Memory Organization and Addressing

CSCI 224 / ECE 317: Computer Architecture

Instructor:
Prof. Jason Fritts

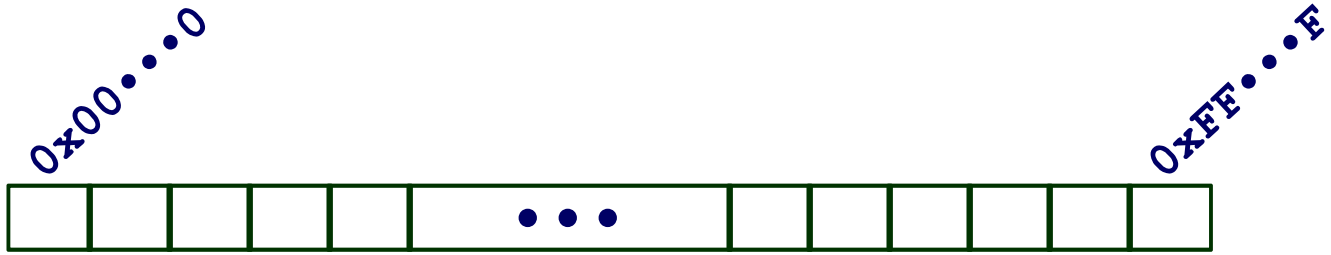
Slides adapted from Bryant & O'Hallaron's slides

Data Representation in Memory

- **Memory organization within a process**

- **Memory addressing and ordering of multi-byte data**
  - Addressing
  - Byte ordering
  - Arrays
  - Data structures
  - Ordering in arrays/structures vs. single multi-byte data elements

Recall: Basic Memory Organization

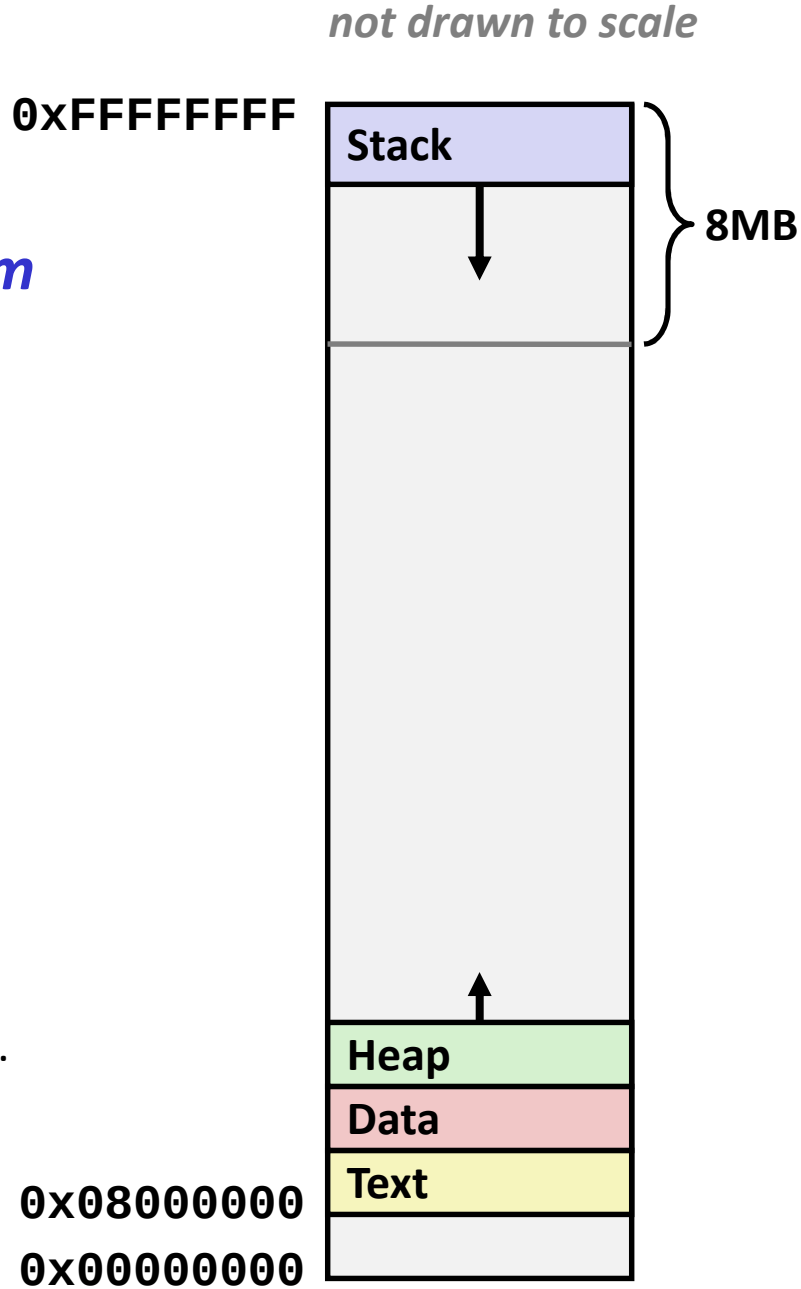


- **Byte-Addressable Memory**
  - Conceptually a very large array, with a unique address for each byte
  - Processor width determines address range:
    - 32-bit processor has 2^{32} unique addresses
    - 64-bit processor has 2^{64} unique addresses

- ***Where does a given process reside in memory?***
  - depends upon the perspective…
    - virtual memory: process can use most any virtual address
    - physical memory: location controlled by OS

Virtual Address Space for IA32 (x86) Linux

- ***All processes have the same uniform view of memory***
- **Stack**
  - Runtime stack (8MB limit)
  - E. g., local variables
- **Heap**
  - Dynamically allocated storage
  - When call *malloc(), calloc(), new()*
- **Data**
  - Statically allocated data
  - E.g., global variables, arrays, structures, etc.
- **Text**
  - Executable machine instructions
  - Read-only data

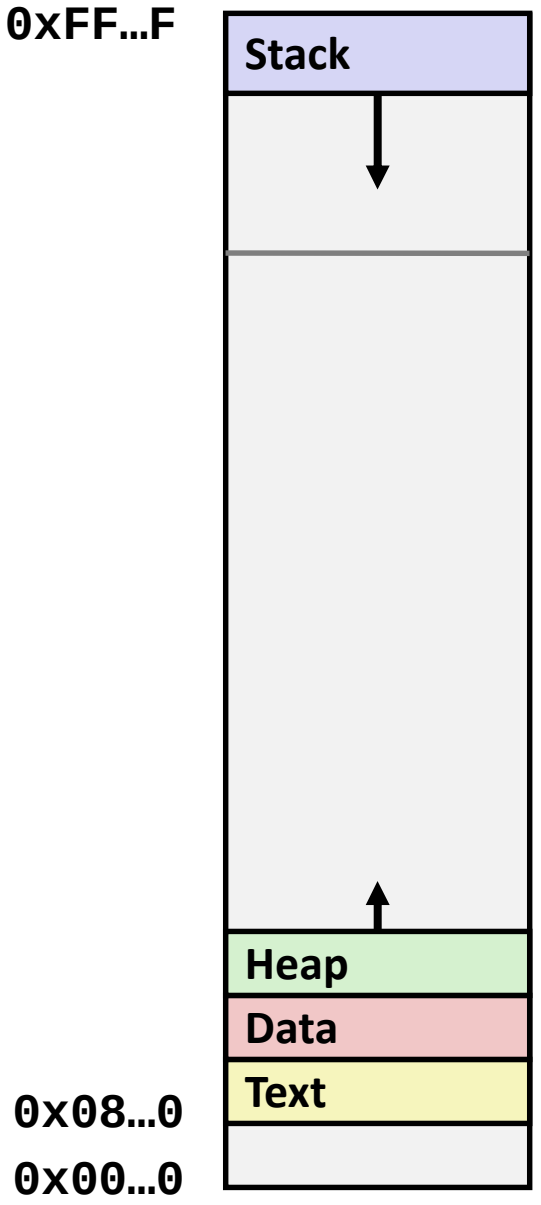


Memory Allocation Example

```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

Where does everything go?

not drawn to scale



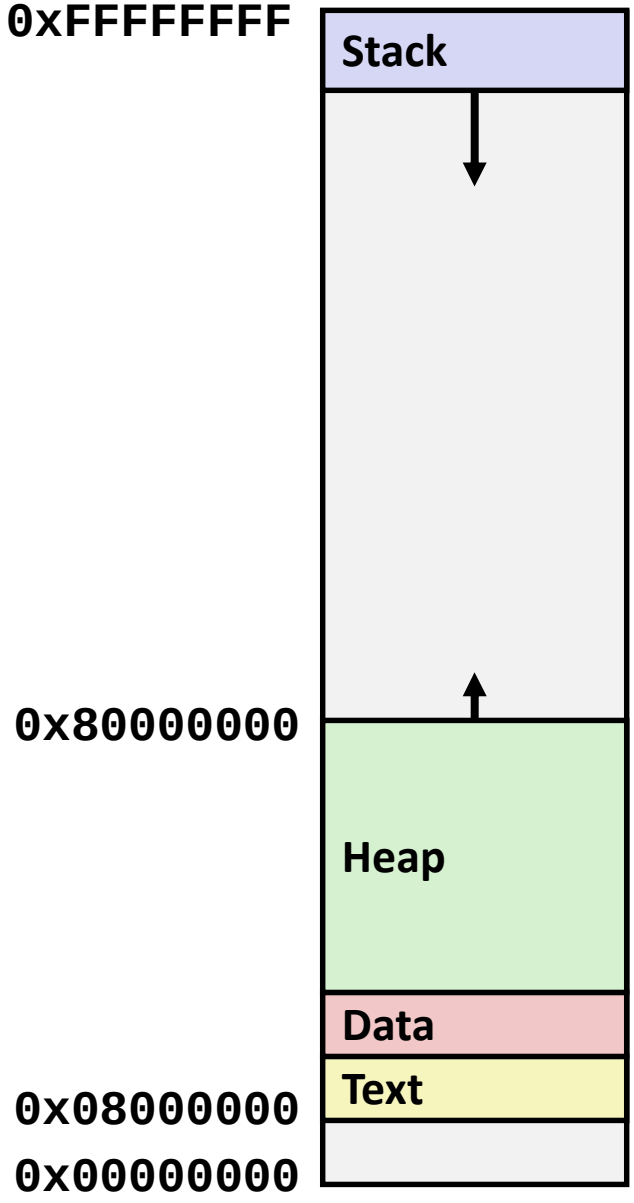
Addresses in IA32 (x86)

address range ~2^{32}

`$esp`	`0xffffbcd0`
`p3`	`0x65586008`
`p1`	`0x55585008`
`p4`	`0x1904a110`
`p2`	`0x1904a008`
`&p2`	`0x18049760`
`&beyond`	`0x08049744`
`big_array`	`0x18049780`
`huge_array`	`0x08049760`
`main()`	`0x080483c6`
`useless()`	`0x08049744`

`malloc()` is dynamically linked
address determined at runtime

not drawn to scale



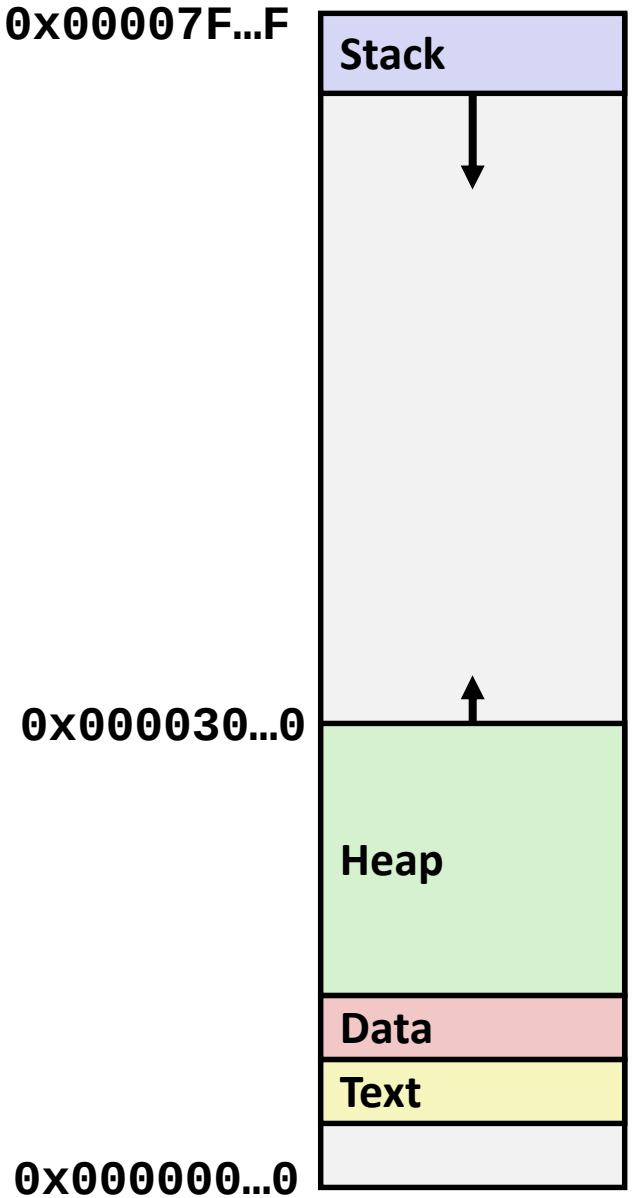
Addresses in x86-64

address range ~2^{47}

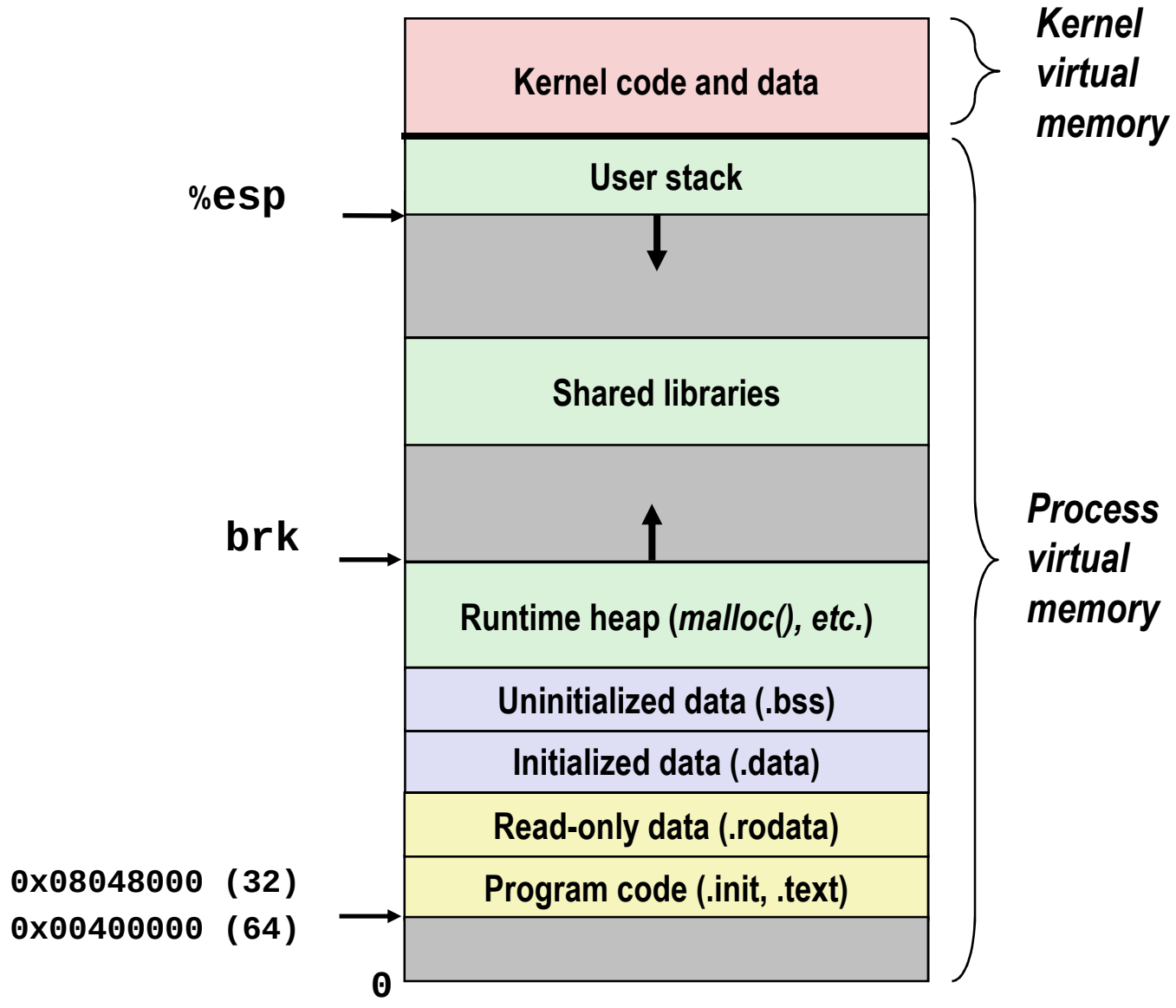
`$rsp`	`0x00007fffffff8d1f8`
`p3`	`0x00002aaabaadd010`
`p1`	`0x00002aaaaaadc010`
`p4`	`0x0000000011501120`
`p2`	`0x0000000011501010`
`&p2`	`0x0000000010500a60`
`&beyond`	`0x0000000000500a44`
`big_array`	`0x0000000010500a80`
`huge_array`	`0x0000000000500a50`
`main()`	`0x0000000000400510`
`useless()`	`0x0000000000400500`

`malloc()` is dynamically linked
address determined at runtime

not drawn to scale



Detailed Virtual Address Space for a Linux Process



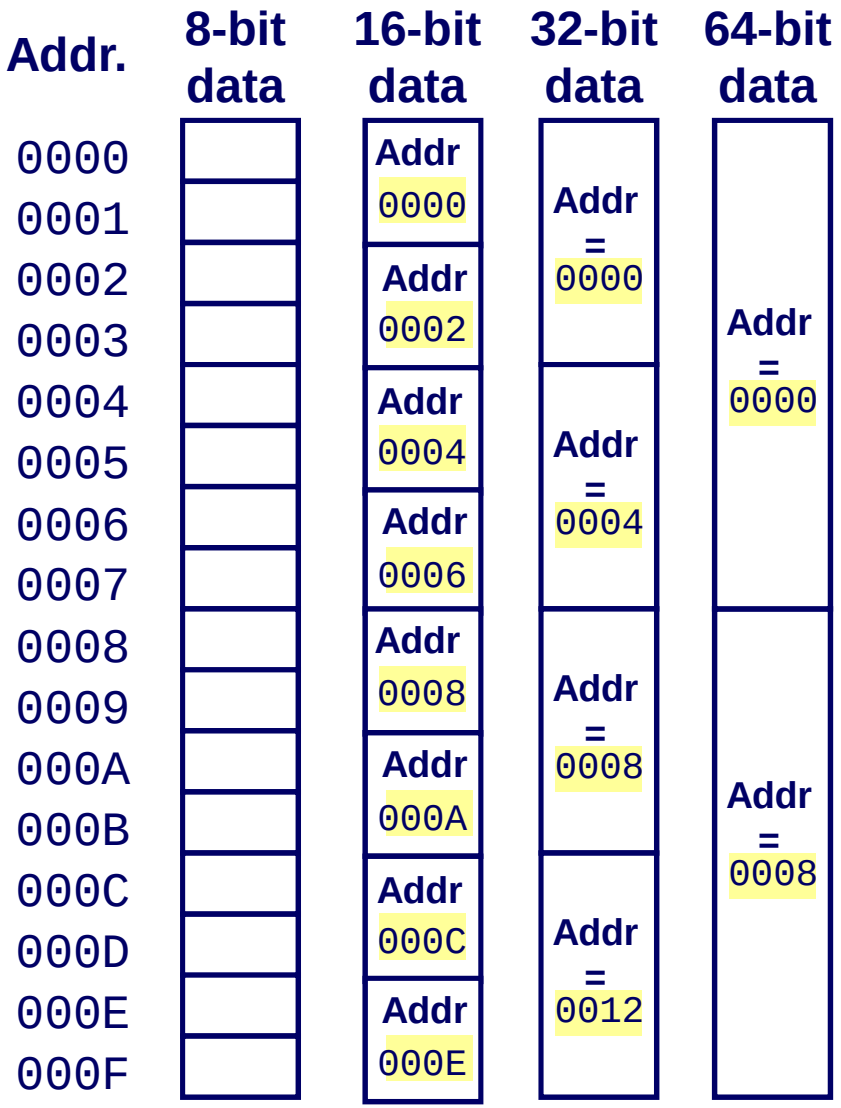
Data Representation in Memory

- Memory organization within a process

- **Memory addressing and ordering of multi-byte data**
  - Addressing
  - Byte ordering
  - Arrays
  - Data structures
  - Ordering in arrays/structures vs. single multi-byte data elements

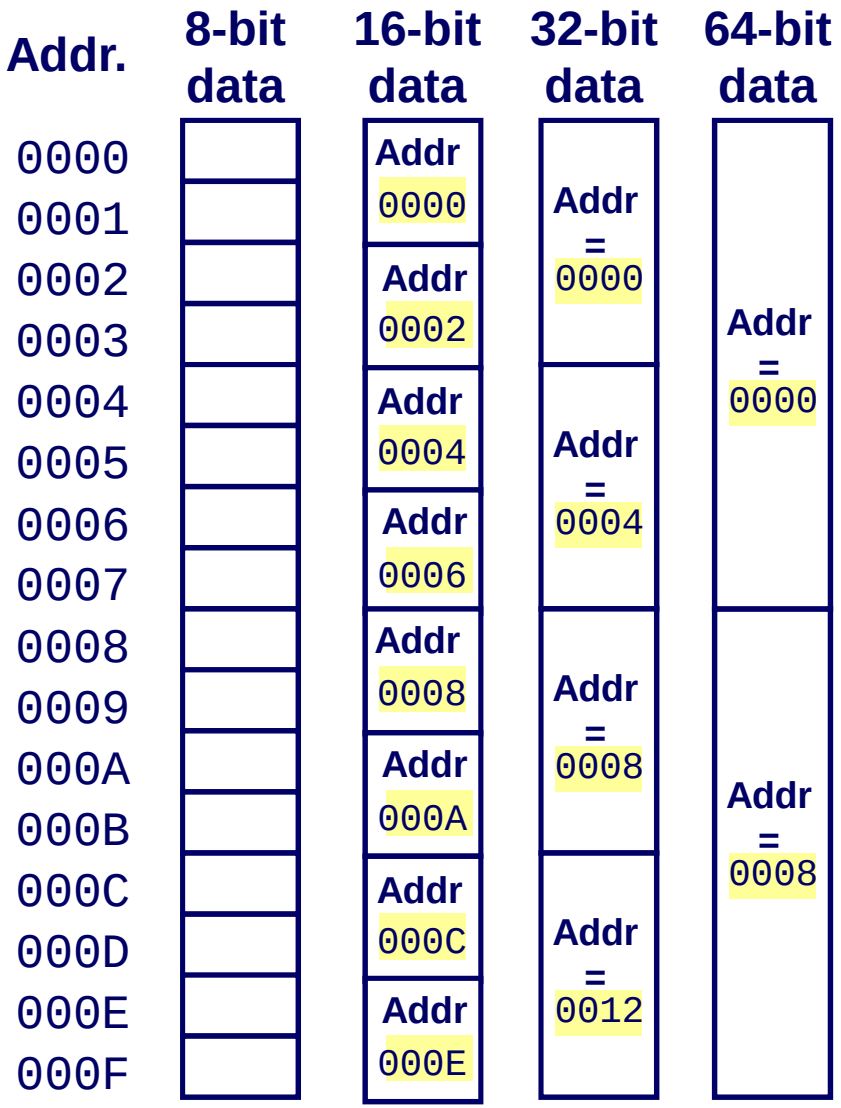
Address of Multi-byte Data

- Every byte has a unique address
- *So, if data spans multiple bytes, what is address?*
- Data always addressed by its lowest address
  - address of first byte in memory



Address of Multi-byte Data

- **Alignment**
  - Data elements are aligned by size
    - for a primitive (single datum) with *K* bits, address must be multiple of *K*
    - **chars, booleans** at any address
    - **shorts** at even addresses
    - **ints, floats,** pointers every 4th addr
    - **doubles** every 8th address
    - etc.
  - Arrays, structures, and classes
    - alignment determined by size of largest primitive (single datum)



Data Representation in Memory

- Memory organization within a process

- **Memory addressing and ordering of multi-byte data**
  - Addressing
  - Byte ordering
  - Arrays
  - Data structures
  - Ordering in arrays/structures vs. single multi-byte data elements

Byte Ordering

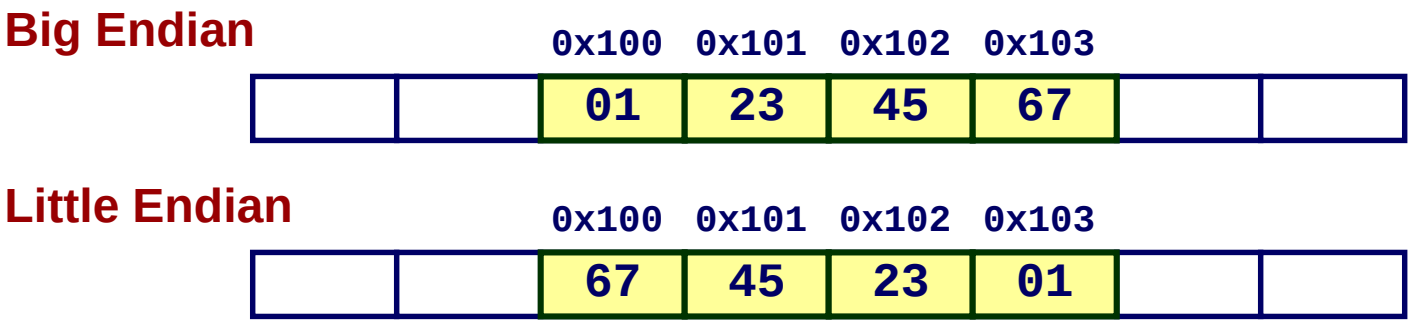
- **How should bytes within a multi-byte word be ordered in memory?**

- **Affects only primitive data elements with multiple bytes**
  - i.e. a *single* data element composed of multiple bytes
    - **short**, **int**, **long**, **float**, **double, boolean, …**
    - does not affect arrays, structs, or classes

- **Conventions**
  - **Big Endian**: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - **Little Endian**: x86
    - Least significant byte has lowest address

Byte Ordering Example

- **Big Endian**
  - Least significant byte has highest address
- **Little Endian**
  - Least significant byte has lowest address
- **Example**
  - Variable *m* has 4-byte representation 0x01234567
  - Address given by &*m* is 0x100



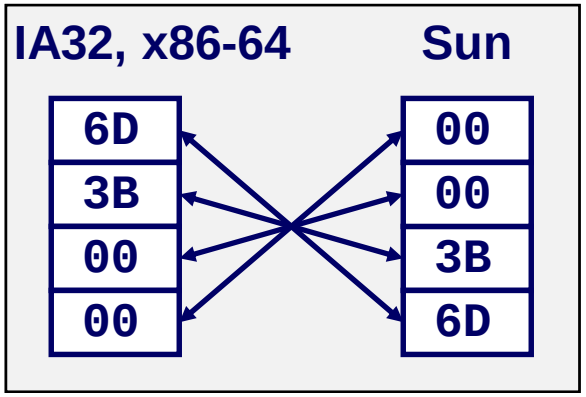
Representing Integers

Decimal: 15213

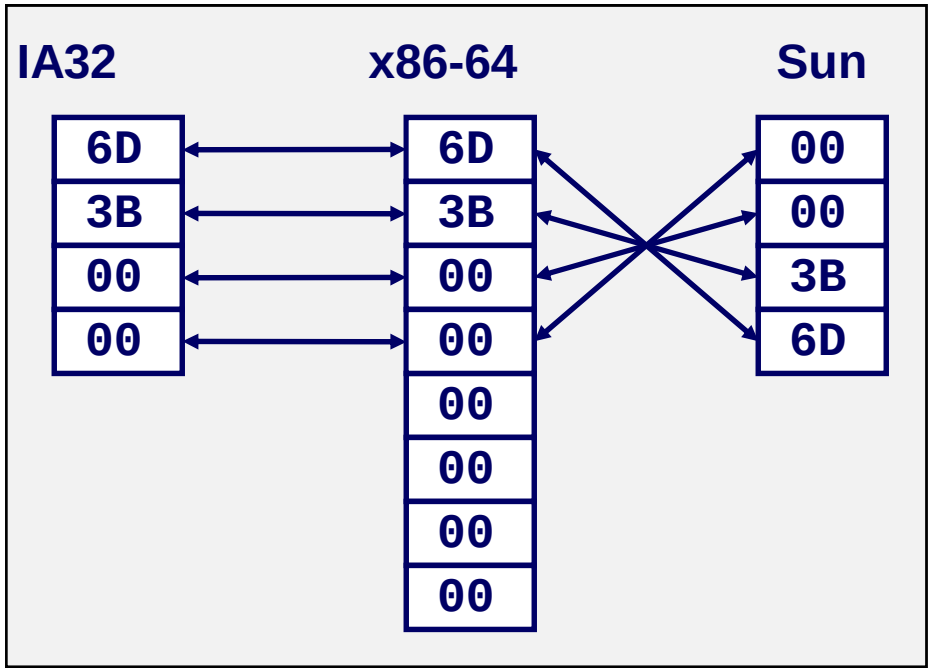
Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

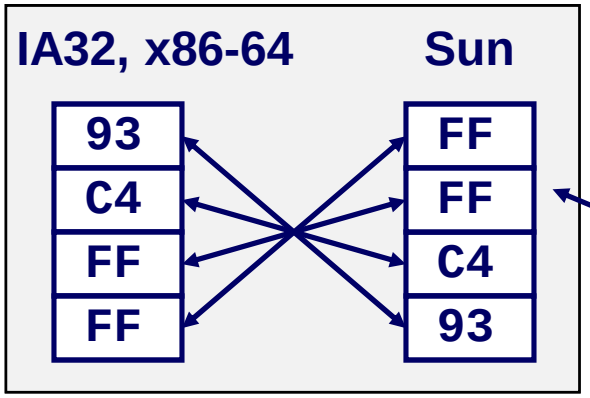
int A = 15213;



long int C = 15213;



int B = -15213;



Signed integer (two's complement) representation

Data Representation in Memory

- Memory organization within a process

- **Memory addressing and ordering of multi-byte data**
  - Addressing
  - Byte ordering
  - Arrays
  - Data structures
  - Ordering in arrays/structures vs. single multi-byte data elements

Basic Data Types

- **Integral**
  - Stored & operated on in general (integer) registers
  - Signed vs. unsigned depends on instructions used

Intel	ASM	Bytes	C
byte	`b`	1	`[unsigned] char`
word	`w`	2	`[unsigned] short`
double word	`l`	4	`[unsigned] int`
quad word	`q`	8	`[unsigned] long int` (x86-64)

- **Floating Point**
  - Stored & operated on in floating point registers

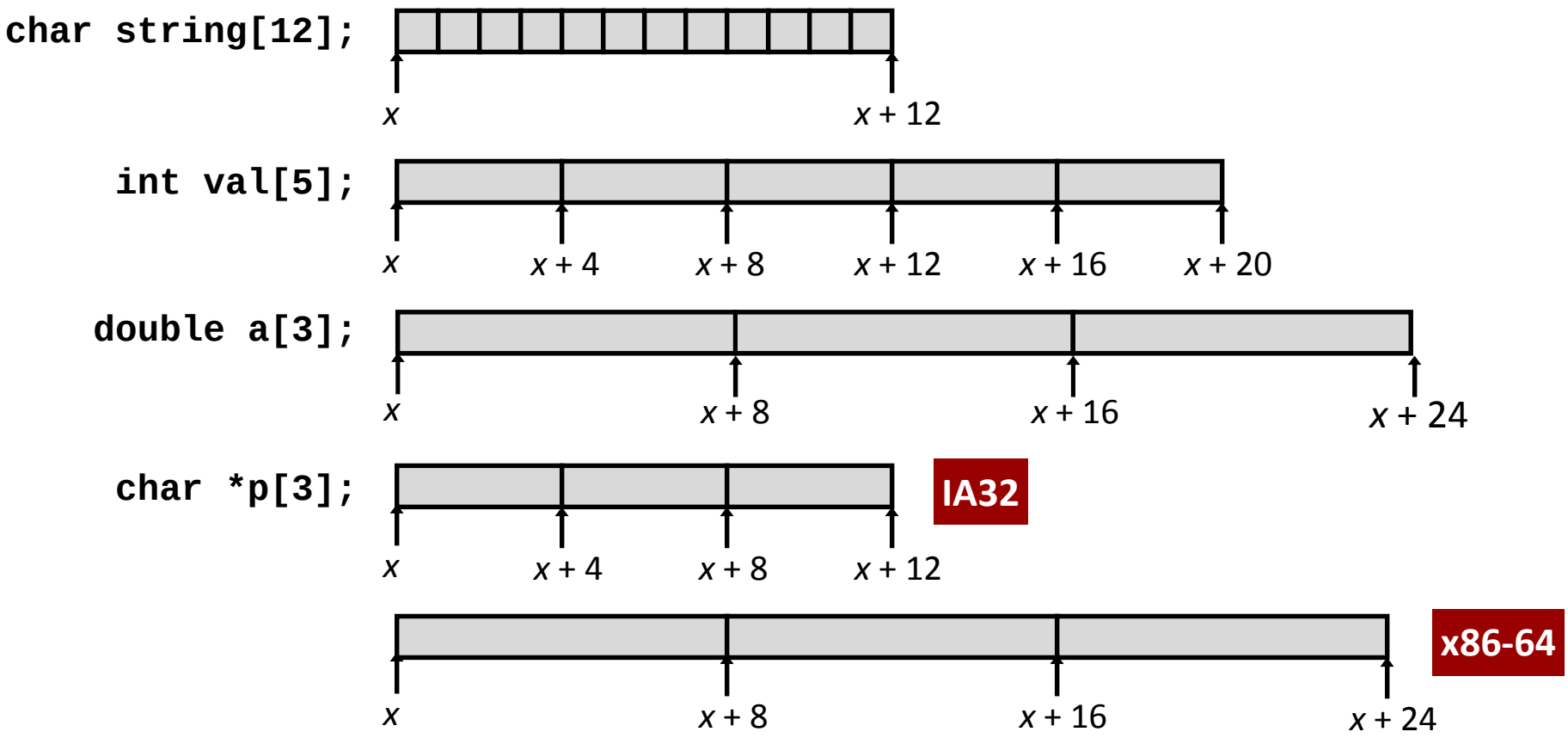
Intel	ASM	Bytes	C
Single	`s`	4	`float`
Double	`l`	8	`double`
Extended	`t`	10/12/16	`long double`

Array Allocation

- **Basic Principle**

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

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

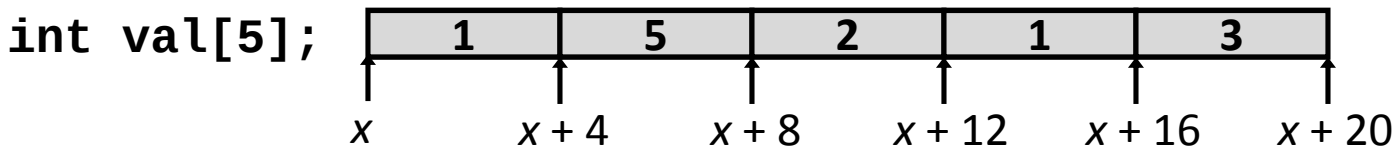


Array Access

- **Basic Principle**

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

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

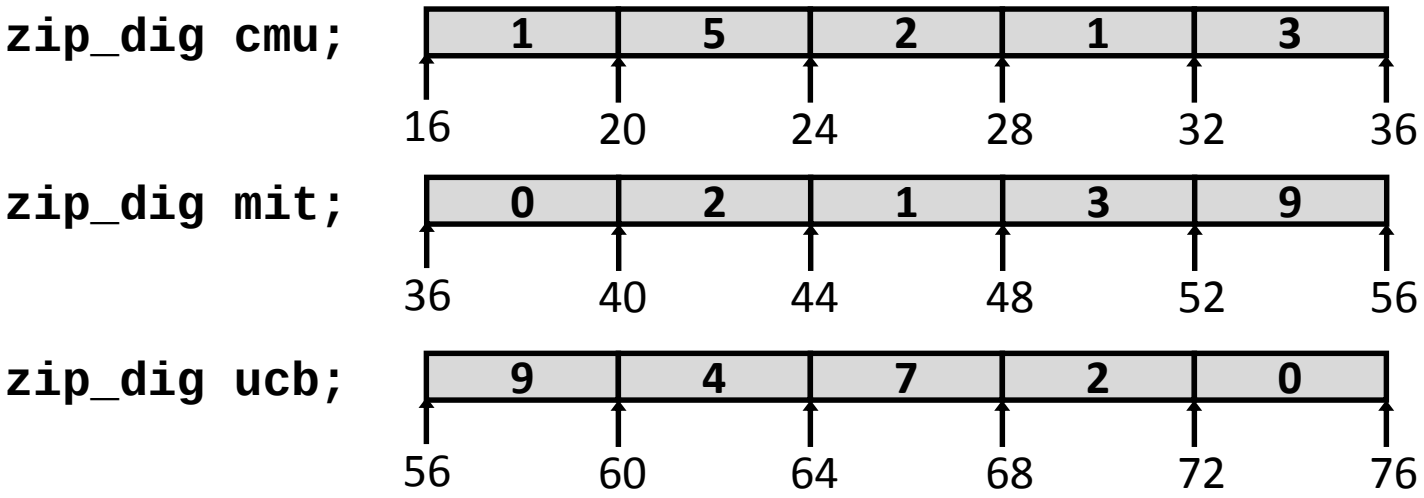


- **Reference** **Type** **Value**

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

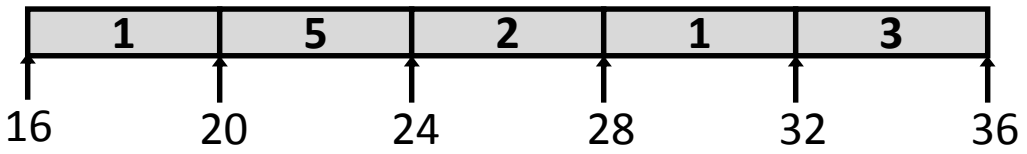
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig 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 dig)
{
  return z[dig];
}
```

IA32

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

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

Array Loop Example (IA32)

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

```
  # edx = z
   movl  $0, %eax            #   %eax = i
.L4:                         # loop:
   addl  $1, (%edx,%eax,4)   #   z[i]++
   addl  $1, %eax            #   i++
   cmpl  $5, %eax            #   i:5
   jne   .L4                 #   if !=, goto loop
```

Pointer Loop Example (IA32)

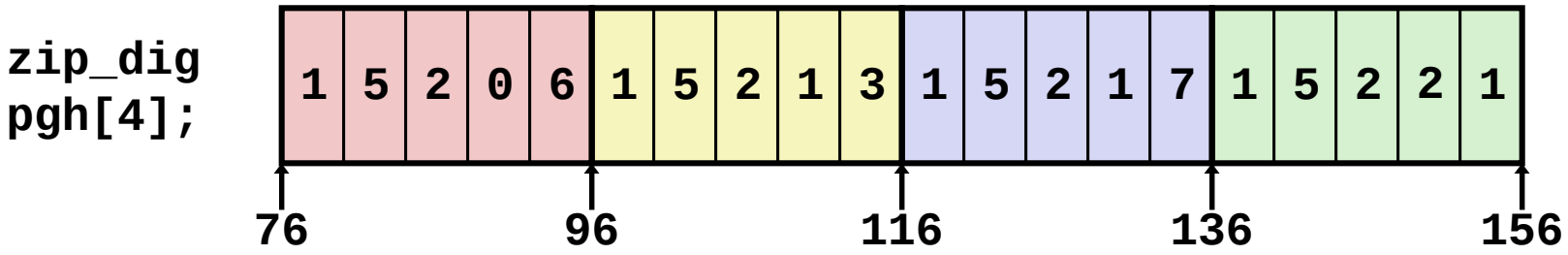
```
void zincr_p(zip_dig z) {
  int *zend = z+ZLEN;
  do {
    (*z)++;
    z++;
  } while (z != zend);
}
```

```
void zincr_v(zip_dig z) {
  void *vz = z;
  int i = 0;
  do {
    (*((int *) (vz+i)))++;
    i += ISIZE;
  } while (i != ISIZE*ZLEN);
}
```

```
  # edx = z = vz
  movl  $0, %eax            #   i = 0
.L8:                        # loop:
  addl  $1, (%edx,%eax)     #   Increment vz+i
  addl  $4, %eax            #   i +=  4
  cmpl  $20, %eax           #   Compare i:20
  jne   .L8                 #   if !=, goto loop
```

Nested Array Example

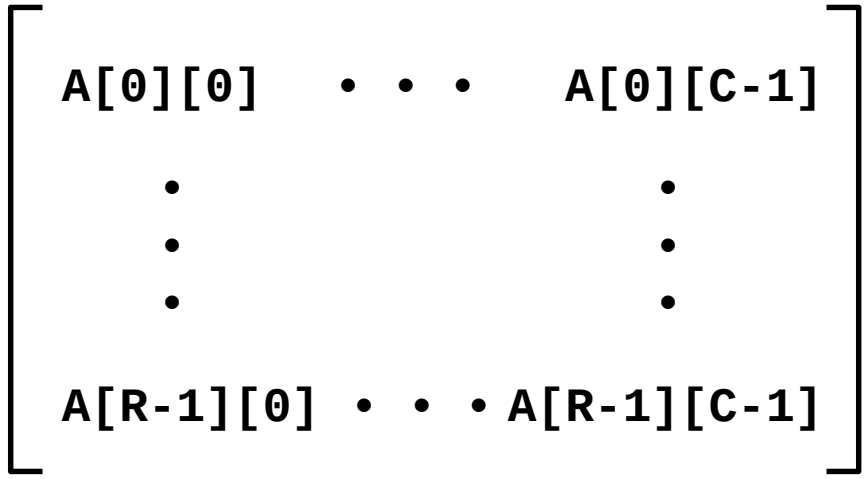
```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



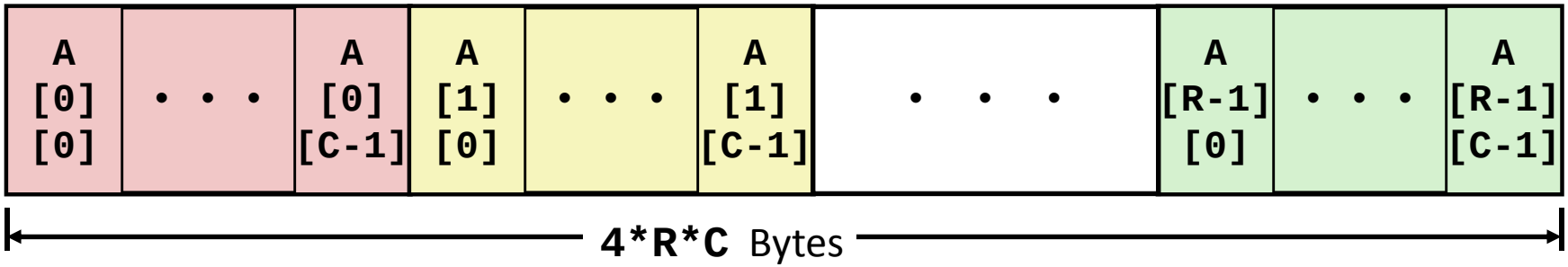
- **"`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"**
  - Variable **pgh**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**'s, allocated contiguously
- **"Row-Major" ordering of all elements guaranteed**

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



`int A[R][C];`



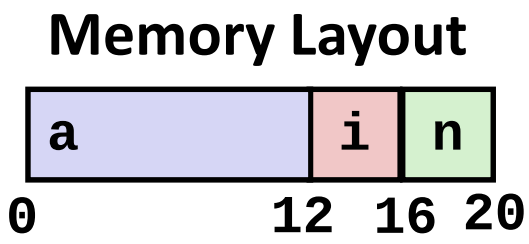
Data Representation in Memory

- Memory organization within a process

- **Memory addressing and ordering of multi-byte data**
  - Addressing
  - Byte ordering
  - Arrays
  - Data structures
  - Ordering in arrays/structures vs. single multi-byte data elements

Structure Allocation

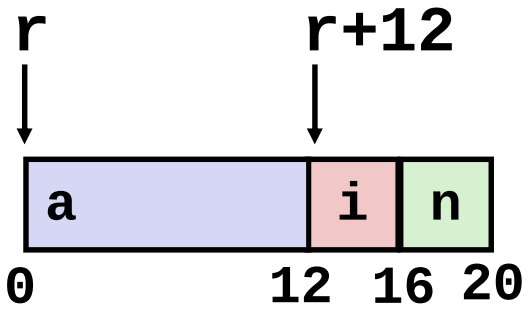
```
struct rec {
  int a[3];
  int i;
  struct rec *n;
};
```



- **Concept**
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

Structure Access

```
struct rec {
  int a[3];
  int i;
  struct rec *n;
};
```



- **Accessing Structure Member**
  - Pointer indicates first byte of structure
  - Access elements with offsets

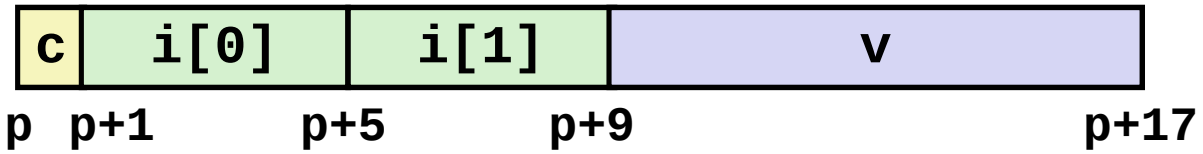
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

IA32 Assembly

```
# %edx = val
# %eax = r
movl %edx, 12(%eax)  # Mem[r+12] = val
```

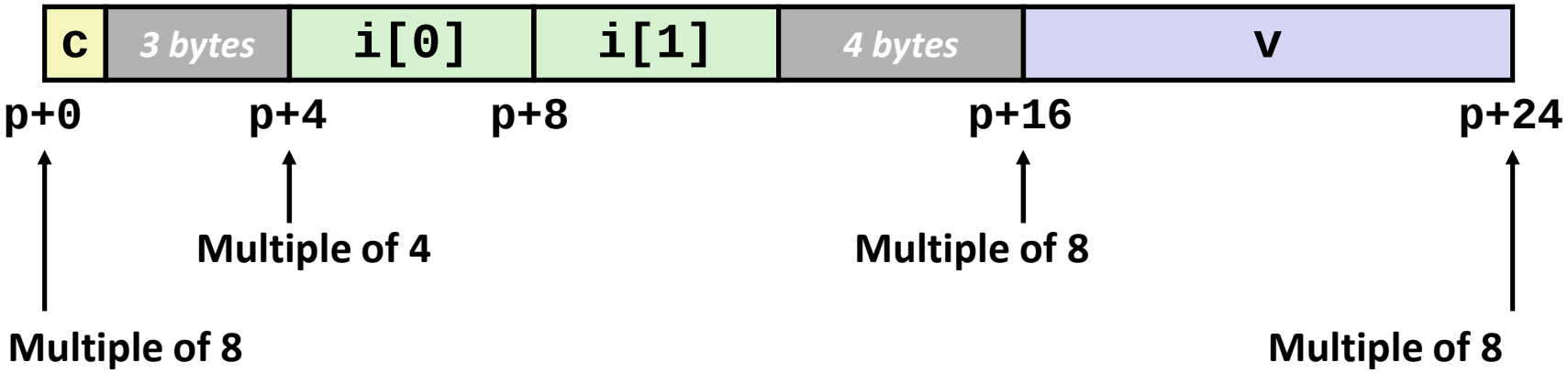
Structures & Alignment

- **Unaligned Data**




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

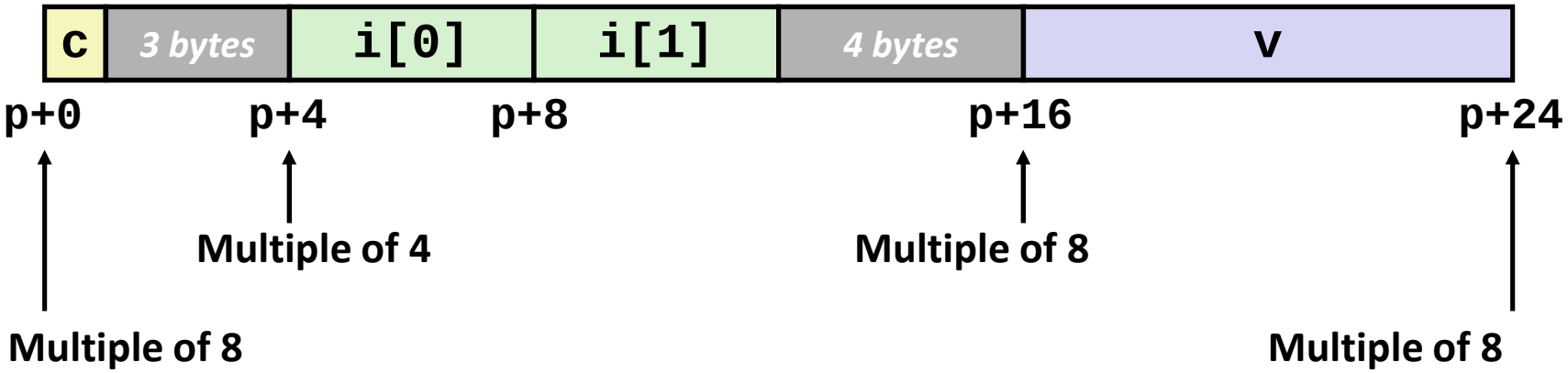
- **Aligned Data**
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***



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 (under Windows or x86-64):**
  - K = 8, due to **`double`** element

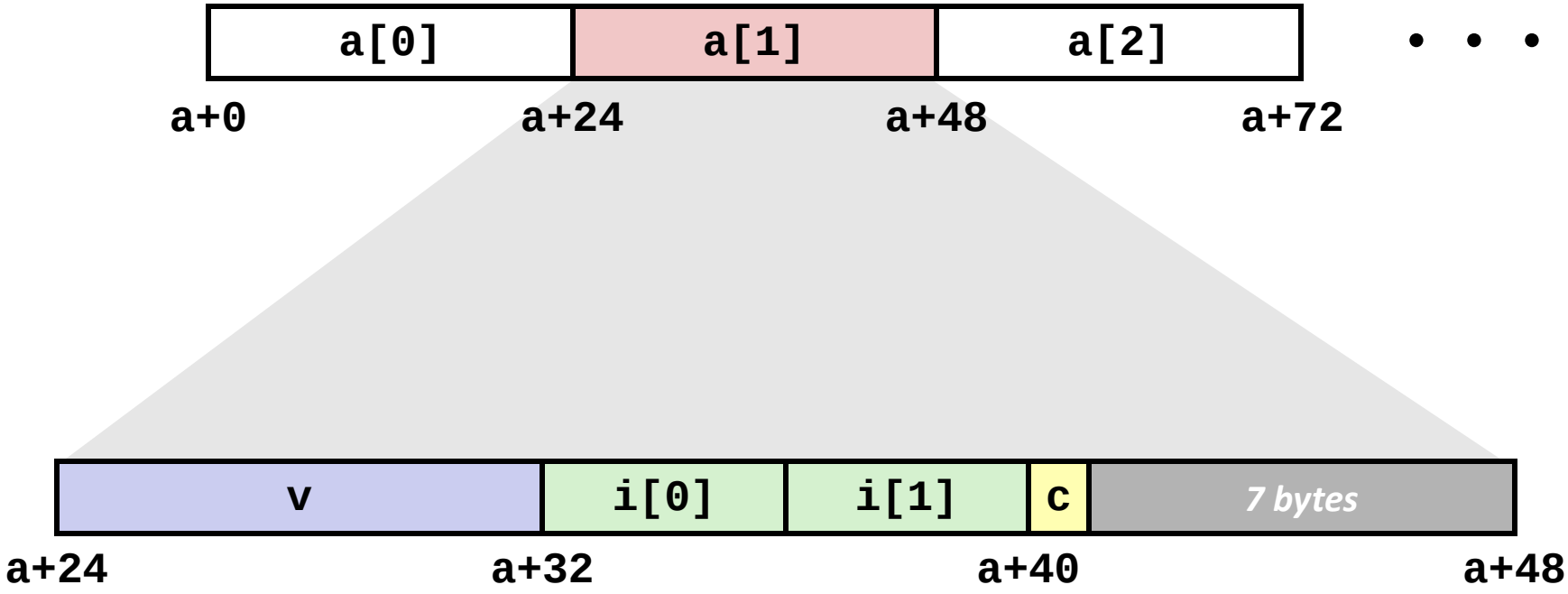
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Arrays of Structures

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

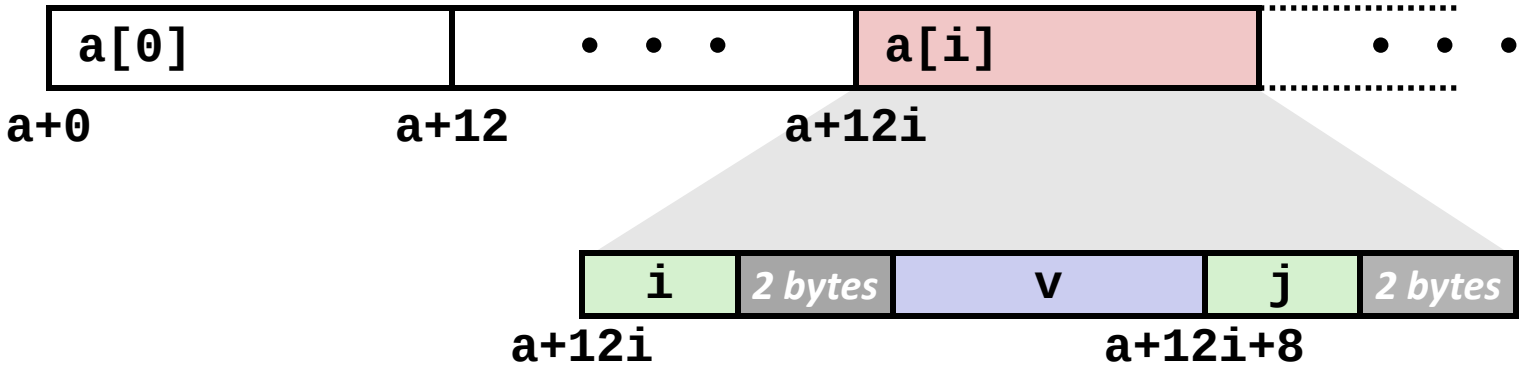
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

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

- **Compute array offset 12i**
  - `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;
}
```

```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```

Data Representation in Memory

- Memory organization within a process

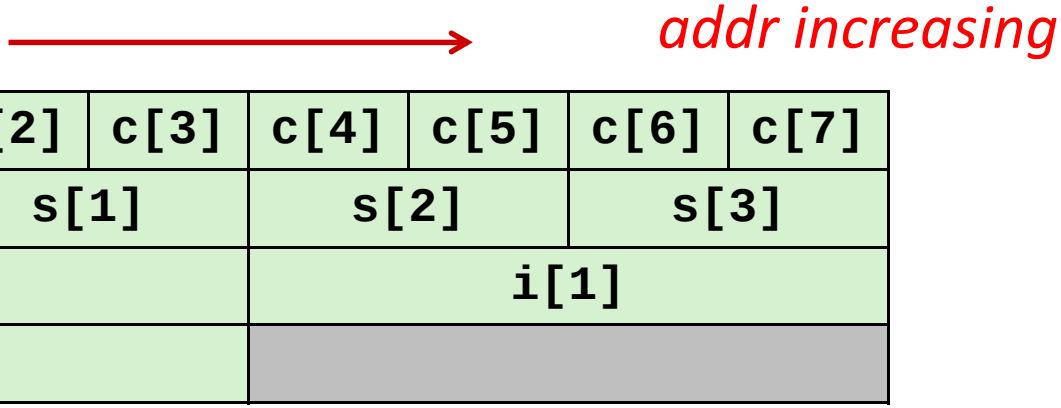
- **Memory addressing and ordering of multi-byte data**
  - Addressing
  - Byte ordering
  - Arrays
  - Data structures
  - Ordering in arrays/structures vs. single multi-byte data elements

Byte Ordering Revisited

- **Idea**
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which is most (least) significant?
  - Can cause problems when exchanging binary data between machines
- **Big Endian**
  - Most significant byte has lowest address
  - Sparc
- **Little Endian**
  - Least significant byte has lowest address
  - Intel x86

Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

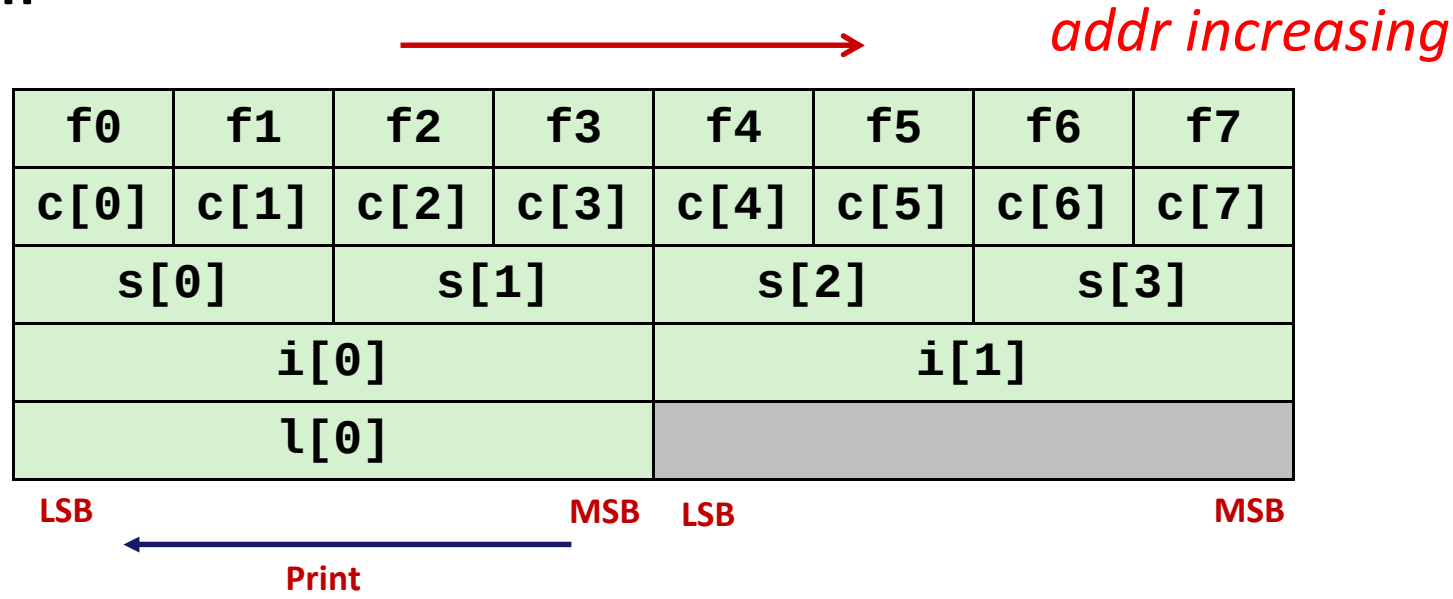


Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

Byte Ordering on IA32

Little Endian

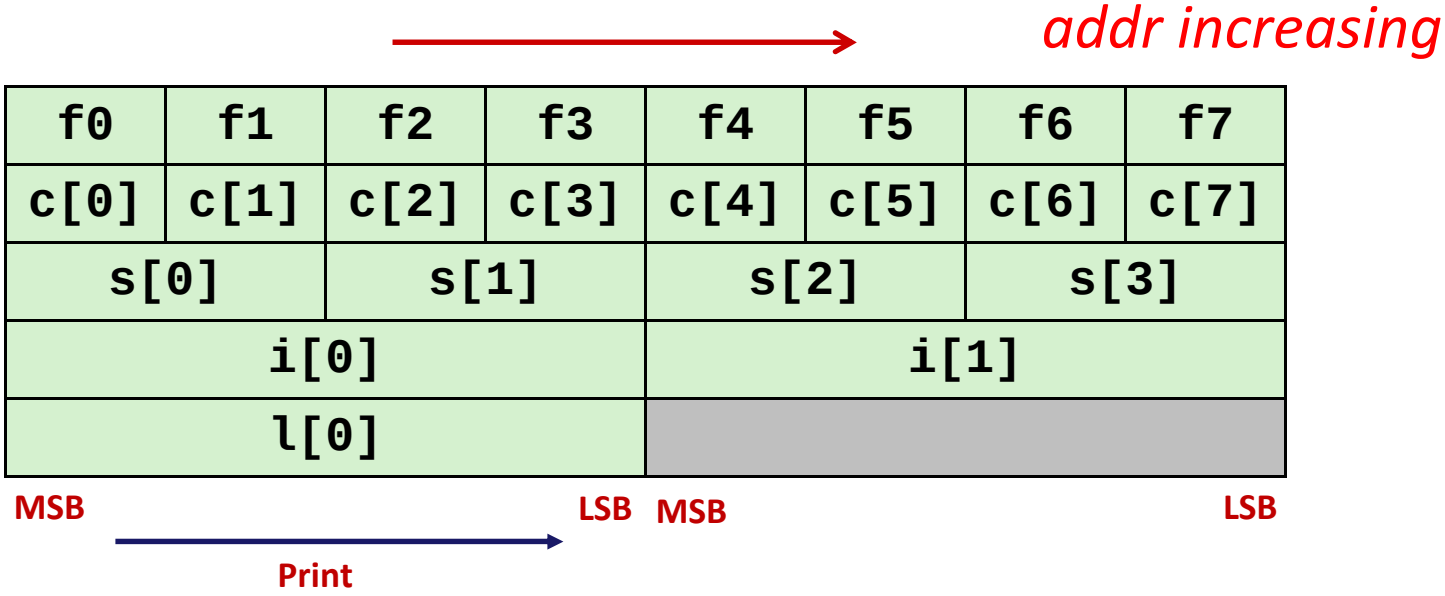


Output:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

Byte Ordering on Sun

Big Endian

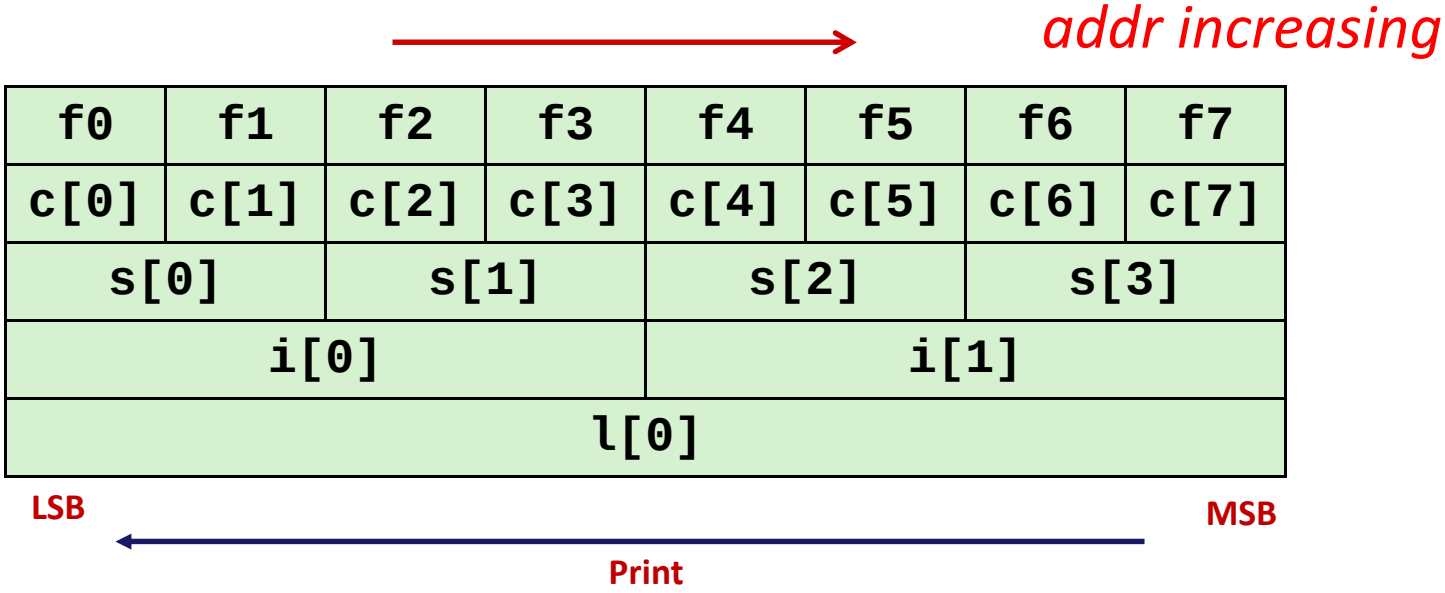


Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

Byte Ordering on x86-64

Little Endian



Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```