15-213

"The course that gives CMU its Zip!"

Structured Data II
Heterogenous Data
Sept. 22, 1998

Topics

- Structure Allocation
- Alignment
- Operating on Byte Strings
- Unions
- Byte Ordering
- Alpha Memory Organization

Basic Data Types

Integral

- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used

Alpha	Bytes	C
byte	1	[unsigned] `char`
word	2	[unsigned] `short`
long word	4	[unsigned] `int`
quad word	8	[unsigned] `long int`, pointers

Floating Point

- Stored & operated on in floating point registers
- Special instructions for four different formats (only 2 we care about)

Alpha	Bytes	C
S_floating	4	`float`
T_floating	8	`double`

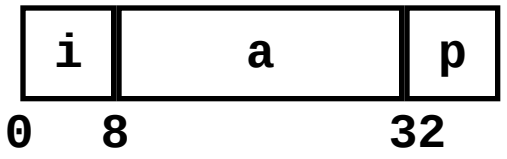
Structures

Concept

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

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

Memory Layout



Accessing Structure Member

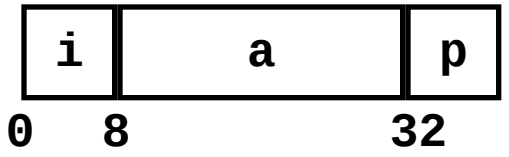
```
void
set_i(struct rec *r,
      long int val)
{
  r->i = val;
}
```

Annotated Assembly

```
set_i:
   stq $17,0($16)  # r->i = val
   ret $31,($26),1
```

Generating Pointer to Structure Member

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



Generating Pointer to Array Element

- Offset of each structure member determined at compile time

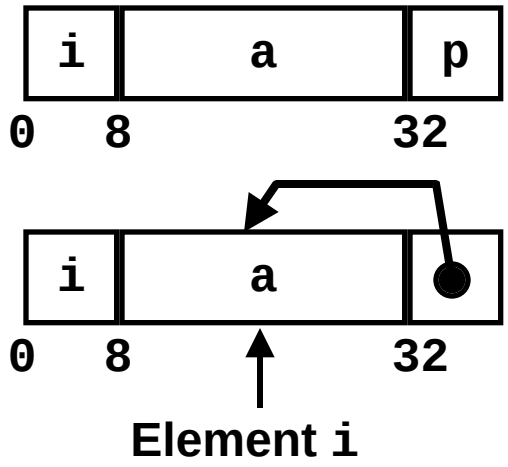
```
long int *
find_a
 (struct rec *r,
  long int idx)
{
  return &r->a[idx];
}
```

```
find_a:
  s8addq $17,8,$0 # $0 = 8*idx +8
  addq $16,$0,$0  # $0 += r
  ret $31,($26),1
```

Structure Referencing (Cont.)

C Code

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




```
void
set_p(struct rec *r,
      long int *ptr)
{
  r->p =
   &r->a[r->i];
}
```

```
set_p:
  ldq $1,0($16)   # get r->i
  s8addq $1,8,$1  # Compute 8*i+8
  addq $1,$16,$1  # compute &a[i]
  stq $1,32($16)  # store in p
  ret $31,($26), 1
```

Alignment

Requirement

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

Specific Cases

- **Long word address must be multiple of 4**
  - Lower 2 bits of address must be 00_2
- **Quad word address must be multiple of 8**
  - Lower 3 bits of address must be 000_2

Reason

- **Memory accessed by (aligned) quadwords**
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory very tricky when datum spans 2 pages

Compiler

- **Inserts gaps in structure to ensure correct alignment of fields**

Satisfying Alignment with Structures

Offsets Within Structure

- **Must satisfy element's alignment requirement**

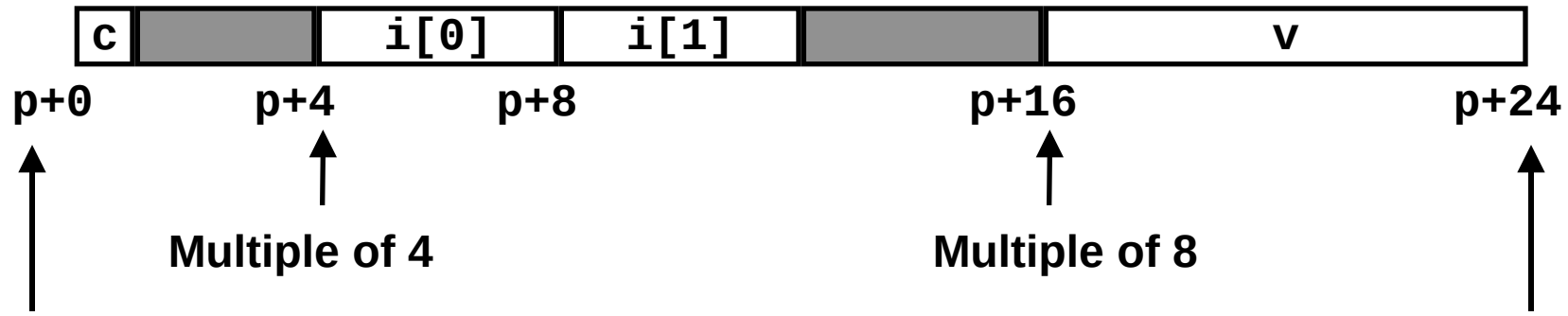
Overall Structure Placement

- **Each structure has alignment requirement K**
  - Largest alignment of any element
- **Initial address + structure length must be multiples of K**

Example

- **K = 8, due to `long int` element**

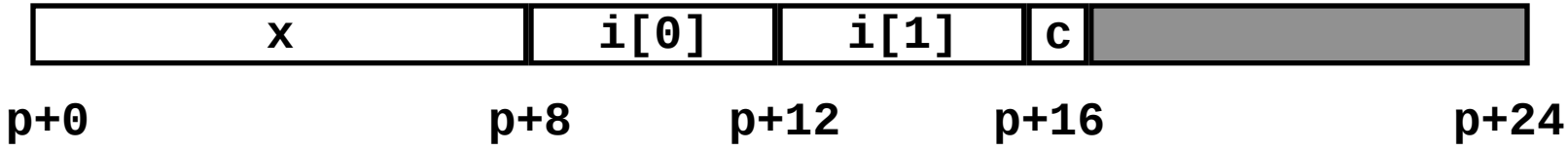
```
struct S1 {
  char c;
  int i[2];
  long int v;
} *p;
```



Effect of Overall Alignment Requirement

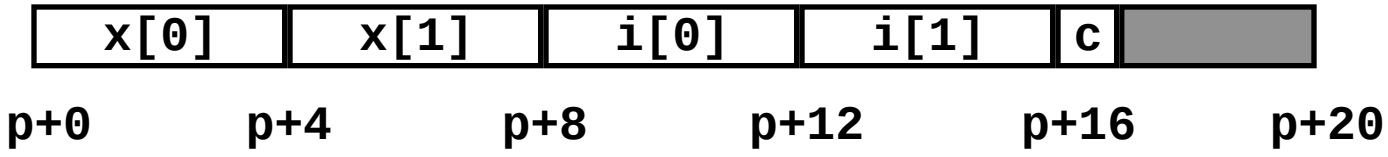
```
struct S2 {
  int *x;
  int i[2];
  char c;
} *p;
```

p must be multiple of 8




```
struct S3 {
  int x[2];
  int i[2];
  char c;
} *p;
```

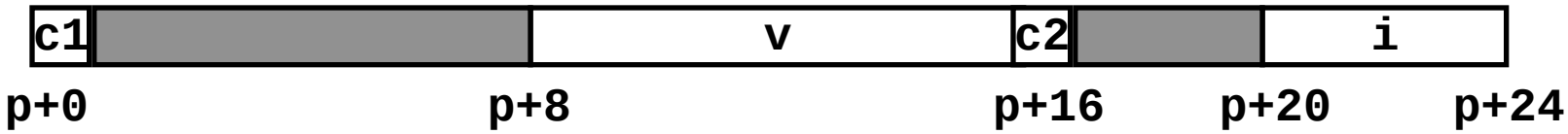
p must be multiple of 4



Ordering Elements Within Structure

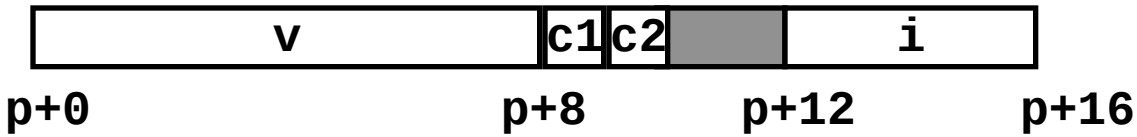
```
struct S4 {
  char c1
  long int v;
  char c2;
  int i;
} *p;
```

10 bytes wasted space




```
struct S5 {
  long int v;
  char c1
  char c2;
  int i;
} *p;
```

2 bytes wasted space

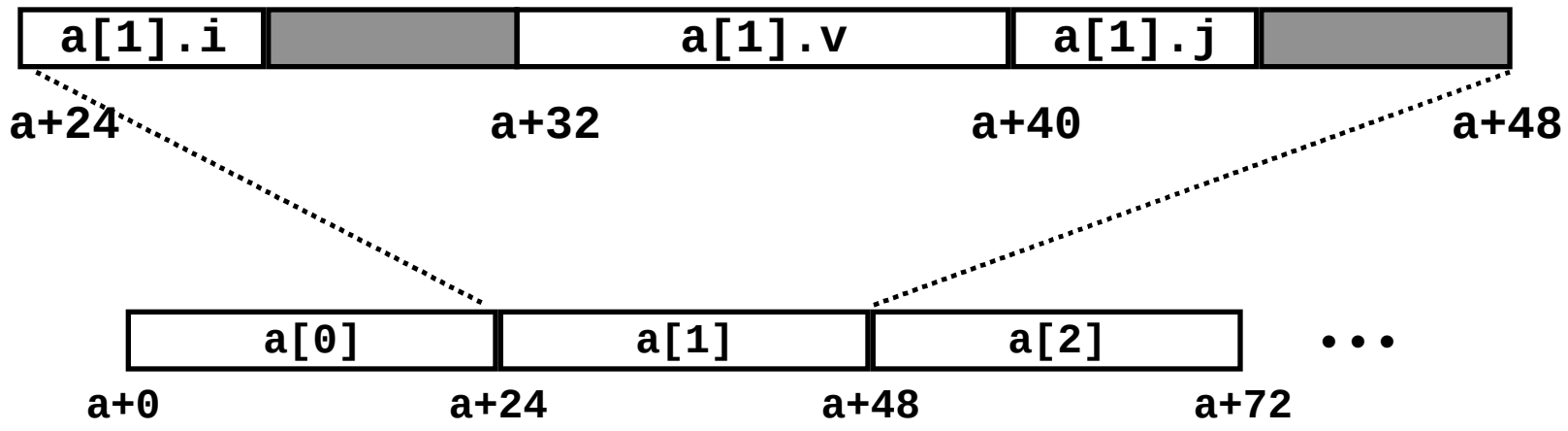


Arrays of Structures

Principle

- Allocated by repeating allocation for array type
- In general, may nest arrays & structures to arbitrary depth

```
struct S6 {
  int i;
  long int v;
  int j;
} a[10];
```



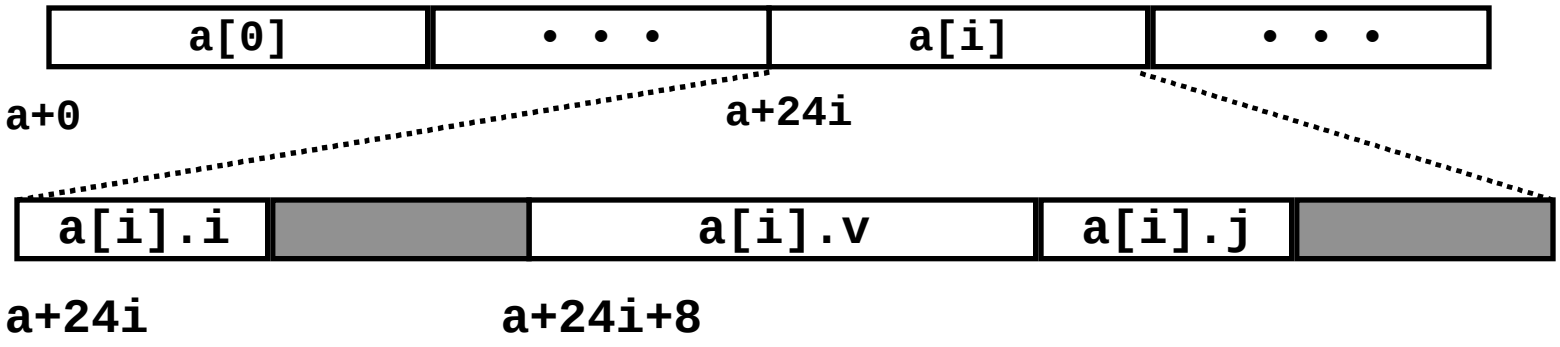
Accessing Element within Array

- **Compute offset to start of structure**
  - Compute 24**i* as (4*i*–*i*)*8
- **Access element according to its offset within structure**
  - Offset by 8

```
struct S6 {
  int i;
  long int v;
  int j;
} a[10];
```

```
long int get_v(int i)
{
  return a[i].v;
}
```

```
  s4subq $16,$16,$16 # i*= 3
  lda $1,a           # $1 = a
  s8addq $16,$1,$16  # $16 = a+i
  ldq $0,8($16)      # a[i].v
```

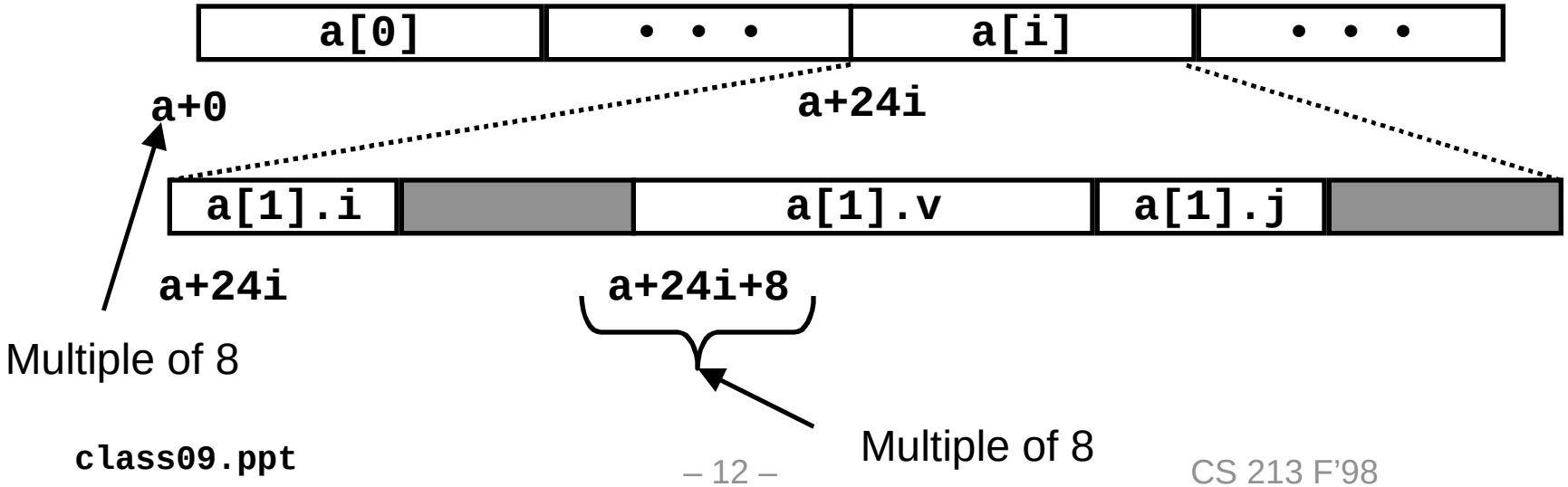


Satisfying Alignment within Structure

Achieving Alignment

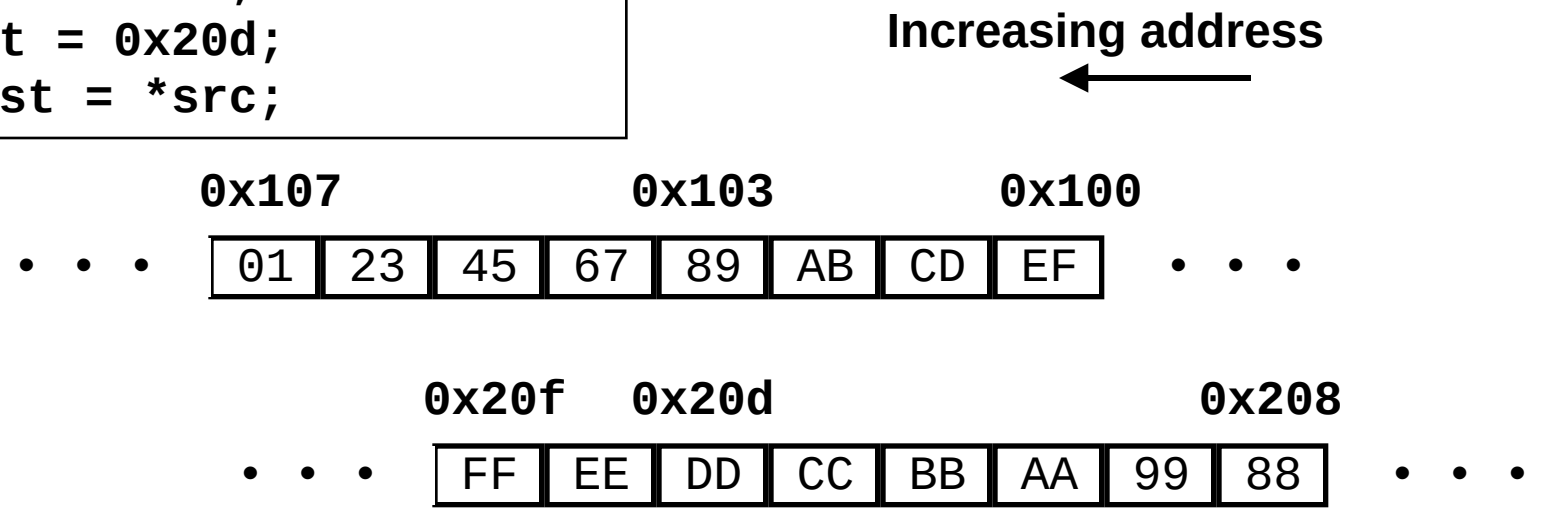
- **Starting address of structure array must be multiple of worst-case alignment for any element**
  - **a** must be multiple of 8
- **Offset of element within structure must be multiple of element's alignment requirement**
  - **v**'s offset of 8 is a multiple of 8
- **Overall size of structure must be multiple of worst-case alignment for any element**
  - Structure padded with unused space to be 24 bytes

```
struct S6 {
  int i;
  long int v;
  int j;
} a[10];
```

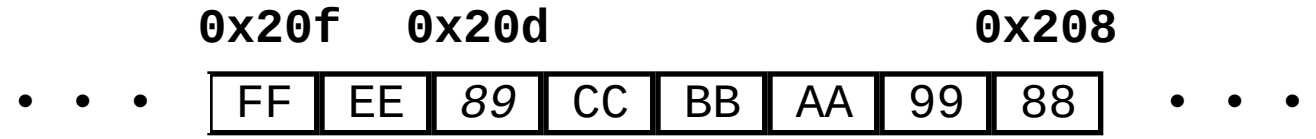


Byte-Level Memory Operation Example

```
char *src, *dest;
src = 0x103;
dest = 0x20d;
*dest = *src;
```



Desired Effect:



Implementing Byte-Level Operations

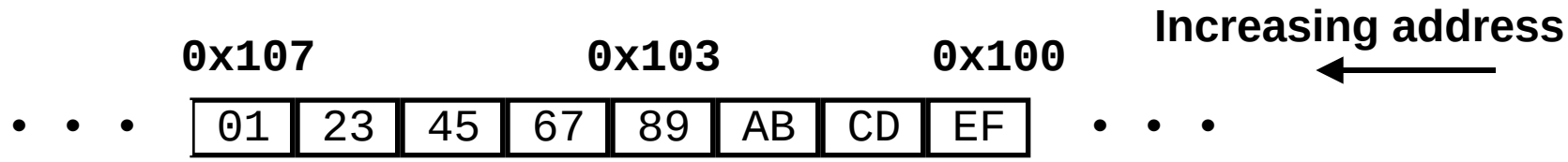
Issue

- **Byte addresses have no alignment requirement**
- **Alpha memory system only designed for 4-byte and 8-byte aligned memory operations**

Method

- **Could use regular quad-word memory operations + masking and shifting**
  - 17 instructions required to implement statement **`*dest = *src`**
  - single byte transfer
- **Instead, provide set of instructions to perform byte extraction and insertion in quad-words**
  - 7 instructions required to implement statement **`*dest = *src`**

Extracting Source Byte



Step 1. Get byte at `src = 0x103`

- **Pointer `src` in register `$17`**

`ldq_u $1, 0($17)`

- Rounds effective address **`0x103`** to nearest quad word boundary (**`0x100`**)
  - » By setting low 3 bits of address to $\mathbf{000_2}$
- Loads entire quad word into memory

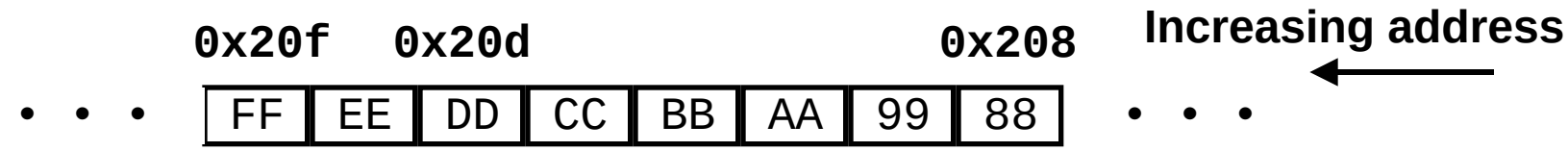
`$1:`	**01**	**23**	**45**	**67**	**89**	**AB**	**CD**	**EF**
Byte Number:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

`extbl $1, $17, $1`

- Uses lower 3 bits of $17 as byte offset (= 3)
- Sets destination register to that byte of source

`$1:`	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**89**

Preparing Destination



Step 2. Zero byte at `dest = 0x20d`

- **Pointer `dest` in register `$16`**

`ldq_u $2, 0($16)`

- Rounds effective address **`0x20d`** down to nearest quad word boundary (**`0x208`**)
- Loads entire quad word into register

$2:	**FF**	**EE**	**DD**	**CC**	**BB**	**AA**	**99**	**88**
Byte Number:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

`mskbl $2, $16, $2`

- Uses lower 3 bits of $16 as byte offset (= 5)
- Copies source to destination, but zeros specified byte

$2:	**FF**	**EE**	***00***	**CC**	**BB**	**AA**	**99**	**88**
Byte Number:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

Merging Source into Destination

Step 3. Position Source Byte

$1:	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**89**

- **Pointer `dest` in register `$16`**

`insbl $1, $16, $1`

- Uses lower 3 bits of **`$16`** as byte offset (= 5)
- Shifts low order byte of source into specified byte position

$1:	**00**	**00**	**89**	**00**	**00**	**00**	**00**	**00**

$2:	**FF**	**EE**	**00**	**CC**	**BB**	**AA**	**99**	**88**

Step 4. Form Destination Quad-Word

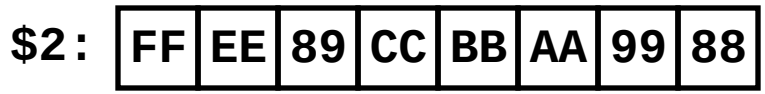
`bis $1, $2, $2`

- Merges new byte into destination

$2:	**FF**	**EE**	**89**	**CC**	**BB**	**AA**	**99**	**88**

Updating Destination

Step 5. Update Destination Quad-Word



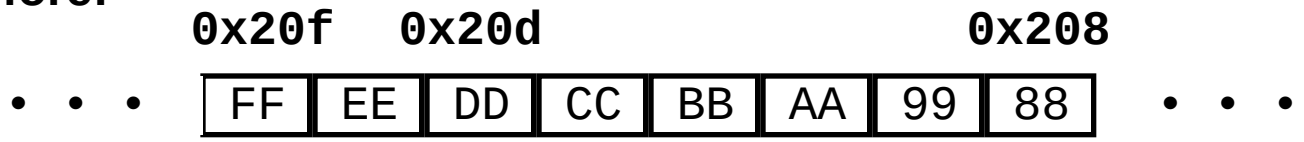
- **Pointer `dest` in register `$16`**

```
stq_u $2, 0($16)
```

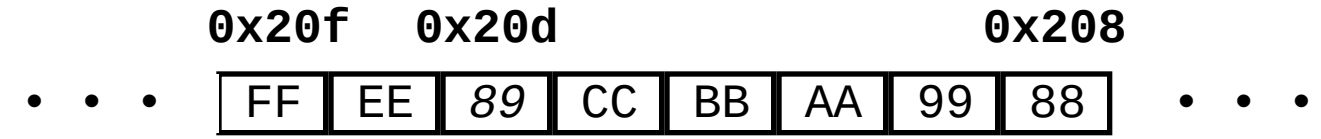
  - Rounds effective address **`0x20d`** down to nearest quad word boundary (**`0x208`**)
  - Stores entire quad word into memory

Net Effect

- **Before:**



- **After:**

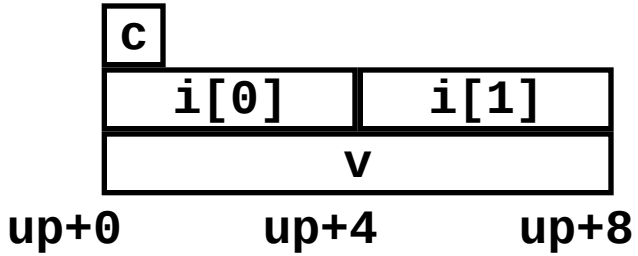


Union Allocation

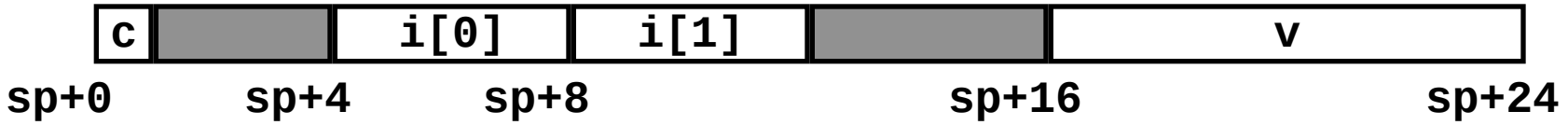
Principles

- Overlay union elements
- Allocate according to largest element
- Can only use one field at a time

```
union U1 {
  char c;
  int i[2];
  long int v;
} *up;
```



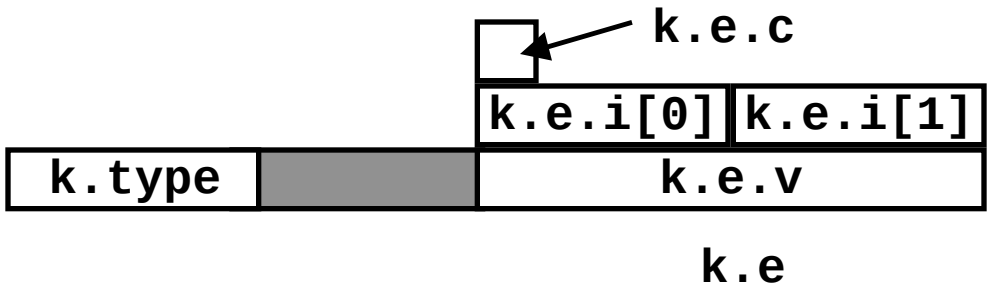

```
struct S1 {
  char c;
  int i[2];
  long int v;
} *sp;
```



Implementing “Tagged” Union

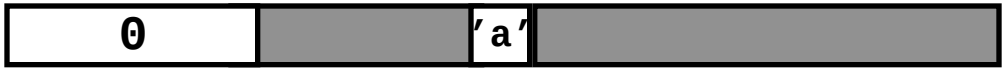
- Structure can hold 3 kinds of data
- Only one form at any given time
- Identify particular kind with flag `type`

```
typedef enum { CHAR, INT, LONG }
  utype;

typedef struct {
  utype type;
  union {
    char c;
    int i[2];
    long int v;
  } e;
} store_ele, *store_ptr;

store_ele k;
```



Using “Tagged” Union

```
store_ele k1;
k1.type = CHAR;
k1.e.c = 'a';
```

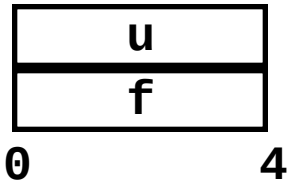



```
store_ele k2;
k2.type = INT;
k2.e.i[0] = 17;
k2.e.i[1] = 47;
```


```
store_ele k3;
k3.type = LONG;
k1.e.v =
0xFF00FF00FF00FF00;
```

Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```



- **Get direct access to bit representation of float**
- **`bit2float` generates float with given bit pattern**
  - NOT the same as **`(float) u`**
- **`float2bit` generates bit pattern from float**
  - NOT the same as **`(unsigned) f`**

```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Byte Ordering

Idea

- Long/quad words stored in memory as 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
- IBM 360/370, Motorola 68K, Sparc

Little Endian

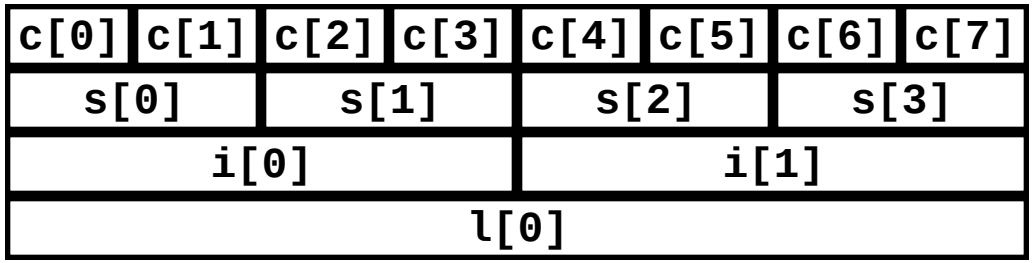
- Least significant byte has lowest address
- Intel x86, VAX

Alpha

- Chip can be configured to operate either way
- Our's are little endian
- Cray T3E Alpha's are big endian

Byte Ordering Example

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

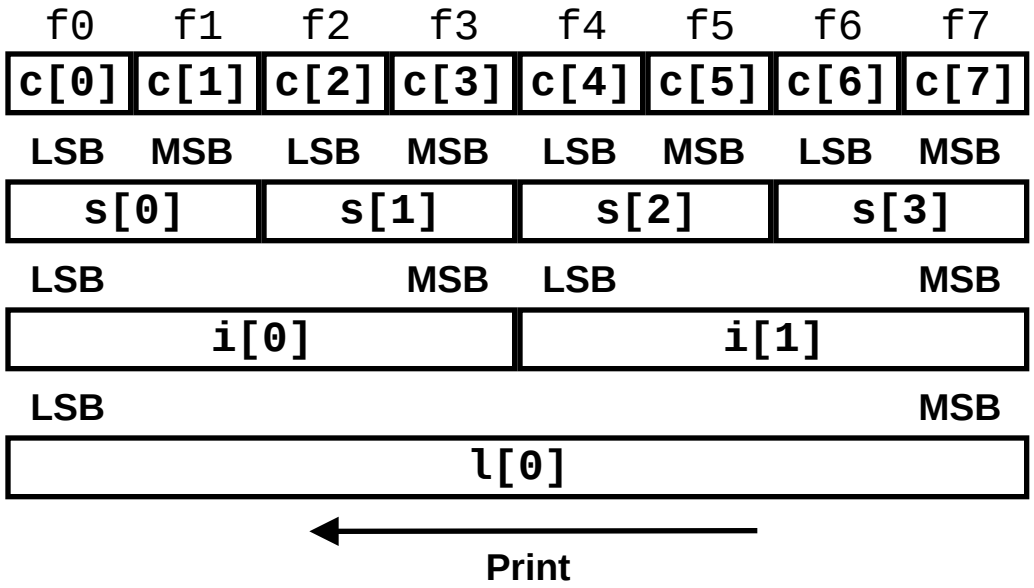


Byte Ordering Example (Cont).

```
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 Alpha

Little Endian

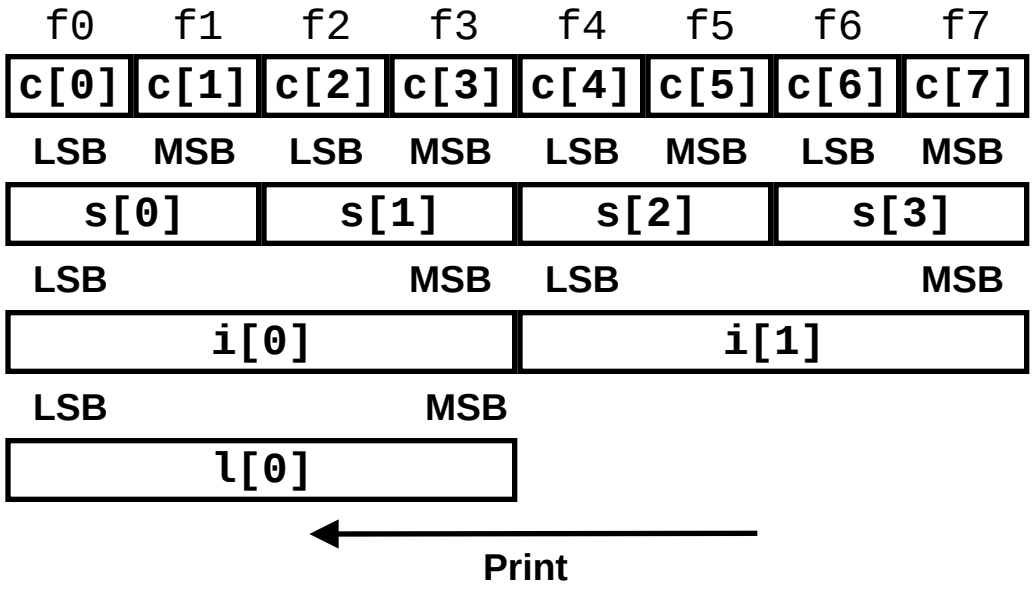


Output on Alpha:

```
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]
```

Byte Ordering on x86

Little Endian

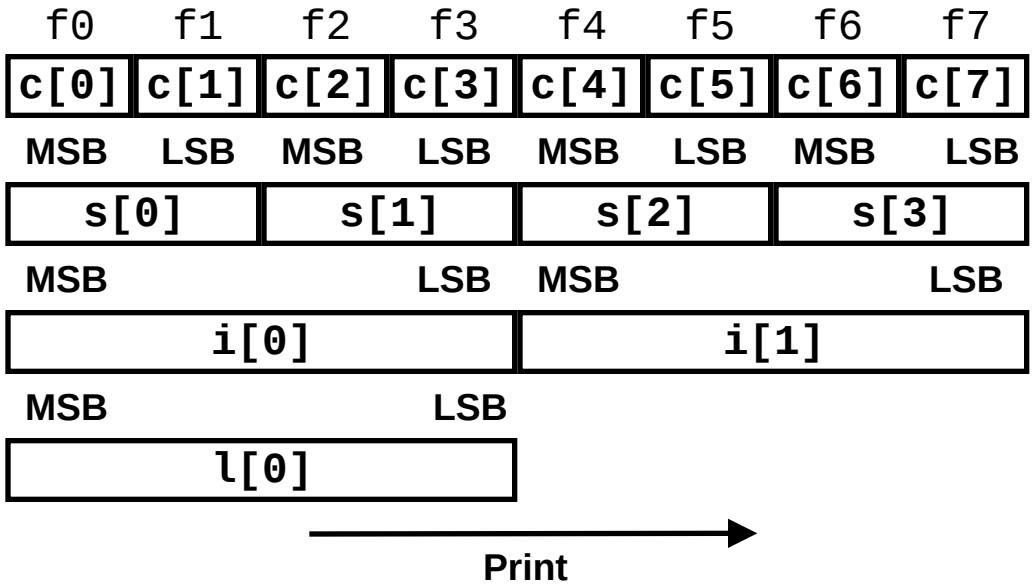


Output on Pentium:

```
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   == [f3f2f1f0]
```

Byte Ordering on Sun

Big Endian

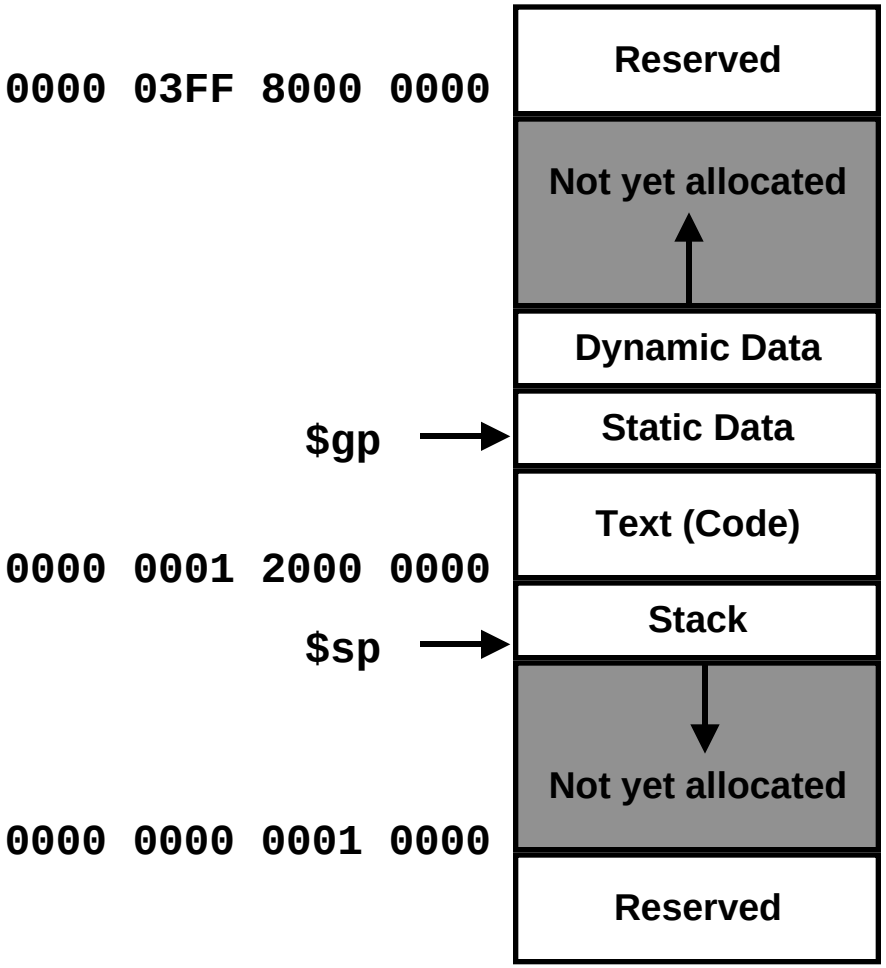


Output on Sun:

```
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]
```

Alpha Memory Layout

Segments



- **Data**
  - Static space for global variables
    - » Allocation determined at compile time
    - » Access via **`$gp`**
  - Dynamic space for runtime allocation
    - » E.g., using **`malloc`**
- **Text**
  - Stores machine code for program
- **Stack**
  - Implements runtime stack
  - Access via **`$sp`**
- **Reserved**
  - Used by operating system
    - » page tables, process info, etc.

Alpha Memory Allocation

Address Range

- **User code can access memory locations in range `0x0000000000010000` to `0x000003FF80000000`**
- **Nearly** $2^{42} \approx 4.3980465 \times 10^{12}$ **byte range**
- **In practice, programs access far fewer**

Dynamic Memory Allocation

- **Virtual memory system only allocates blocks of memory ("pages") as needed**
- **As stack reaches lower addresses, add to lower allocation**
- **As break moves toward higher addresses, add to upper allocation**
  - Due to calls to **`malloc`**, **`calloc`**, etc.