MACHINE-LEVEL PROGRAMMING IV: DATA

CS 045

Computer Organization and Architecture

Prof. Donald J. Patterson

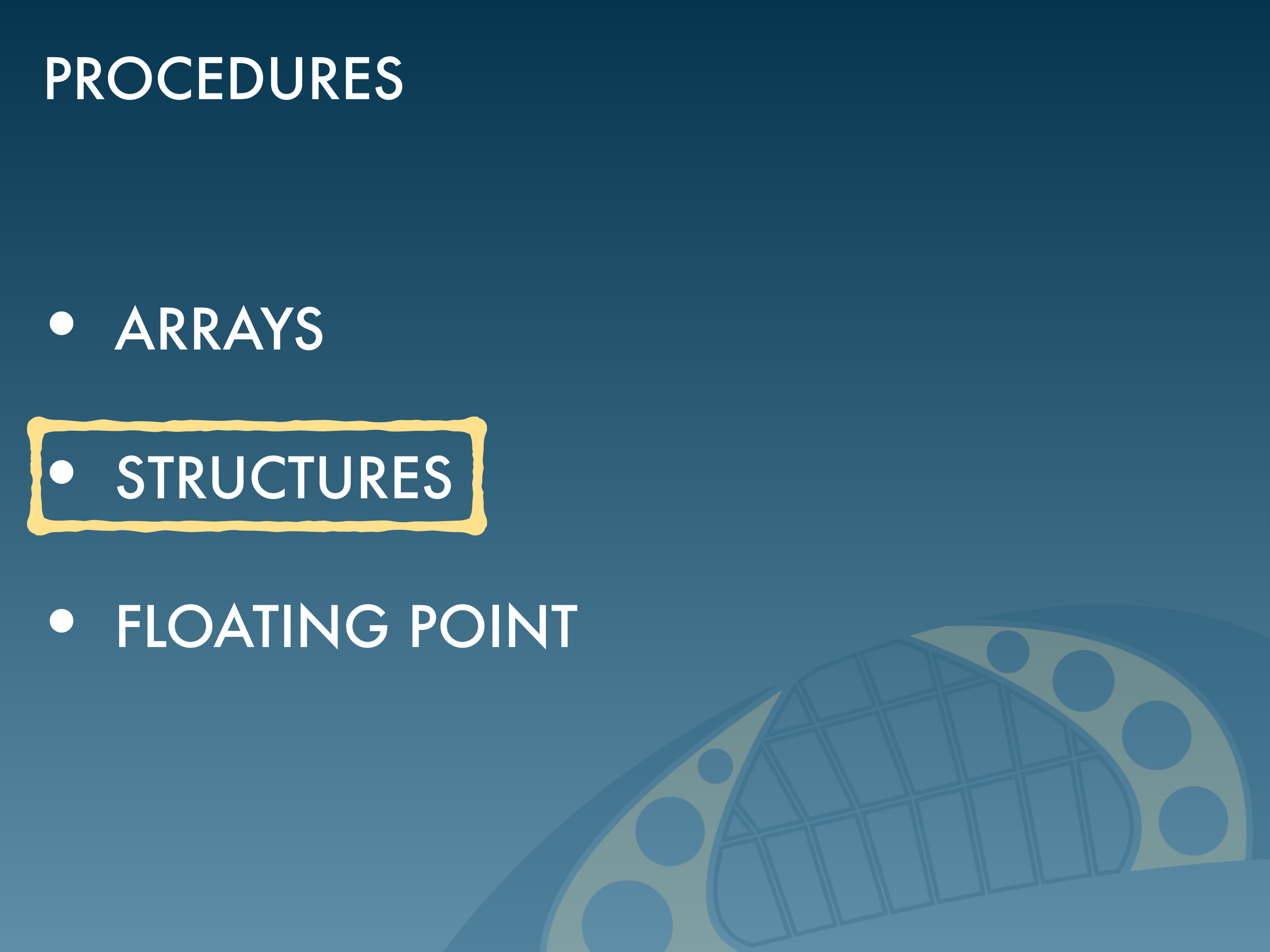
Adapted from Bryant and O'Hallaron,
Computer Systems:
A Programmer's Perspective, Third Edition

PROCEDURES

- ARRAYS
- STRUCTURES
- FLOATING POINT

PROCEDURES

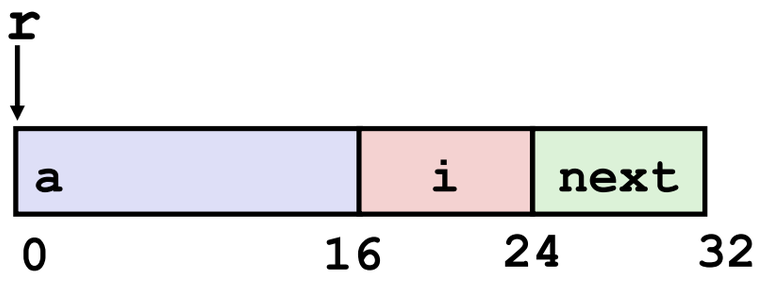
- ARRAYS
- STRUCTURES
- FLOATING POINT



DATA

STRUCTURE REPRESENTATION

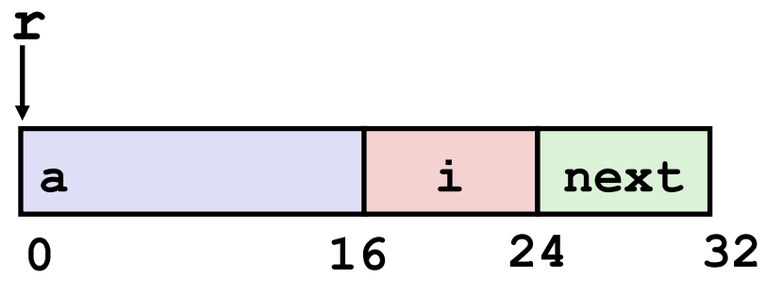
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



DATA

STRUCTURE REPRESENTATION

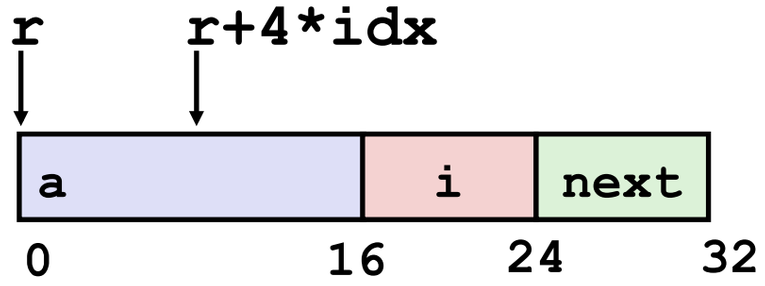
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- 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;
};
```



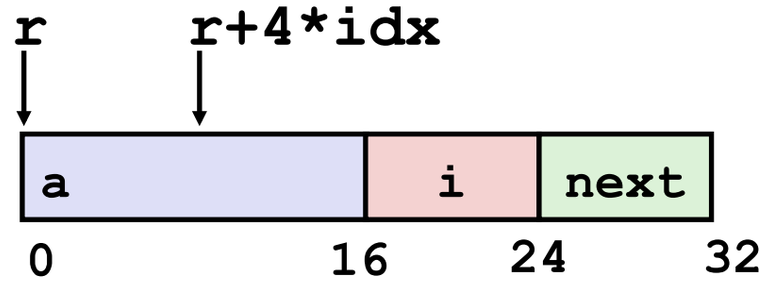

```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &r->a[idx];
}
```

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

DATA

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];
}
```

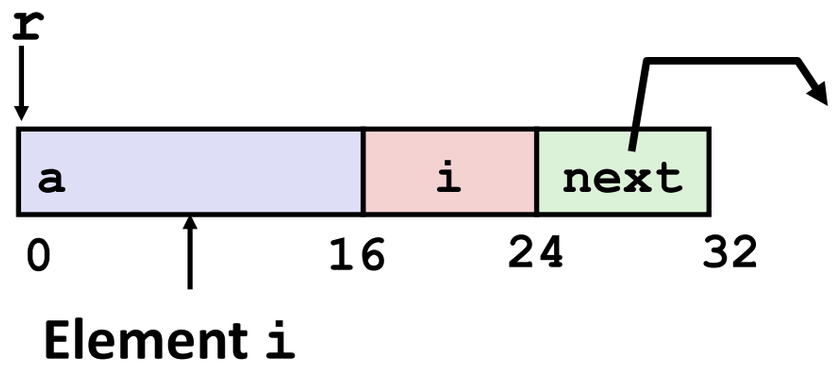
```
# r in %rdi, idx in %rsi
leaq  (%rdi,%rsi,4), %rax
ret
```

DATA

FOLLOWING LINKED LIST

```
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;
  }
}
```

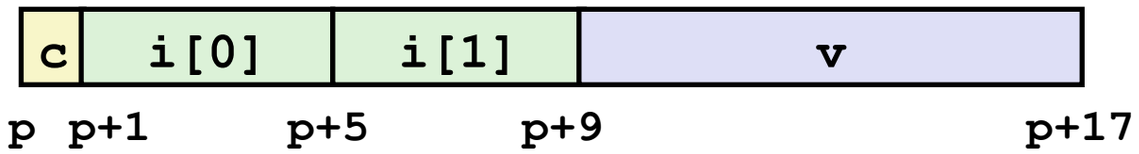


Register	Value
%rdi	r
%rsi	val

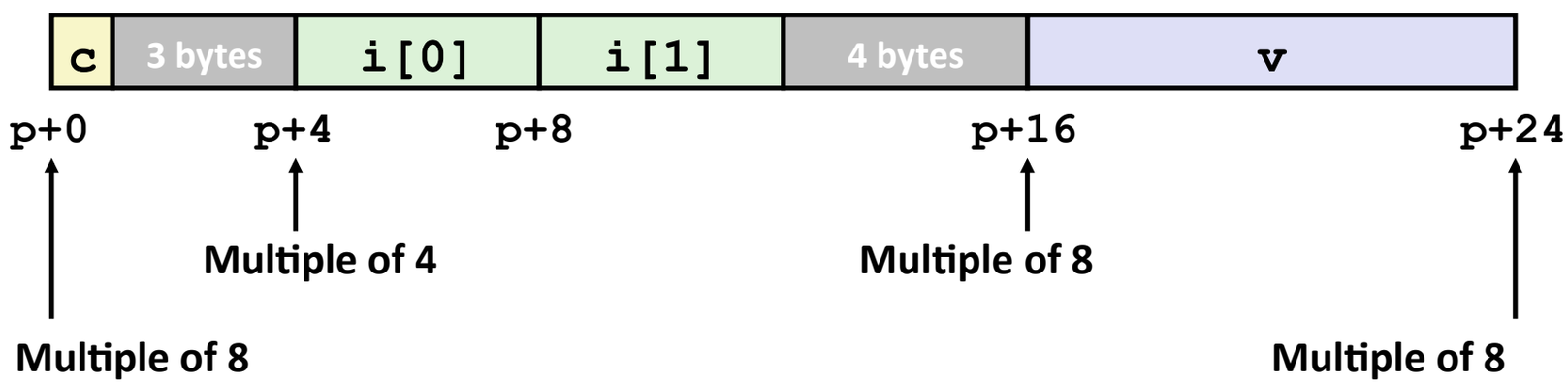
```
.L11:                                  # loop:
  movslq  16(%rdi), %rax               #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)          #   M[r+4*i] = val
  movq    24(%rdi), %rdi               #   r = M[r+24]
  testq   %rdi, %rdi                   #   Test r
  jne     .L11                 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

DATA

STRUCTURES & ALIGNMENT

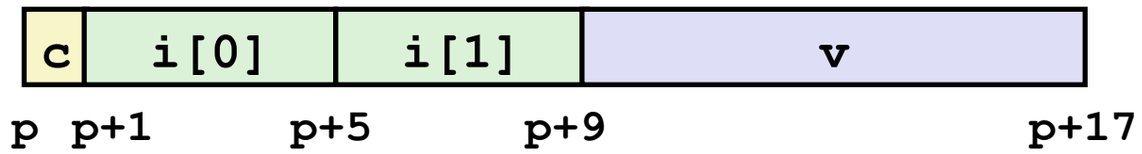



```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



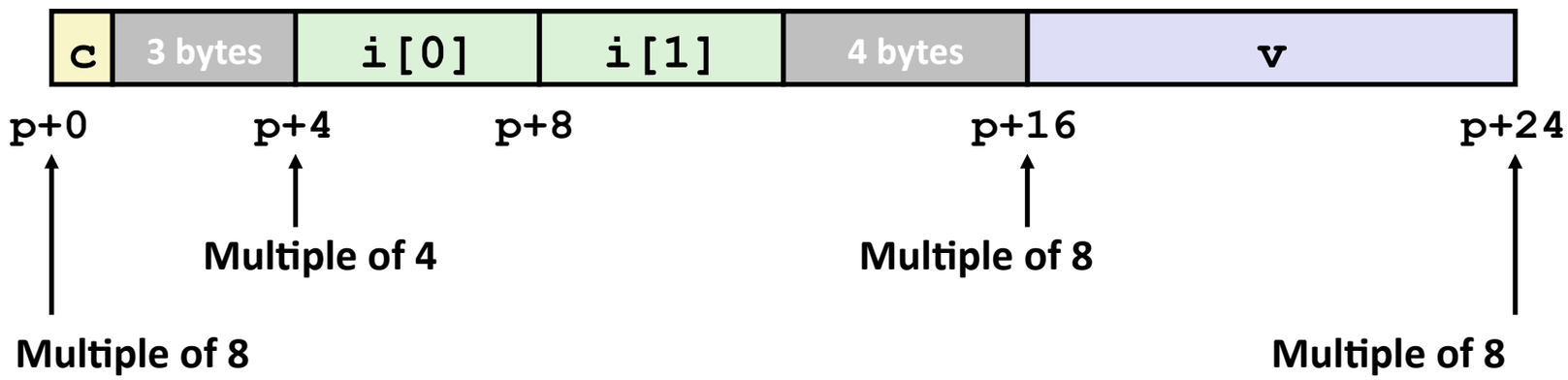
STRUCTURES & ALIGNMENT

- 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



DATA

ALIGNMENT PRINCIPLES

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

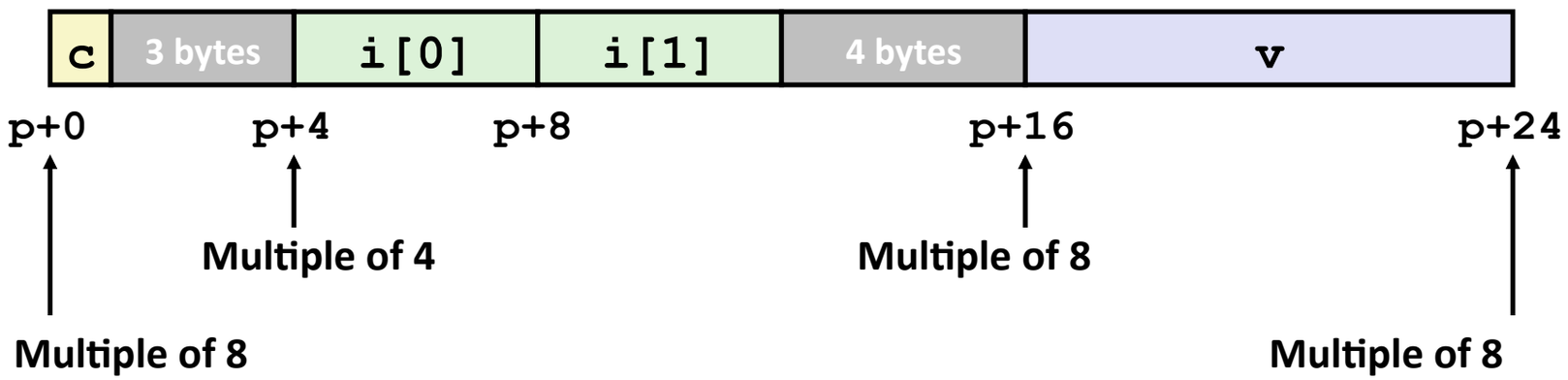
SPECIFIC CASES OF ALIGNMENT (X86-64)

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

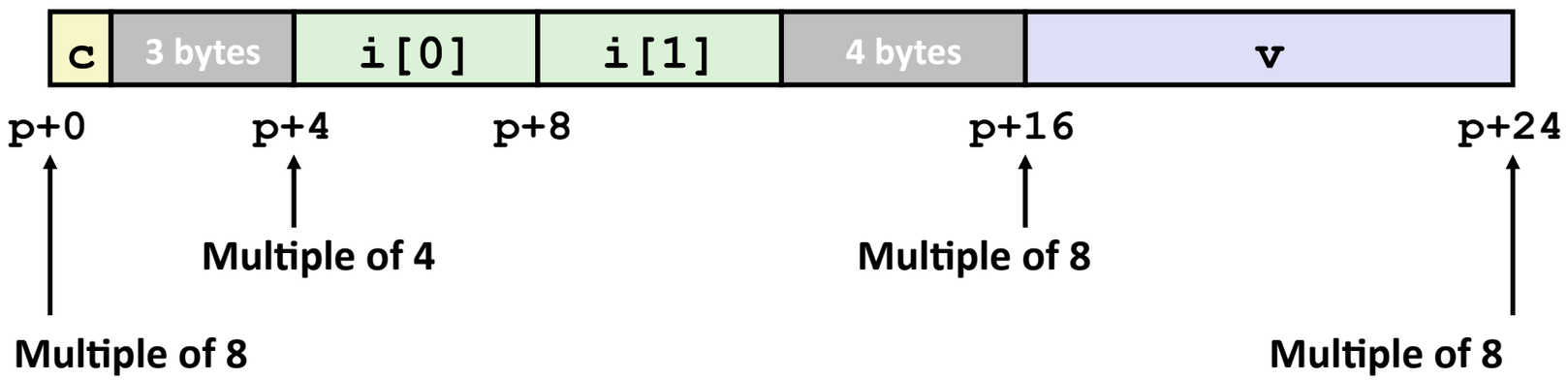
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



SATISFYING ALIGNMENT WITH STRUCTURES

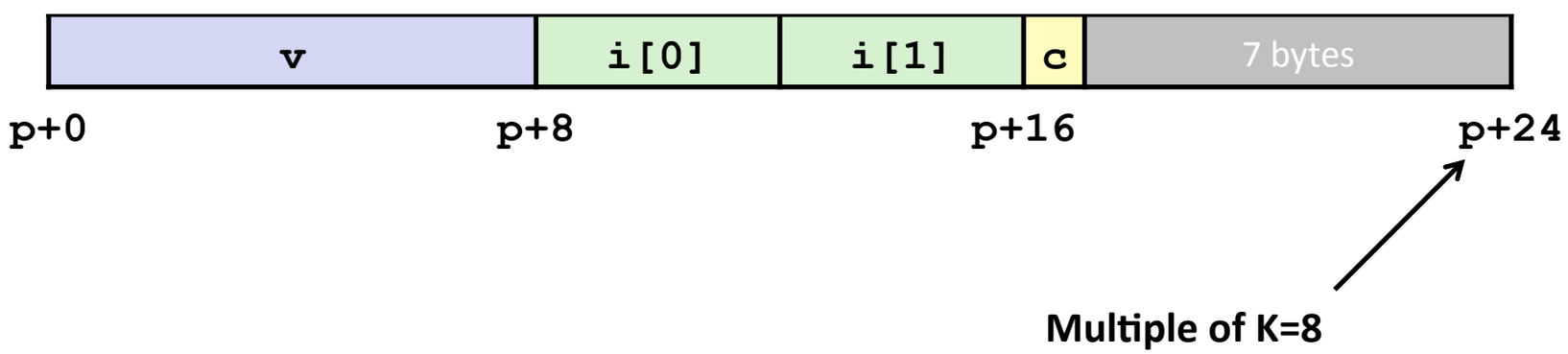
- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement K
    - K = Largest alignment of any element
  - Initial address & structure length must be multiples of K
- Example:
  - K = 8, due to double element

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



MEETING OVERALL ALIGNMENT REQUIREMENT

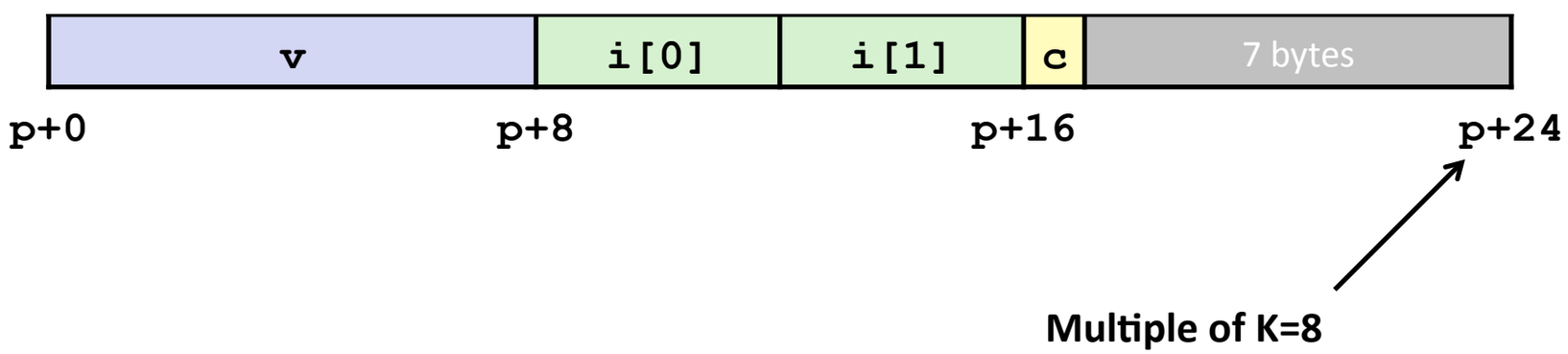
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



MEETING OVERALL ALIGNMENT REQUIREMENT

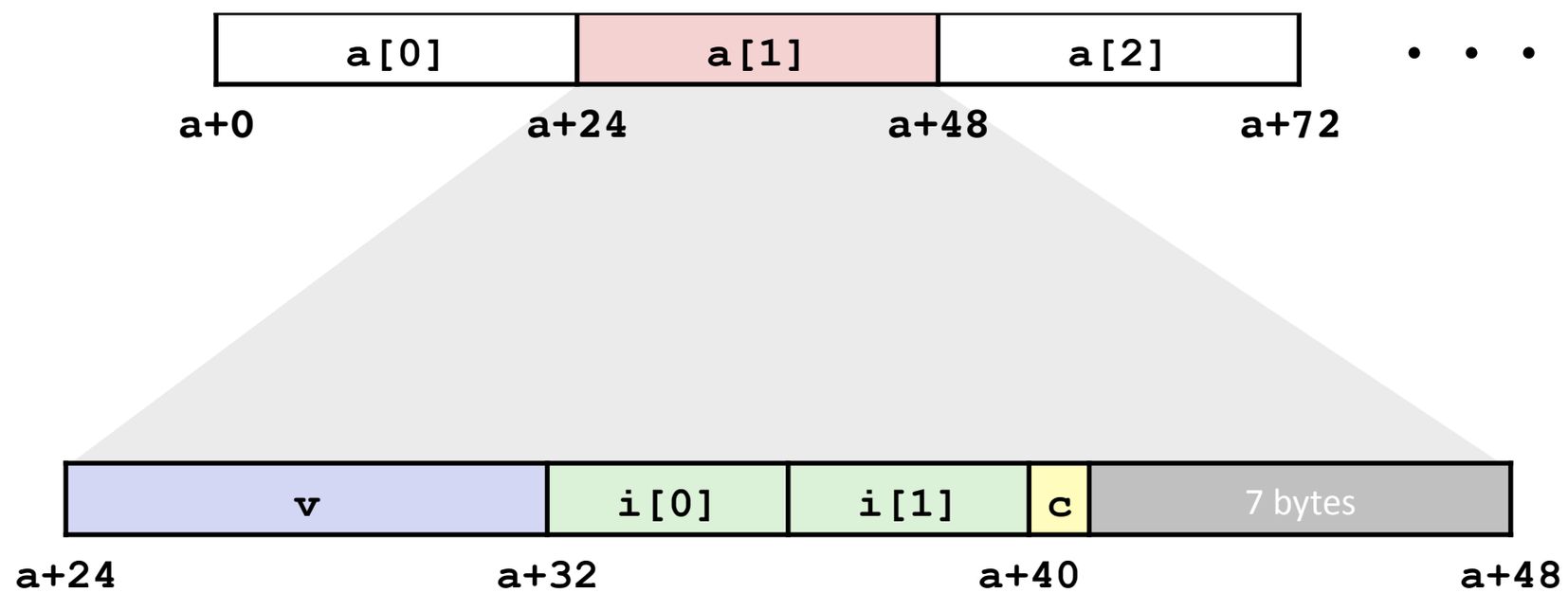
- For largest alignment requirement K
- Overall structure must be multiple of K

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



ARRAYS OF STRUCTURES

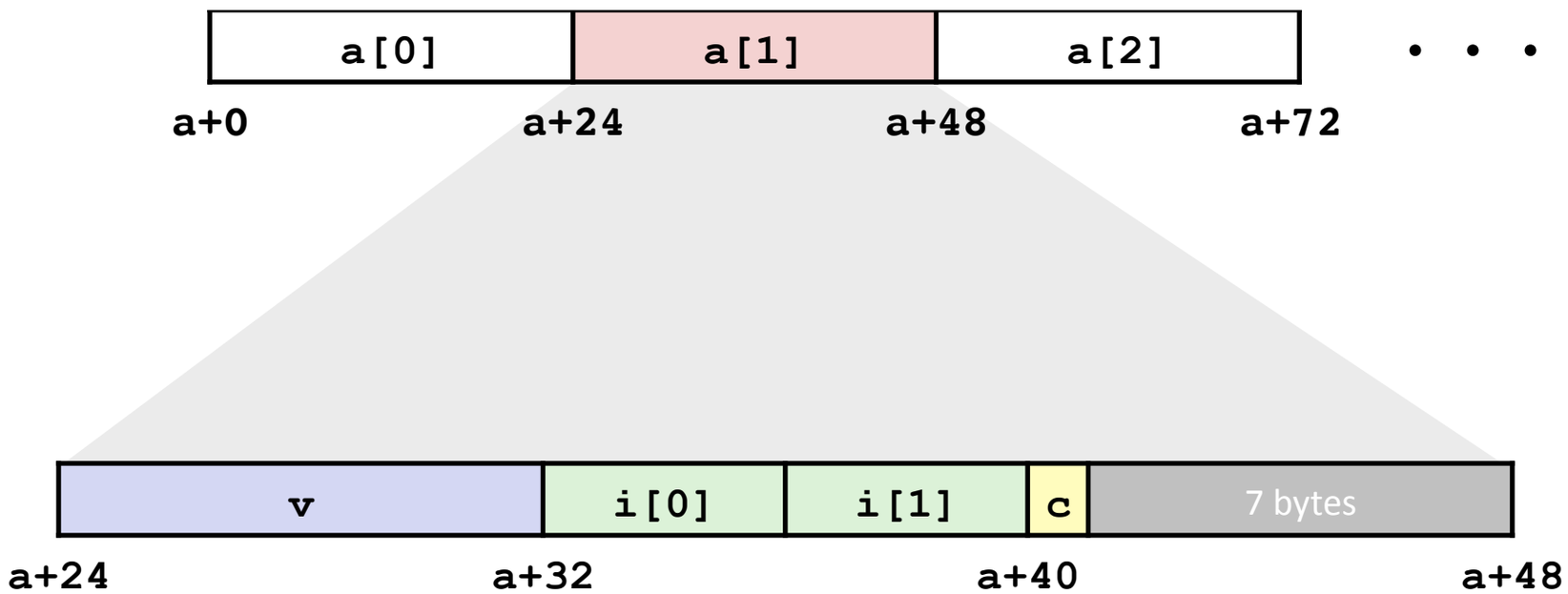
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



ARRAYS OF STRUCTURES

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

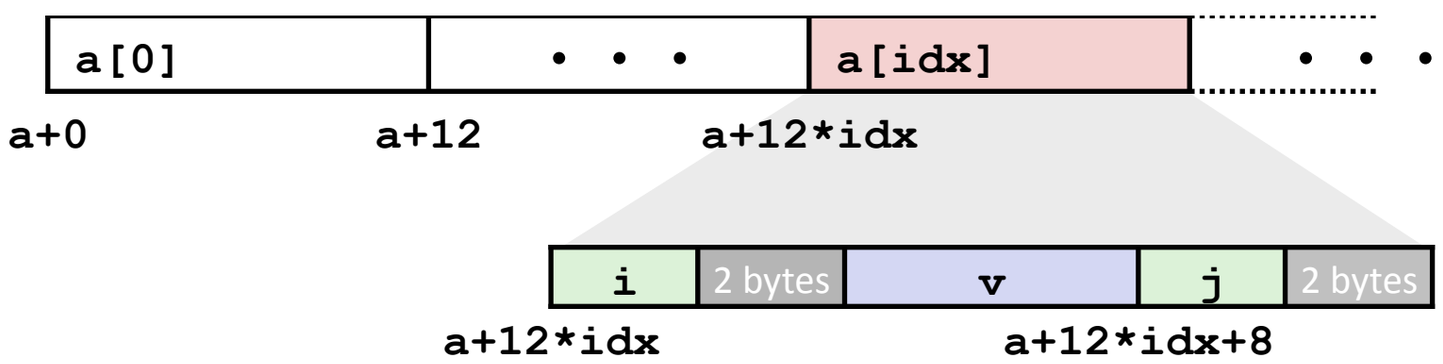
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



DATA

ACCESSING ARRAY ELEMENTS

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



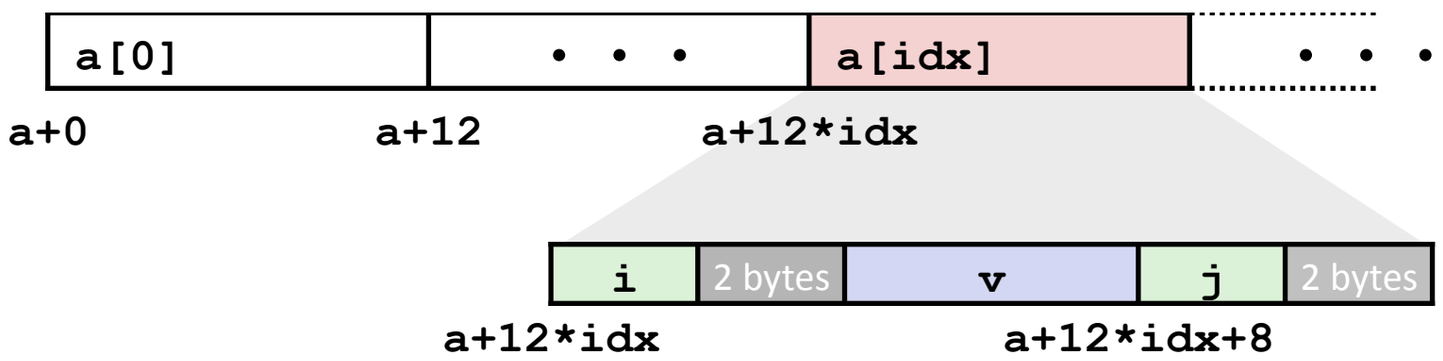

```
short get_j(int idx)
{
  return a[idx].j;
}
```

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

ACCESSING ARRAY ELEMENTS

- 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

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



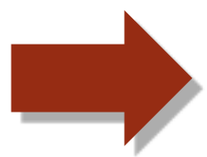

```
short get_j(int idx)
{
  return a[idx].j;
}
```

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

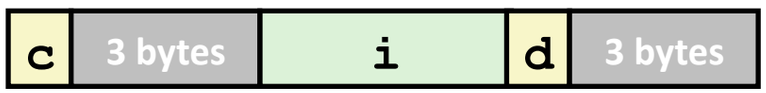
DATA

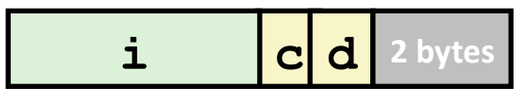
SAVING SPACE

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



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



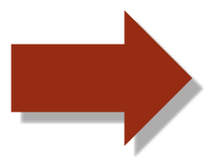


DATA

SAVING SPACE

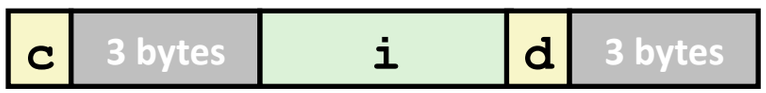
- Put large data types first

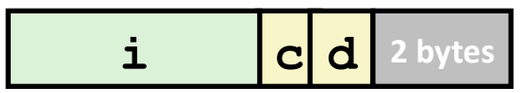
```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```



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

- Effect (K=4)





WORK IT OUT

WORK IT OUT

WHAT ARE M & N?

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

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

```
0000000000400474 <sum_element>:
  400474:       48 8d 04 7f             lea    (%rdi,%rdi,2),%rax
  400478:       48 8d 04 46             lea    (%rsi,%rax,2),%rax
  40047c:       48 8d 14 b7             lea    (%rdi,%rsi,4),%rdx
  400480:       48 8b 04 c5 60 08 60    mov    0x600860(,%rax,8),%rax
  400487:       00
  400488:       48 2b 04 d5 20 09 60    sub    0x600920(,%rdx,8),%rax
  40048f:       00
  400490:       c3                      retq
```

WORK IT OUT

WHAT ARE M & N?

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

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

```
0000000000400474 <sum_element>:
  400474:       48 8d 04 7f             lea    (%rdi,%rdi,2),%rax
  400478:       48 8d 04 46             lea    (%rsi,%rax,2),%rax
  40047c:       48 8d 14 b7             lea    (%rdi,%rsi,4),%rdx
  400480:       48 8b 04 c5 60 08 60    mov    0x600860(,%rax,8),%rax
  400487:       00
  400488:       48 2b 04 d5 20 09 60    sub    0x600920(,%rdx,8),%rax
  40048f:       00
  400490:       c3                      retq
```

WORK IT OUT

WHAT ARE M & N?

$M = 4$

$N = 6$

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

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

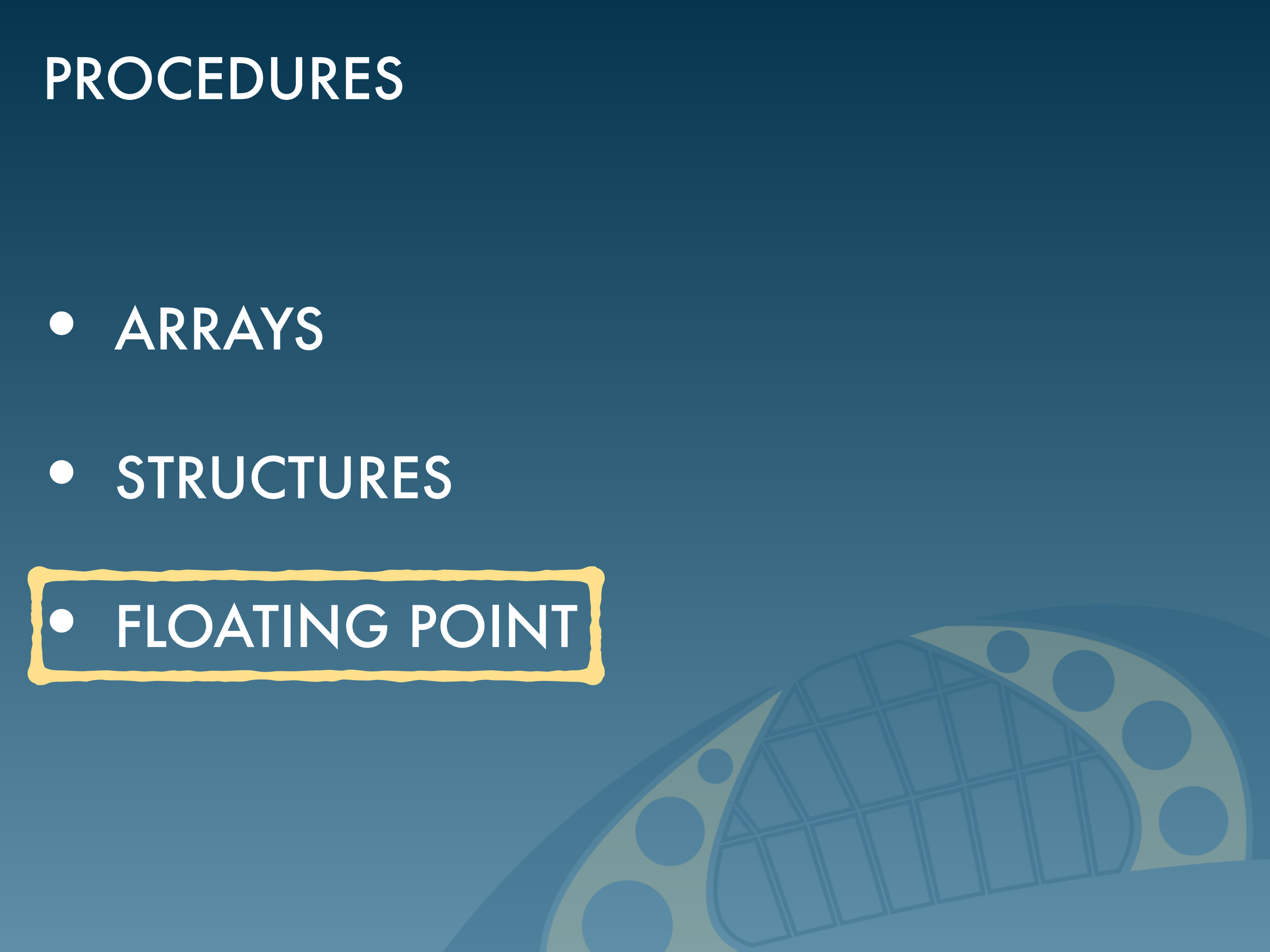
```
0000000000400474 <sum_element>:
  400474:       48 8d 04 7f             lea    (%rdi,%rdi,2),%rax
  400478:       48 8d 04 46             lea    (%rsi,%rax,2),%rax
  40047c:       48 8d 14 b7             lea    (%rdi,%rsi,4),%rdx
  400480:       48 8b 04 c5 60 08 60    mov    0x600860(,%rax,8),%rax
  400487:       00
  400488:       48 2b 04 d5 20 09 60    sub    0x600920(,%rdx,8),%rax
  40048f:       00
  400490:       c3                      retq
```

PROCEDURES

- ARRAYS
- STRUCTURES
- FLOATING POINT

PROCEDURES

- ARRAYS
- STRUCTURES
- FLOATING POINT



FLOATING POINT

BACKGROUND

- History
  - x87 FP
    - Legacy, very ugly
    - separate chip
  - SSE FP
    - Special case use of vector instructions
    - integrated into main chip package
- AVX FP
  - Newest version
  - Similar to SSE
  - Documented in book

Image Source: Bumper12

FLOATING POINT

BACKGROUND

- History
  - x87 FP
    - Legacy, very ugly
    - separate chip
  - SSE FP
    - Special case use of vector instructions
    - integrated into main chip package
- AVX FP
  - Newest version
  - Similar to SSE
  - Documented in book

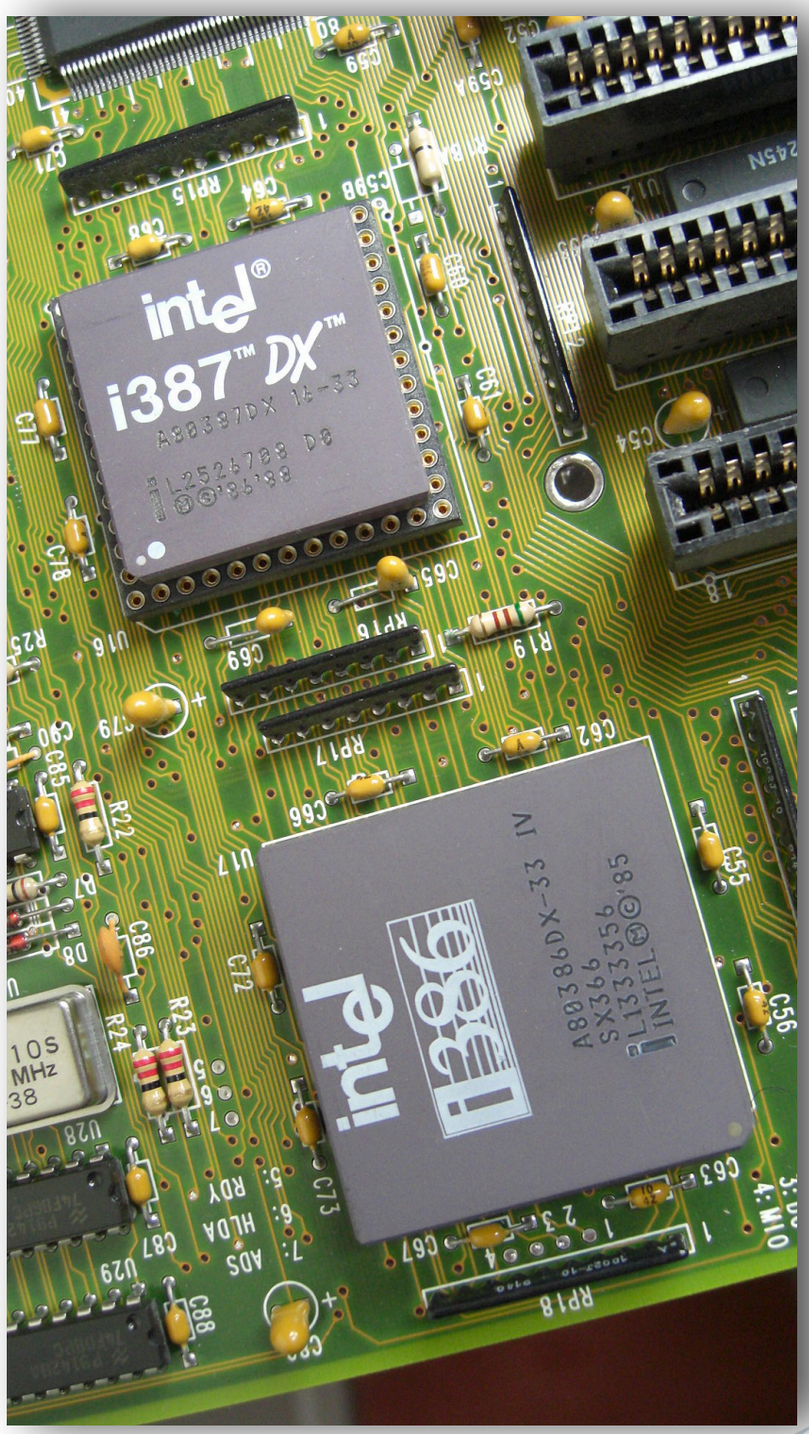


Image Source: Bumper12

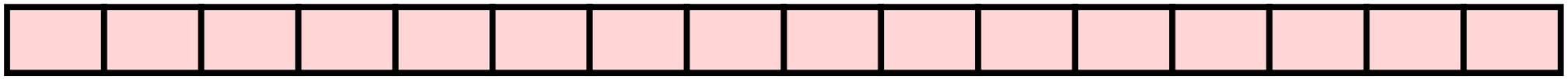
PROGRAMMING WITH SSE3

- 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

FLOATING POINT

PROGRAMMING WITH SSE3

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

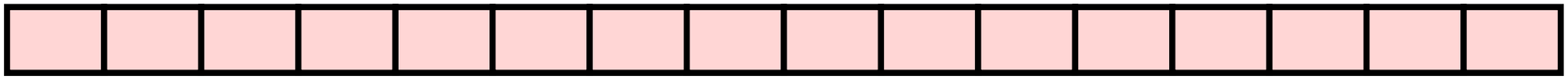


  - 8 16-bit integers
  - 4 32-bit integers
  - 4 single-precision floats
  - 2 double-precision floats
  - 1 single-precision float
  - 1 double-precision float

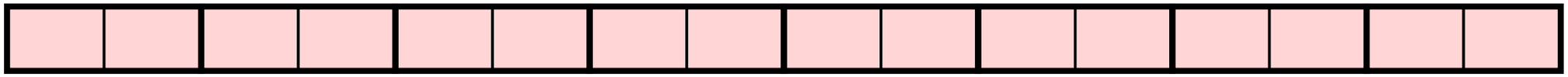
FLOATING POINT

PROGRAMMING WITH SSE3

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



  - 8 16-bit integers

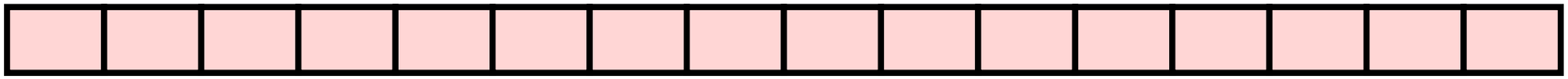


  - 4 32-bit integers
  - 4 single-precision floats
  - 2 double-precision floats
  - 1 single-precision float
  - 1 double-precision float

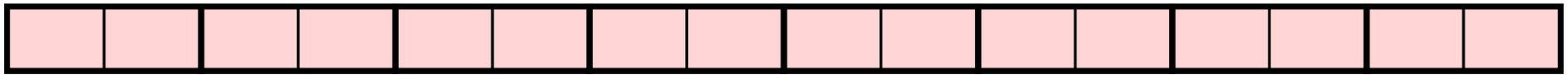
FLOATING POINT

PROGRAMMING WITH SSE3

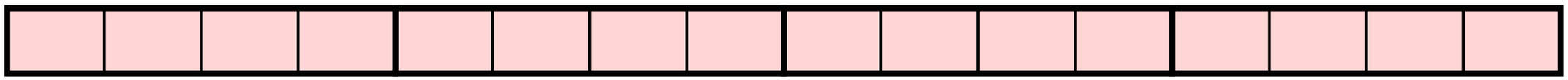
- XMM Registers
  - 16 total, each 16 bytes
  - 16 single-byte integers



  - 8 16-bit integers



  - 4 32-bit integers

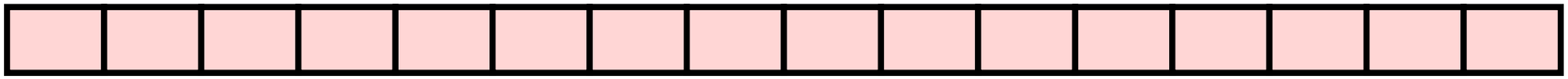


  - 4 single-precision floats
  - 2 double-precision floats
  - 1 single-precision float
  - 1 double-precision float

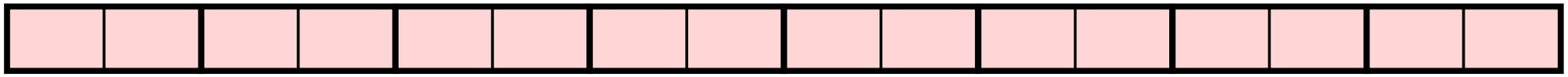
FLOATING POINT

PROGRAMMING WITH SSE3

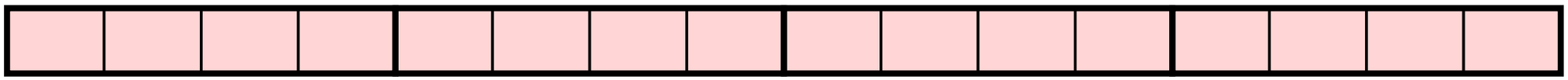
- XMM Registers
  - 16 total, each 16 bytes
  - 16 single-byte integers



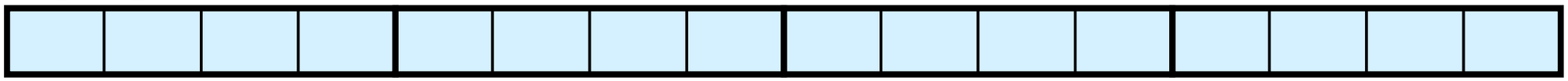
  - 8 16-bit integers



  - 4 32-bit integers



  - 4 single-precision floats

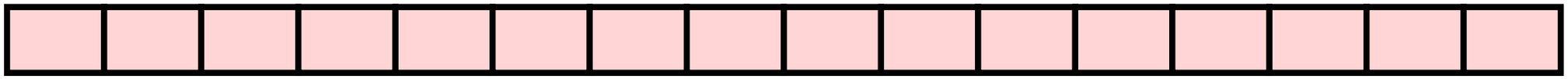


  - 2 double-precision floats
  - 1 single-precision float
  - 1 double-precision float

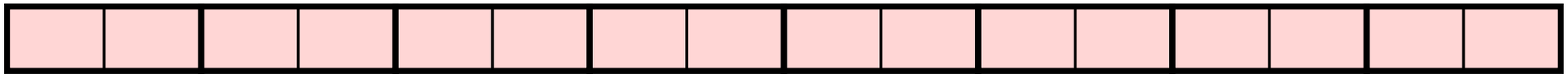
FLOATING POINT

PROGRAMMING WITH SSE3

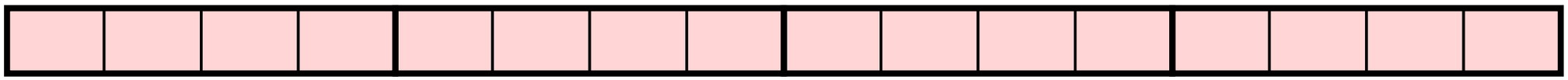
- XMM Registers
  - 16 total, each 16 bytes
  - 16 single-byte integers



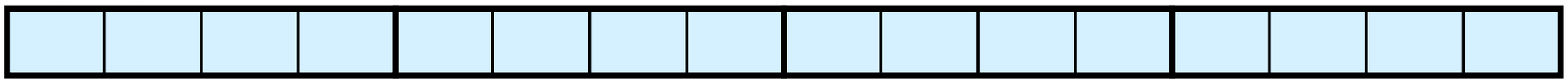
  - 8 16-bit integers



  - 4 32-bit integers



  - 4 single-precision floats



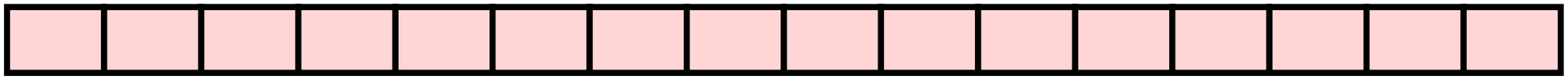
  - 2 double-precision floats

  - 1 single-precision float
  - 1 double-precision float

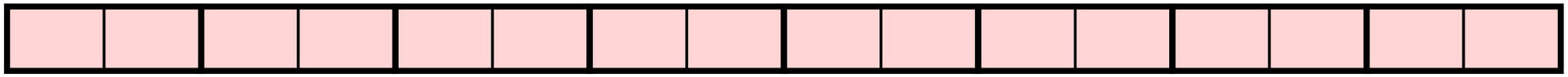
FLOATING POINT

PROGRAMMING WITH SSE3

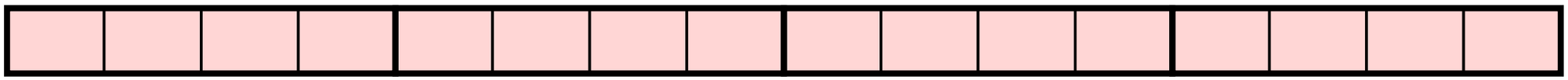
- XMM Registers
  - 16 total, each 16 bytes
  - 16 single-byte integers



  - 8 16-bit integers



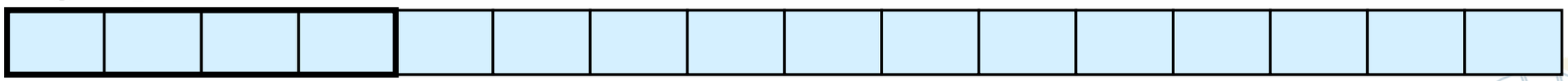
  - 4 32-bit integers



  - 4 single-precision floats

  - 2 double-precision floats

  - 1 single-precision float

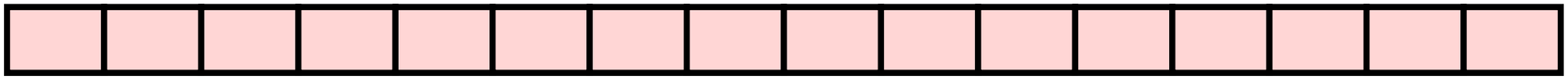


  - 1 double-precision float

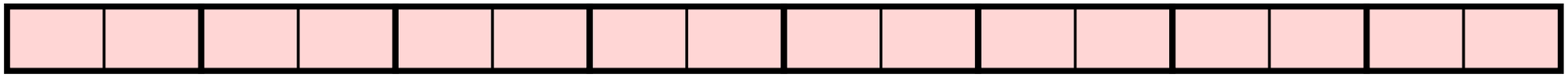
FLOATING POINT

PROGRAMMING WITH SSE3

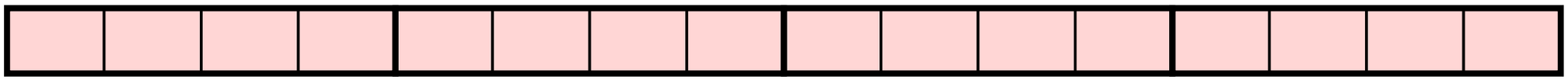
- XMM Registers
  - 16 total, each 16 bytes
  - 16 single-byte integers



  - 8 16-bit integers



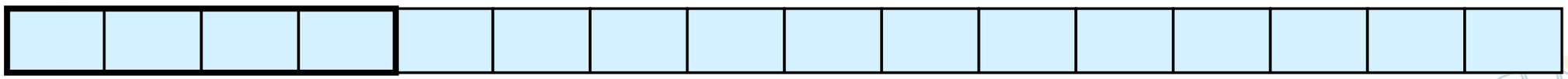
  - 4 32-bit integers



  - 4 single-precision floats

  - 2 double-precision floats

  - 1 single-precision float



  - 1 double-precision float

FLOATING POINT

SCALAR & SIMD OPERATIONS

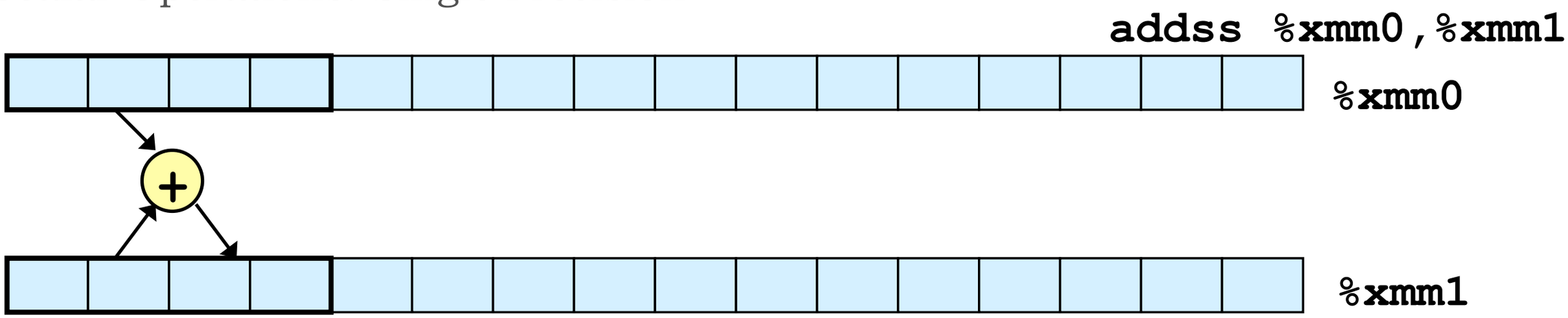
- Scalar vs Vector

- SIMD
  - Single Instruction Multiple Data

FLOATING POINT

SCALAR & SIMD OPERATIONS

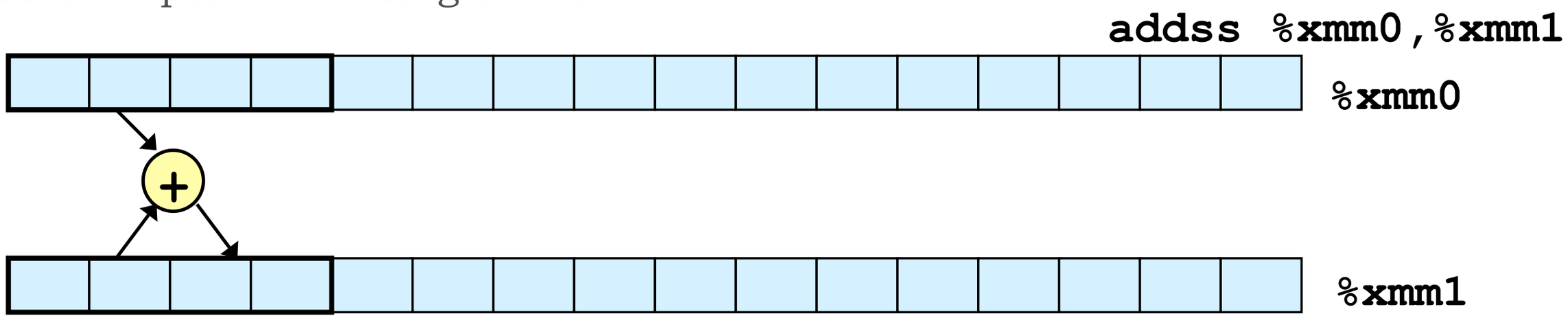
- Scalar Operations: Single Precision



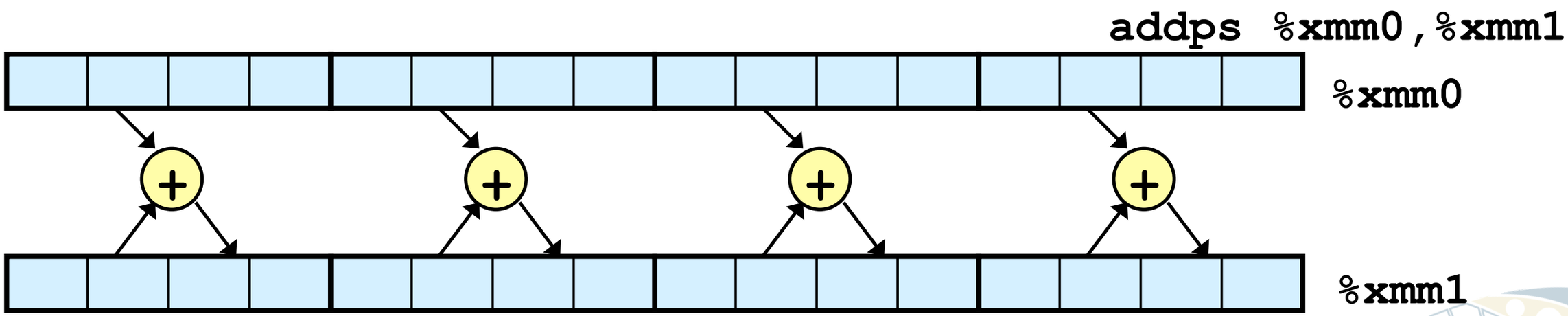
FLOATING POINT

SCALAR & SIMD OPERATIONS

- Scalar Operations: Single Precision



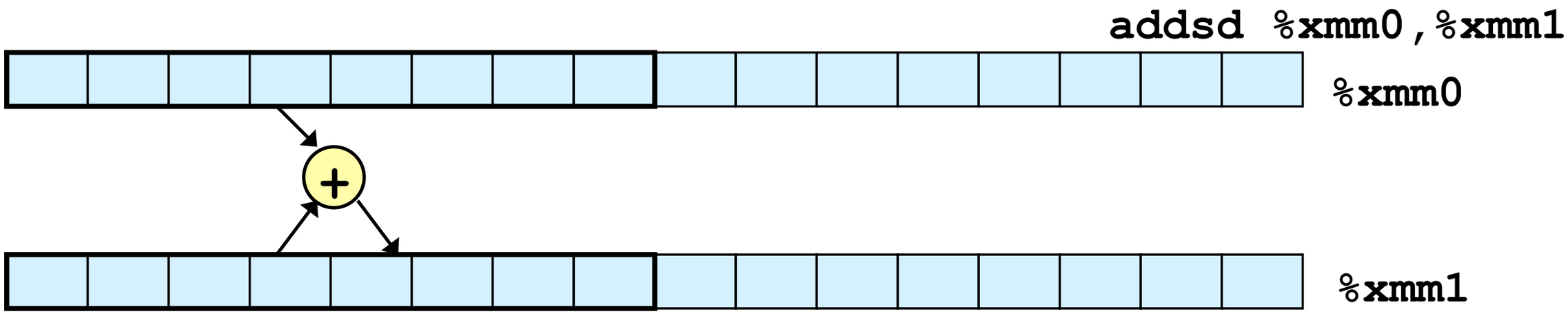
- SIMD Operations: Single Precision



FLOATING POINT

SCALAR & SIMD OPERATIONS

- Scalar Operations: Double Precision



FLOATING POINT

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
```

FLOATING POINT

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
  - No immediate values for floating point code
  - Set XMM0 register to 0 with instruction xorpd %xmm0, %xmm0
  - Others loaded from memory

PROCEDURES

- ARRAYS
- STRUCTURES
- FLOATING POINT
- SUMMARY

PROCEDURES

- ARRAYS
- STRUCTURES
- FLOATING POINT
- SUMMARY

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

WORK IT OUT

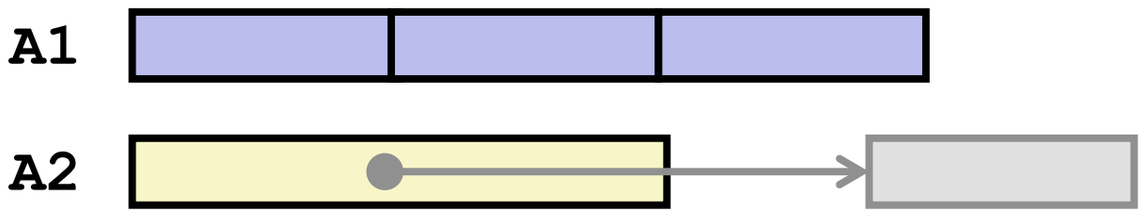
UNDERSTANDING POINTERS & ARRAYS #1

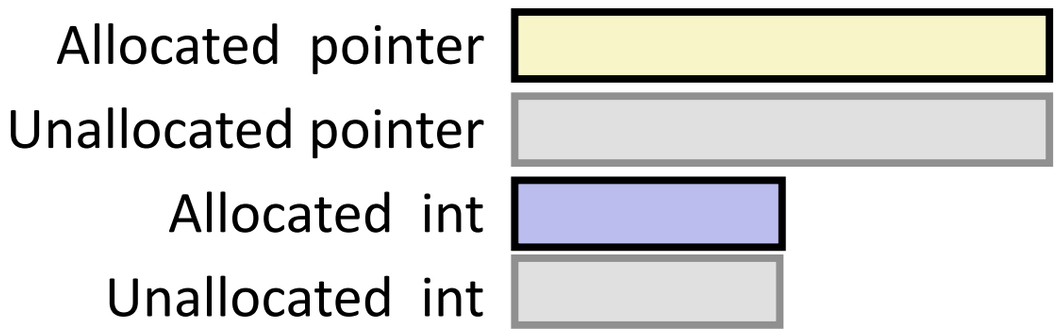
Decl	*An*			**An*		
	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`						
`int *A2`						

- **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`**

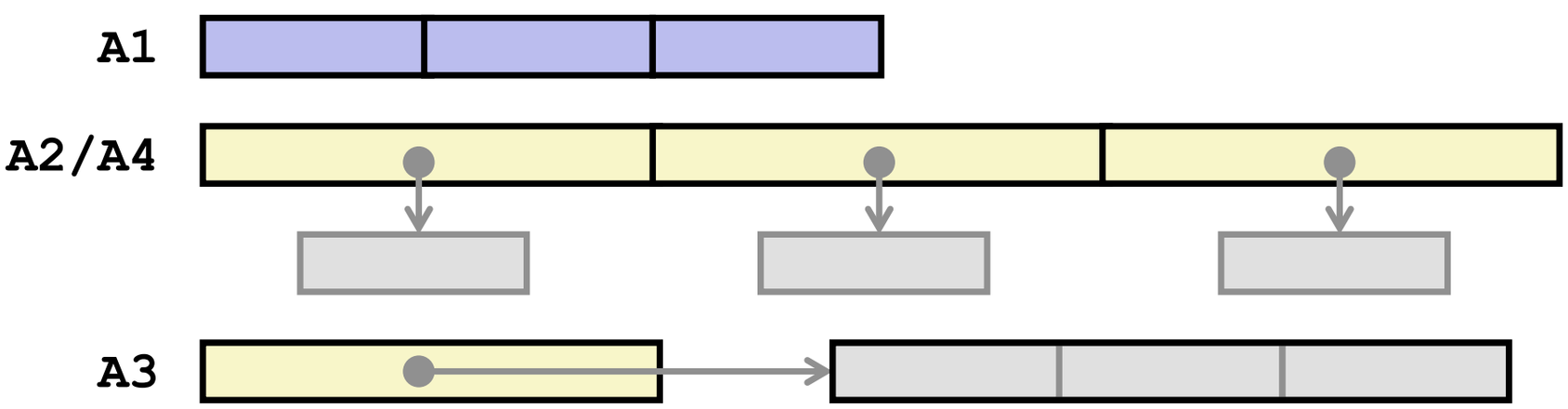
UNDERSTANDING POINTERS & ARRAYS #2

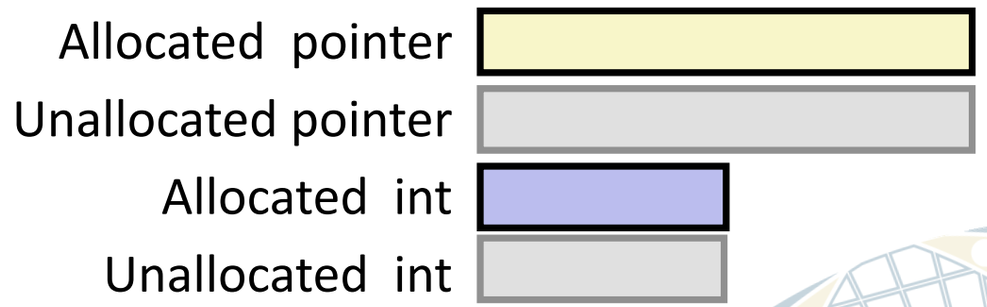
Decl	*An*			**An*			***An*		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`									
`int *A2[3]`									
`int (*A3)[3]`									
`int (*A4[3])`									

- **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





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

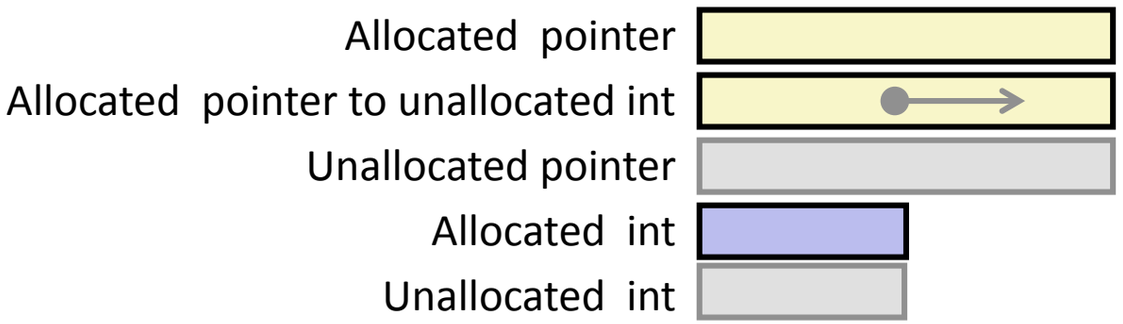
UNDERSTANDING POINTERS & ARRAYS #3

Decl	*An*			**An*			***An*		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3][5]`									
`int *A2[3][5]`									
`int (*A3)[3][5]`									
`int *(A4[3][5])`									
`int (*A5[3])[5]`									

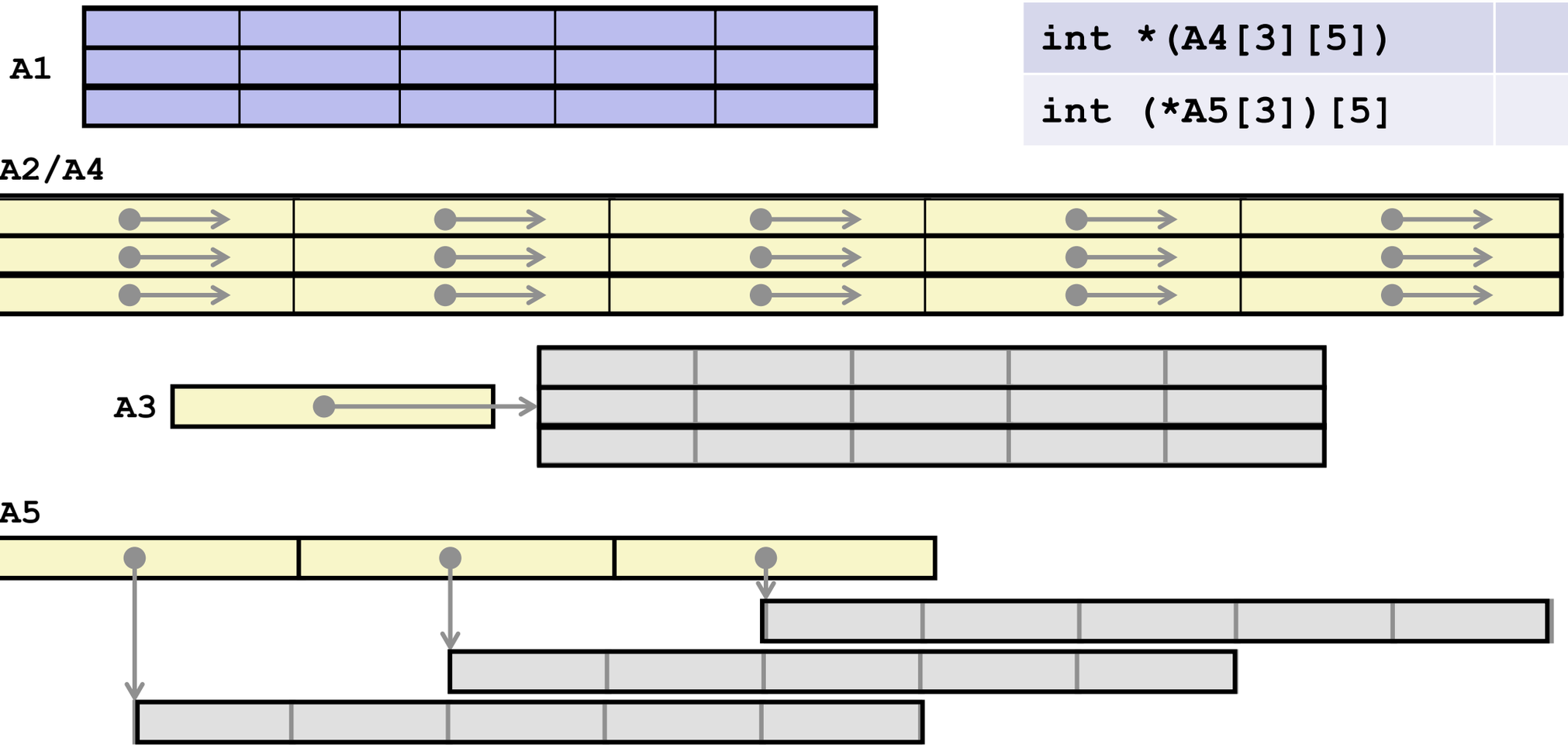
Decl	****An*		
	Cmp	Bad	Size
`int A1[3][5]`			
`int *A2[3][5]`			
`int (*A3)[3][5]`			
`int *(A4[3][5])`			
`int (*A5[3])[5]`			

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

UNDERSTANDING POINTERS & ARRAYS #3



Decl	***An		
	Cmp	Bad	Size
`int A1[3][5]`			
`int *A2[3][5]`			
`int (*A3)[3][5]`			
`int *(A4[3][5])`			
`int (*A5[3])[5]`			



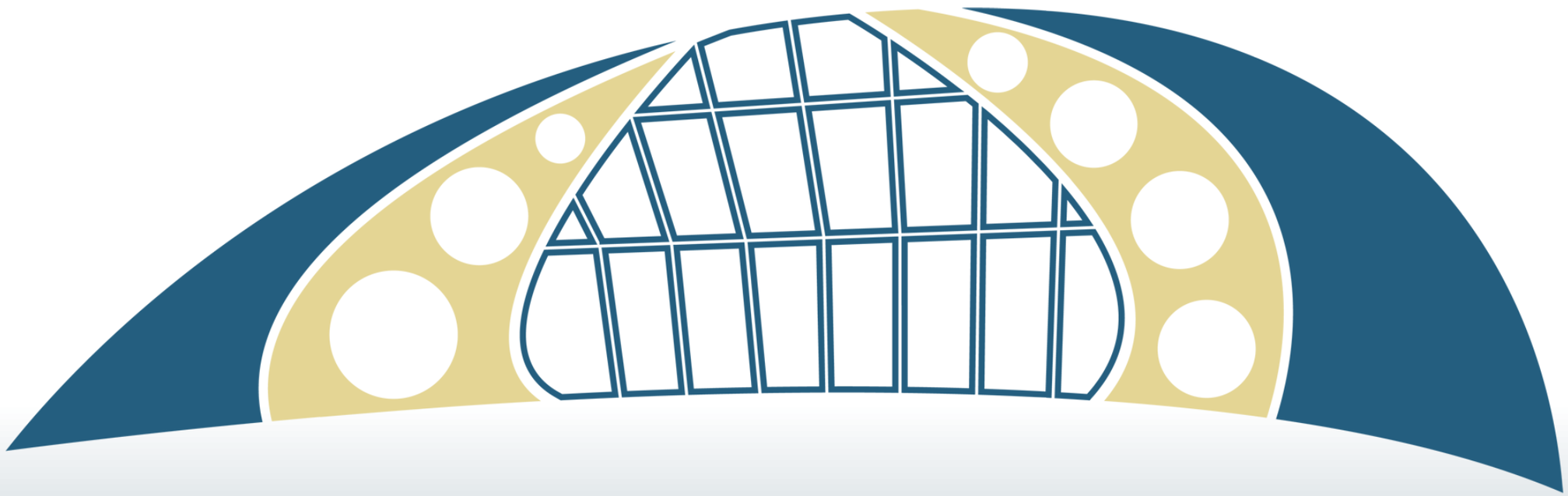
- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

UNDERSTANDING POINTERS & ARRAYS #3

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3][5]`	Y	N	60	Y	N	20	Y	N	4
`int *A2[3][5]`	Y	N	120	Y	N	40	Y	N	8
`int (*A3)[3][5]`	Y	N	8	Y	Y	60	Y	Y	20
`int *(A4[3][5])`	Y	N	120	Y	N	40	Y	N	8
`int (*A5[3])[5]`	Y	N	24	Y	N	8	Y	Y	20

Decl	***An		
	Cmp	Bad	Size
`int A1[3][5]`	N	-	-
`int *A2[3][5]`	Y	Y	4
`int (*A3)[3][5]`	Y	Y	4
`int *(A4[3][5])`	Y	Y	4
`int (*A5[3])[5]`	Y	Y	4

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



WESTMONT INSPIRED

COMPUTING LAB