Machine-Level Representation: Data

Announcement 2/19/2016

- HW#2 Problem 5 has been updated
  - Download the up-to-date version on Moodle
  - Check General Discussion Forum on Moodle for details.
- HW#1 has been graded
  - Returned during recitation sessions yesterday.
  - Verify your grades on Moodle.
  - Re-grading should be done within 7 calendar days (see course syllabus for policy details)
- Remaining un-returned HW#1 will be distributed after the class.

Outline

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Floating Point

Example Data Representations

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Typical 64-bit</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>-</td>
<td>-</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Example Data Representations

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Typical 64-bit</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>-</td>
<td>-</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Basic Data Types

- Integer
  - Stored & operated on in general (integer) registers
  - Signed vs. unsigned depends on instructions used (e.g. imull vs. mull)
    | Intel | ASM | Bytes | C |
    |-------|-----|-------|---|
    | byte  | b   | 1     | [unsigned] char |
    | word  | w   | 2     | [unsigned] short |
    | double | d  | 4     | [unsigned] int |
    | quad word | q | 8     | [unsigned] long (x86-64) |

- Floating Point
  - Stored & operated on in floating point registers
    | Intel | ASM | Bytes | C |
    |-------|-----|-------|---|
    | Single | s  | 4     | float |
    | Double | d  | 8     | double |
    | Extended | e | 10/12/16 | long double |

Array Allocation

- Basic Principle
  - Array of data type T and length L
  - Contiguously allocated region of L * sizeof(T) bytes in memory

Example Data Representations
### Array Access

- **Basic Principle**
  - `T A[L]`
  - Array `A` of data type `T` and length `L`.
  - Identifier `A` can be used as a pointer to array element 0: Type `T`.

<table>
<thead>
<tr>
<th><code>int val[5]</code></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

- **Reference**
  - Type
  - Value

<table>
<thead>
<tr>
<th><code>val[4]</code></th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>val+1</code></th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x + 4</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>val+2</code></th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x + 8</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>*(val+1)</code></th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>val[i]</code></th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x + 4*i</code></td>
<td></td>
</tr>
</tbody>
</table>

### Array Accessing Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];
typedef int ucb[ZLEN];

zip_dig cmu;
```

<table>
<thead>
<tr>
<th><code>int get_digit</code></th>
<th><code>cin &gt;&gt; digit;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int val[ZLEN]</code></td>
<td></td>
</tr>
<tr>
<td><code>int z[4]</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>for (i = 0; i &lt; ZLEN; i++)</code></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>z[i] = digit;</code></td>
<td></td>
</tr>
</tbody>
</table>

```
zip_dig mit;
```

<table>
<thead>
<tr>
<th><code>int get_digit</code></th>
<th><code>cin &gt;&gt; digit;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int val[ZLEN]</code></td>
<td></td>
</tr>
<tr>
<td><code>int z[4]</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>for (i = 0; i &lt; ZLEN; i++)</code></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>z[i] = digit;</code></td>
<td></td>
</tr>
</tbody>
</table>

### Array Example

```
#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 };
```

### Array Loop Example

```
#define ZLEN 5
void zincor(zip_dig z) {
    # i:4
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

### Multidimensional (Nested) Arrays

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

<table>
<thead>
<tr>
<th><code>int A[R][C]</code></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A[0][0]</code></td>
<td><code>A[0][R-1]</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>A[R-1][0]</code></td>
<td><code>A[R-1][C-1]</code></td>
</tr>
</tbody>
</table>

- **Array Size**
  - `R * C * K` bytes.

- **Arrangement**
  - Row-Major vs. Column-Major Ordering

- **Nested Array Example**
  - `zip_dig pgh[PCOUNT]`;
  - `PCOUNT` integer.

<table>
<thead>
<tr>
<th><code>int A[R][C]</code></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A[0][0]</code></td>
<td><code>A[0][R-1]</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>A[R-1][0]</code></td>
<td><code>A[R-1][C-1]</code></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th><code>int A[R][C]</code></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A[0][0]</code></td>
<td><code>A[0][R-1]</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>A[R-1][0]</code></td>
<td><code>A[R-1][C-1]</code></td>
</tr>
</tbody>
</table>

```
int pgh[PCOUNT] = {
    {1, 5, 2, 0, 6},
    {1, 5, 2, 1, 3},
    {1, 5, 2, 1, 7},
    {1, 5, 2, 1, 1}
};
```
Nested Array Row Access

- **Row Vectors**
  - \( A[i] \) is an array of \( C \) elements
  - Each element of type \( T \) requires \( K \) bytes
  - Starting address \( A + i \cdot (C \times K) \)

```c
int A[R][C];
```

```c
\ldots \ldots \ldots \ldots \ldots \ldots \ldots 
```

Nested Array Element Access

- **Array Elements**
  - \( A[i][j] \) is an element of type \( T \), which requires \( K \) bytes
  - Address \( A + i \cdot (C \times K) + j \cdot K = A + (i \cdot C + j) \cdot K \)

```c
int A[R][C];
```

```c
\ldots \ldots \ldots \ldots \ldots \ldots \ldots 
```

Multi-Level Array Example

- Variable `univ` denotes an array of 3 elements
  - Each element is a pointer to 8 bytes
  - Each pointer points to an array of `int`

```c
int *univ[UCOUNT] = (MIT, CMO, UCH);`n```

Element Access in Multi-Level Array

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

```c
salq %d, %r8
addq univ(%rdi,8), %r8 # p = univ[index] + 4 * digit
movl (%r8), %eax # *p
ret
```

Nested Array Row Access Code

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

```c
leaq (%rdi,%rdi,4),%rax # pgh + 20 * index
```

```c
leaq (%rdi,%rdi,4),%rsi # pgh + 5 * index
```

```c
movl (%r8),%rax # pgh + 4 * (5 * index + dig)
```

```c
movl (%rdi,8),%r9 # p = univ[index] + 4 * digit
```

```c
addq %d, %r9
```

```c
salq %r9, %r8
```

```c
movl (%r8),%eax # *p
```

```c
ret
```

Nested Array Element Access Code

```c
int get_pgh_zip
(int index) {
    return pgh[index];
}
```

```c
leaq (%rdi,%rdi,4),%rax # pgh + 20 * index
```

```c
leaq (%rdi,%rdi,4),%rsi # pgh + 5 * index
```

```c
movl (%r8),%rax # pgh + 4 * (5 * index + dig)
```

```c
movl (%rdi,8),%r9 # p = univ[index] + 4 * digit
```

```c
addq %d, %r9
```

```c
salq %r9, %r8
```

```c
movl (%r8),%eax # *p
```

```c
ret
```
Array Element Accesses

Nested array

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

Multi-level array

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

Access looks similar in C, but address computations very different:

```
Mem[pgh+20*index+4*digit] Mem[Mem[univ+8*index]+4*digit]
```

16 x 16 Matrix Access

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

Announcement 2/22/2016

- HW#2 due Wednesday 2/24/2016 BEFORE the class, i.e. it will be considered as late submission after the class begins.
- First mid-term exam will be held next Monday 2/29/2016 in class. Coverage will be Chapter 1 – Chapter 3.
- Reminder: Instructors are required to report plagiarism and academic dishonesty to the university (OSCAI). Please read the course syllabus carefully.

Outline

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Floating Point
**Review: 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 * C) * K bytes

- **Arrangement**
  
  - Row-Major vs. Column-Major Ordering

**Outline**

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Floating Point

**Generating Pointer to Structure Member**

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

- Generating Pointer to Array Element of `a`
  - Offset of each structure member determined at compile time
  - Compute as `r = 4 * idx`

**Review: Nested vs. Multi-Level Arrays**

- Nested array
  ```
  int get_pgh_digit
  ```
  ```
  (size_t index, size_t digit)
  ```
  ```
  {
    return pgh[index][digit];
  }

- Multi-level array
  ```
  int get_univ_digit
  ```
  ```
  (size_t index, size_t digit)
  ```
  ```
  {
    return univ[index][digit];
  }

**Structure Representation**

- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

**Following Linked List**

```c
struct rec {
  int a[4];
  struct rec *next;
};
```
Structures & Alignment

• Unaligned Data
  
<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+12</td>
</tr>
</tbody>
</table>

• Aligned Data
  
<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+12</td>
</tr>
</tbody>
</table>

Alignment Principles

• Aligned Data
  
  - Primitive data type requires K bytes
  - Address must be multiple of K

  • Motivation for Aligning Data
    
    - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory trickier when datum spans 2 pages

  • Compiler
    
    - Inserts gaps in structure to ensure correct alignment of fields

Specific Cases of Alignment (x86-64)

• 1 byte: char, ...
  
  - no restrictions on address

• 2 bytes: short, ...
  
  - lowest 1 bit of address must be 0:
  
  - 4 bytes: int, float, ...
    
    - lowest 2 bits of address must be 00:

• 8 bytes: double, long, char *, ...
  
  - lowest 3 bits of address must be 000:

• 16 bytes: long double (GCC on Linux)
  
  - lowest 4 bits of address must be 0000;

Satisfying Alignment with Structures

• Within structure:
  
  - Must satisfy each element's alignment requirement

• Overall structure placement
  
  - Each structure has alignment requirement K
    
    - K = Largest alignment of any element

• Example:
  
  - K = 8, due to double element

Arrays of Structures

• Overall structure length multiple of K

• Satisfy alignment requirement for every element
Accessing Array Elements

- Compute array offset: 12*idx
- `sizeof(S3)`, including alignment spacers
- Element j is at offset 8 within structure
- Assembler gives offset: a+8
- Resolved during linking

```c
short get_j(int idx)
{
    return a[idx].j;
}
```

-saving space

- Put large data types first
- Effect (K=4)

```c
struct S4 {
    char c;
    int i;
    char d;
} *p;
```

Outline

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Floating Point

Saving Space

- Put large data types first

```c
struct S5 {
    int i;
    char c;
    char d;
} *p;
```

-effect (K=4)

Background

- History
  - x87 FP
    - Legacy, very ugly
  - SSE FP
    - Supported by Shark machines
    - Special case use of vector instructions
  - AVX FP
    - Newest version
    - Similar to SSE
    - Documented in book

Programming with AVX – YMM Registers

- 16 total, each 32 bytes
- 32 single-byte integers
- 16 16-bit integers
- 8 32-bit integers
- 8 single-precision floats
- 4 double-precision floats

Scalar & SIMD Operations

- Scalar Operations: Single Precision  `addss %xmm0, %xmm1`
- SIMD Operations: Single Precision  `addps %xmm0, %xmm1`
- Scalar Operations: Double Precision  `addsd %xmm0, %xmm1`
FP Basics

• Arguments passed in %xmm0, %xmm1, ...
• Result returned in %xmm0
• All XMM registers are caller-saved

```cpp
float fadd(float x, float y) {
    return x + y;
}
```

```cpp
double dadd(double x, double y) {
    return x + y;
}
```

$\# x \text{ in } %\text{xmm0}, y \text{ in } %\text{xmm1}$

```assembly
addss %xmm1, %xmm0
```

```assembly
addsd %xmm1, %xmm0
```

FP Memory Referencing

• Integer (and pointer) arguments passed in regular registers
• FP values passed in XMM registers
• Different mov instructions to move between XMM registers, and between memory and XMM registers

```assembly
double dincr(double *p, double v) {
    double x = *p;
    *p = x + v;
    return x;
}
```

$\# p \text{ in } %\text{rdi}, v \text{ in } %\text{xmm0}$

```assembly
movapd %xmm0, %xmm1   # Copy v
movsd (%rdi), %xmm0  # x = *p
addsd %xmm0, %xmm1   # t = x + v
movsd %xmm1, (%rdi)  # *p = t
```

Other Aspects of FP Code

• Lots of instructions
  • Different operations, different formats, ...
• Floating-point comparisons
  • Instructions ucomiss and ucomisd
  • Set condition codes CF, ZF, and PF
• Using constant values
  • Set xmm0 register to 0 with instruction
    xorpd %xmm0, %xmm0
  • Others loaded from memory

Summary

• Arrays
  • Elements packed into contiguous region of memory
  • Use index arithmetic to locate individual elements
• Structures
  • Elements packed into single region of memory
  • Access using offsets determined by compiler
  • Possible require internal and external padding to ensure alignment
• Combinations
  • Can nest structure and array code arbitrarily
• Floating Point
  • Data held and operated on in XMM registers