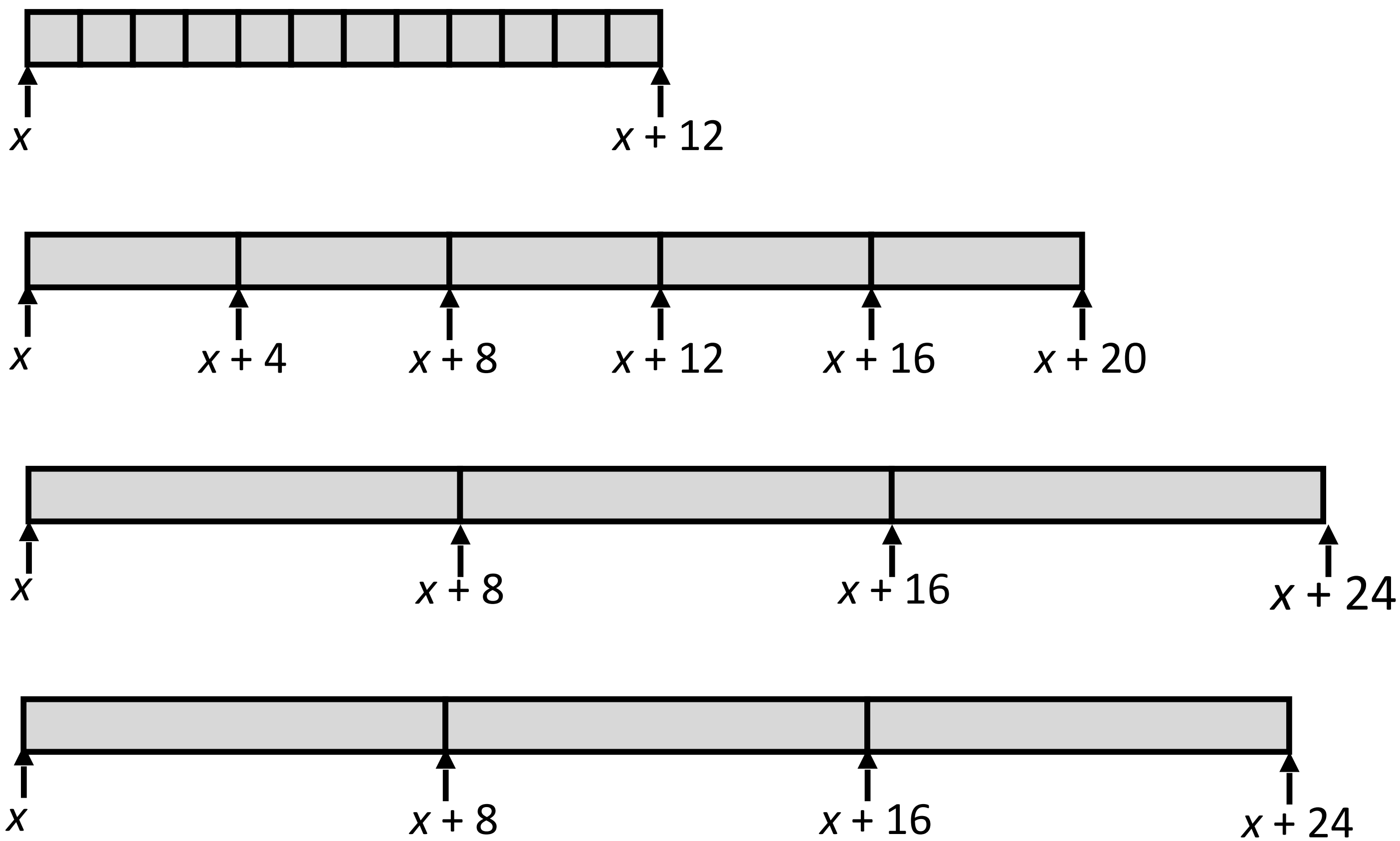
Array Allocation

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];
int val[5];
double a[3];
char *p[3];
```



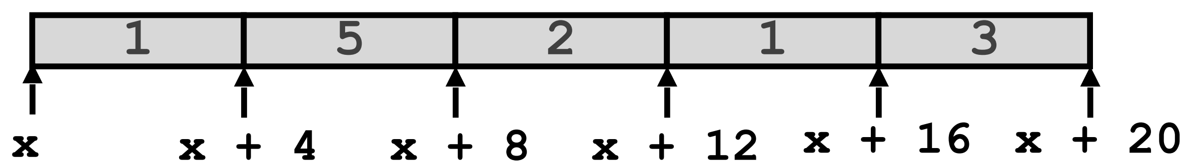
Array Access

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**

```
int val[5];
```

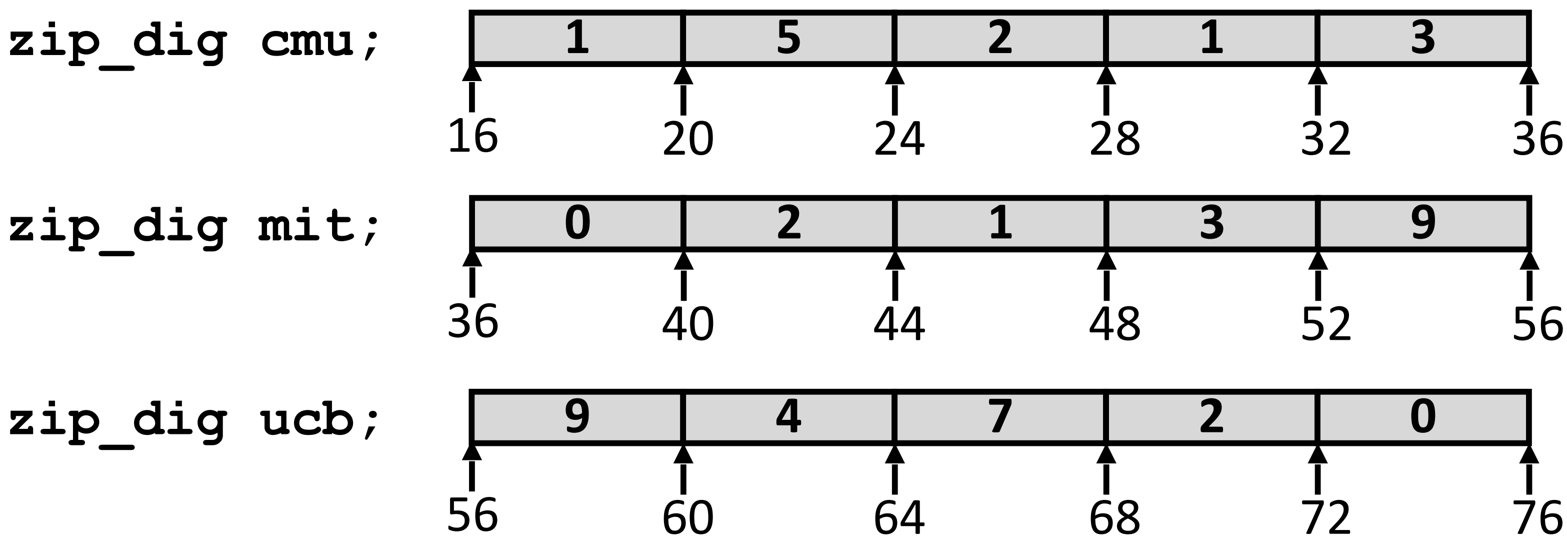


Reference Type Value

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[1]`
`val + i`	`int *`	`x + 4 * i`	`//&val[i]`

Array Example

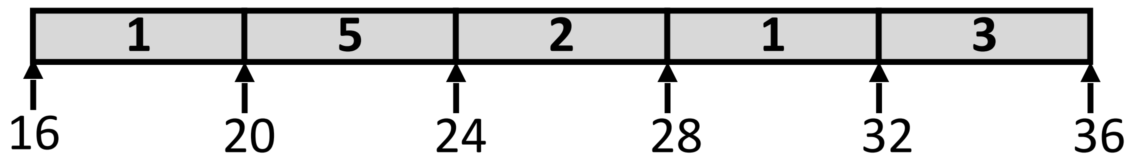
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



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

Array Accessing Example

```
zip_dig cmu;
```




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

x86-64

```
  # %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

```
  # %rdi = z
  movl     $0, %eax
  jmp      .L3
.L4:
  addl     $1, (%rdi,%rax,4)
  addq     $1, %rax
.L3:
  cmpq     $4, %rax
  jbe      .L4
  rep; ret
```

Array Loop Example

```
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
```

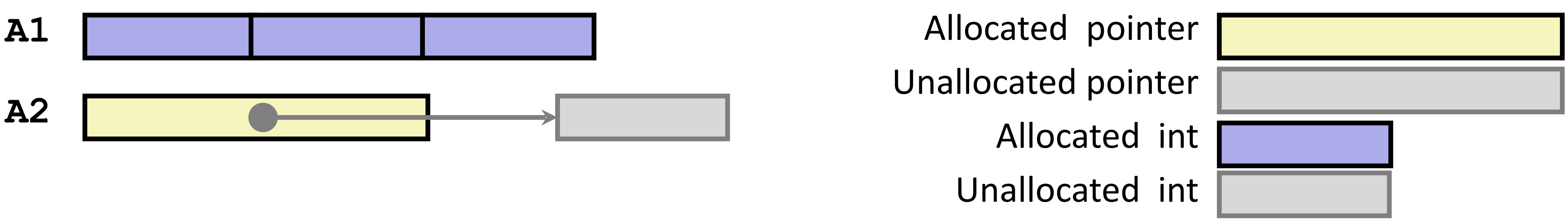
Understanding Pointers & Arrays #1

Decl	An			*An		
	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`						
`int *A2`						

- Cmp: Compiles (Y/N)
- Bad: Possible bad pointer reference (Y/N)
- Size: Value returned by `sizeof`

Understanding Pointers & Arrays #1

Decl	An			*An		
	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4
`int *A2`	Y	N	8	Y	Y	4



- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

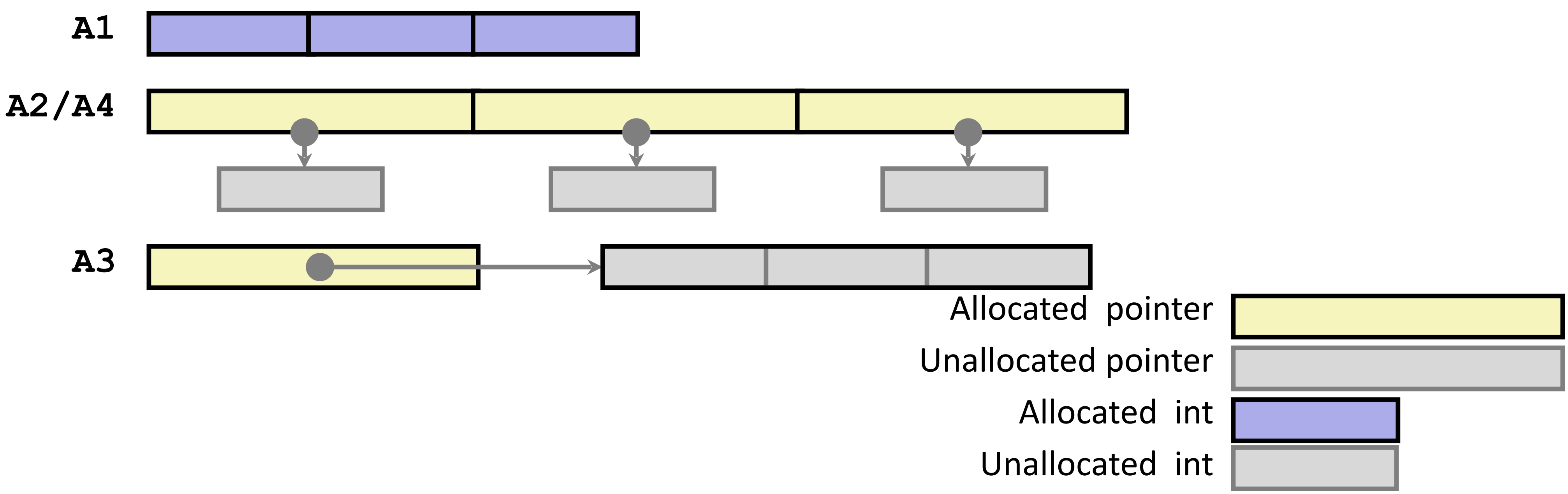
Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`									
`int *A2[3]`									
`int (*A3)[3]`									
`int (*A4[3])`									

- Cmp: Compiles (Y/N)
- Bad: Possible bad pointer reference (Y/N)
- Size: Value returned by `sizeof`

Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4	N	-	-
`int *A2[3]`	Y	N	24	Y	N	8	Y	Y	4
`int (*A3)[3]`	Y	N	8	Y	Y	12	Y	Y	4
`int (*A4[3])`	Y	N	24	Y	N	8	Y	Y	4

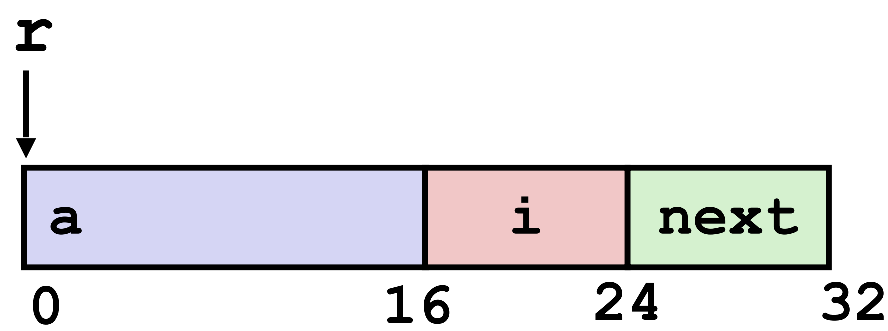


Machine-Level Programming IV: Data

- Partial recap: Integers
  - Word size
  - Addresses
- One-Dimensional Arrays
- **Structs**
  - Alignment
  - Arrays of Structs
- Multi-Dimensional Arrays
  - Nested (Arrays of Arrays)
  - (Arrays of) Pointers to Arrays
- Floating Point Machine Code

Structure Representation

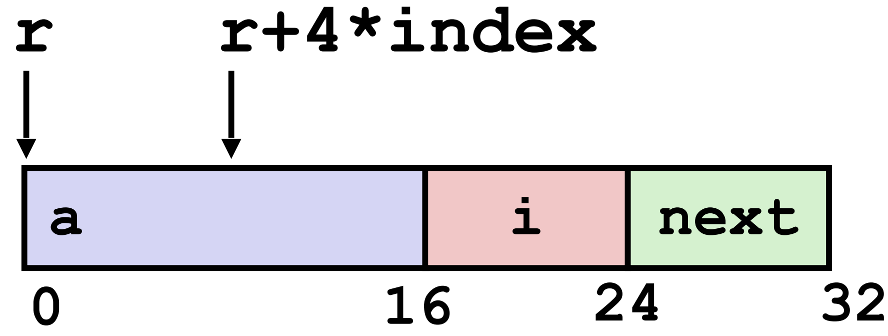
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields **ordered** according to declaration (***always in order***)
  - 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];
    size_t i;
    struct rec *next;
};
```



- **Generating Pointer to Array Element**
  - **Offset** of each structure member **determined at compile time**
  - Compute as `r + 4*index`

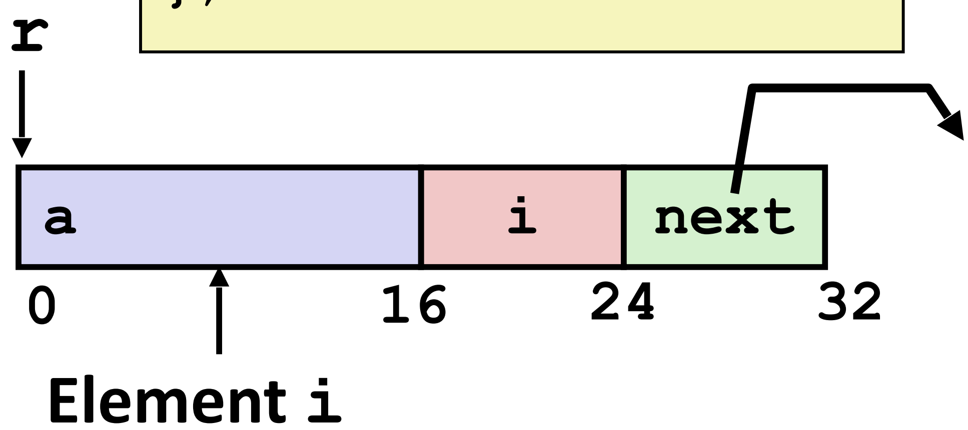
```
int *get_ap
 (struct rec *r, size_t index)
{
  return &r->a[index];
}
```

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

Following Linked List

- C Code

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



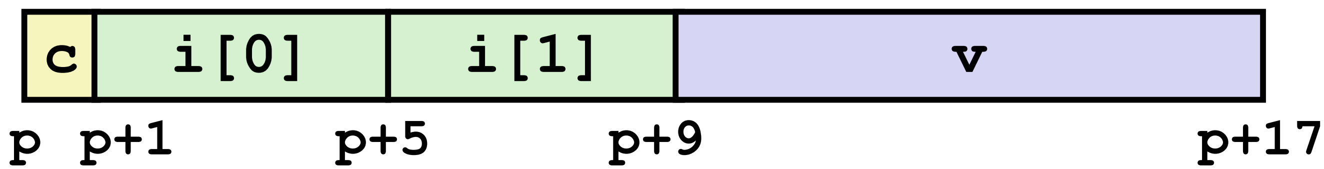

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

Register	Value
%rdi	r
%rsi	val

```
.L11:                              # loop:
  movslq  16(%rdi), %rax           #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)      #   M[r+4*i] = val
  movq    24(%rdi), %rdi           #   r = M[r+24]
  testq   %rdi, %rdi               #   Test r
  jne     .L11                     #   if !=0 goto loop
```

Structures & Alignment

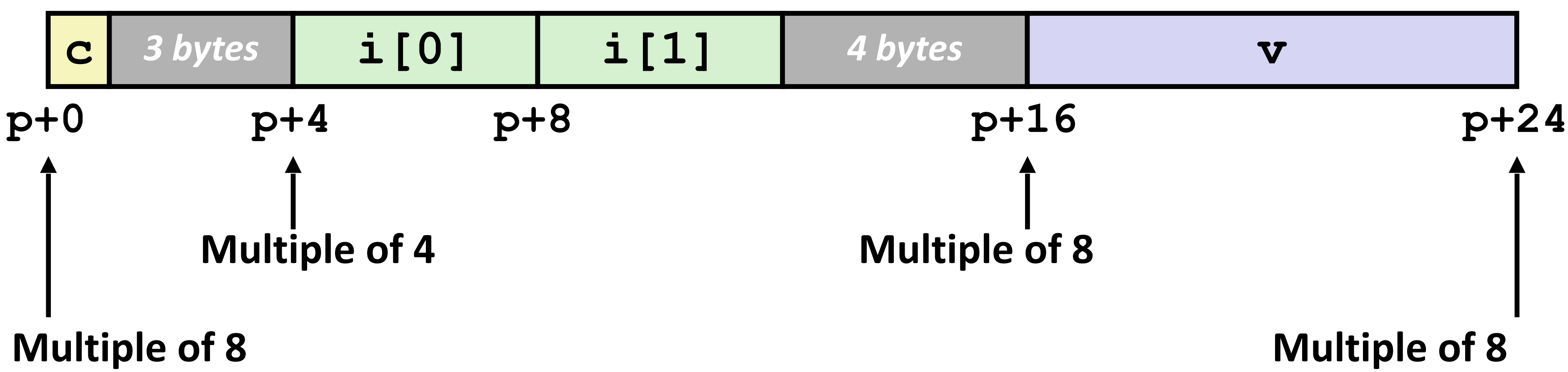
Unaligned Data




```
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
    - E.g., a *double* value that is *saved at an address that is not a multiple of 8* will *require 2 memory accesses* to be read instead of a single read if it was saved at an address that is a multiple of 8.
  - *Virtual memory trickier* when datum spans 2 pages
  - Alignment simplifies the design of the hardware

Compiler

- Inserts gaps in structure to ensure correct alignment of fields

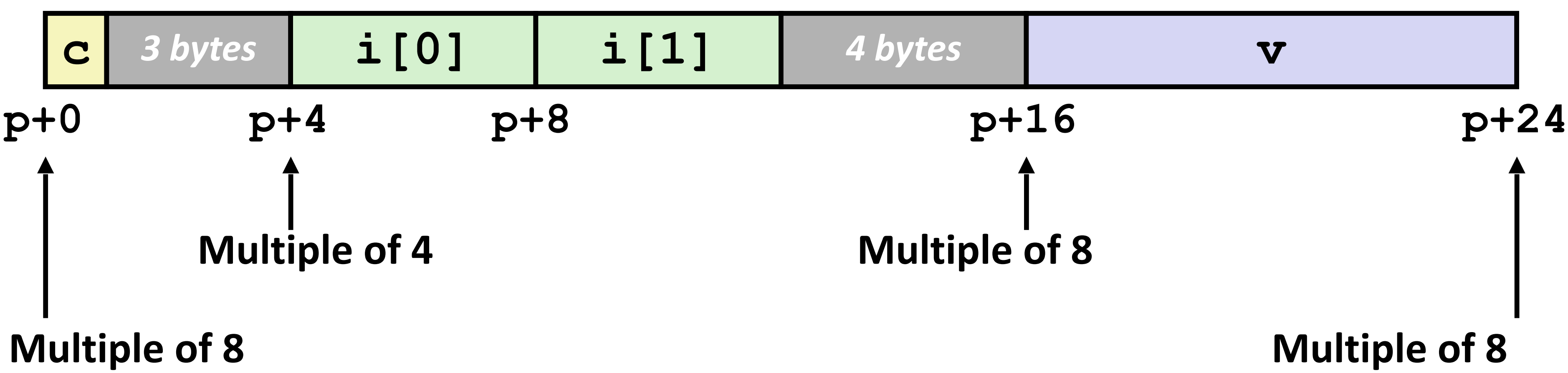
Specific Cases of Alignment (x86-64)

- **1 byte: `char`, …**
  - K = 1 —> Address must be a multiple of 1
  - no restrictions on address
- **2 bytes: `short`, …**
  - K = 2 —> Address must be a multiple of 2
  - lowest 1 bit of address must be 0_2
- **4 bytes: `int`, `float`, …**
  - K = 4 —> Address must be a multiple of 4
  - lowest 2 bits of address must be 00_2
- **8 bytes: `double`, `long`, `char *`, …**
  - K = 8 —> Address must be a multiple of 8
  - lowest 3 bits of address must be 000_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

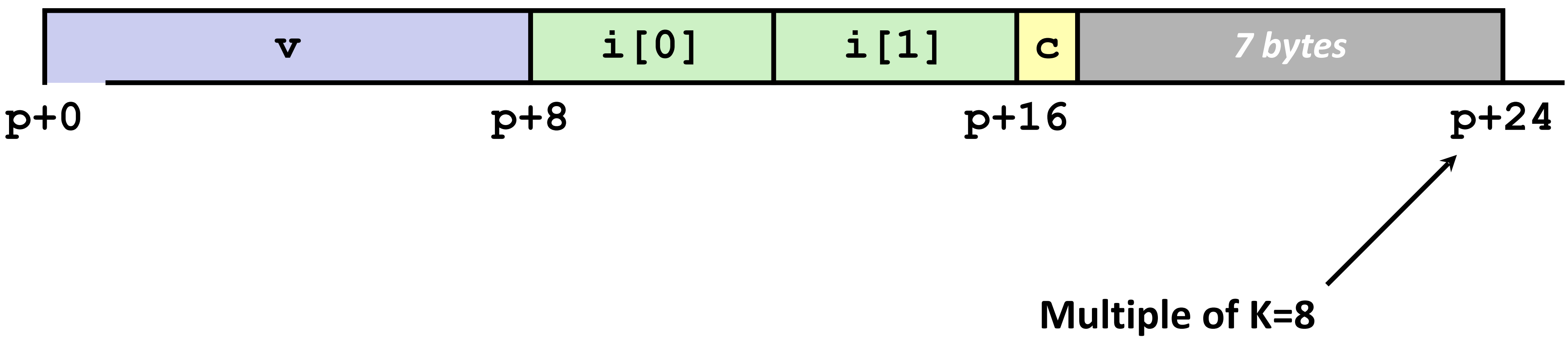
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Meeting Overall Alignment Requirement

- For largest alignment requirement K
- Overall structure must be multiple of K
- Add **padding** at the **end** if needed

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



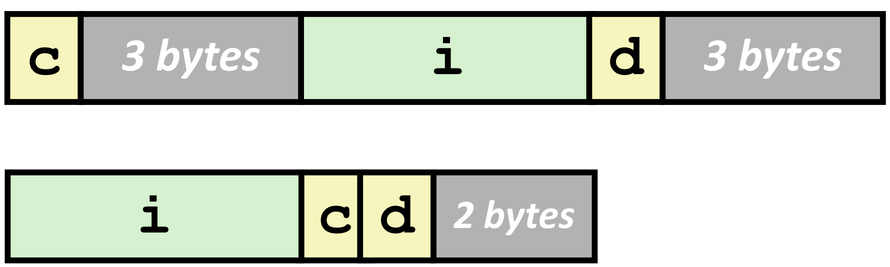
Saving Space

- **Put large data types first**

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

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

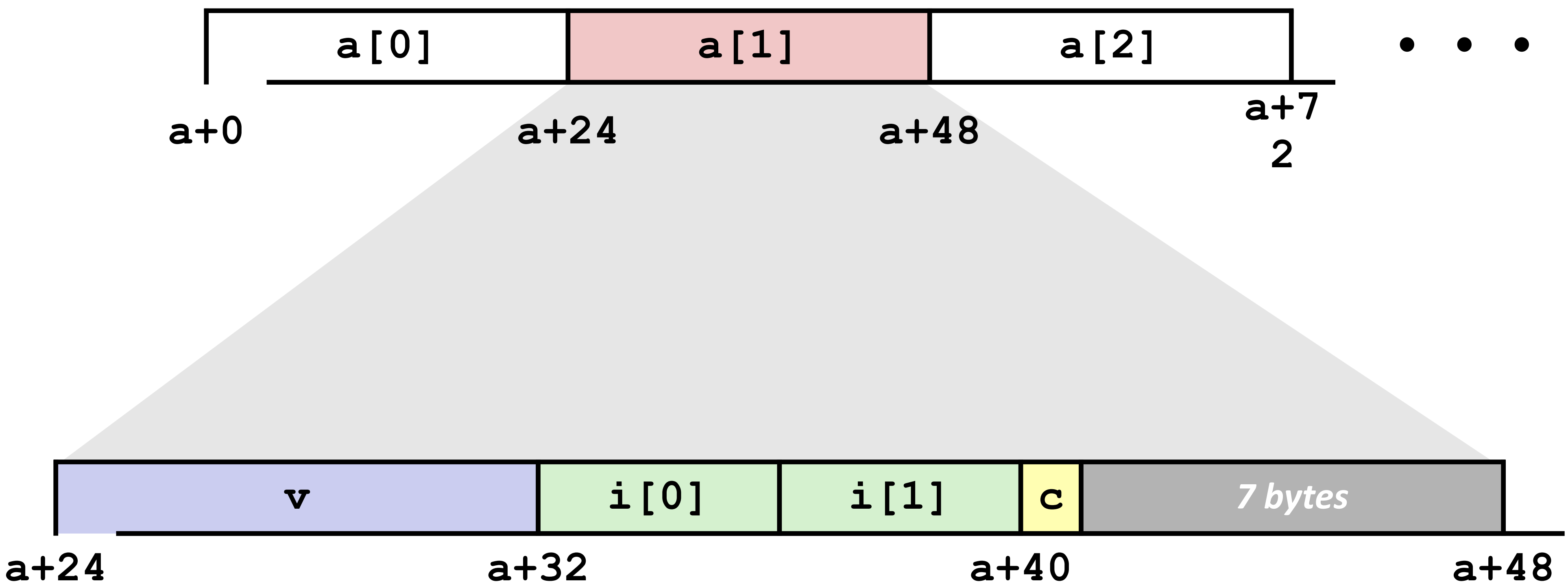
- **Effect (K=4)**



Arrays of Structures

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

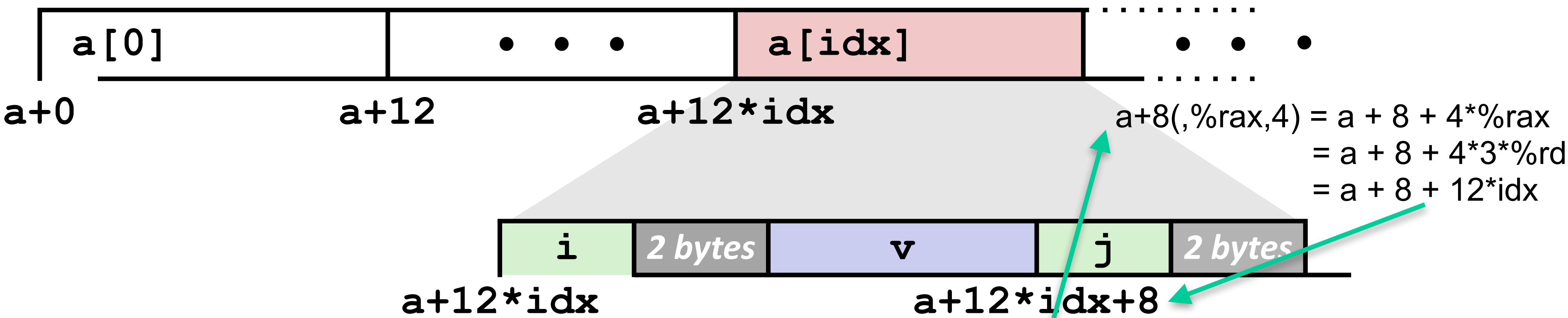
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Array of Structures (Accessing Elements Example)

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

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




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

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

Machine-Level Programming IV: Data

- Partial recap: Integers
  - Word size
  - Addresses
- One-Dimensional Arrays
- Structs
  - Alignment
  - Arrays of Structs
- **Multi-Dimensional Arrays**
  - Nested (Arrays of Arrays)
  - (Arrays of) Pointers to Arrays
- Floating Point Machine Code

Multidimensional (Nested) Arrays

Declaration

T **A**[*R*][*C*];

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

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

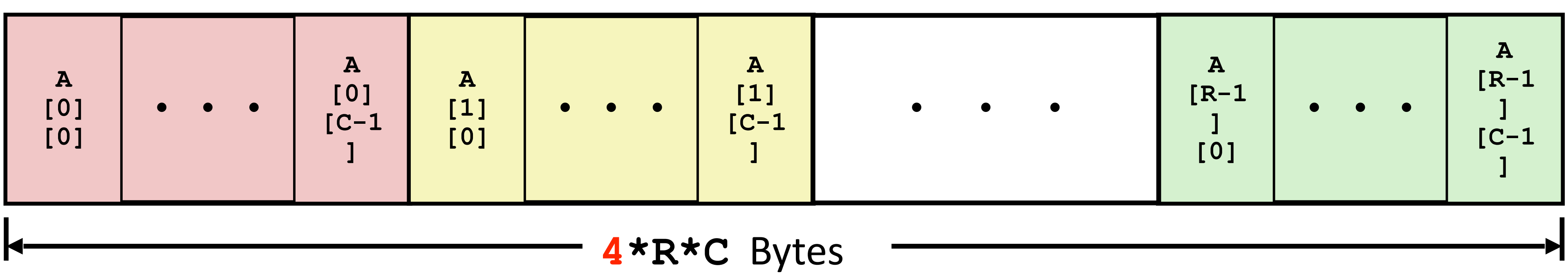
Array Size

- *R* * *C* * *K* bytes

Arrangement

- ***Row-Major Ordering****: lay out matrix* ***by row*** *in the one dimensional memory (common convention)*

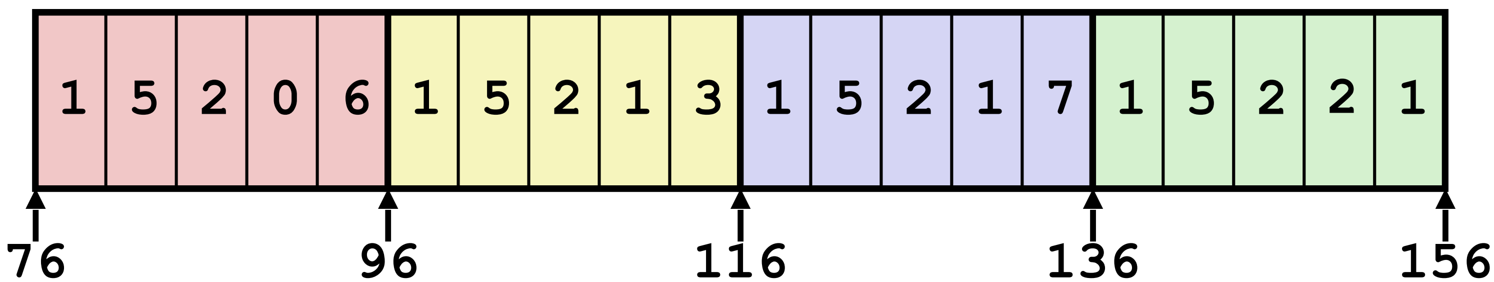
```
int A[R][C];
```



Nested Array Example

```
#define PCOUNT 4
#typedef int zip_dig[5];

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];`



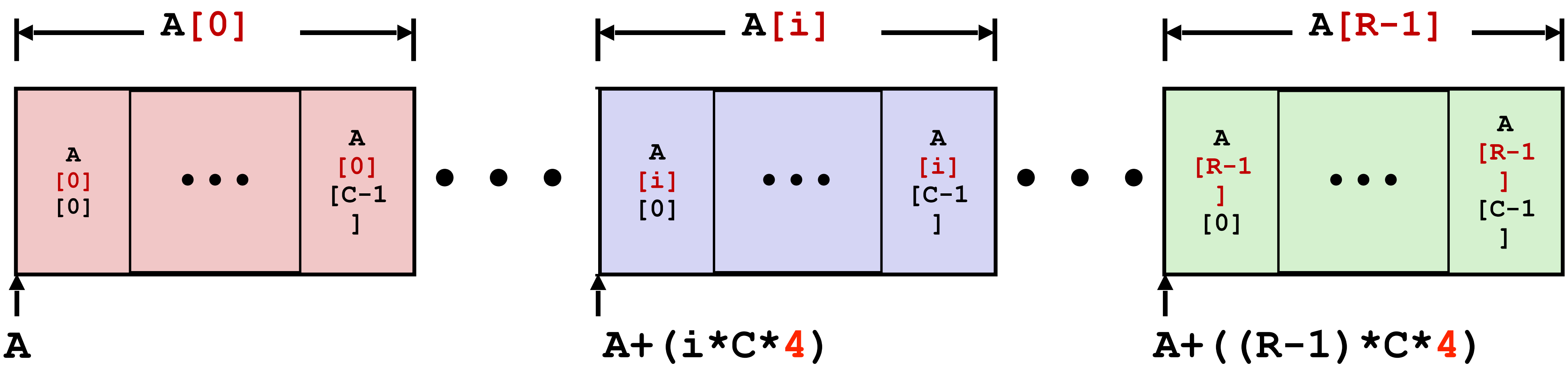
- "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  - Variable **`pgh`**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**'s, allocated contiguously
- "Row-Major" ordering of all elements in memory

Nested Array Row Access

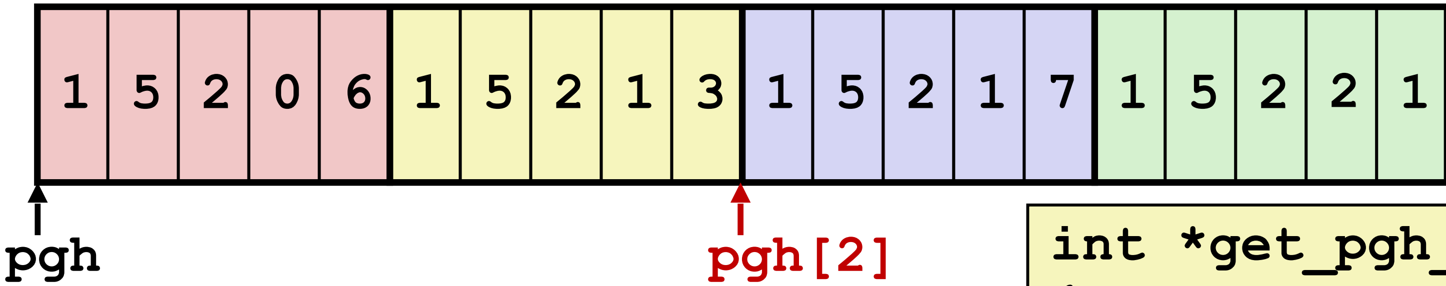
Row Vectors

- `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

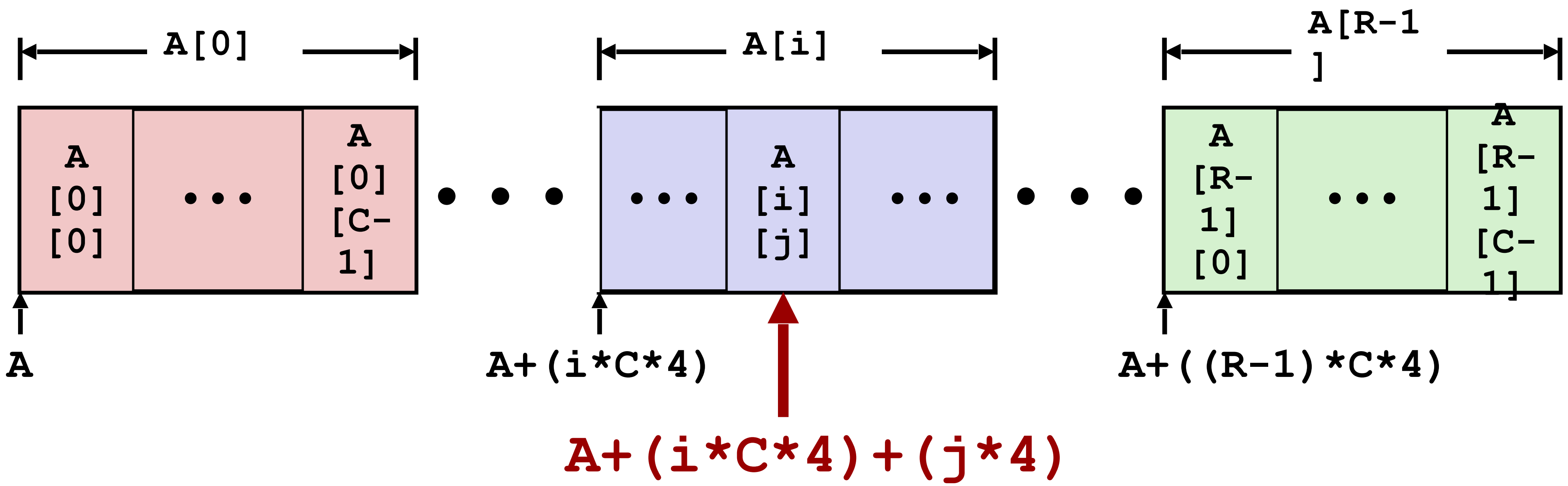
- Computes and returns address
- Compute as `pgh + 4*(index+4*index)`

Nested Array Element Access

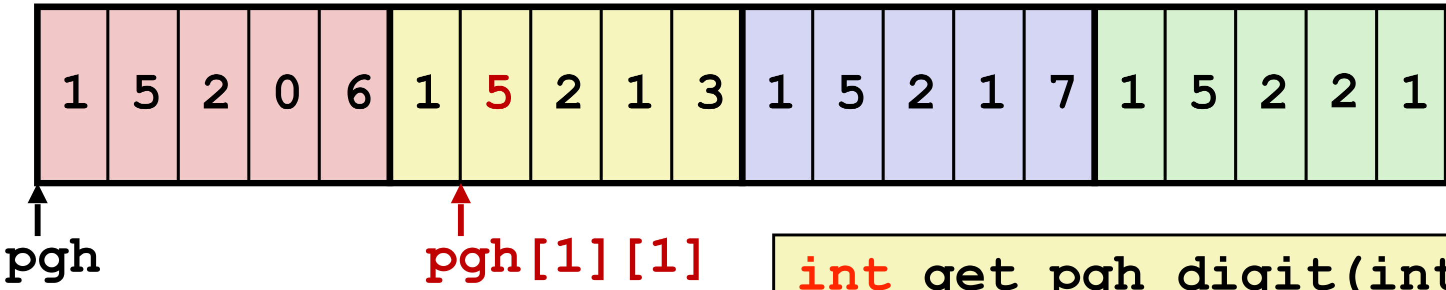
Array Elements

- `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];
```



Nested Array Element Access Code




```
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`

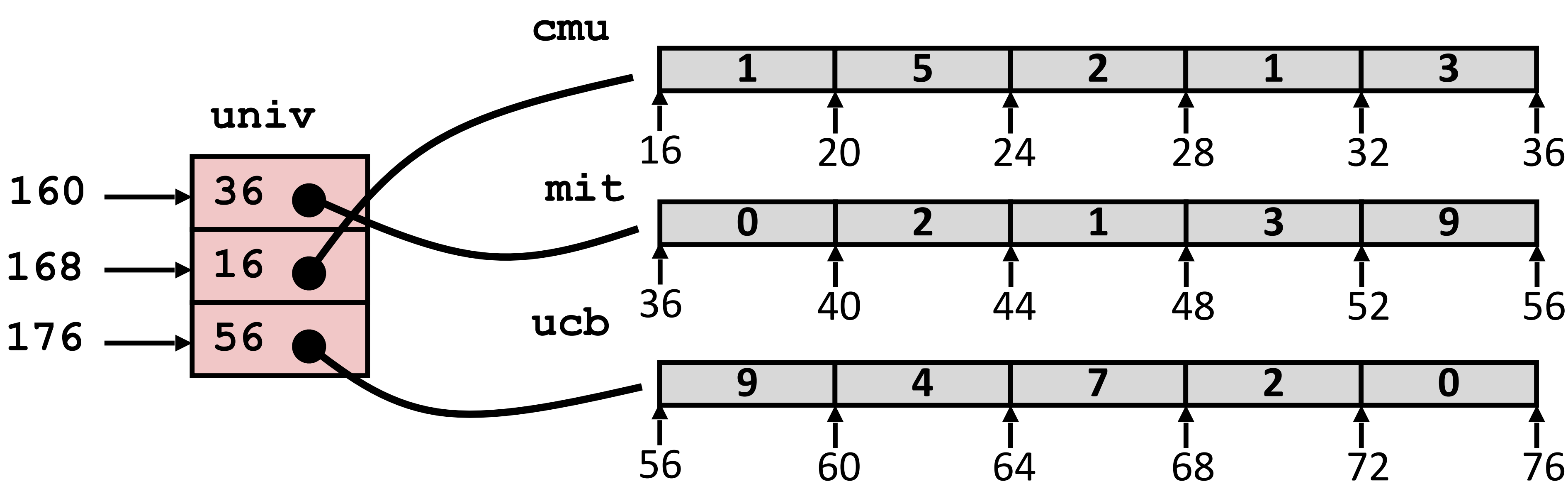
  `= pgh + 4*(5*index + dig)`

Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

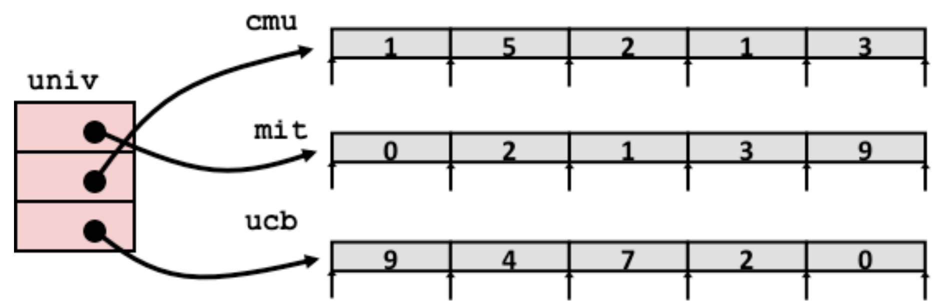
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- 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[8*index] + 4*digit
movl    (%rsi), %eax          # return *p
ret
```

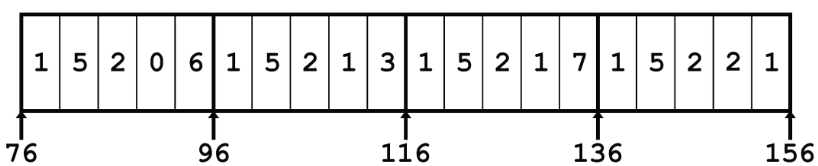
Computation

- Element access **Mem[Mem[univ+8*index]+4*digit]**
- Requires two memory reads (less efficient)
  - First get pointer to row array
  - Then access element within array

Array Element Accesses (Comparison)

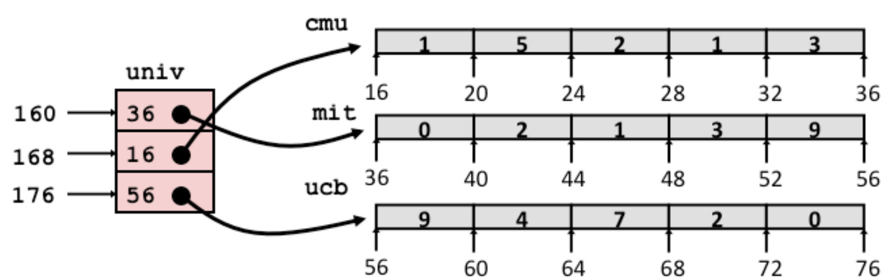
Nested array

```
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];
}
```



Accesses looks similar in C, but address computations very different:

One memory read:
`Mem[pgh+20*index+4*digit]`

Two memory reads:
`Mem[Mem[univ+8*index]+4*digit]`

N X *N* Matrix Code

- ***Fixed* dimensions**
  - Know value of *N* (# of Columns) 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
  - **Programmer** needs to explicitly define indexing

```
#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**
  - "New" feature in C99
  - gcc compiler does the work for you, allowing you to simply use **same** accessing format **as** in arrays with **fixed dimensions**

```
/* 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];
}
```

Machine-Level Programming IV: Data

- **Partial recap: Integers**
  - Word size
  - Addresses
- **One-Dimensional Arrays**
- **Structs**
  - Alignment
  - Arrays of Structs
- **Multi-Dimensional Arrays**
  - Nested (Arrays of Arrays)
  - (Arrays of) Pointers to Arrays
- **Floating Point Machine Code**

Floating Point Machine Code - Background

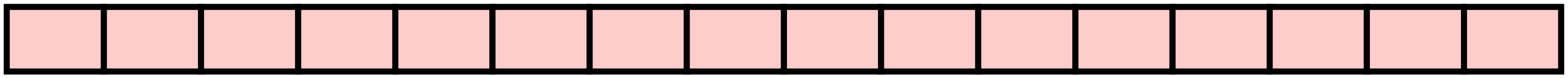
History

- x87 FP
  - Legacy, very ugly
- **SSE** (**S**treaming **S**IMD **E**xtensions) FP
  - Special case use of vector instructions
    - Instructions that allow multiple operations to be performed in a **parallel** mode known as **SIMD** ‘**sim-dee**’ (**S**ingle **I**nstruction, **M**ultiple **D**ata)
  - **SSE2**, version 2 of SSE, is supported by all processors capable of executing x86-64 code
  - Will briefly cover **SSE3** here
- **AVX** (**A**dvanced **V**ector E**x**tensions) FP
  - Newest version
  - AVX2, version 2 of AVX, introduced in **iCore 7 Haswell**
  - Similar to SSE (but registers are 32 bytes instead of 16)
  - Not covered here, but documented in book

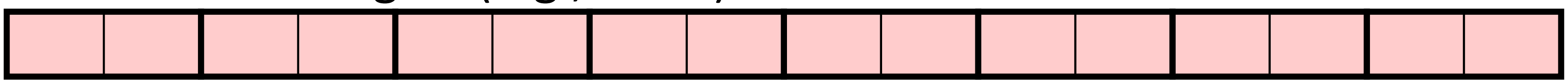
Programming with SSE3

XMM Registers

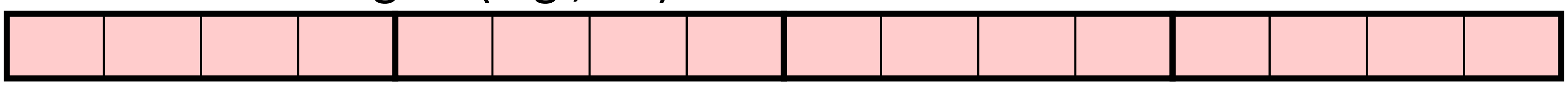
- **16** total, **each 16 bytes (128 bits)**
- **16** single-byte integers (e.g., char)



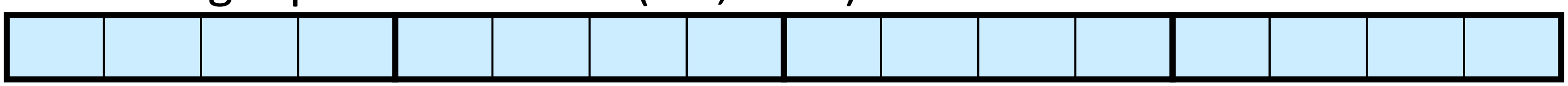
- **8** 16-bit integers (e.g., short)



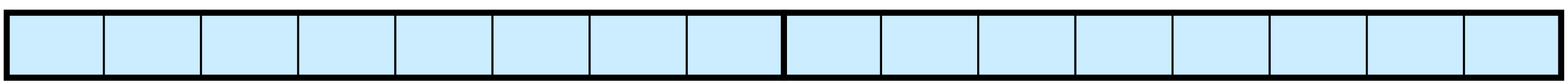
- **4** 32-bit integers (e.g., int)



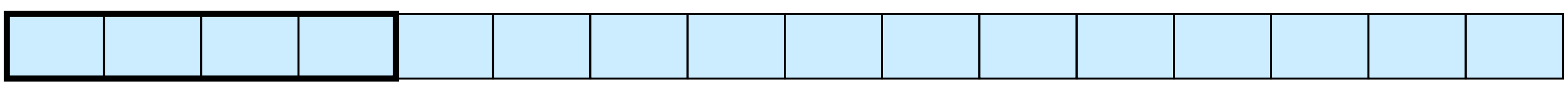
- **4** single-precision floats (i.e., float)



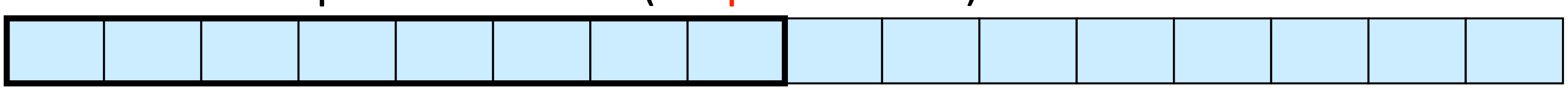
- **2** double-precision floats (i.e., double)



- **1** single-precision float (no parallelism)

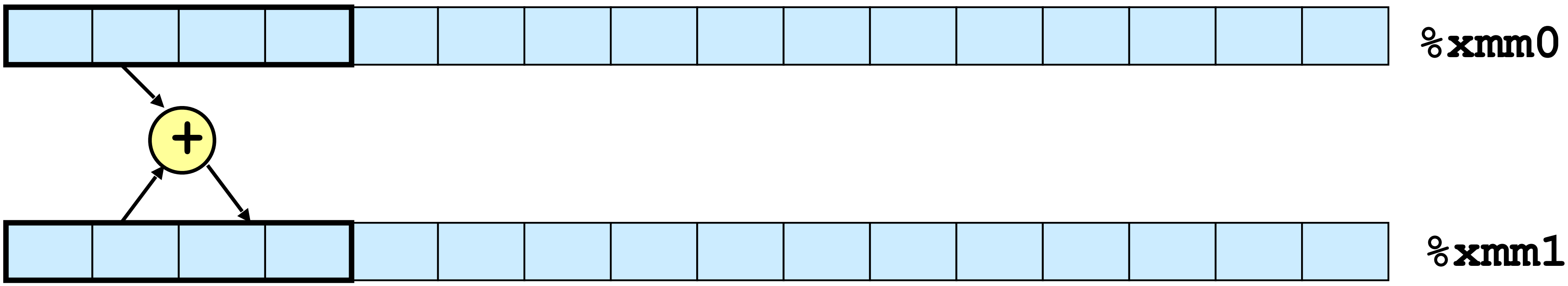


- **1** double-precision float (no parallelism)

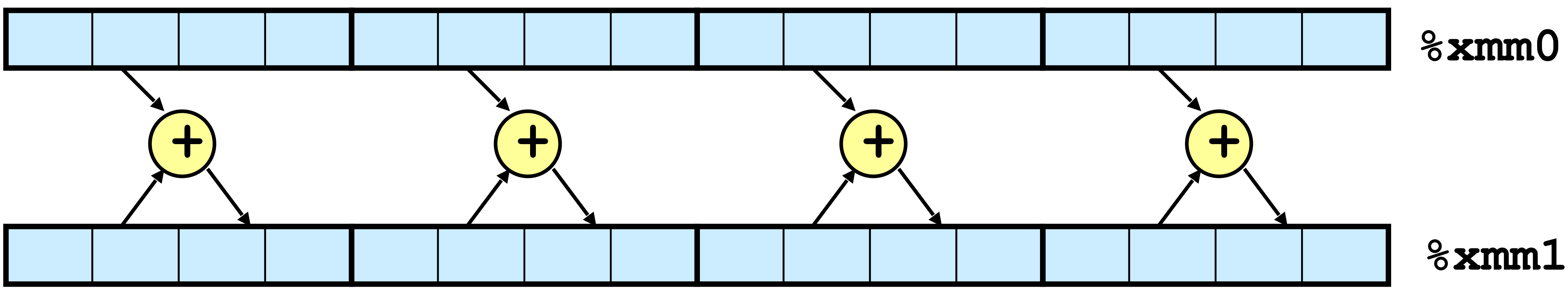


Scalar & SIMD Operations

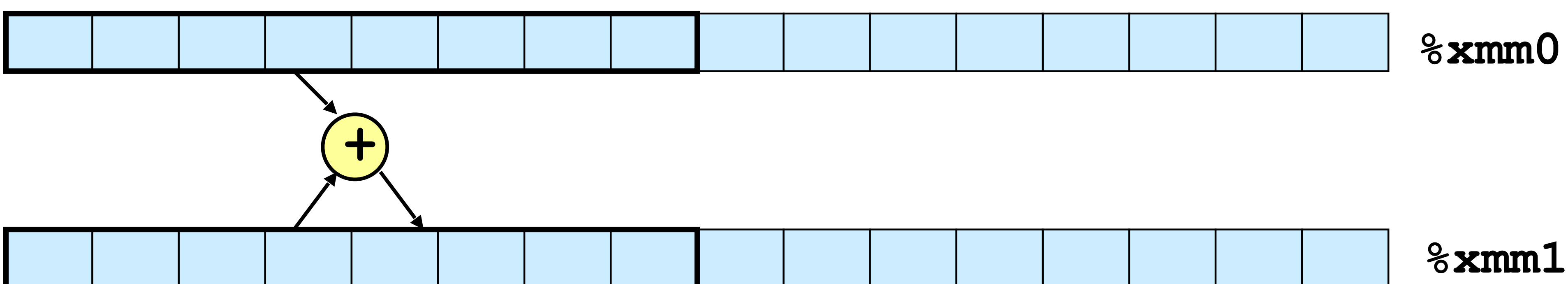
- Scalar Operations: Single Precision `addss %xmm0,%xmm1`



- SIMD Operations: Single Precision `addps %xmm0,%xmm1`



- Scalar Operations: Double Precision `addsd %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;
}
```

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

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

```
  # 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
```

Summary

- **Arrays**
  - Elements packed into contiguous region of memory
  - Use index arithmetic to locate individual elements
- **Structures**
  - Elements packed into single region of memory
  - Access using offsets determined by compiler
  - Possible require internal and external padding to ensure alignment
- **Combinations**
  - Can nest structure and array code arbitrarily
- **Floating Point Machine Code**