Machine-Level Programming: Data Structures CS 347

Feb. 6, 1996

Topics

- **Basic data types**
- **Structures, unions, and arrays**
  - Allocation
  - Accessing elements
  - Alignment & byte ordering
- **Memory layout**
- **RISC Principles Summary**

Basic Data Types

Integral

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

MIPS	Bytes	C
byte	1	`[unsigned] char`
half word	2	`[unsigned] short`
word	4	`[unsigned] [long] int`
		pointer

Floating Point

- Stored & operated on in floating point registers
- Special instructions for two different precisions

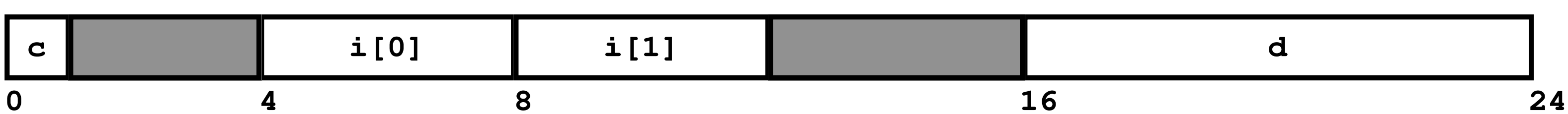
MIPS	Bytes	C
single precision	4	`float`
double precision	8	`double`

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

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

Specific Cases

- **Word data address must be multiple of 4**
- **Doubleword data address must be multiple of 8**

Reason

- **Memory accessed by (aligned) words**
  – Inefficient to load or store data that spans word boundaries

Compiler

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

Structure Access

C Code

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

```
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;
}
```

Assembler

```
  lb     $2,0($4)    # 1st byte
  j      $31
```

```
  J      $31
  addu   $2,$4,4     # addr of 4th byte
```

```
  lw     $2,8($4)    # word at 8th byte
  j      $31
```

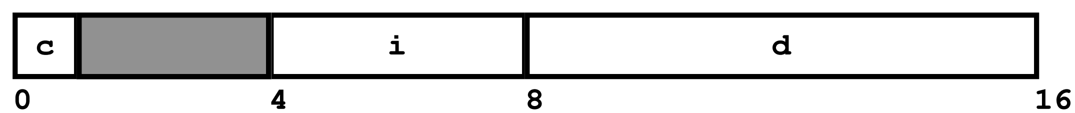
```
  l.d    $f0,16($4) # dblewd at 16th by.
  j      $31
```

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;
}
```

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

Assembler

```
  lw     $2,4($4)   # word at 4th byte
  j      $31
```

```
  # i = word at 4th byte from p
  lw     $2,4($4)
  # Retrieve word at 4th byte from i
  lw     $2,4($2)
  j      $31
```

Arrays

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];
}
```

Assembler

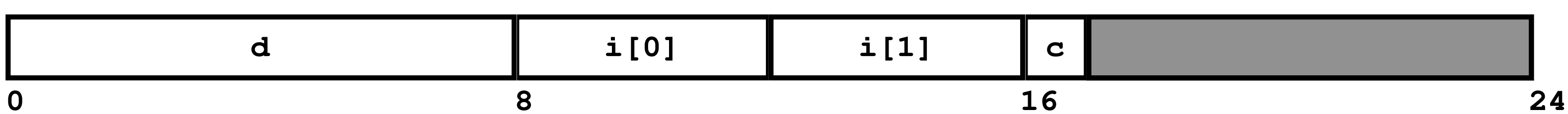
```
  sll    $2,$5,1
  addu   $2,$2,$5
  sll    $2,$2,3     # $2 = 24 * idx
  J      $31         # return
    # with a + (scaled) idx
  addu   $2,$4,$2
```

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;
```



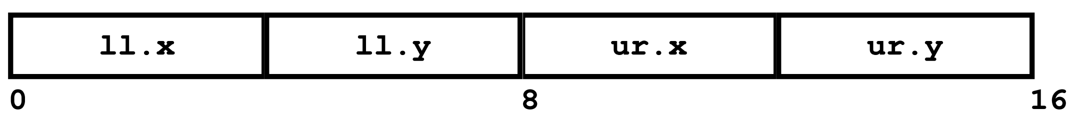
Nested Allocations

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

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

Assembler

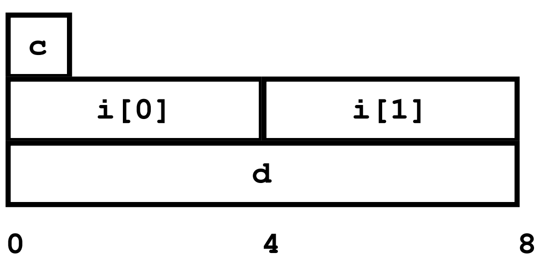
```
    lw      $2,8($4)     # $2 = ur.x
    lw      $3,0($4)     # $3 = ll.x
    subu    $2,$2,$3     # $2 = width
    lw      $3,12($4)    # $3 = ur.y
    lw      $4,4($4)     # $4 = ll.y
    subu    $3,$3,$4     # $3 = height
    mult    $2,$3
    mflo    $2           # width * height
    j       $31
```

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;
```



Byte Ordering

Idea

- Bytes in 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

MIPS

- Can be configured either way
- DECStation’s are little endian

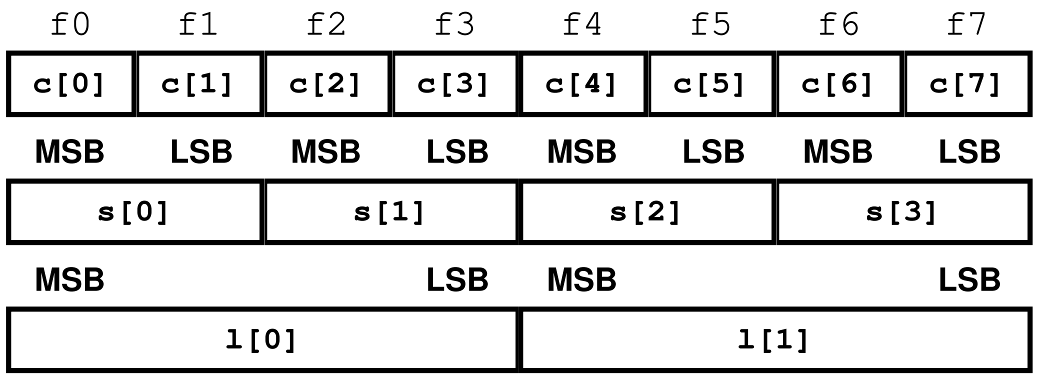
Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned long l[2];
} dw;
int i;
for (i = 0; i < 8; i++)
  dw.c[i] = 0xf0 + i;
printf("Characters 0-7 == [0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
       (long) dw.c[0], (long) dw.c[1], (long) dw.c[2], (long) dw.c[3],
       (long) dw.c[4], (long) dw.c[5], (long) dw.c[6], (long) dw.c[7]);
printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
       (long) dw.s[0], (long) dw.s[1], (long) dw.s[2], (long) dw.s[3]);
printf("Longs 0-1 == [0x%x,0x%x]\n",
       dw.l[0], dw.l[1]);
```

<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">l[0]</td><td colspan="4">l[1]</td></tr>
</table>

Byte Ordering Example (cont.)

Big Endian

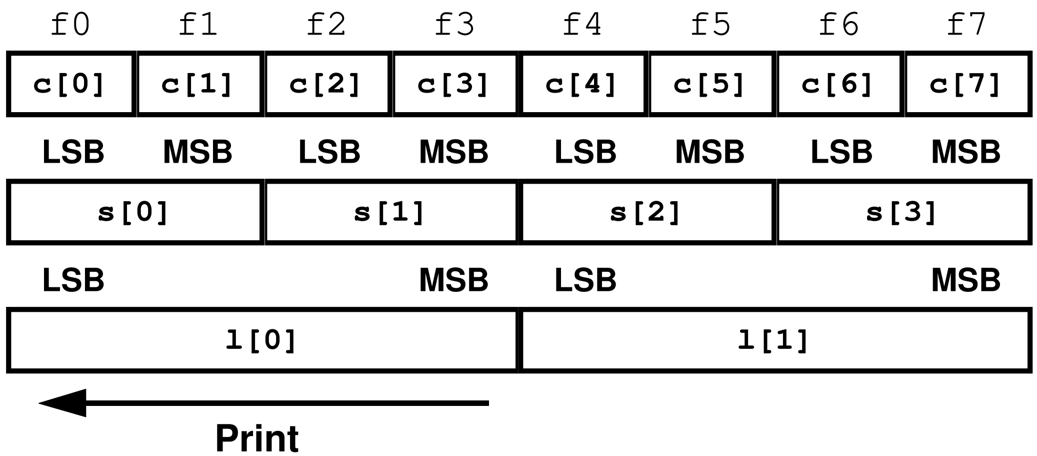


Output on Sun:

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

Byte Ordering Example (cont.)

Little Endian



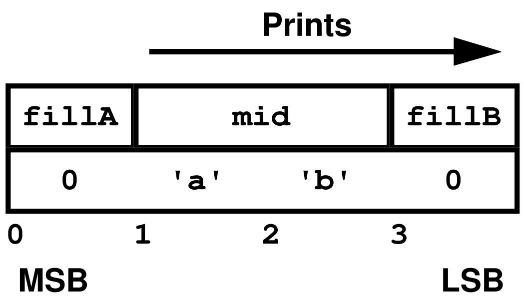
Output on DecStation:

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

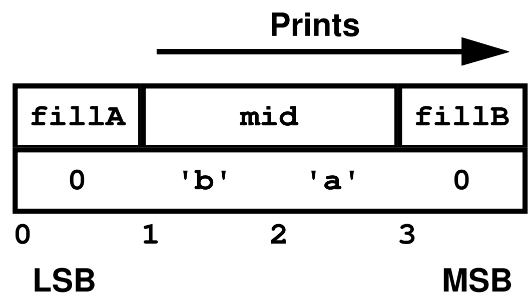
Byte Ordering Example #2

```
union {
  struct {
    char fillA;
    char mid[2];
    char fillB;
} bv;
  int  iv;
} word;
word.iv = ('a' << 16) | ('b' << 8);
printf("The word is '%s'\n", word.bv.mid);
```

Big Endian



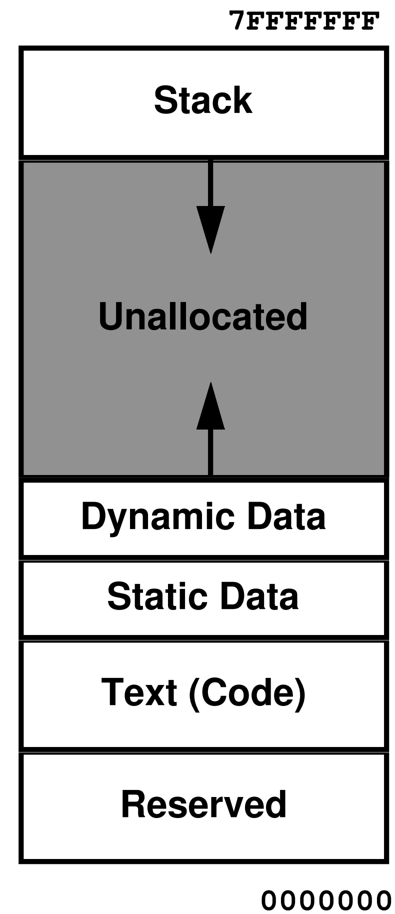
Little Endian



MIPS Memory Layout

Segments

- **Stack**
  - Grows down from top
  - Implements runtime stack
- **Data**
  - Static space for global variables
    - » Allocation determined at compile time
  - Dynamic space for runtime allocation
    - » E.g., using malloc
- **Text**
  - Stores machine code for program
- **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**
- **Dealing with implementation artifacts**
  - e.g., branch delay slots