# Alpha Programming (Part 2)
# CS 347
# Feb 26
# 1998

**Topics**

- Flavors of integers
- Floating point
- Data structures
- Byte ordering

# 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` |

# Int vs. Long Int

## Difference Data Types

- **Long int uses quad (8-byte) word**
- **Int uses long (4-byte) word**

## Visible to C Programmer

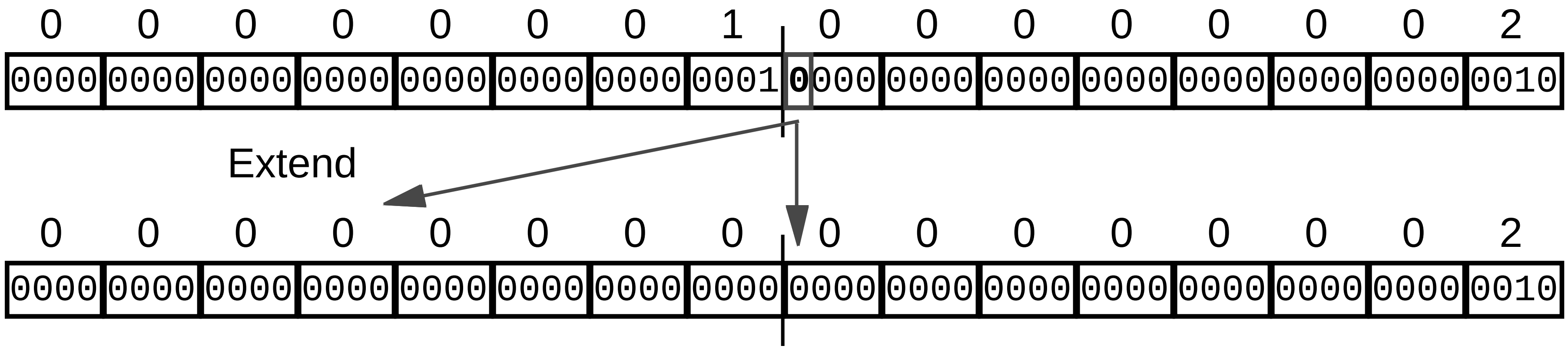
- **Long constants should be suffixed with “L”**

```
0x0000000100000002L    -->    4294967298
0x0000000100000002     -->             2    (truncated)
0x0000000080000001L    -->    2147483649
0x0000000080000001     -->   -2147483647    (extended)
```

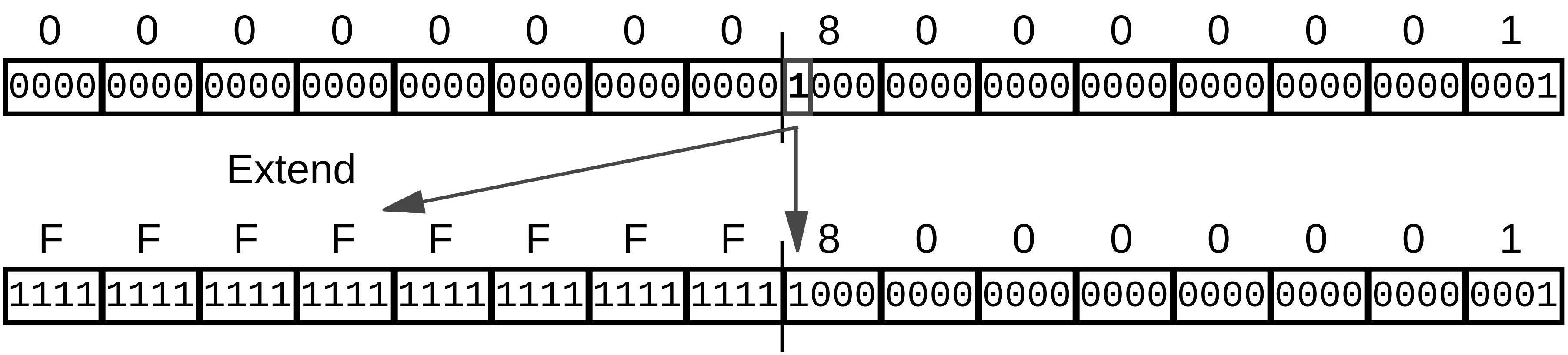
- **Printf format string should use `%ld` and `%lu`**
- **Don’t try to pack pointers into space declared for integer**
  - Pointer will be corrupted
  - Seen in code that manipulates low-level data structures

# A Closer Look at Quad --> Long

**0x0000000100000002 --> 2 (truncated)**




**0x0000000080000001 --> -2147483647 (extended)**



# Internal Representation

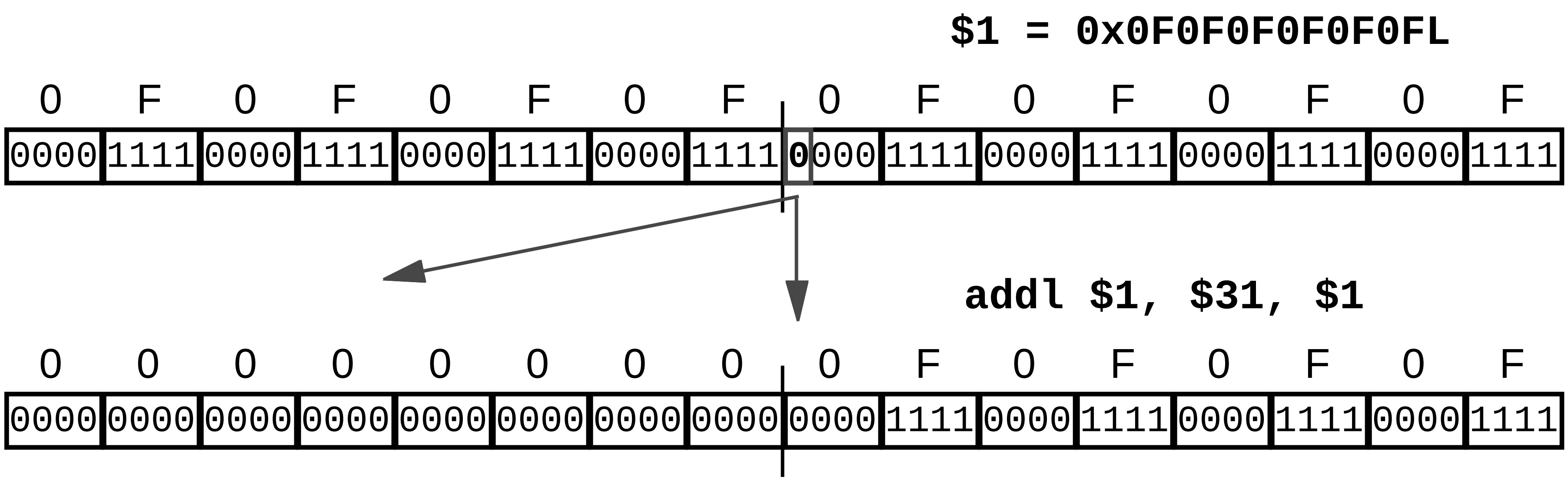
## All General Purpose Registers 8 bytes

- **Long (unsigned) int's stored in full precision form**
- **Int's stored in signed-extended form**
  - High order 33 bits all match sign bit
- **Unsigned's also stored in sign-extended form**
  - Even though really want high order 32 bits to be zero
  - Special care taken with these values

## Separate Quad and Long Word Arithmetic Instructions

- **`addq` computes sum of 8-byte arguments**
- **`addl` computes sign-extended sum of 4-byte arguments**
  - **`addl` $16, $31, $16** handy way to sign extend int in register $16
- **`ldq` reads 8 bytes from memory into register**
- **`ldl` reads 4 bytes from memory and sign extends into register**

# ADDL Example



# Integer Conversion Examples

**C Code**

**Return Value Computation**

```
int long2int(long int li)
{
  return (int) li;
}
```

```
addl $16,$31,$0  # sign extend
```

[Replace high order bits with sign]

```
long int2long(int i)
{
  return (long) i;
}
```

```
bis $16,$16,$0 # Verbatim copy
```

[Already in proper form]

```
unsigned
  ulong2uint(long unsigned ul)
{
  return (unsigned) ul;
}
```

```
addl $16,$31,$0 # sign extend
```

[Replace high order bits with sign.
Even though really want 0's]

```
long unsigned
  uint2ulong(unsigned int u)
{
  return (unsigned long) u;
}
```

```
zapnot $16,15,$0 # zero high bytes
```

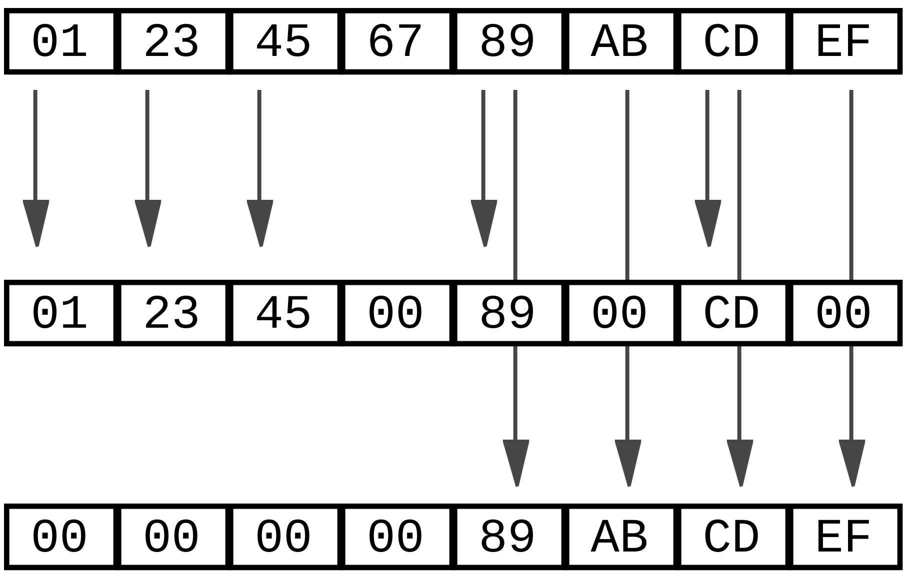
[Clear high order bits]

# Byte Zapping

## Set selected bytes to zero

- **`zap a, b, c`**
  - Low order 8 bits of b acts as mask
  - Copy nonmasked bytes from a to c
- **`zapnot a, b, c`**
  - Copy masked bytes from a to c

**`$1 = 0x0123456789ABCDEF`**




**`zap $1, 37, $2`**

**$37_{10} = 00010101_2$**

**`zapnot $1, 15, $2`**

**$15_{10} = 00001111_2$**

# Floating Point Unit

## Implemented as Separate Unit

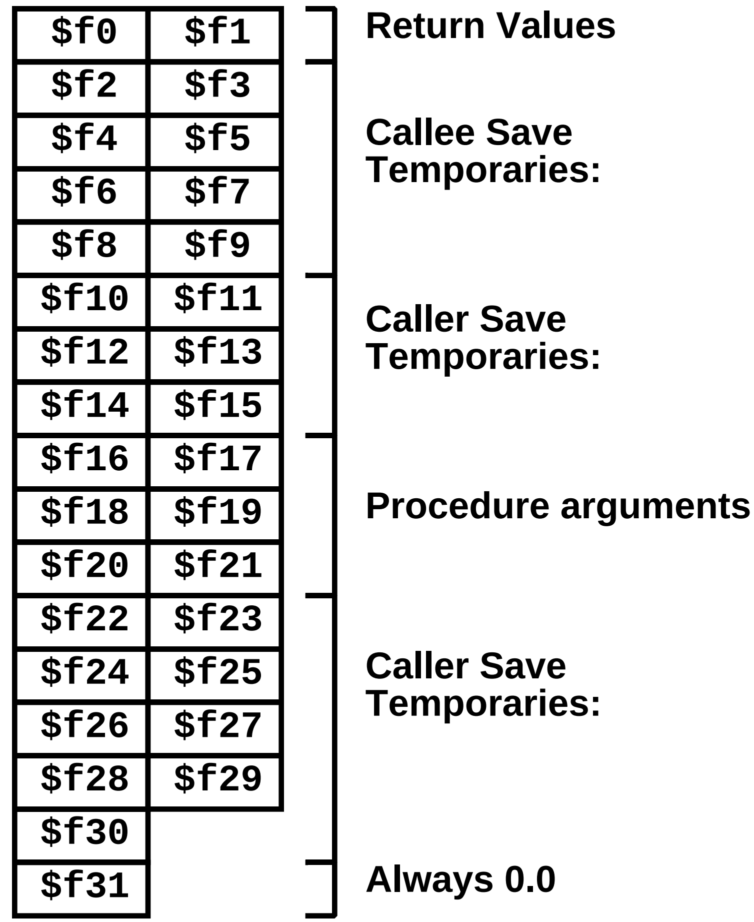
- Hardware to add, multiply, and divide
- Floating point data registers
- Various control & status registers

## Floating Point Formats

- S_Floating (C `float`): 32 bits
- T_Floating (C `double`): 64 bits

## Floating Point Data Registers

- 32 registers, each 8 bytes
- Labeled $f0 to $f31
- $f31 is always 0.0



# Floating Point Code Example

## Compute Inner Product of Two Vectors

- **Single precision**

```
float inner_prodF
  (float x[], float y[],
   int n)
{
  int i;
  float result = 0.0;
  for (i = 0; i < n; i++) {
    result += x[i] * y[i];
  }
  return result;
}
```

```
     cpys $f31,$f31,$f0 # result = 0.0
     bis $31,$31,$3     # i = 0
     cmplt $31,$18,$1   # 0 < n?
     beq $1,$102        # if not, skip loop
     .align 5
$104:
     s4addq $3,0,$1     # $1 = 4 * i
     addq $1,$16,$2     # $2 = &x[i]
     addq $1,$17,$1     # $1 = &y[i]
     lds $f1,0($2)      # $f1 = x[i]
     lds $f10,0($1)     # $f10 = y[i]
     muls $f1,$f10,$f1  # $f1 = x[i] * y[i]
     adds $f0,$f1,$f0   # result += $f1
     addl $3,1,$3       # i++
     cmplt $3,$18,$1    # i < n?
     bne $1,$104        # if so, loop
$102:
     ret $31,($26),1    # return
```

# Double Precision

```
double inner_prodD
  (double x[],
   double y[], int n)
{
  int i;
  double result = 0.0;
  for (i = 0; i < n; i++) {
    result += x[i] * y[i];
  }
  return result;
}
```

```
     cpys $f31,$f31,$f0 # result = 0.0
     bis $31,$31,$3     # i = 0
     cmplt $31,$18,$1   # 0 < n?
     beq $1,$102        # if not, skip loop
     .align 5
$104:
     s8addq $3,0,$1     # $1 = 4 * i
     addq $1,$16,$2     # $2 = &x[i]
     addq $1,$17,$1     # $1 = &y[i]
     ldt $f1,0($2)      # $f1 = x[i]
     ldt $f10,0($1)     # $f10 = y[i]
     mult $f1,$f10,$f1  # $f1 = x[i] * y[i]
     addt $f0,$f1,$f0   # result += $f1
     addl $3,1,$3       # i++
     cmplt $3,$18,$1    # i < n?
     bne $1,$104        # if so, loop
$102:
     ret $31,($26),1    # return
```

# Numeric Format Conversion

## Between Floating Point and Integer Formats

- **Special conversion instructions `cvttq`, `cvtqt`, `cvtts`, `cvtst`, …**
- **Convert source operand in one format to destination in other**
- **Both source & destination must be FP register**
  - Transfer to & from GP registers via stack store/load

**C Code**

```
float double2float(double d)
{
  return (float) d;
}
```

**Conversion Code**

```
  cvtts $f16,$f0
```

[Convert T_Floating to S_Floating]

```
double long2double(long i)
{
  return (double) i;
}
```

```
  stq $16,0($30)
  ldt $f1,0($30)
  cvtqt $f1,$f0
```

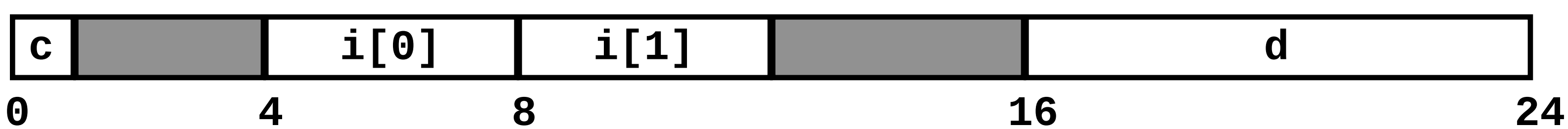
[Pass through stack and convert]

# Structure Allocation

## Principles

- **Allocate space for structure elements contiguously**
- **Access fields by offsets from initial location**
  - Offsets determined by compiler

```
typedef struct {
  char c;
  int i[2];
  double d;
} struct_ele, *struct_ptr;
```



# Alignment

## Requirements

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

## Specific Cases

- **Long word data address must be multiple of 4**
- **Quad word data address must be multiple of 8**

## Reason

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

## Compiler

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

# Structure Access

## C Code

```
int *struct_i(struct_ptr p)
{
  return p->i;
}
```

```
int struct_i1(struct_ptr p)
{
  return p->i[1];
}
```

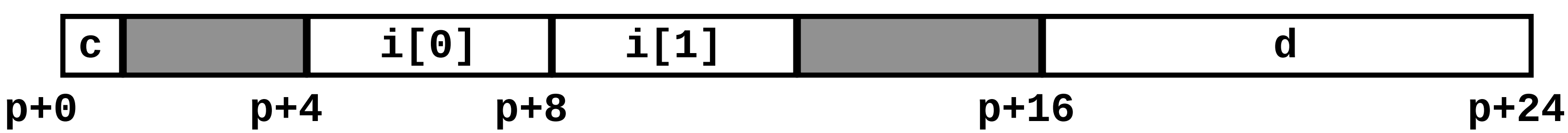
```
double struct_d(struct_ptr p)
{
  return p->d;
}
```

## Result Computation

```
# address of 4th byte
  addq $16,4,$0
```

```
# Long word at 8th byte
  ldl $0,8($16)
```

```
# Double at 16th byte
   ldt $f0,16($16)
```



# Accessing Byte in Structure

## C Code

```
char struct_c(struct_ptr p)
{
  return p->c;
}
```

## Result Computation

```
ldq_u $0,0($16)  # unaligned load
extbl $0,$16,$0  # extract byte p%8
sll $0,56,$0
sra $0,56,$0     # Sign extend char
```

## Retrieving Single Byte From Memory

**`$1 = 0x103`**

| 0x107 | | | | 0x103 | | | 0x100 |
|---|---|---|---|---|---|---|---|
| 01 | 23 | 45 | 67 | 89 | AB | CD | EF |

- **`ldq_u $2, 0($1)` loads quad word at address `0x100`**
  - Aligned quad word containing address **`0x103`**

| $2 | | | | | | | |
|---|---|---|---|---|---|---|---|
| 01 | 23 | 45 | 67 | 89 | AB | CD | EF |

# Byte Retrieval (Cont)

**$2**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 01 | 23 | 45 | 67 | 89 | AB | CD | EF |

- **`extbl $2, $1, $6` extracts byte 3 and copies into $6**
  - Uses low order 3 bits of $1 as byte number

**$6**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 00 | 00 | 00 | 00 | 00 | 00 | 00 | 89 |

- **`sll $6, 56, $6` moves low order byte to high position**

**$6**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 89 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

- **`sra $6, 56, $6` completes sign extension of selected byte**

**$6**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| FF | FF | FF | FF | FF | FF | FF | 89 |

# Arrays vs. Pointers

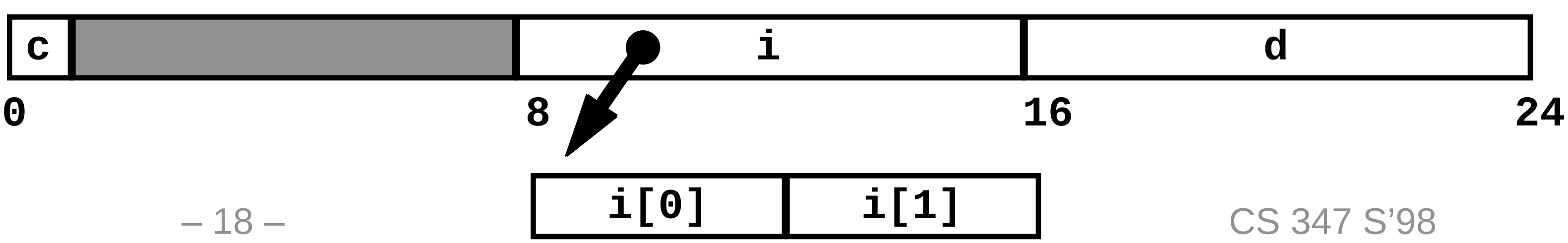
## Recall

- **Can access stored data either with pointer or array notation**
- **Differ in how storage allocated**
  - Array declaration allocates space for array elements
  - Pointer declaration allocates space for pointer only

```
typedef struct {
  char c;
  int *i;
  double d;
} pstruct_ele,
  *pstruct_ptr;
```

## C Code for Allocation

```
pstruct_ptr pstruct_alloc(void)
{
  pstruct_ptr result = (pstruct_ptr)
            malloc(sizeof(pstruct_ele));
  result->i = (int *)
            calloc(2, sizeof(int));
  return result;
}
```



# Accessing Through Pointer

## C Code

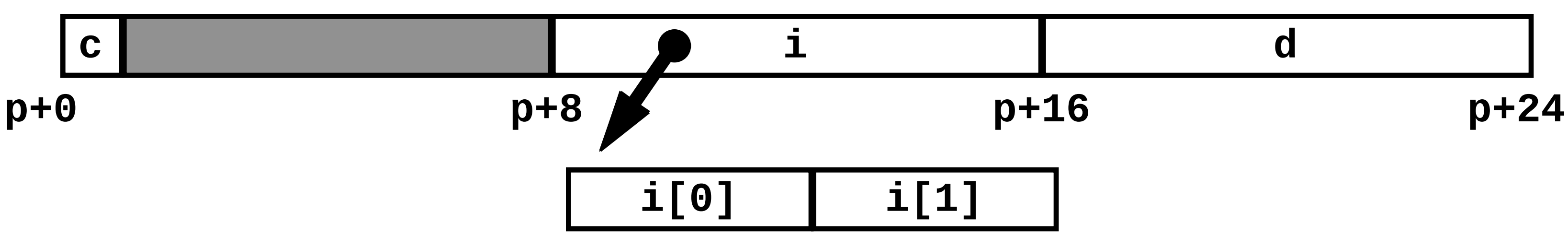
```
int *pstruct_i(pstruct_ptr p)
{
  return p->i;
}
```

## Result Computation

```
# quad word at 8th byte
  ldq $0,8($16)
```

```
int pstruct_i1(pstruct_ptr p)
{
  return p->i[1];
}
```

```
# i = quad word at 8th byte from p
  ldq $1,8($16)
# Retrieve i[1]
  ldl $0,4($1)
```



# Arrays of Structures

## Principles

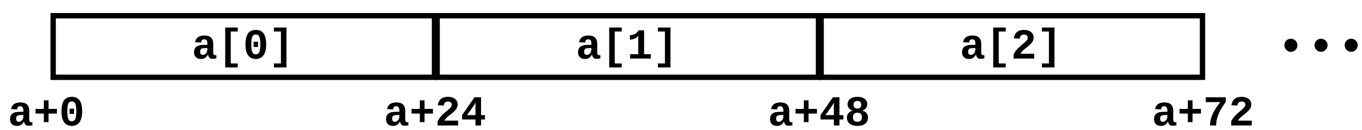
- **Allocated by repeating allocation for array type**
- **Accessed by computing address of element**
  - – Attempt to optimize
    - » Minimize use of multiplication
    - » Exploit values determined at compile time

## C Code

```
/* Index into array of
   struct_ele's */
struct_ptr a_index
    (struct_ele a[], int idx)
{
  return &a[idx];
}
```

## Address Computation

```
s4subq $17,$17,$0   # 3 * idx
s8addq $0,$16,$0    # 24*idx + a
```

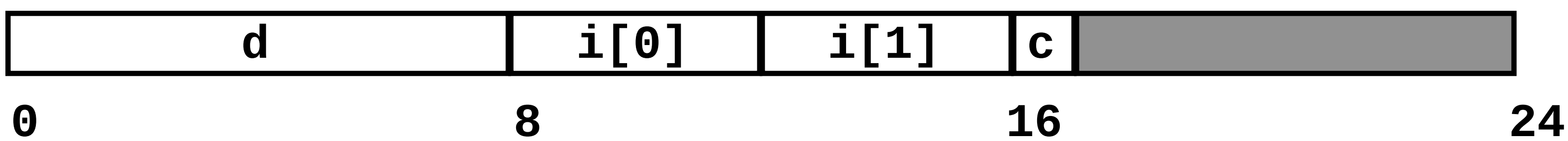


# Aligning Array Elements

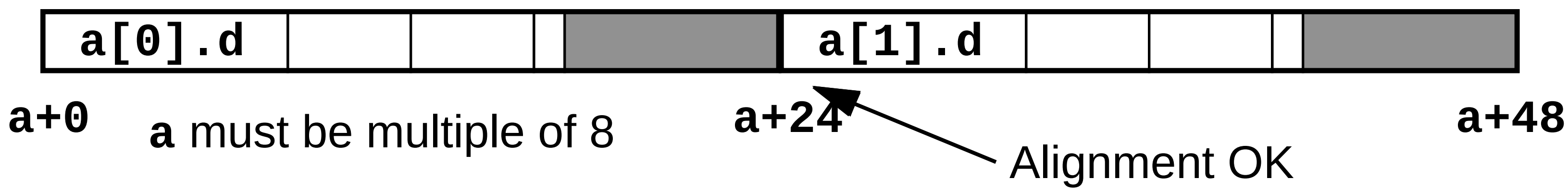
## Requirement

- Must make sure alignment requirements met when allocate array of structures
- May require inserting unused space at end of structure

```
typedef struct {
  double d;
  int i[2];
  char c;
} rev_ele, *rev_ptr;
```




```
rev_ele a[2];
```

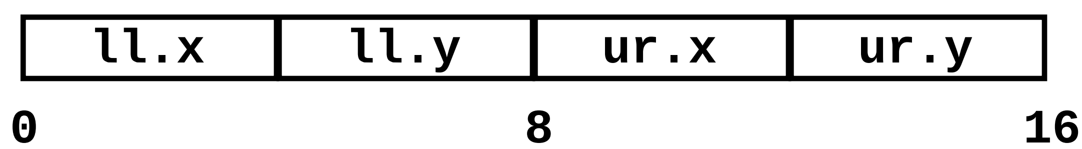


# Nested Allocations

## Principles

- **Can nest declarations of arrays and structures**
- **Compiler keeps track of allocation and access requirements**

```
typedef struct {
  int x;
  int y;
} point_ele, *point_ptr;

typedef struct {
  point_ele ll;
  point_ele ur;
} rect_ele, *rect_ptr;
```



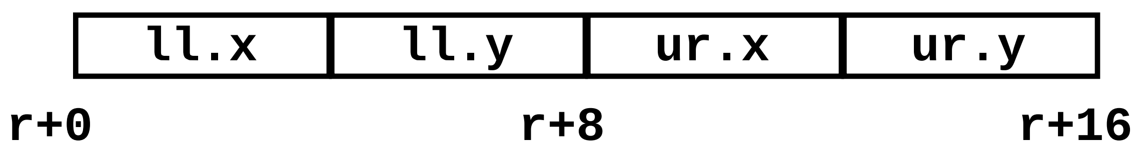
# Nested Allocation (cont.)

## C Code

```
int area(rect_ptr r)
{
  int width =
      r->ur.x - r->ll.x;
  int height =
      r->ur.y - r->ll.y;
  return width * height;
}
```

## Computation

```
ldl $2,8($16)  # $2 = ur.x
ldl $1,0($16)  # $1 = ll.x
subl $2,$1,$2  # $2 = width
ldl $0,12($16) # $0 = ur.y
ldl $1,4($16)  # $1 = ll.y
subl $0,$1,$0  # $0 = height
mull $2,$0,$0  # $0 = area
```

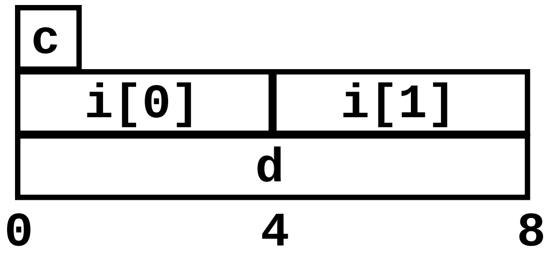


# Union Allocation

## Principles

- Overlay union elements
- Allocate according to largest element
- Programmer responsible for collision avoidance

```
typedef union {
  char c;
  int i[2];
  double d;
} union_ele, *union_ptr;
```



# Example Use of Union

- Structure can hold 3 kinds of data
- Never use 2 forms simultaneously
- Identify particular kind with flag `type`

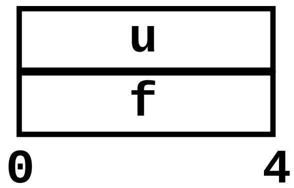
```
typedef enum { CHAR, INT, DOUBLE } utype;

typedef struct {
  utype type;
  union_ele e;
} store_ele, *store_ptr;
```

```
void print_store(store_ptr p)
{
  switch (p->type) {
  case CHAR:
    printf("Char = %c\n", p->e.c);
    break;
  case INT:
    printf("Int[0] = %d, Int[1] = %d\n",
           p->e.i[0], p->e.i[1]);
    break;
  case DOUBLE:
    printf("Double = %g\n", p->e.d);
  }
}
```

# Using Union to Access Bit Patterns

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

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




***AFS347*/asst/h2/ftest.c**

- **Get direct access to bit representation of `float`**
- **`bit2float` generates float with given bit pattern**
  - –NOT the same as `(float) u`
- **`show_parts` extracts different components of float**

```
void show_parts(float f) {
  int sign, exp, significand;
  bit_float_t arg;
  arg.f = f;
  /* Get bit 31 */
  sign = (arg.u >> 31) & 0x1;
  /* Get bits 30 .. 23 */
  exp = (arg.u >> 23) & 0xFF;
  /* Get bits 22 .. 0 */
  significand = arg.u & 0x7FFFFF;
  • • •
}
```

# Byte Ordering

## Idea

- Bytes in long word numbered 0 to 3
- Which is most (least) significant?
- Can cause problems when exchanging binary data between machines

## Big Endian

- Byte 0 is most, 3 is least
- IBM 360/370, Motorola 68K, Sparc

## Little Endian

- Byte 0 is least, 3 is most
- 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;
```

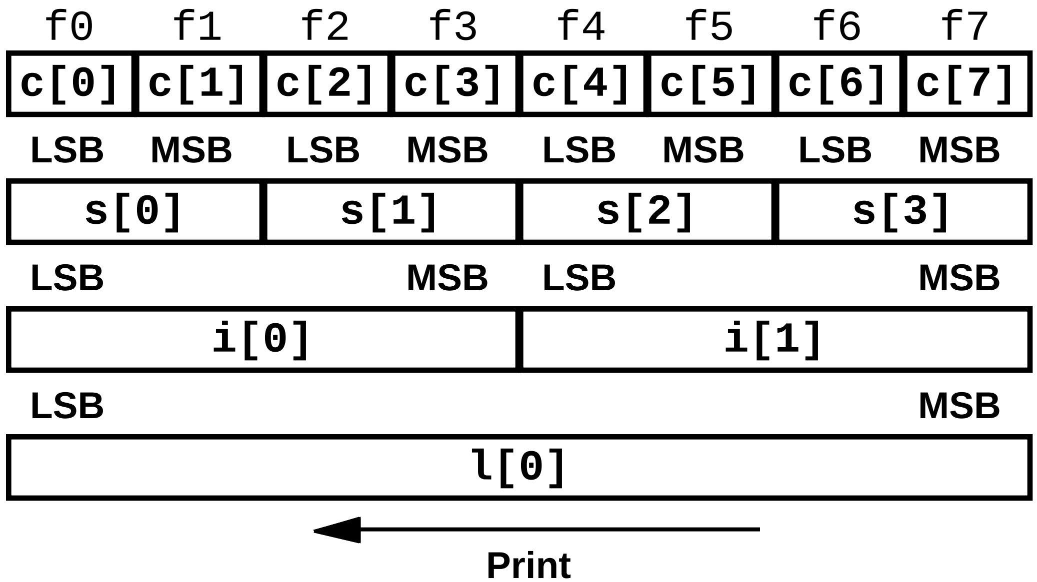
<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

# 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

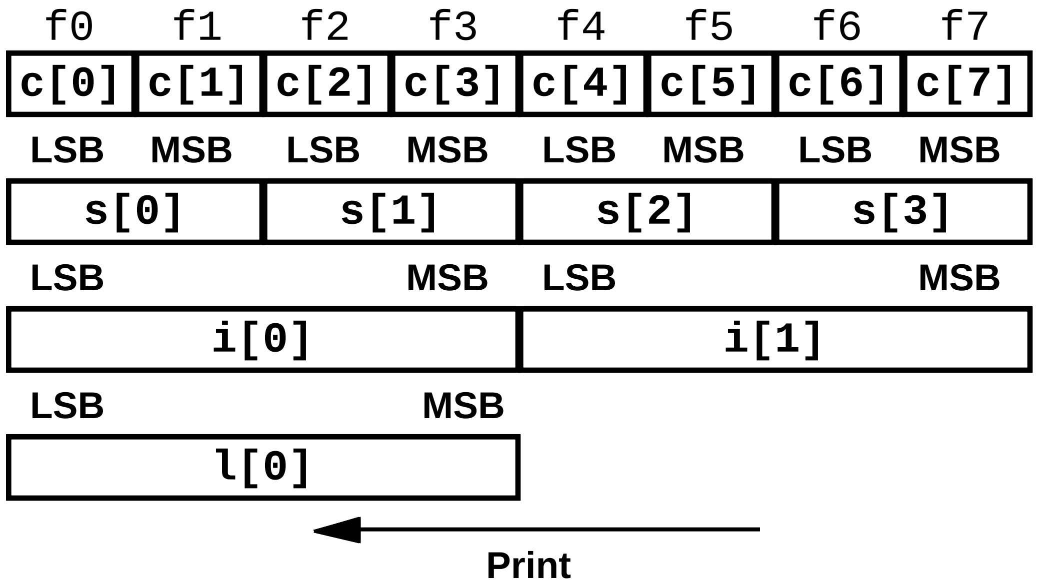



## Output on Alpha:

```
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

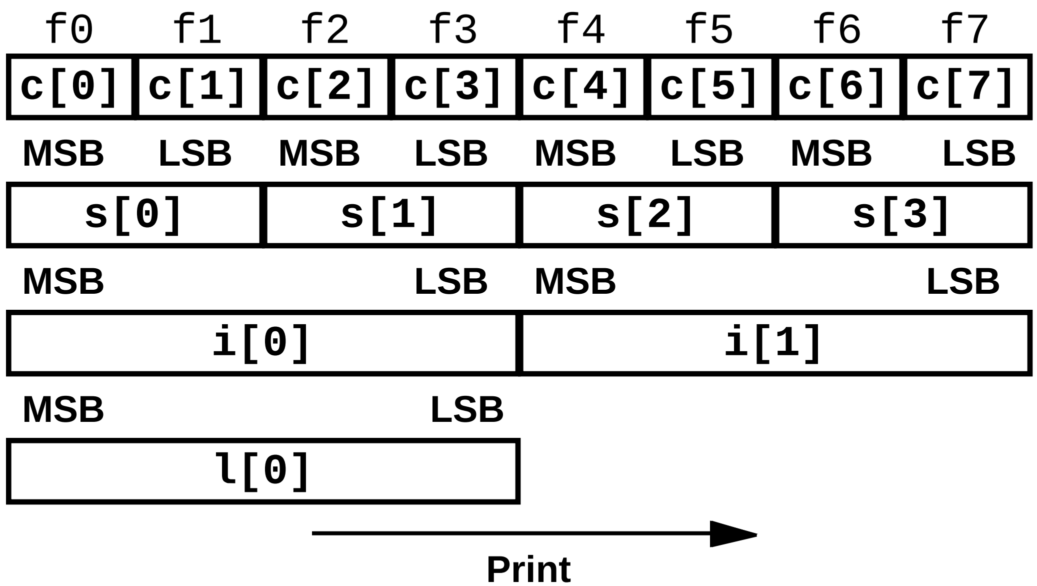



## Output on Pentium:

```
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



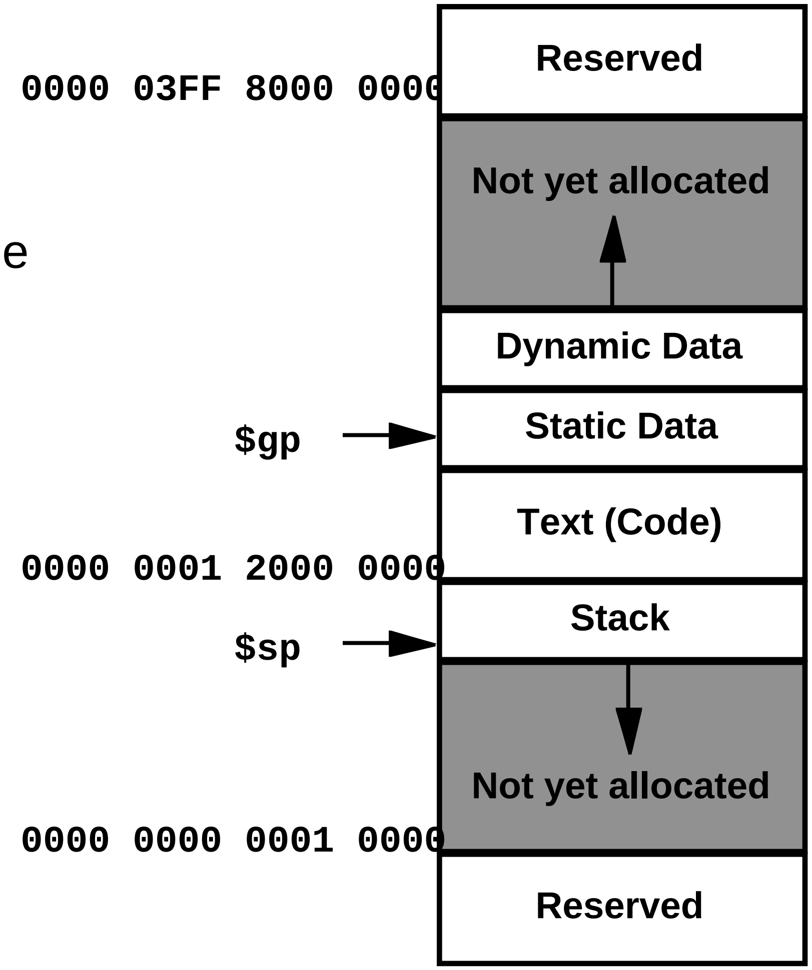

## Output on Sun:

```
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
    - » I/O devices, process info, etc.



# RISC Principles Summary

## Simple & Regular Instructions

- **Small number of uniform formats**
- **Each operation does just one thing**
  - Memory access, computation, conditional, etc.

## Encourage Register Usage over Memory

- **Operate on register data**
  - Load/store architecture
- **Procedure linkage**

## Rely on Optimizing Compiler

- **Data allocation & referencing**
- **Register allocation**
- **Improve efficiency of user’s code**