Computer Organization: A Programmer's Perspective

Machine-Level Programming
(4: Data Structures)

Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**
  - Allocation
  - Access
  - Alignment
- **Optional: Floating Point**

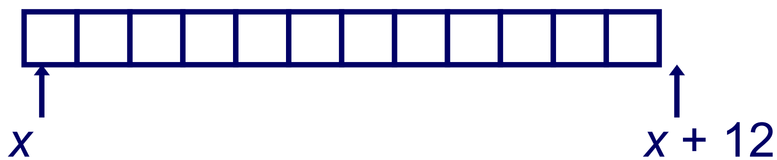
Array Allocation

Basic Principle

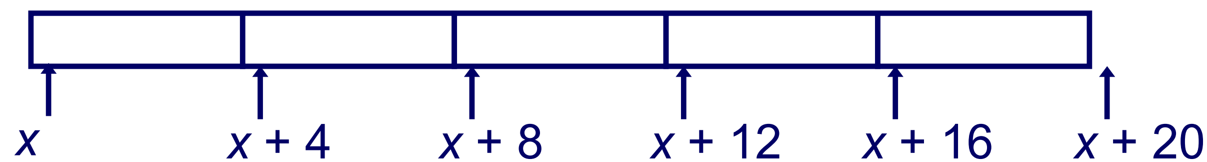
T `A[`L`];`

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

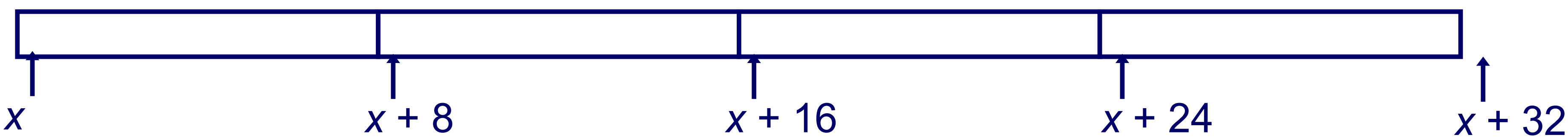
```
char string[12];
```



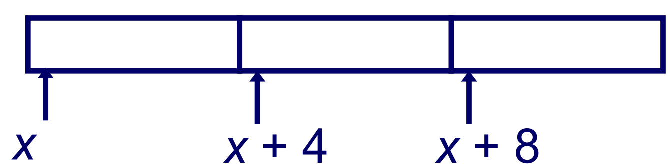

```
int val[5];
```




```
double a[4];
```



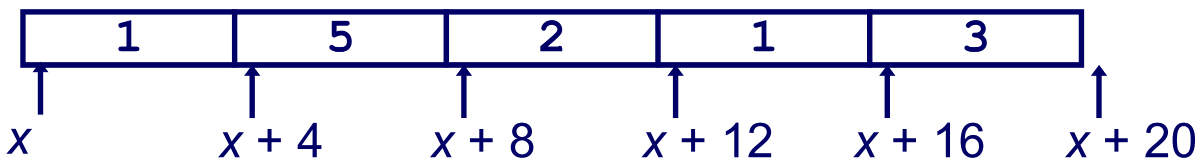

```
char *p[3];
```



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

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



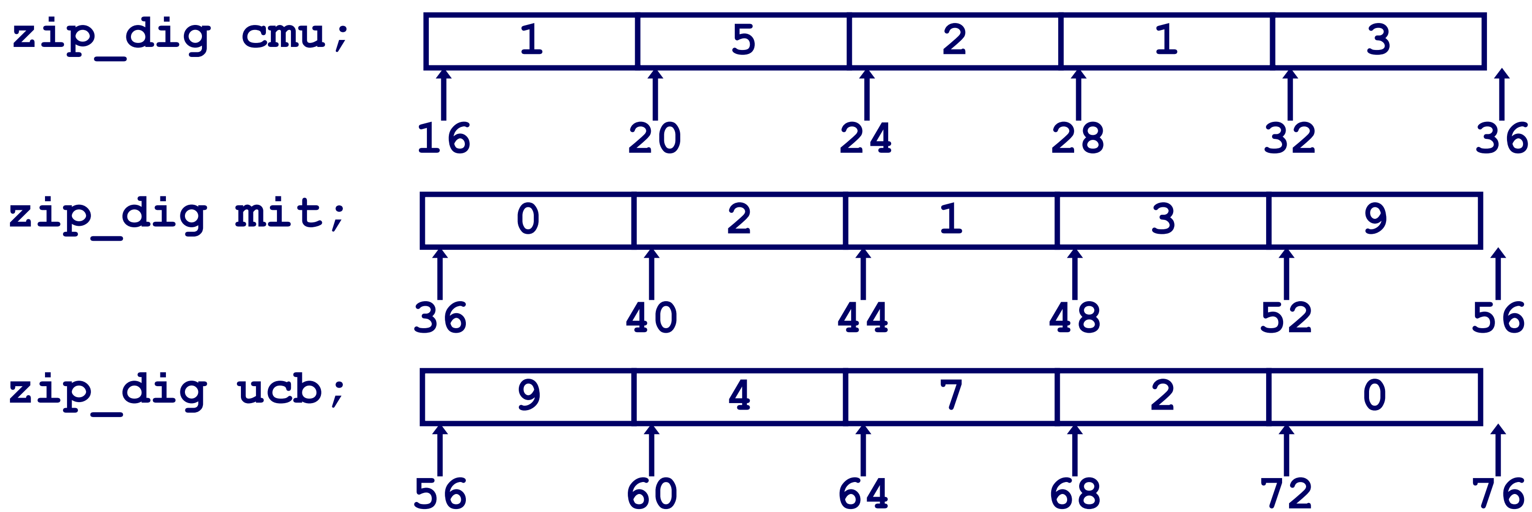
- Reference / Type / Value

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 +` *i*	`int *`	*x* + 4 *i*

Array Example

```c
#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 };
```



Notes

- 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

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

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

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

Array Loop Example

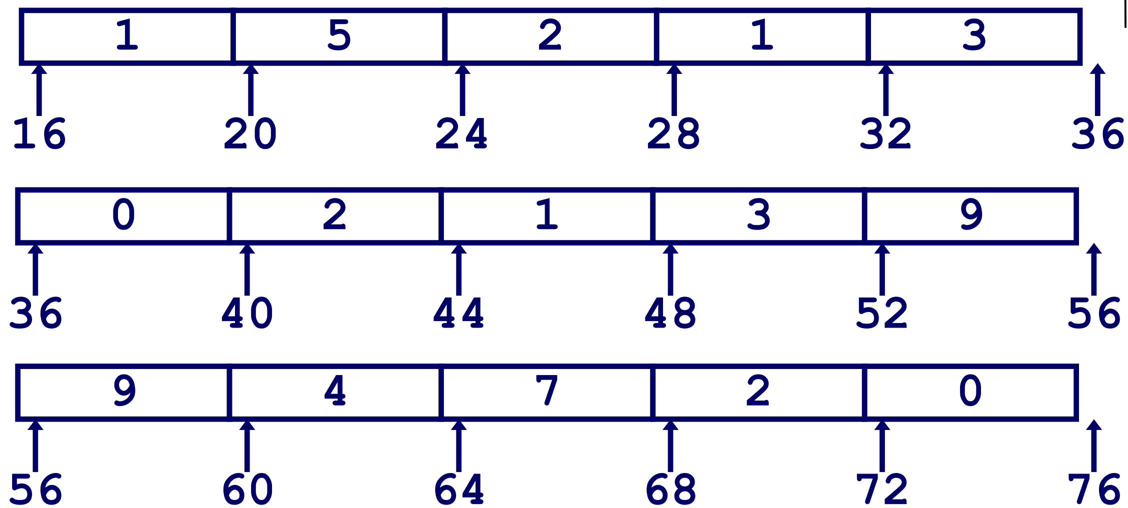
```
void zincr(zip_dig z) {
  size_t i;

  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
```

Referencing Examples

```
zip_dig cmu;
zip_dig mit;
zip_dig ucb;
```



Code Does Not Do Any Bounds Checking!

Reference	Address	Value	Guaranteed?
`mit[3]`	`36 + 4* 3 = 48`	`3`	**Yes**
`mit[5]`	`36 + 4* 5 = 56`	`9`	**No**
`mit[-1]`	`36 + 4*-1 = 32`	`3`	**No**
`cmu[15]`	`16 + 4*15 = 76`	`??`	**No**

- Out of range behavior implementation-dependent
  - No guaranteed relative allocation of different arrays

Nested Array Allocation

Declaration

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

- 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

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

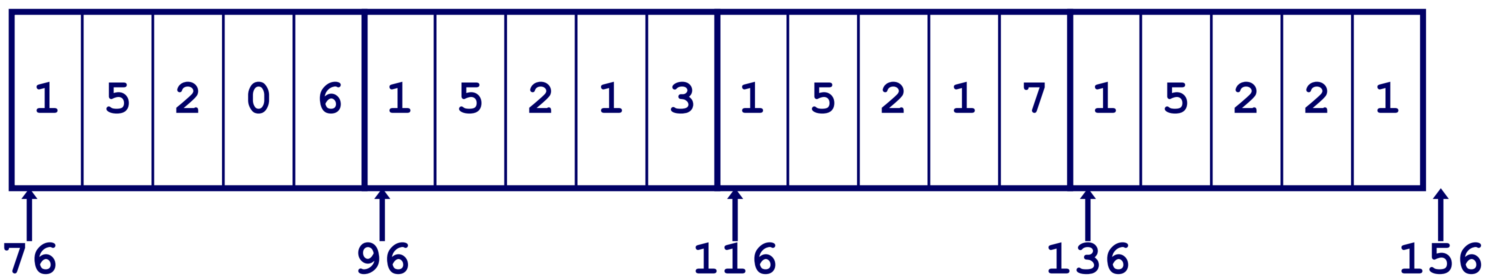
A [0] [0]	• • •	A [0] [C-1]	A [1] [0]	• • •	A [1] [C-1]	• • •	A [R-1] [0]	• • •	A [R-1] [C-1]

`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];
```



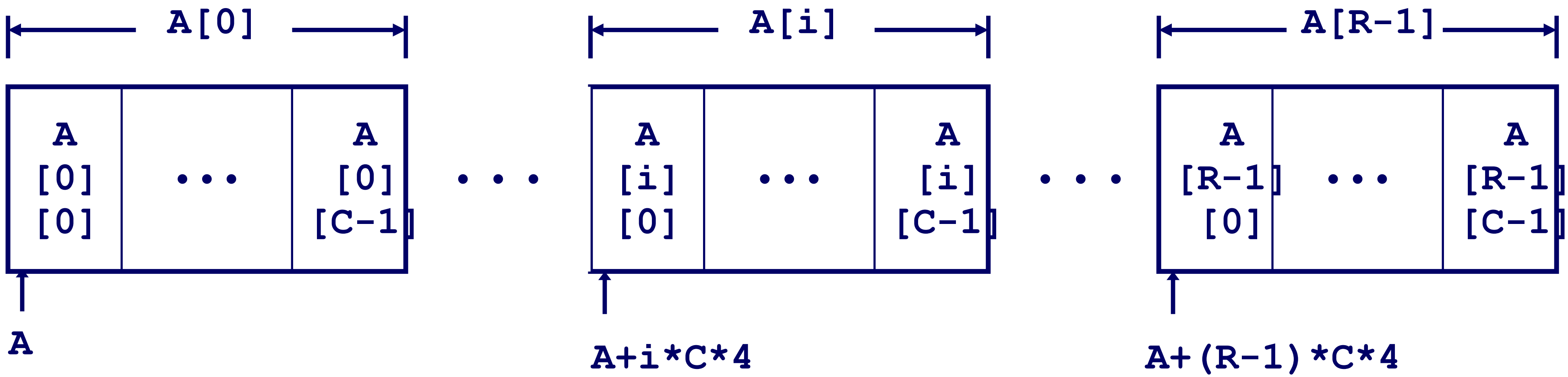
- Declaration "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
- Variable `pgh` denotes array of 4 elements, allocated contiguously
  - Each element is an array of 5 `int`'s, allocated contiguously
- "Row-Major" ordering of all elements guaranteed

Nested Array Row Access

Row Vectors

- `A[i]` is array of *C* elements
- Each element of type *T*
- Starting address `A +` *i* * *C* * *K*

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



Nested Array Row Access Code

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

Row Vector

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

Code

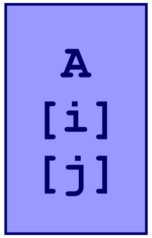
- Computes and returns address
- Compute as `pgh + 4*(index+4*index)`

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

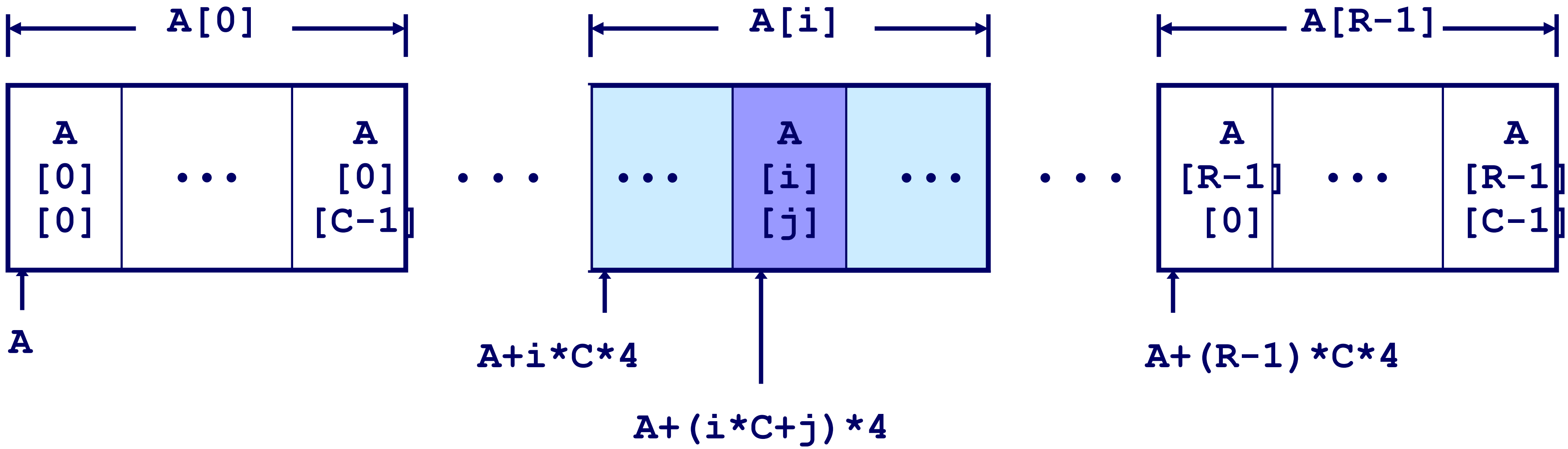
Nested Array Element Access

Array Elements

- `A[i][j]` is element of type *T*
- Address `A +` (*i* * *C* + *j*) * *K*




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



Nested Array Element Access Code

Array Elements

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

Code

- Computes address as: `pgh + 4*dig + 4*(index+4*index)`
- `movl` performs memory reference

```
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)]
```

Referencing Examples

```
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

Reference	Address	Value	Guaranteed?
`Pgh[3][3]`	`76+20*3+4*3 = 148`	`2`	Yes
`Pgh[2][5]`	`76+20*2+4*5 = 136`	`1`	Yes
`Pgh[2][-1]`	`76+20*2+4*-1 = 112`	`3`	Yes
`Pgh[4][-1]`	`76+20*4+4*-1 = 152`	`1`	Yes
`Pgh[0][19]`	`76+20*0+4*19 = 152`	`1`	Yes
`Pgh[0][-1]`	`76+20*0+4*-1 = 72`	`??`	No

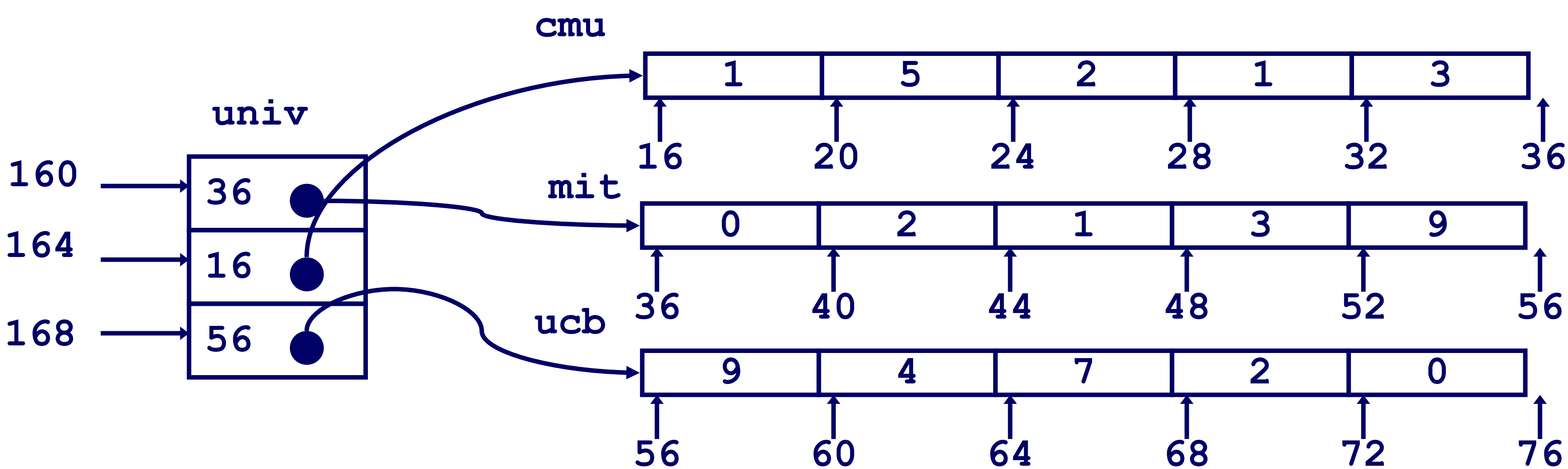
- Code does not do any bounds checking
- Ordering of elements within array guaranteed

Multi-Level Array Example

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of `int`'s

```
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};
```



Element Access in Multi-Level Array

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

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

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

Array Element Accesses

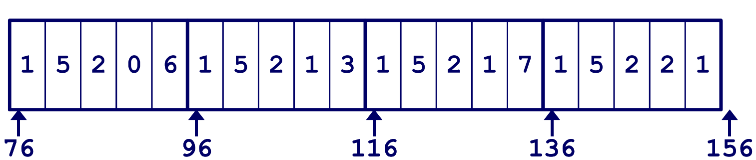
- Similar C references

Nested Array

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

- Element at

`Mem[pgh+20*index+4*dig]`



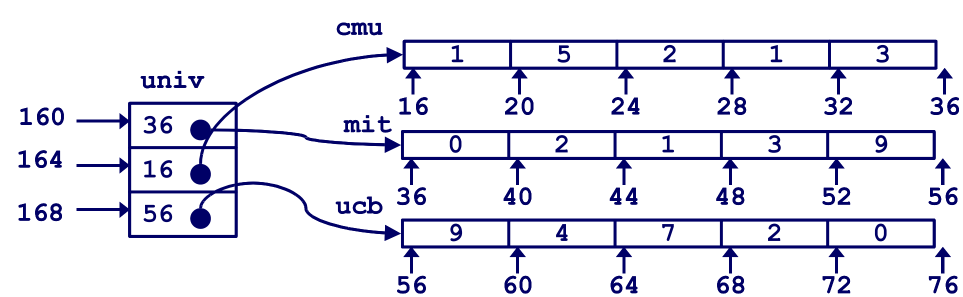
- Different address computation

Multi-Level Array

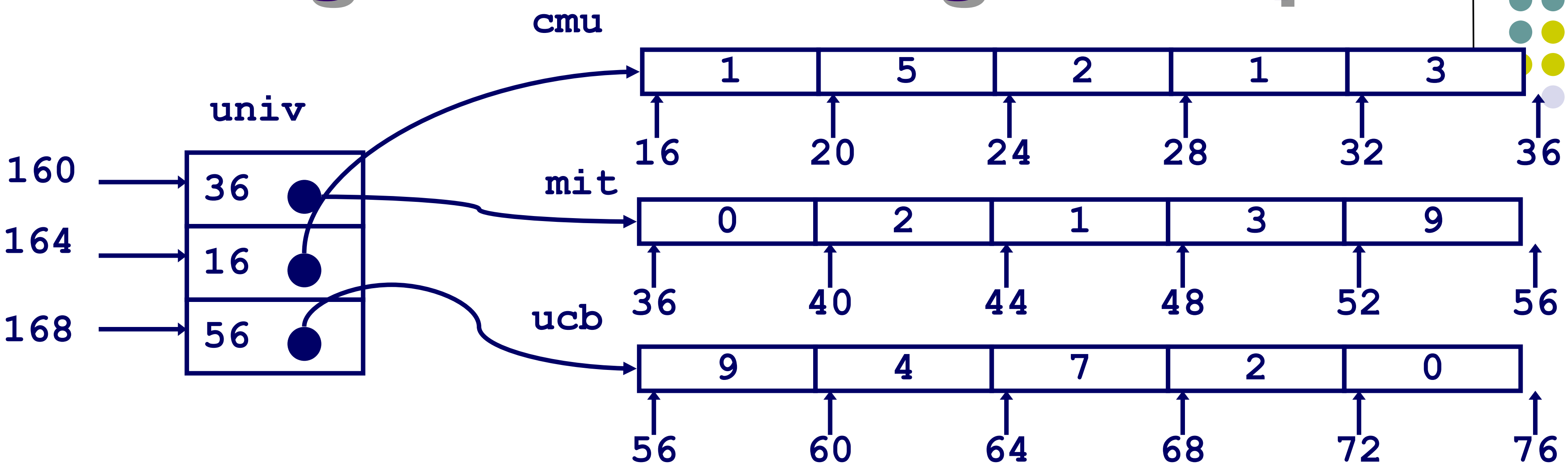
```
int get_univ_digit
  (int index, int dig)
{
  return univ[index][dig];
}
```

- Element at

`Mem[Mem[univ+8*index]+4*dig]`



Strange Referencing Examples



Reference	Address	Value	Guaranteed?
`univ[2][3]`	`56+4*3 = 68`	`2`	**Yes**
`univ[1][5]`	`16+4*5 = 36`	`0`	**No**
`univ[2][-1]`	`56+4*-1 = 52`	`9`	**No**
`univ[3][-1]`	`??`	`??`	**No**
`univ[1][12]`	`16+4*12 = 64`	`7`	**No**

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed

N X 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
  - ISO C99 standard
  - Try at your own risk

```
/* 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 X 16 Matrix Access

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

```
/* 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
```

n X n Matrix Access

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

```
/* 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];
}
```

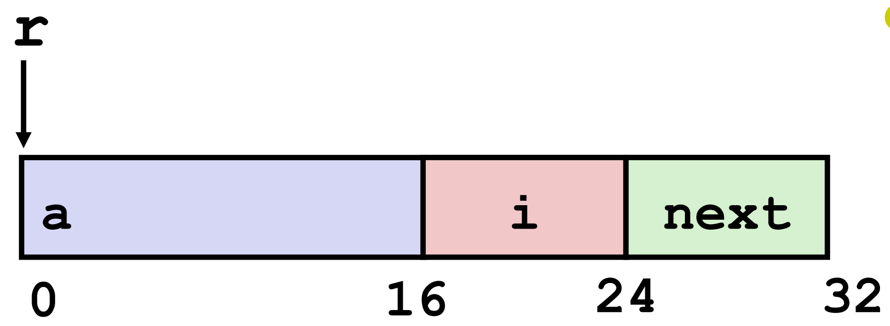
```
  # n in %rdi, a in %rsi, i in %rdx, j in %rcx
  imulq   %rdx, %rdi             # n*i
  leaq    (%rsi,%rdi,4), %rax    # a + 4*n*i
  movl    (%rax,%rcx,4), %eax    # a + 4*n*i + 4*j
  ret
```

Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**
  - Allocation
  - Access
  - Alignment
- **Optional: Floating Point**

Structure Representation

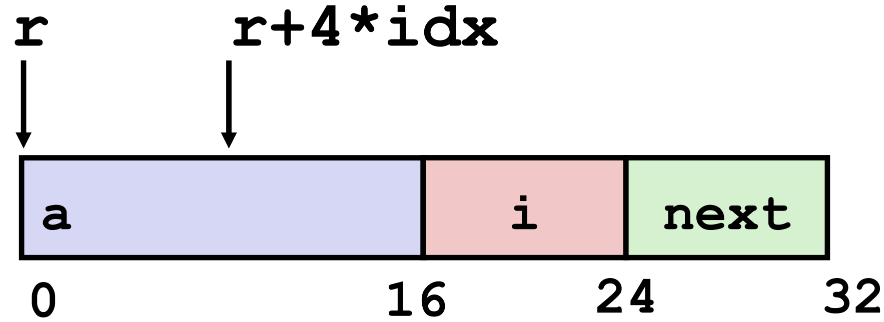
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- **Structure represented as block of memory**
  - Big enough to hold all of the fields
- **Fields ordered according to declaration**
  - 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*idx`

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

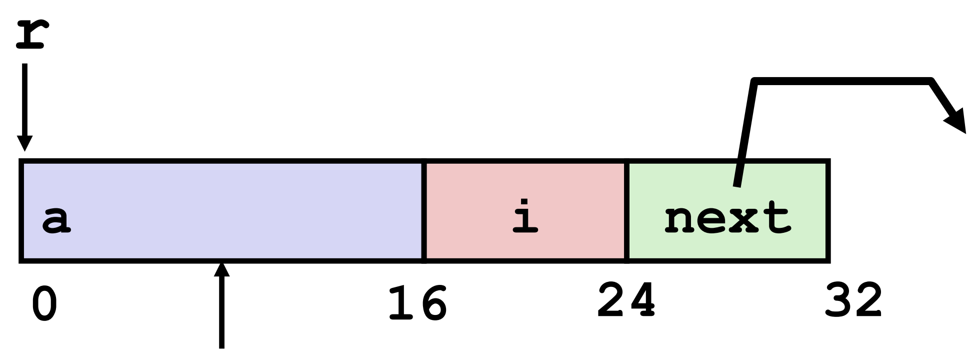
```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

Following Linked List

- C 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;
  }
}
```



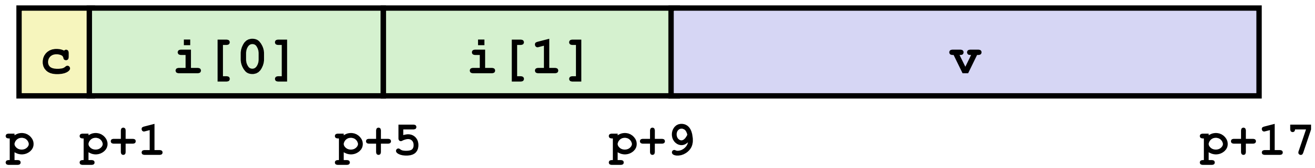
Element `i`

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
```

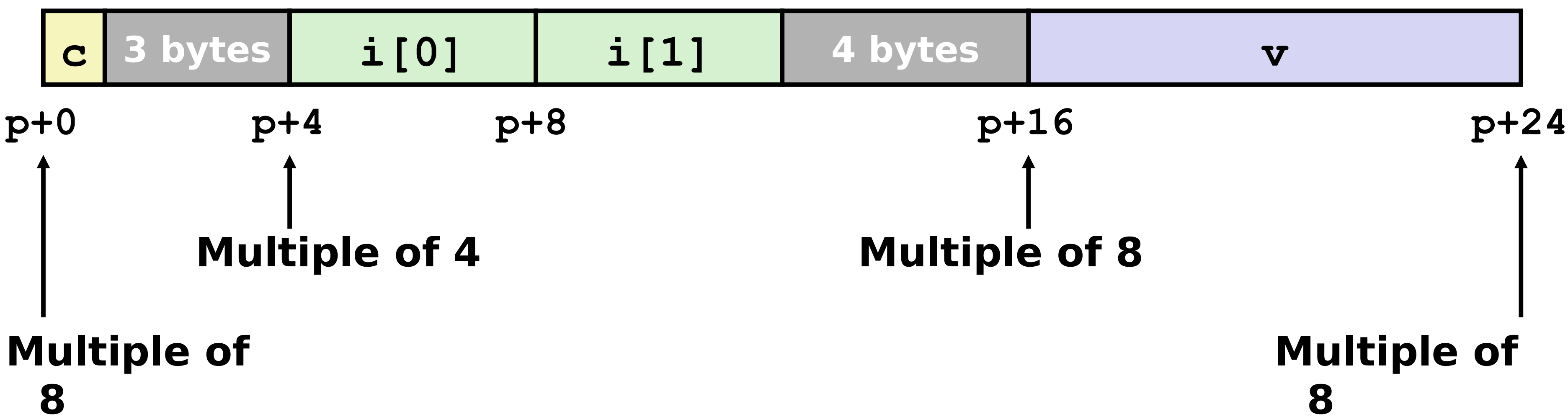
Structures & Alignment

- **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

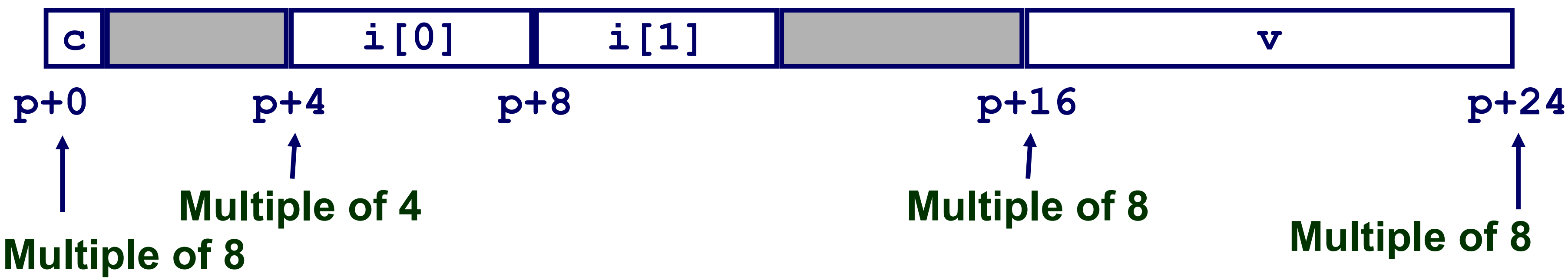
- **Aligned Data**
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on x86-64
- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans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

Linux vs. Windows (IA32 bits)

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

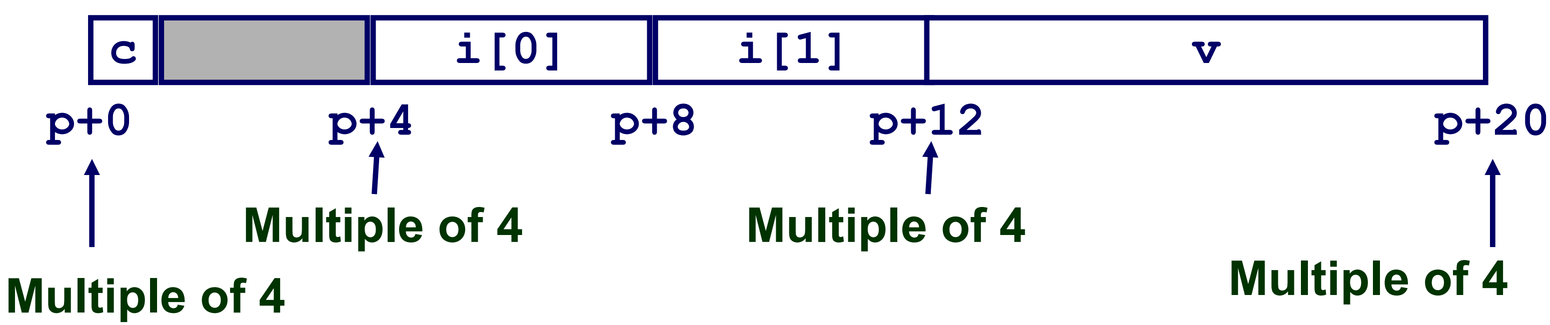
Windows (including Cygwin):

- K = 8, due to `double` element



Linux:

- K = 4; `double` treated like a 4-byte data type



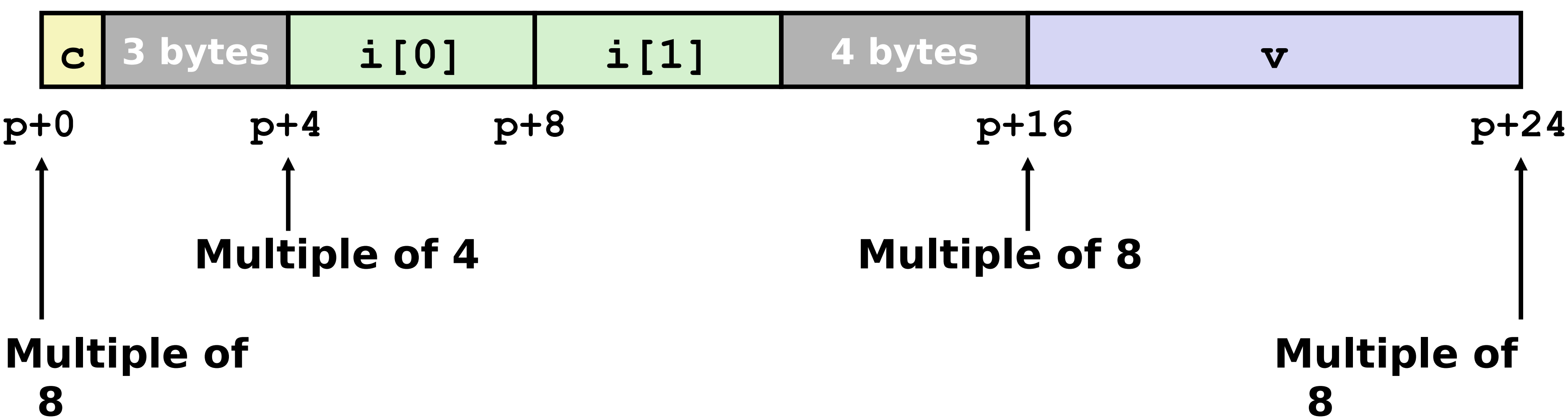
Specific Cases of Alignment (x86-64)

- 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

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

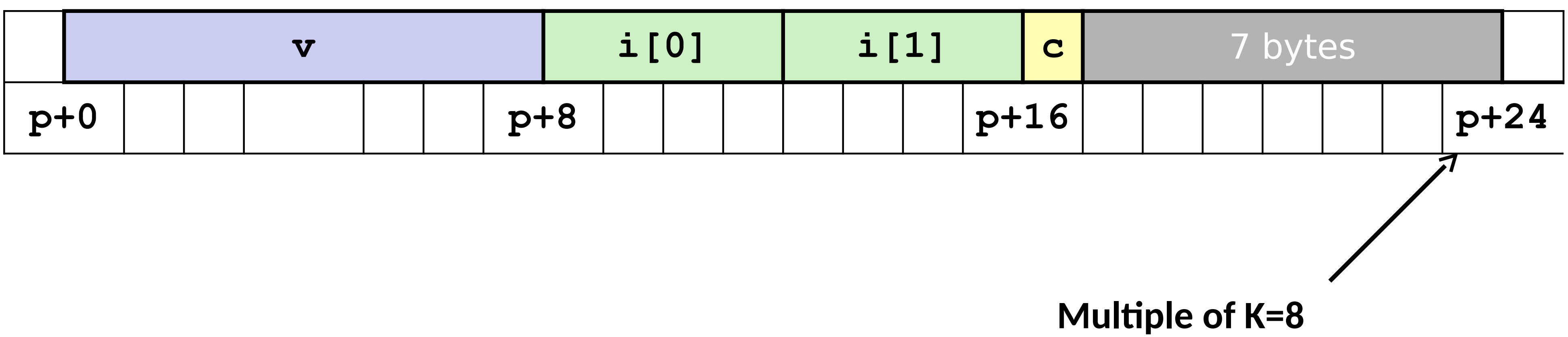
- **Within structure:**
  - Must satisfy each element's alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement K
    - K = Largest alignment of any element
  - Initial address & structure length must be multiples of K
- **Example:**
  - K = 8, due to `double`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



Meeting Overall Alignment Requirement

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

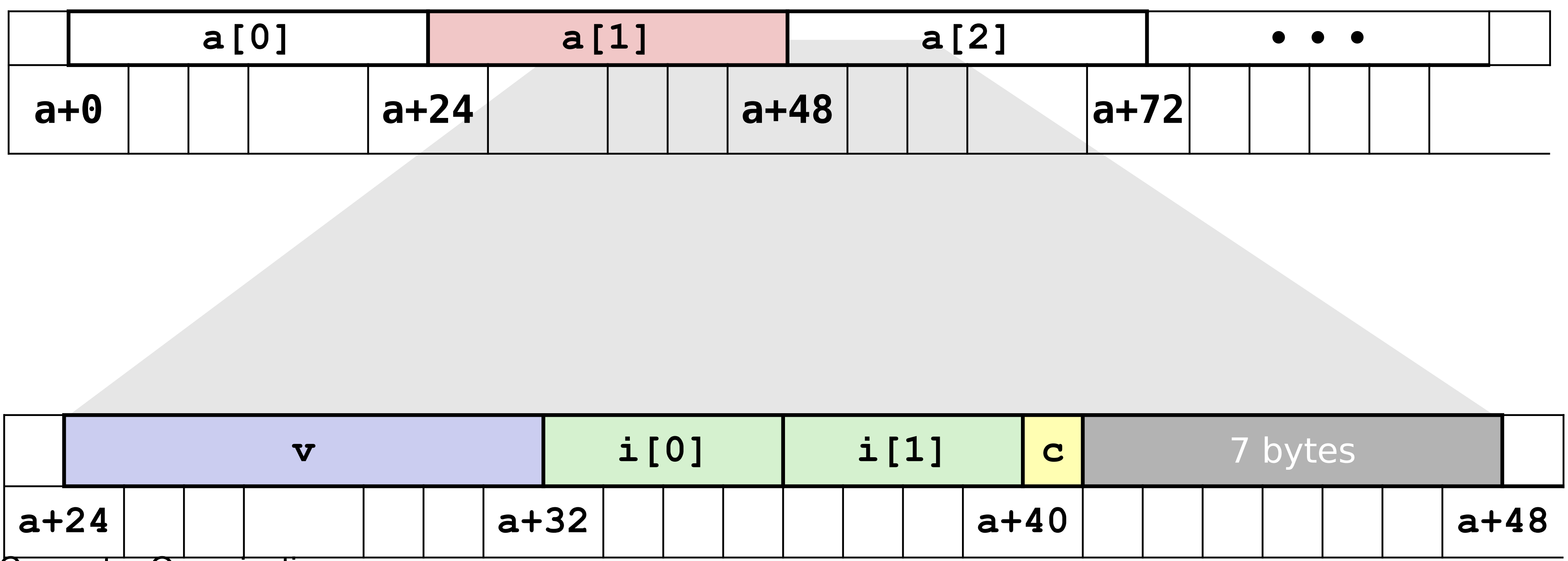
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures

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

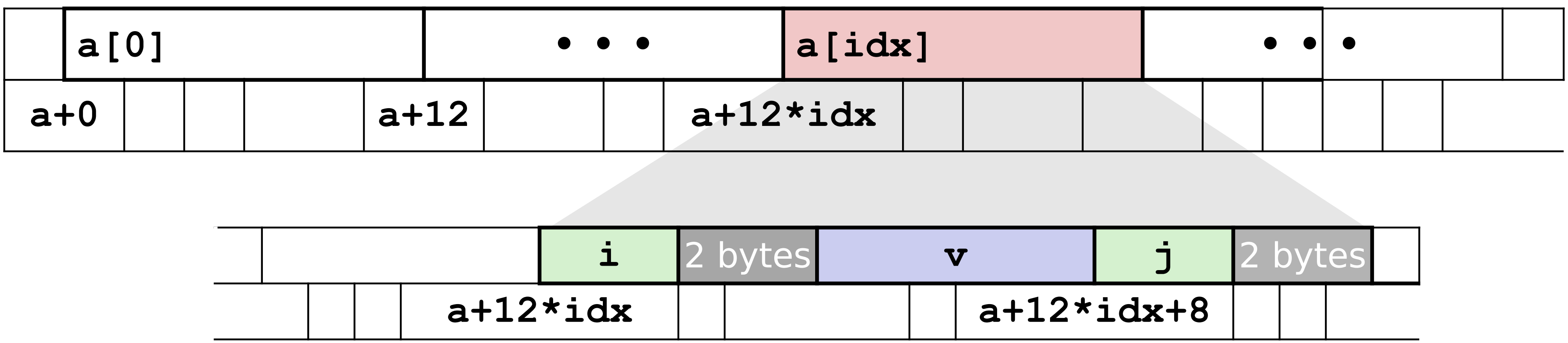
- Overall structure length multiple of K
- Satisfy alignment requirement for every element



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




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

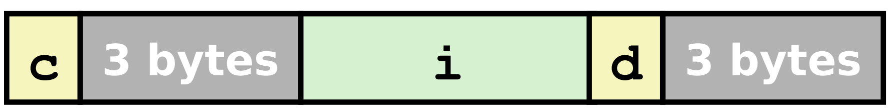
```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

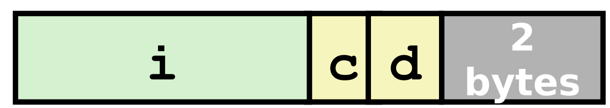
Saving Space

- Put large data types first
- Effect (K=4)

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

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





שאלות?

Optional Materials

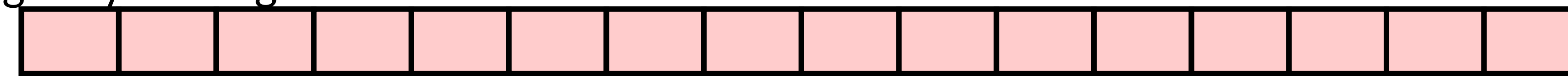
Background

- **History**
  - x87 FP
    - Legacy, very ugly
  - SSE FP
    - Supported by Shark machines
    - Special case use of vector instructions
  - AVX FP
    - Newest version
    - Similar to SSE
    - Documented in book

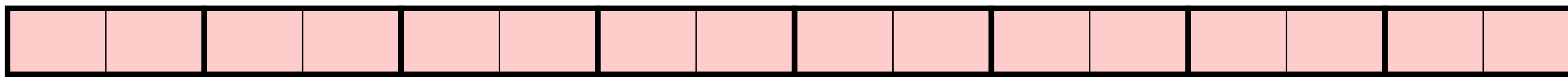
Programming with SSE3

XMM Registers: 16 total

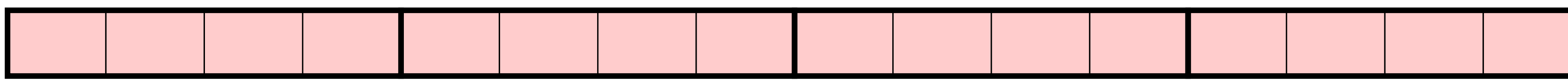
- Each 16 bytes
- 16 single-byte integers



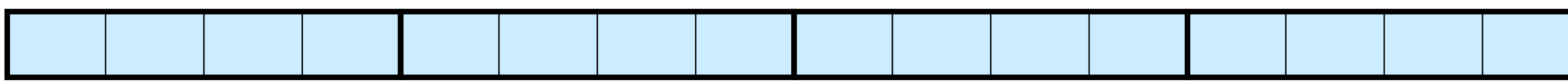
- 8 16-bit integers



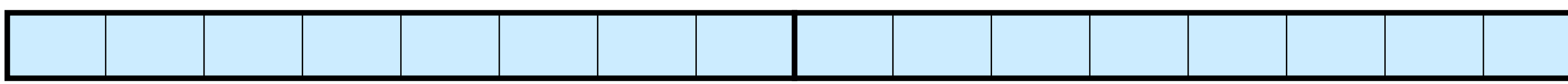
- 4 32-bit integers



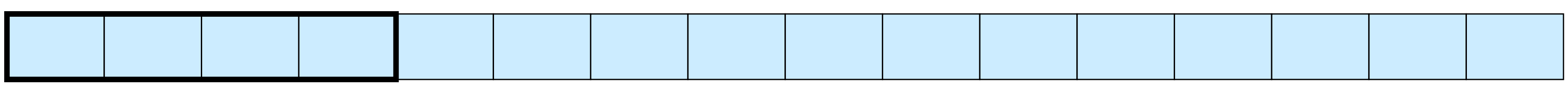
- 4 single-precision floats



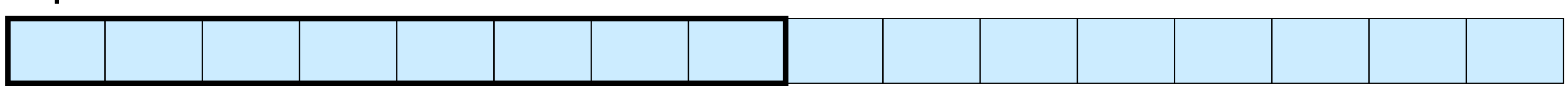
- 2 double-precision floats



- 1 single-precision float



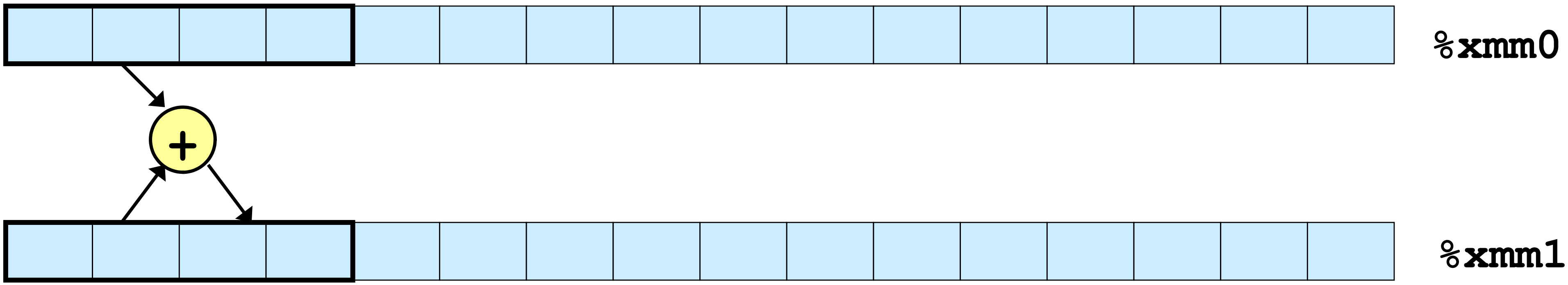
- 1 double-precision float



Scalar & SIMD Operations

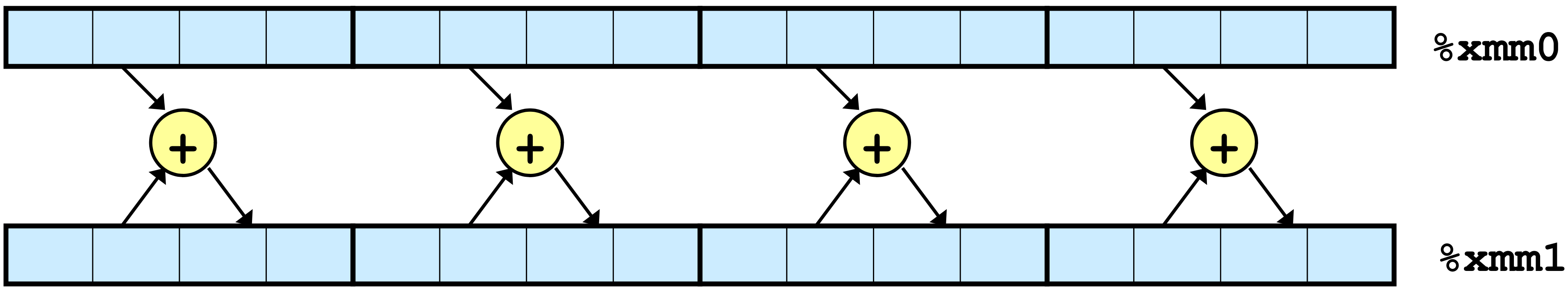
- Scalar Operations: Single Precision

`addss %xmm0,%xmm1`



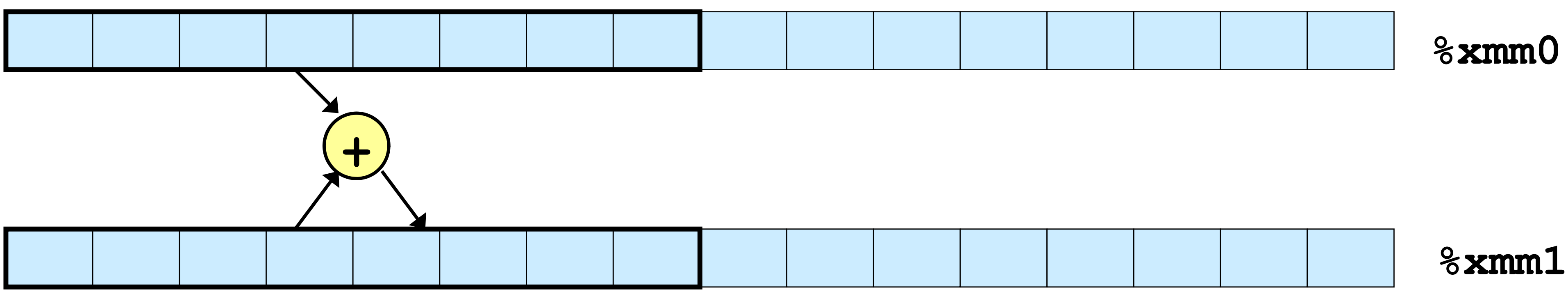
- SIMD Operations: Single Precision

`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
```

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