211: Computer Architecture
Summer 2016

Liu Liu

Topic:
- Assembly Programming
Recap

- Assembly Programming:
  - Control Flow: If-then-else / do-while / while / switch-case
Today’s Topic

- Assembly Programming:
  - Representing Array / Structure in Assembly
  - Structure Alignment
  - revisit stack (call - chain)
Basic Data Types

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

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

Floating Point
- Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th>Type</th>
<th>Intel</th>
<th>GAS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td></td>
<td>s</td>
<td>4</td>
<td>float</td>
</tr>
<tr>
<td>Double</td>
<td></td>
<td>l</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>Extended</td>
<td></td>
<td>t</td>
<td>10/12</td>
<td>long double</td>
</tr>
</tbody>
</table>

GAS: GNU Assembly
GNU: www.gnu.org/home.en.html
Array Allocation

Basic Principle

- Array of data type \( T \) and length \( L \)
- Contiguously allocated region of \( L \times \text{sizeof}(T) \) bytes

```plaintext
T A[L];
```

char string[12];

int val[5];

double a[4];

char *p[3];
**Array Example**

```c
typedef int zip_dig[5];

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

**Notes**

- **Declaration** “`zip_dig cmu`” equivalent to “`int cmu[5]`”
- **Example arrays** were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
Array Accessing Example

Computation

- Register %edx contains starting address of array
- Register %eax contains array index
- Desired digit at 4*%eax + %edx
- Use memory reference (%edx, %eax, 4)

Memory Reference Code

```c
int get_digit(zip_dig z, int dig) {
    return z[dig];
}
```

```assembly
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax # z[dig]
```
Referencing Examples

Code Does Not Do Any Bounds Checking!

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>mit[3]</td>
<td>36 + 4* 3 = 48</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>mit[5]</td>
<td>36 + 4* 5 = 56</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td>mit[-1]</td>
<td>36 + 4*-1 = 32</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>cmu[15]</td>
<td>16 + 4*15 = 76</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>

- Out of range behavior implementation-dependent
  - No guaranteed relative allocation of different arrays
Multi-Level Array Example

- Variable `univ` denotes array of 3 elements.
- Each element is a pointer.
  - 4 bytes
- Each pointer points to array of int’s

```c
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```
Element Access in Multi-Level Array

### Computation

- **Element access** \( \text{Mem[Mem[univ +4*index]]+4*dig]} \)
- **Must do two memory reads**
  - First get pointer to row array
  - Then access element within array

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

```assembly
# %ecx = index
# %eax = dig
leal 0(%ecx,4),%edx     # 4*index
movl univ(%edx),%edx    # Mem[univ+4*index]
movl (%edx,%eax,4),%eax # Mem[...+4*dig]
```
Structures

Concept

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

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

Accessing Structure Member

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

Assembly

```assembly
# %eax = val
# %edx = r
movl %eax, (%edx)  # Mem[r] = val
```
Generating Pointer to Struct. Member

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

Generating Pointer to Array Element

- Offset of each structure member determined at compile time

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

```assembly
# %ecx = idx
# %edx = r
leal 0(%ecx,4),%eax     # 4*idx
leal 4(%eax,%edx),%eax  # r+4*idx+4
```
# Structure Referencing (Cont.)

## C Code

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

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

```assembly
# %edx = r
movl (%edx),%ecx     # r->i
leal 0(%ecx,4),%eax  # 4*(r->i)
leal 4(%edx,%eax),%eax # r+4+4*(r->i)
movl %eax,16(%edx)    # Update r->p
```
Alignment

Aligned Data

- Primitive data type requires $K$ bytes
- Address must be multiple of $K$
- Required on some machines; advised on IA32
  - treated differently by Linux and Windows!

Motivation for Aligning Data

- Memory accessed by (aligned) double or quad-words
  - Inefficient to load or store datum that spans quad word boundaries

Compiler

- Inserts gaps in structure to ensure correct alignment of fields