Machine-Level Programming IV: Data

adapted for CS367@GMU

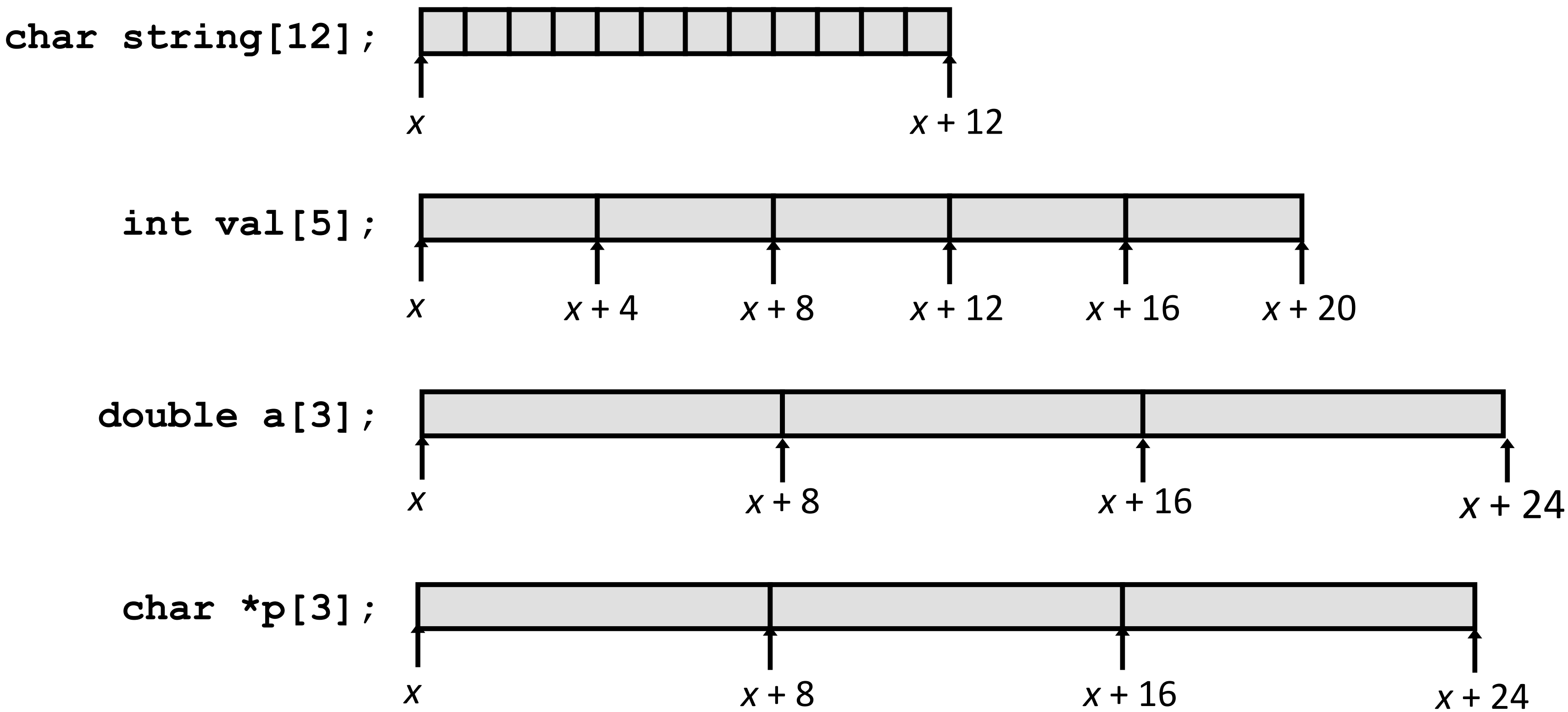
Arrays

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



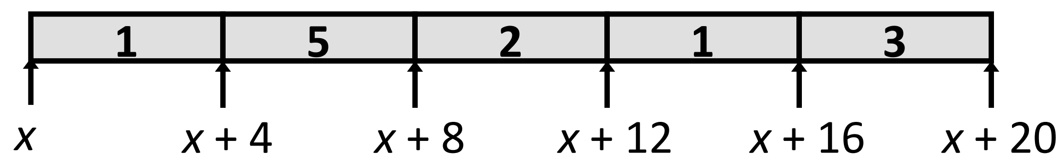
Array Access

- **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**

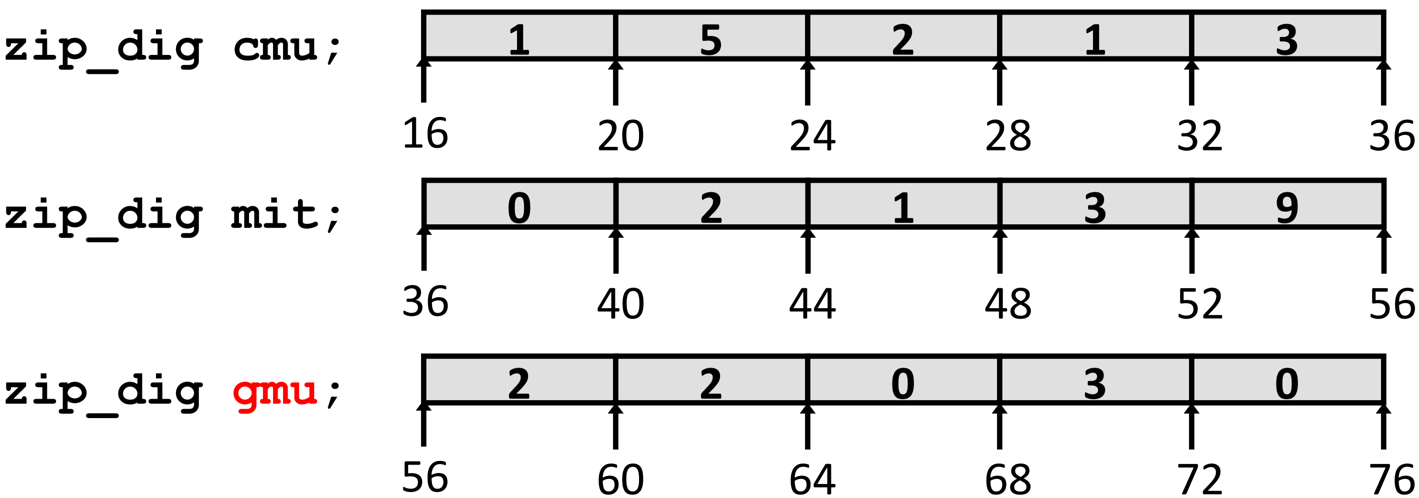
`int val[5];`



Reference	Type	Value
`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

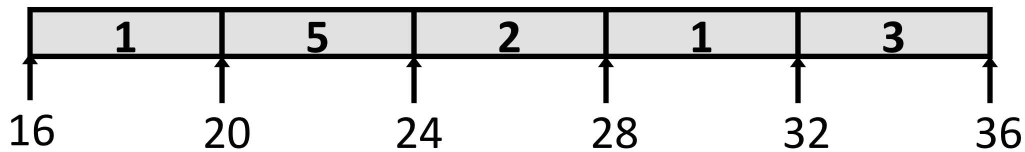
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig gmu = { 2, 2, 0, 3, 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];
}
```

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

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index

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
```

Multidimensional (Nested) Arrays

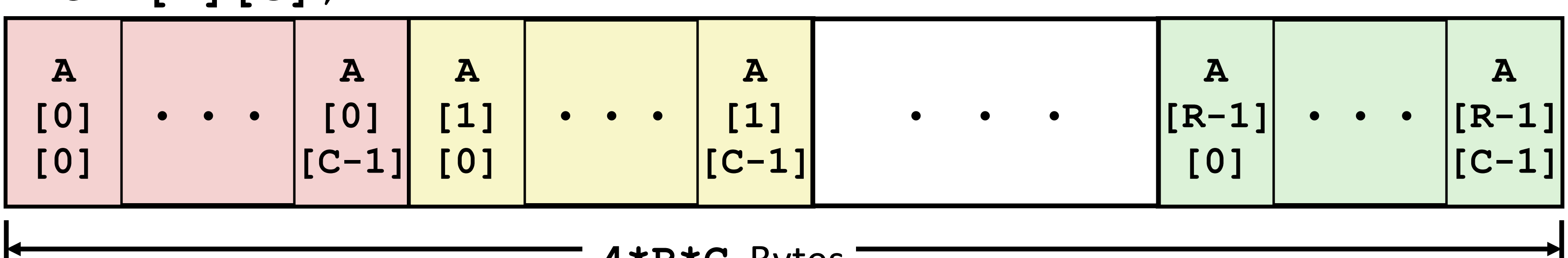
- **Declaration**

  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

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

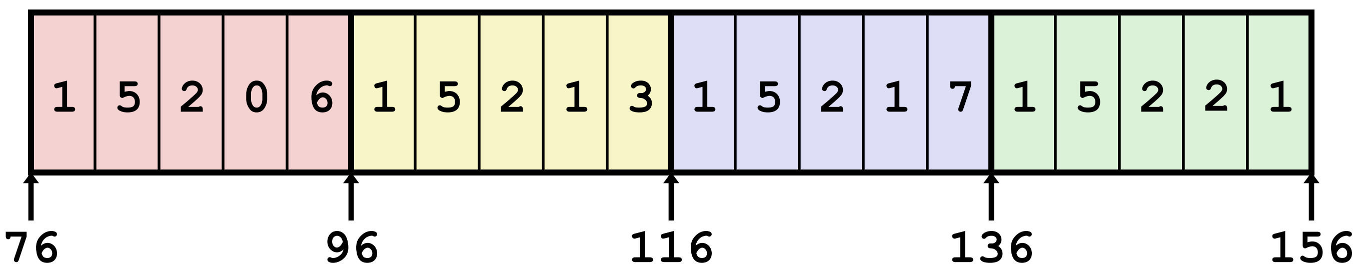
```
int A[R][C];
```



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];`

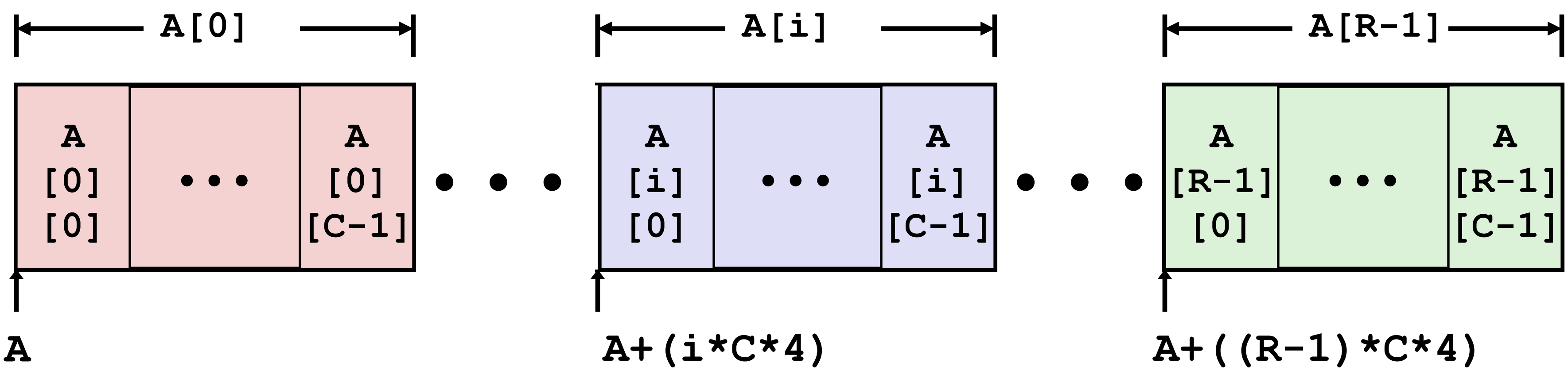


- **"`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"**
  - Variable **`pgh`**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**'s, allocated contiguously
- **"Row-Major" ordering of all elements in memory**

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*)

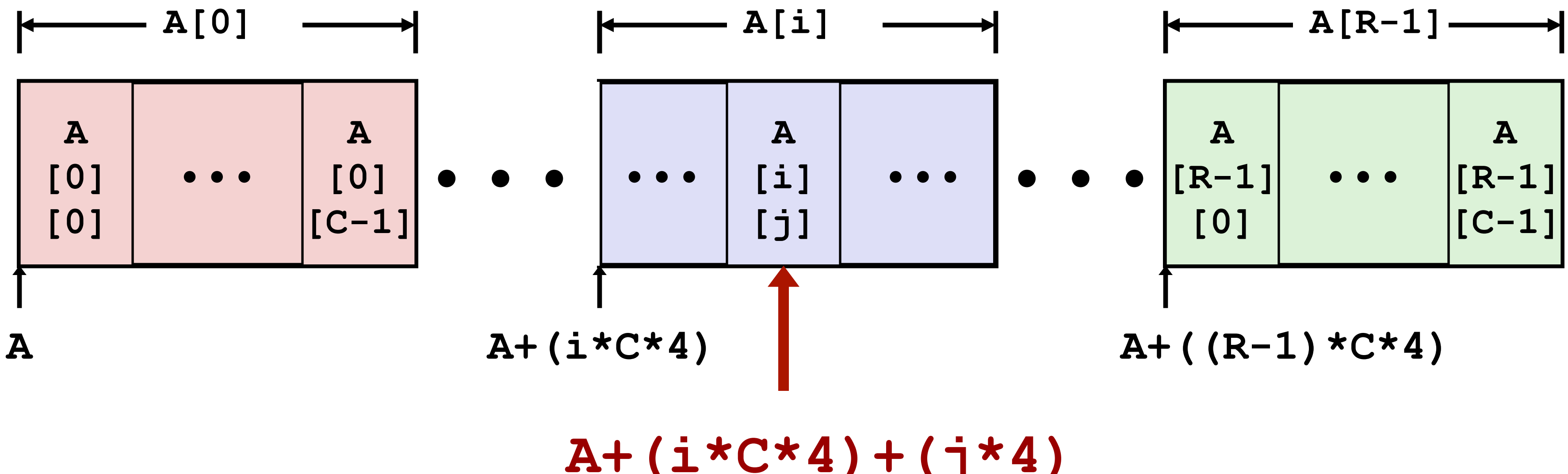
```
int A[R][C];
```



Nested Array 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];
```

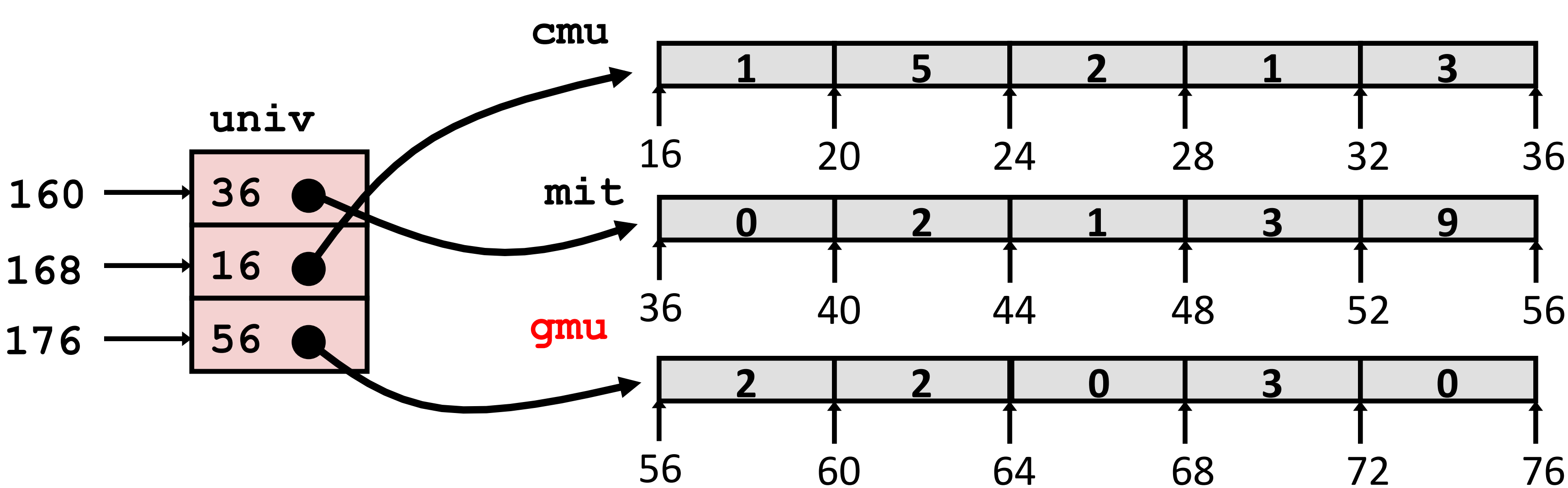


Multi-Level Array Example

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

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

- 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

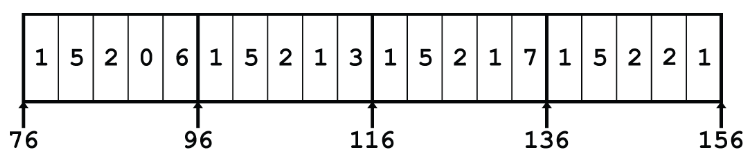
Array Element Accesses

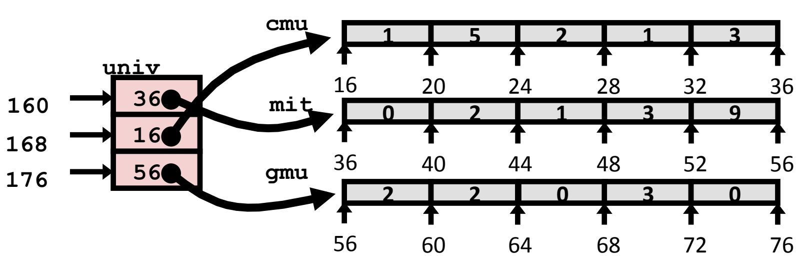
Nested array

```
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];
}
```





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

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

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

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

```
/* 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

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

fix_matrix is always int[16][16]

```
/* 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

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

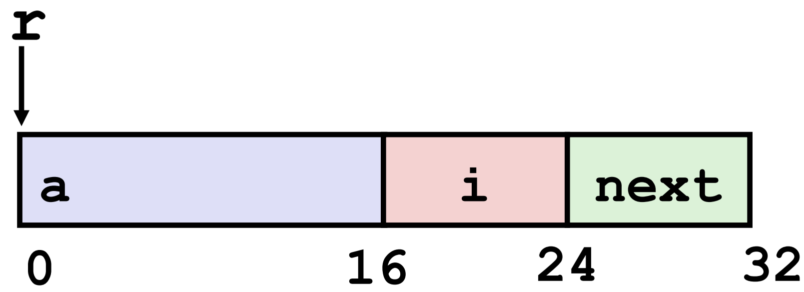
```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n], size_t i, size_t j)
{
  return a[i][j];
}
```

```
  # 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
```

Structures

Structure Representation

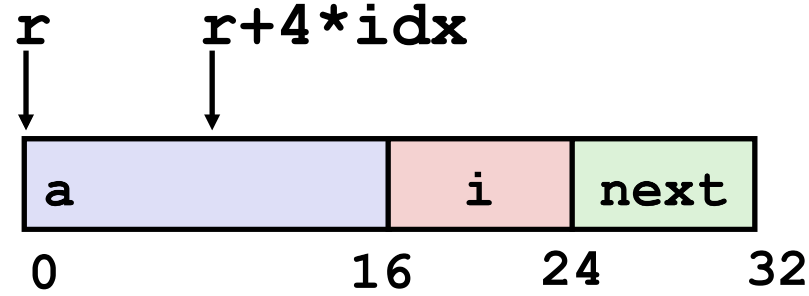
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- **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];
}
```

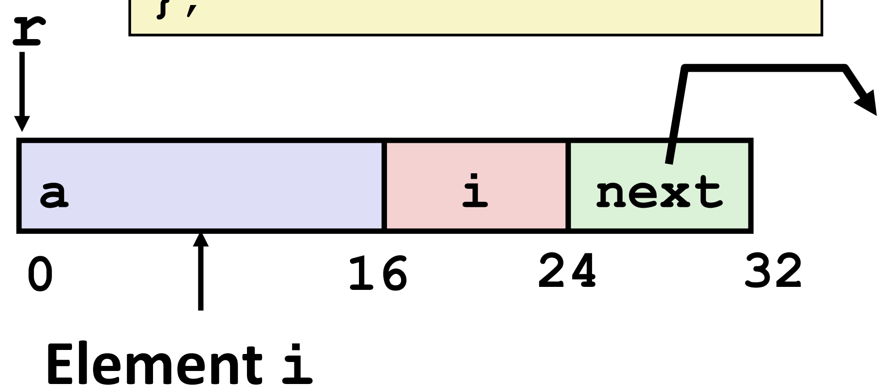
```
# r in %rdi, idx in %rsi
leaq  (%rdi,%rsi,4), %rax
ret
```

Following Linked List

- C Code

```
struct rec {
    int a[3];
    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;
  }
}
```

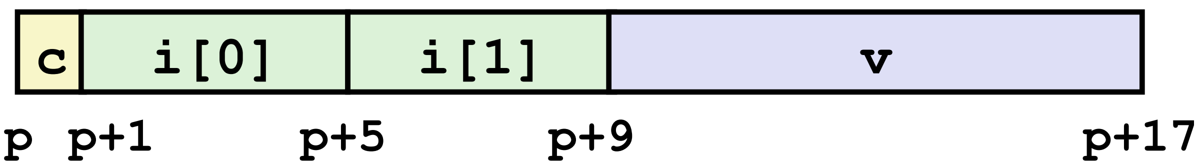


Register	Value
%rdi	r
%rsi	val

```
.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
```

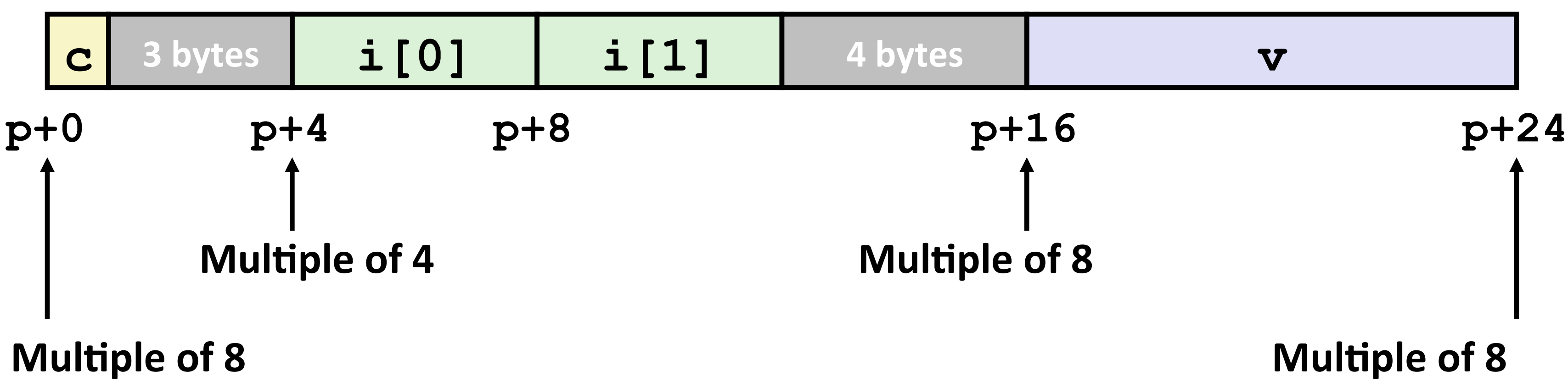
Structures & Alignment

- **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
    - Virtual memory trickier when datum spans 2 pages
- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields

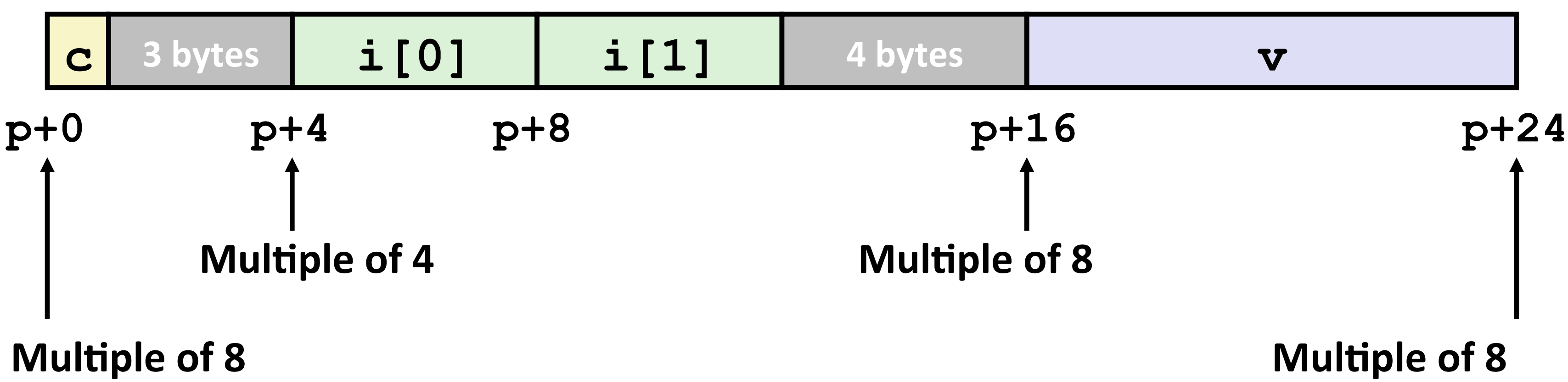
Specific Cases of Alignment (x86-64)

- **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

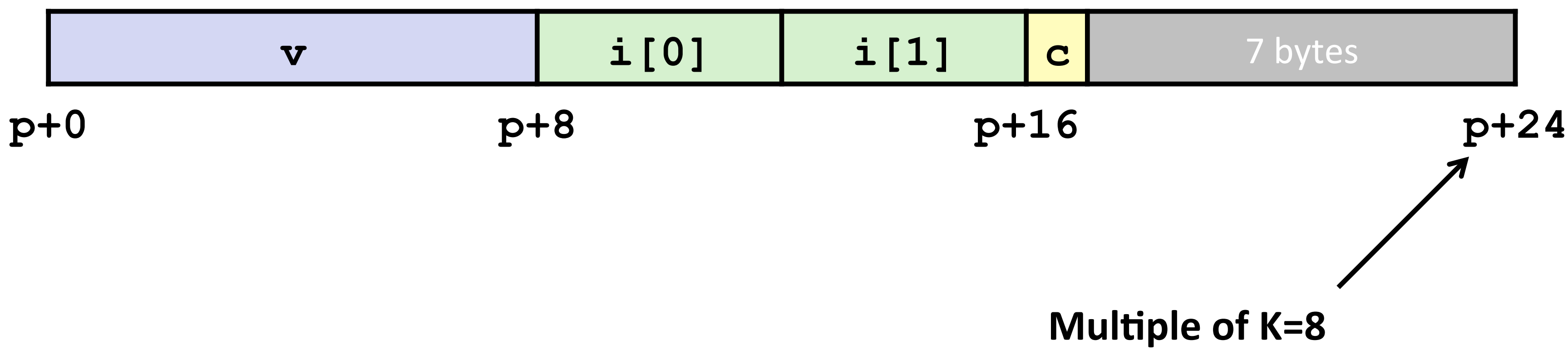
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Meeting Overall Alignment Requirement

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

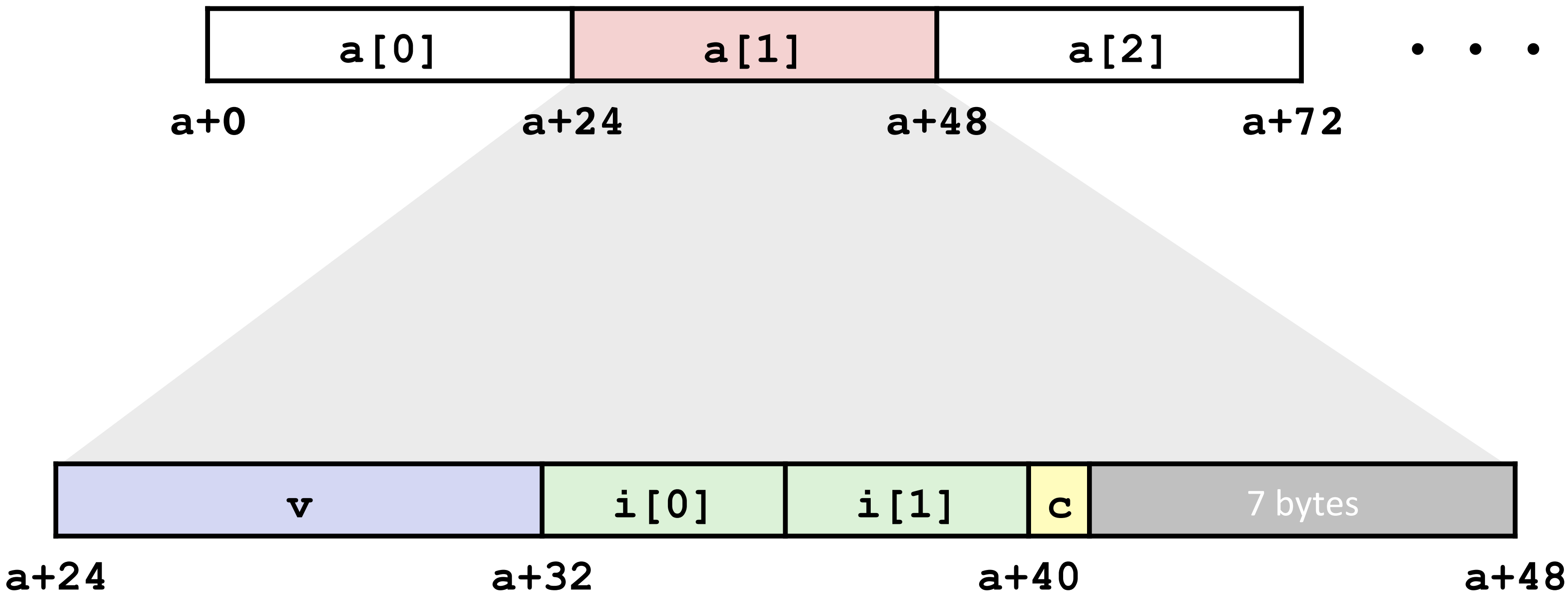
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures

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

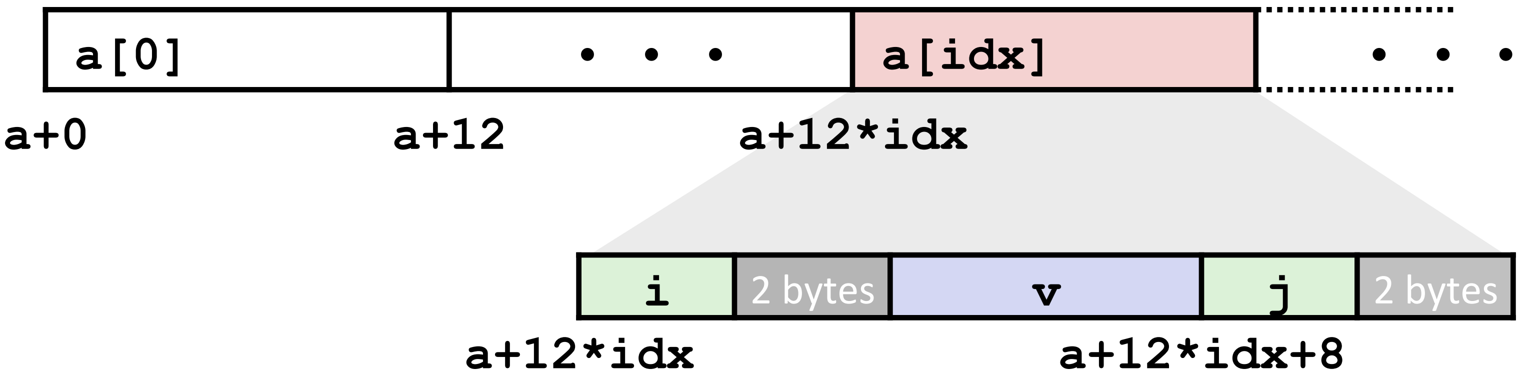
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



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
```

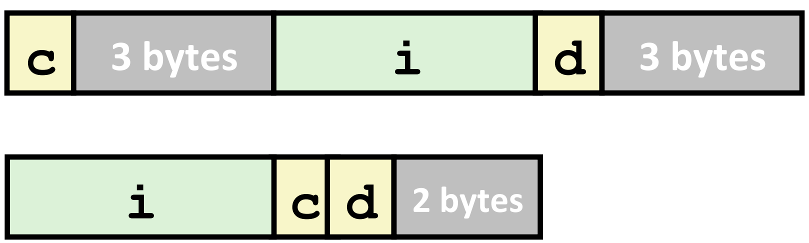
Saving Space

- **Put large data types first**

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

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

- **Effect (K=4)**



Floating Point

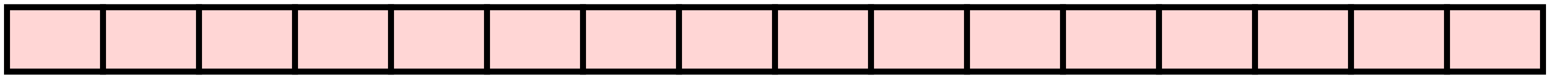
Background

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

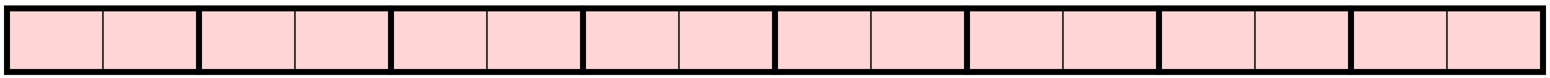
Programming with SSE3

XMM Registers

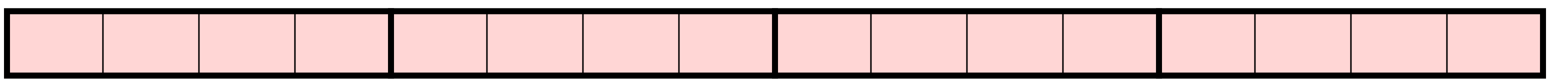
- 16 total, each 16 bytes
- 16 single-byte integers



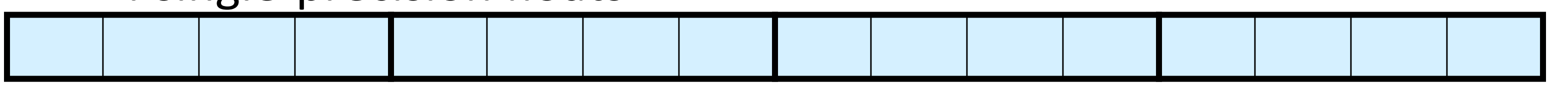
- 8 16-bit integers



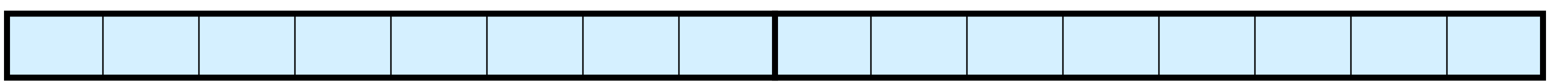
- 4 32-bit integers



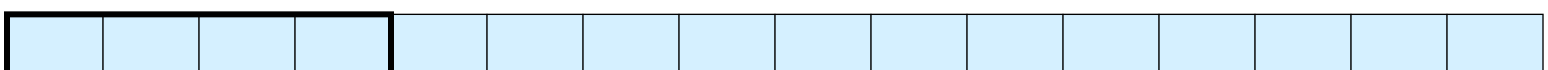
- 4 single-precision floats



- 2 double-precision floats



- 1 single-precision float

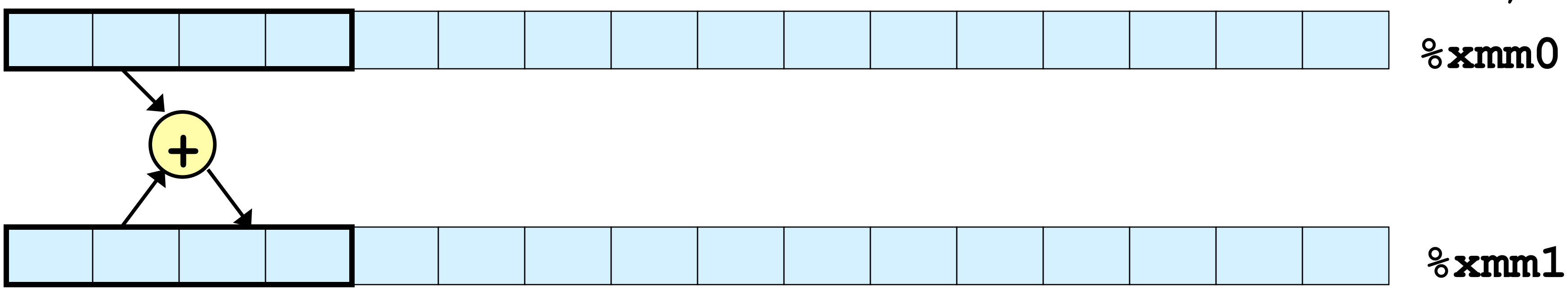


- 1 double-precision float

Scalar & SIMD Operations

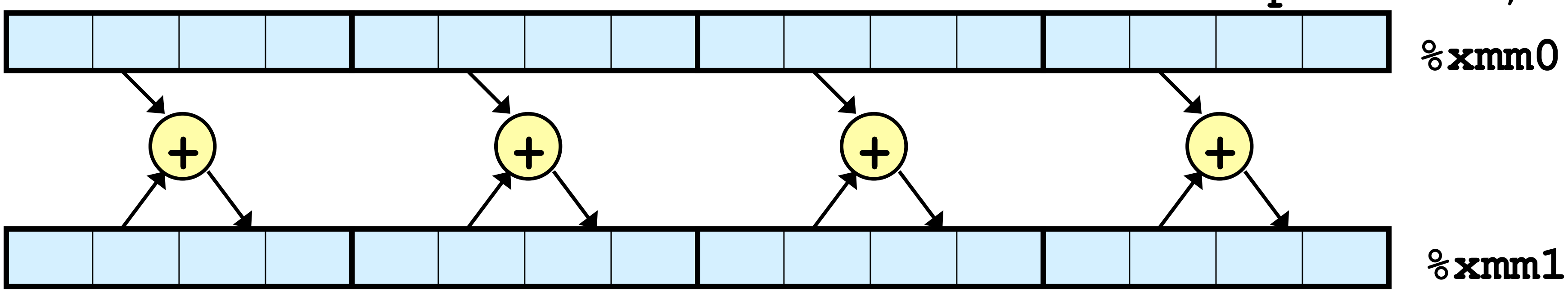
- Scalar Operations: Single Precision

`addss %xmm0,%xmm1`



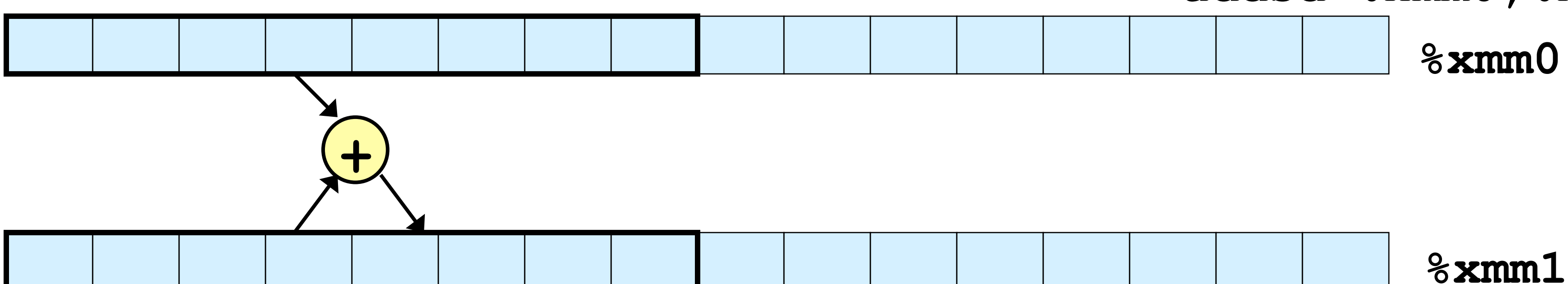
- SIMD Operations: Single Precision

`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`**
    - ***`Unordered COMpare (I?) Scalar (Single/Double)`***
    - ***`Unordered: allows <, >, ==, unordered (NaN args)`***
  - Set condition codes CF, ZF, and PF (parity flag – set if NaN arg)
- **Using constant values**
  - Set XMM0 register to 0 with instruction **`xorpd %xmm0, %xmm0`**
  - Others loaded from memory

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**
  - Data held and operated on in XMM registers