CS429: Computer Organization and Architecture

Instruction Set Architecture V

Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: October 31, 2017 at 09:37
Basic Data Types

Integral

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

<table>
<thead>
<tr>
<th>Intel</th>
<th>GAS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>b</td>
<td>1</td>
<td>[unsigned] char</td>
</tr>
<tr>
<td>word</td>
<td>w</td>
<td>2</td>
<td>[unsigned] short</td>
</tr>
<tr>
<td>double word</td>
<td>l</td>
<td>4</td>
<td>[unsigned] int</td>
</tr>
<tr>
<td>quad word</td>
<td>q</td>
<td>8</td>
<td>[unsigned] long int</td>
</tr>
</tbody>
</table>

Floating Point

Stored and operated on in floating point registers.

<table>
<thead>
<tr>
<th>Intel</th>
<th>GAS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>s</td>
<td>4</td>
<td>float</td>
</tr>
<tr>
<td>Double</td>
<td>l</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>Extended</td>
<td>t</td>
<td>10/12</td>
<td>long double</td>
</tr>
</tbody>
</table>
Basic Principle: \( T \ [A[L]] \)
- Array (named A) of data type \( T \) and length \( L \).
- Contiguously allocated region of \( L \times \text{sizeof}(T) \) bytes.

```c
char string[12];
int val[5];
double a[3];
char *p[3];
```
int val[5];

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td>x</td>
</tr>
<tr>
<td>val+1</td>
<td>int *</td>
<td>x + 4</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td>x + 8</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>??</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>val+j</td>
<td>int *</td>
<td>x + 4j</td>
</tr>
</tbody>
</table>

Note the use of pointer arithmetic.
Array Example

Example arrays were allocated in successive 20 byte block.

That’s not guaranteed to happen in general.

#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

Declaration zip_dig cmu is equivalent to int cmu[5].
int get_digit
    ( zip_digit z, int dig )
{
    return z[dig];
}

Memory Reference Code

# %rdi = z
# %rsi = dig
movl (%rdi,%rsi,4),%eax # z[dig]

Computation

- Register %rdi contains the starting address of the array.
- Register %rsi contains the array index.
- The desired digit is at %rdi + (4 * %rsi).
- User memory reference (%rdi,%rsi,4).
Array Loop Example

```c
void zincr( zipped z ) {
    size_t i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

```
# %rdi = z
movl $0, %eax
jmp .L3    # i = 0
.L4:
    addl $1, (%rdi, %rax, 4)    # z[i]++
    addq $1, %rax    # i++
.L3:
    cmpq $4, %rax    # i:4
    jbe .L4    # if <=, goto loop
    ret    # return
```
Multidimensional (Nested) Arrays

**Declaration:** \( T \ A[R][C] \);
- 2D array of data type \( T \)
- \( R \) rows, \( C \) columns
- Type \( T \) element requires \( K \) bytes

**Array Size:** \( R \times C \times K \) bytes

**Arrangement:** Row-Major ordering (guaranteed)

Row major order means the elements are stored in the following order:

\[
[A_0,0, \ldots, A_0,C-1, A_1,0, \ldots, A_1,C-1, \ldots, A_{R-1},0, \ldots, A_{R-1},C-1].
\]
**Declaration:** \( T \ A[R][C] \);

- 2D array of data type \( T \)
- \( R \) rows, \( C \) columns
- Type \( T \) element requires \( K \) bytes

To access element \( A[i][j] \), perform the following computation:

\[
A + i \times C \times K + j \times K
\]
Nested Array Example

```c
#define PCOUNT 4
zip_dig pgh[PCOUNT] = 
{ {1, 5, 2, 0, 6},
  {1, 5, 2, 1, 3},
  {1, 5, 2, 1, 7},
  {1, 5, 2, 2, 1} };
```

- Declaration “zip_dig pgh[4]” is equivalent to “int pgh[4][5].”
- Variable `pgh` denotes an array of 4 elements allocated contiguously.
- Each element is an array of 5 ints, which are allocated contiguously.
- This is “row-major” ordering of all elements, guaranteed.
Row Vectors:

Given a nested array declaration `T A[R][C]`, you can think of this as an array of arrays.

- `A[i]` is an array of `C` elements.
- Each element of `A[i]` has type `T`, and requires `K` bytes.
- The starting address of `A[i]` is `A + i * C * K`.

```
  A[0][0] ... A[0][C-1]
    |       |        |
    +---+---+----+
    |       |        |       |
    +---+---+----+
```

```
A+i*C*4
```

```
A+(R-1)*C*4
```
Array Elements

- \( A[i][j] \) is an element of type \( T \), which requires \( K \) bytes.
- The address is \( A + (i \times C + j) \times K \).
Multi-Level Array Example

```c
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- Variable `univ` denotes an array of 3 elements.
- Each element is a pointer (8 bytes).
- Each pointer points to an array of ints (may vary in length).

![Diagram of multi-level array example]
Element Access in a Multi-Level Array

```c
int get_univ_digit(size_t index, size_t dig)
{
    return univ[index][dig];
}
```

**Computation**

- **Element access**
  
  Mem[Mem[univ+8*index] + 4*dig]

- **Must do two memory reads:**
  
  - First get pointer to row array.
  - Then access element within the row.

```assembly
salq $2, %rsi          # 4*dig
addq univ(,%rdi,8),%rsi # p = univ[dig] + 4*dig
movl (%rsi), %eax     # return *p
ret
```
Array Element Accesses

Nested Array

```c
int get_pgh_digit
(size_t index,
 size_t dig)
{
    return pgh[index][dig];
}
```

Element at
Mem[pgh+20*index+4*dig]

Multi-Level Array

```c
int get_univ_digit
(size_t index,
 size_t dig)
{
    return univ[index][dig];
}
```

Element at
Mem[Mem[univ+8*index]+4*dig]

Similar C references, but different address computations.
**N x N Matrix Code**

**Fixed dimensions:**
Know value of N at compile time.

**Variable dimensions, explicit indexing:**
Traditional way to implement dynamic arrays

**Variable dimensions, implicit indexing:**
Now supported by gcc

```c
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele( fix_matrix a,
           size_t i, size_t j ) {
    return a[i][j];
}

#define IDX(n, i, j) ((i) * (n) + (j))
/* Get element a[i][j] */
int vec_ele( size_t n, int *a,
            size_t i, size_t j ) {
    return a[IDX(n, i, j)];
}

/* Get element a[i][j] */
int var_ele( size_t n, int a[n][n],
            size_t i, size_t j ) {
    return a[i][j];
}
```
Array Elements

- Address \( A + i \times (C \times K) + j \times K \)
- \( C = 16, K = 4 \)

```c
/* Get element a[i][j] */
int fix_ele(fix_matrix a, size_t i, size_t j) {
    return a[i][j];
}
```

```plaintext
# a in %rdi, i in %rsi, j in %rdx
salq $6, %rsi       # 64*i
addq %rsi, %rdi     # a + 64*i
movl (%rdi, %rdx, 4), %eax  # M[a + 64*i + 4*j]
```
Array Elements

- Address $A + i \times (C \times K) + j \times K$
- $C = n, K = 4$
- Must perform integer multiplication

```c
/* Get element a[i][j] */
int var_ele( size_t n, int a[n][n], size_t i, size_t j )
{
    return a[i][j];
}
```

```assembly
# n in %rdi, a in %rsi, i in %rdx, j in %rcx
imulq %rdx, %rdi # n*i
leaq (%rsi, %rdi, 4), %rax # a + 4*n*i
movl (%rax, %rcx, 4), %eax # a + 4*n*i + 4*j
ret
```
Structure represented as block of memory

- Big enough to hold all the fields

Fields ordered according to declaration

- Even if another ordering could yield a more compact representation

Compiler determines overall size and position of fields

- Machine-level program has no understanding of the structures in the source code.
Generating Pointer to Structure Member

```c
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Compute as `r + 4*idx`

BTW: why does `r->i` need 8 bytes? Alignment. (Next slide set)
Aside on Structures: Arrow vs. Dot

If you have a pointer \( r \) to a structure, use \( r->x \) to access component \( x \).

If you have the structure \( s \) itself, use \( s.x \).

\( r->x \) is just syntactic sugar for \( (*r).x \).
```c
void set_val 
  (struct rec *r, int val)
{
    while (r) {
      int i = r->i;
      r->a[i] = val;
      r = r->next;
    }
}
```

```
struct rec {
  int a[4];
  size_t i;
  struct rec *next;
};
```

```
.L11:
  testq %rdi, %rdi
  je .L12
  movq 16(%rdi), %rax
  movl %esi, (%rdi, %rax, 4)
  movq 24(%rdi), %rdi
  jmp .L11

.L12:
```