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

<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>
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

0x0000000010000002 --> 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

$1 = 0x0F0F0F0F0F0FL

```
0  F  O  F  O  F  O  F  O  F  O  F  O  F  O  F  O  F  O  F  O  F  O  F
0000111100001111000011110000111100001111000011110000111100001111
```

```
addl $1, $31, $1
0  0  0  0  0  0  0  0  0  0  0  0  F  O  F  O  F  O  F  O  F  O  F  O  F  O  F
00000000000000000000000000000000000000000000001111000011110000111100001111
```
Integer Conversion Examples

C Code

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

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

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

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

Return Value Computation

```c
addl $16,$31,$0  # sign extend
[Replace high order bits with sign]
```

```c
bis $16,$16,$0 # Verbatim copy
[Already in proper form]
```

```c
addl $16,$31,$0 # sign extend
[Replace high order bits with sign. Even though really want 0’s]
```

```c
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

<table>
<thead>
<tr>
<th>Return Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f0$</td>
</tr>
<tr>
<td>$f2$</td>
</tr>
<tr>
<td>$f4$</td>
</tr>
<tr>
<td>$f6$</td>
</tr>
<tr>
<td>$f8$</td>
</tr>
<tr>
<td>$f10$</td>
</tr>
<tr>
<td>$f12$</td>
</tr>
<tr>
<td>$f14$</td>
</tr>
<tr>
<td>$f16$</td>
</tr>
<tr>
<td>$f18$</td>
</tr>
<tr>
<td>$f20$</td>
</tr>
<tr>
<td>$f22$</td>
</tr>
<tr>
<td>$f24$</td>
</tr>
<tr>
<td>$f26$</td>
</tr>
<tr>
<td>$f28$</td>
</tr>
<tr>
<td>$f30$</td>
</tr>
<tr>
<td>$f31$</td>
</tr>
</tbody>
</table>

Callee Save Temporaries:

Caller Save Temporaries:

Procedure arguments

Caller Save Temporaries:

Always 0.0
Floating Point Code Example

Compute Inner Product of Two Vectors

- Single precision

```c
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;
}
```

```assembly
    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;
}
Numeric Format Conversion

Between Floating Point and Integer Formats

- Special conversion instructions `cvttq`, `cvtqt`, `cvtt`, `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

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

Conversion Code

```asm
  cvtts $f16,$f0  
  [Convert T_Floating to S_Floating]
```

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

```asm
  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

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

```
c i[0] i[1] d
0   4   8  16  24
```
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

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

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

```c
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)
  ```

Diagram:

```
c i[0] i[1] d
p+0 p+4 p+8 p+16 p+24
```
# Accessing Byte in Structure

## C Code

```c
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

$1 = 0x103$

<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>

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

```c
$2$
```

<table>
<thead>
<tr>
<th>01</th>
<th>23</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>89</td>
<td>AB</td>
</tr>
<tr>
<td>CD</td>
<td>EF</td>
<td></td>
</tr>
</tbody>
</table>
Byte Retrieval (Cont)

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

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

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

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

C Code for Allocation
Accessing Through Pointer

C Code

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

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

Result Computation

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

# 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

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

Address Computation

```plaintext
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];

Alignment OK
Nested Allocations

Principles

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

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

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

<table>
<thead>
<tr>
<th>ll.x</th>
<th>ll.y</th>
<th>ur.x</th>
<th>ur.y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
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;
}
```

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
mul $2,$0,$0      # $0 = area
```

<table>
<thead>
<tr>
<th>ll.x</th>
<th>ll.y</th>
<th>ur.x</th>
<th>ur.y</th>
</tr>
</thead>
<tbody>
<tr>
<td>r+0</td>
<td>r+0</td>
<td>r+8</td>
<td>r+16</td>
</tr>
</tbody>
</table>
Union Allocation

Principles

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

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

```
<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Example Use of Union

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

```c
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;
}

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 & 0x7FFFFFFF;
    ...}

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
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>
<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>
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]\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%1x]\n",
    dw.l[0]);
Byte Ordering on Alpha

Little Endian

```
<table>
<thead>
<tr>
<th>f0</th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
<th>f5</th>
<th>f6</th>
<th>f7</th>
</tr>
</thead>
</table>

LSB | MSB | LSB | MSB | LSB | MSB | LSB | MSB |
---|-----|-----|-----|-----|-----|-----|-----|

LSB | MSB | LSB | MSB |
---|-----|-----|-----|
| i[0]| i[1]|

LSB | MSB |
---|-----|
| l[0]|
```

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

\[
\begin{array}{cccccccc}
 f_0 & f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\end{array}
\]

\[
\begin{array}{cccc}
 i[0] & i[1] \\
\end{array}
\]

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

\[ \begin{array}{cccccccc}
  f_0 & f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 \\
\end{array} \]

\[ \begin{array}{cccccc}
  MSB & LSB & MSB & LSB & MSB & LSB \\
\end{array} \]

\[ \begin{array}{cccc}
  i[0] & i[1] \\
  MSB & LSB \\
\end{array} \]

\[ \begin{array}{cccc}
  l[0] \\
  MSB & LSB \\
\end{array} \]

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