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

<table>
<thead>
<tr>
<th>Alpha</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>1</td>
<td>[unsigned] char</td>
</tr>
<tr>
<td>word</td>
<td>2</td>
<td>[unsigned] short</td>
</tr>
<tr>
<td>long word</td>
<td>4</td>
<td>[unsigned] int</td>
</tr>
<tr>
<td>quad word</td>
<td>8</td>
<td>[unsigned] long int, pointers</td>
</tr>
</tbody>
</table>

Floating Point

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

<table>
<thead>
<tr>
<th>Alpha</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_floating</td>
<td>4</td>
<td>float</td>
</tr>
<tr>
<td>T_floating</td>
<td>8</td>
<td>double</td>
</tr>
</tbody>
</table>
Structures

Concept

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

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

Accessing Structure Member

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

Annotated Assembly

```assembly
set_i:
    stq $17,0($16)  # r->i = val
    ret $31,($26),1
```
Generating Pointer to Structure Member

```c
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

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

```asm
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

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

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

```assembly
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
```

Diagram:
```
<table>
<thead>
<tr>
<th>i</th>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>32</td>
</tr>
</tbody>
</table>
```

Element i
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;
```

```
c i[0] i[1] v
```

```
p+0  p+4  p+8  p+16  p+24
```

- Multiple of 4
- Multiple of 8
- Multiple of 8
- Multiple of 8

class09.ppt
Effect of Overall Alignment Requirement

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

A pointer `p` must be a multiple of 8 to fit all members of `S2`.

---

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

A pointer `p` must be a multiple of 4 to fit all members of `S3`. 
Ordering Elements Within Structure

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

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

10 bytes wasted space

2 bytes wasted space
Arrays of Structures

Principle

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

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

```
<table>
<thead>
<tr>
<th>a[1].i</th>
<th>a[1].v</th>
<th>a[1].j</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+24</td>
<td>a+32</td>
<td>a+40</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>a[0]</th>
<th>a[1]</th>
<th>a[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+0</td>
<td>a+24</td>
<td>a+48</td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a+48</td>
<td>a+72</td>
<td></td>
</tr>
</tbody>
</table>
```
Accessing Element within Array

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

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

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

```c
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
Byte-Level Memory Operation Example

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

Increasing address

<table>
<thead>
<tr>
<th>0x107</th>
<th>0x103</th>
<th>0x100</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>23</td>
<td>45</td>
</tr>
<tr>
<td>67</td>
<td>89</td>
<td>AB</td>
</tr>
<tr>
<td>CD</td>
<td>EF</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x20f</th>
<th>0x20d</th>
<th>0x208</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF</td>
<td>EE</td>
<td>DD</td>
</tr>
<tr>
<td>CC</td>
<td>BB</td>
<td>AA</td>
</tr>
<tr>
<td>99</td>
<td>88</td>
<td></td>
</tr>
</tbody>
</table>

Desired Effect:

<table>
<thead>
<tr>
<th>0x20f</th>
<th>0x20d</th>
<th>0x208</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF</td>
<td>EE</td>
<td>89</td>
</tr>
<tr>
<td>CC</td>
<td>BB</td>
<td>AA</td>
</tr>
<tr>
<td>99</td>
<td>88</td>
<td></td>
</tr>
</tbody>
</table>
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 \texttt{src} = \texttt{0x103}

- Pointer \texttt{src} in register $17$
  
  \texttt{ldq$_u$ \$1, 0($\$17$)}
  
  - Rounds effective address \texttt{0x103} to nearest quad word boundary (\texttt{0x100})
    
    » By setting low 3 bits of address to \texttt{000}_2
  
  - Loads entire quad word into memory

  \begin{center}
  $1: \begin{array}{ccccccccccc}
  01 & 23 & 45 & 67 & 89 & AB & CD & EF \\
  \end{array}$
  \end{center}

- \texttt{extbl \$1, \$17, \$1}$
  
  - Uses lower 3 bits of \texttt{$17$} as byte offset (= 3)
  
  - Sets destination register to that byte of source

  \begin{center}
  $1: \begin{array}{ccccccccccc}
  00 & 00 & 00 & 00 & 00 & 00 & 00 & 89 \\
  \end{array}$
  \end{center}
Preparing Destination

Step 2. Zero byte at \texttt{dest} = \texttt{0x20d}

- Pointer \texttt{dest} in register \$16
  
  \texttt{ldq\_u \$2, 0\(\$16\)}
  
  - Rounds effective address \texttt{0x20d} down to nearest quad word boundary (\texttt{0x208})
  
  - Loads entire quad word into register

  \begin{center}
  $2$: FF EE DD CC BB AA 99 88
  \end{center}

  Byte Number: 7 6 5 4 3 2 1 0

- \texttt{mskbl \$2, \$16, \$2}

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

  \begin{center}
  $2$: FF EE 00 CC BB AA 99 88
  \end{center}

  Byte Number: 7 6 5 4 3 2 1 0
Merging Source into Destination

Step 3. Position Source Byte

$1: \begin{array}{cccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 89
\end{array}$

- Pointer dest in register $\$16$

```plaintext
insbl $1, \$16, \$1$
```

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

$1: \begin{array}{cccccccccc}
0 & 0 & 0 & 89 & 00 & 00 & 00 & 00 & 00 & 00
\end{array}$

$2: \begin{array}{cccccccccc}
FF & EE & 00 & CC & BB & AA & 99 & 88
\end{array}$

Step 4. Form Destination Quad-Word

```plaintext
bis $1, \$2, \$2$
```

- Merges new byte into destination

$2: \begin{array}{cccccccccc}
FF & EE & 89 & CC & BB & AA & 99 & 88
\end{array}$
Updating Destination

Step 5. Update Destination Quad-Word

$2: \text{FF EE } 89 \text{ CC BB AA 99 88}

- Pointer $\text{dest}$ in register $\$16$

\text{stq}_\text{u} \$2, 0(\$16)

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

Net Effect

- Before:

  $0x20f \quad 0x20d \quad 0x208$

  \[
  \begin{array}{cccccccc}
  \text{FF} & \text{EE} & \text{DD} & \text{CC} & \text{BB} & \text{AA} & 99 & 88 \\
  \end{array}
  \]

- After:

  $0x20f \quad 0x20d \quad 0x208$

  \[
  \begin{array}{cccccccc}
  \text{FF} & \text{EE} & 89 & \text{CC} & \text{BB} & \text{AA} & 99 & 88 \\
  \end{array}
  \]
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

```c
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;
```

```
+---+----+----+
| k.e | k.e.i[0] | k.e.i[1] |
+---+----+----+
    | k.e.c |
```

```
+----+----+----+
| k.type | k.e.v |
+----+----+----+
    | k.e |
```
Using “Tagged” Union

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

```
0 'a'
```

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

```
1 17 47
```

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

```
2 FF00FF00FF00FF00
```
Using Union to Access Bit Patterns

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

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

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

- 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`
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;

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i[0]</td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Byte Ordering Example (Cont).

```c
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.