Machine-Level Representation: Data

CSCI 2021: Machine Architecture and Organization

Pen-Chung Yew
Department Computer Science and Engineering
University of Minnesota

With Slides from Bryant, O’Hallaron
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>
Basic Data Types

**• Integer**

- Stored & operated on in **general (integer) registers**
- **Signed** vs. **unsigned** depends on instructions used (e.g. `imull` vs. `mull`)

<table>
<thead>
<tr>
<th>Intel</th>
<th>ASM</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 (x86-64)</td>
</tr>
</tbody>
</table>

**• Floating Point**

- Stored & operated on in **floating point registers**

<table>
<thead>
<tr>
<th>Intel</th>
<th>ASM</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>10/12/16</td>
<td>long double</td>
</tr>
</tbody>
</table>
Array Allocation

- Basic Principle
  
  ```
  T A[L];
  ```

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

Index start with 0
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 \( T^* \)

\[
\text{int val[5];}
\]

<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+i</td>
<td>int *</td>
<td>x + 4i</td>
</tr>
</tbody>
</table>

0  1  2  3  4
\[
\begin{array}{c|c|c|c|c|c}
0 & 1 & 5 & 2 & 1 & 3 \\
\hline
x & x + 4 & x + 8 & x + 12 & x + 16 & x + 20 \\
\end{array}
\]
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 };  
```

- 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

```
int get_digit(zip_digit z, int digit)
{
    return z[digit];
}
```

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

```
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %rax  # z[digit]
```
# Array Loop Example

```c
#define ZLEN 5
void zincr(zip_dig z) {
    size_t i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

```
# %rdi = z
movl $0, %rax            # i = 0
jmp .L3                   # goto middle
.L4:                      # loop:
    addl $1, (%rdi,%rax,4) # z[i]++
    addq $1, %rax          # i++
.L3:                      # middle
    cmpq $4, %rax          # i:4
    jle .L4                # if <=, goto loop
rep; ret                  # rep is a nop
```
### 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 vs. Column-Major** Ordering

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

zip_dig pgh[4];

- “zip_dig pgh[4]” equivalent to “int pgh[4][5]”
- Variable pgh: array of 4 elements, allocated contiguously
- Each element is an array of 5 int’s, allocated contiguously, i.e. array of arrays
- “Row-Major” ordering of all elements in memory
Nested Array Row Access

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

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

- Starting address: \(A + i*(C*4)\) for \(A[i]\)
- Starting address: \(A + (R-1)*(C*4)\) for \(A[R-1]\)
Nested Array Row Access Code

- **Row Vector**
  - `pgh[index]` is array of 5 int's
  - Starting address `pgh+20*index`

- **Machine Code**
  - Computes and returns address
  - Compute as `pgh + 4*(index+4*index)`

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

```assembly
# %rdi = index
leaq (%rdi,%rdi,4),%rax # 5 * index
leaq pgh(,%rax,4),%rax # pgh + (20 * index)
```
Nested Array Element Access

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

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

\[ A[0][0] \quad \cdots \quad A[0][C-1] \]
\[ A[1][0] \quad \cdots \quad A[1][C-1] \]
\[ \vdots \]
\[ A[R-1][0] \quad \cdots \quad A[R-1][C-1] \]

\[ A \]
\[ A+(i*C*4) \]
\[ A+(i*C*4)+(j*4) \]
\[ A+((R-1)*C*4) \]
Nested Array Element Access Code

Array Elements

- \texttt{pgh[index][dig]} is int
- Address: \texttt{pgh + 20*index + 4*dig} = \texttt{pgh + 4*(5*index + dig)}
Multi-Level Array Example

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

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

```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 };  
```
Element Access in Multi-Level Array

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

```
salq $2, %rsi  # 4*digit
addq univ(,%rdi,8), %rsi # p = univ[index] + 4*digit
movl (%rsi), %eax  # return *p
ret
```

- Computation
  - Element access: \[ \text{Mem[Mem[univ+8*index]+4*digit]} \]
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array
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];
}
```

Accesses looks similar in C, but address computations very different:
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];
}
```
16 x 16 Matrix Access

- **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];
}
```

```assembly
# 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]
ret
```
n x n Matrix Access

- Array Elements
  - Address: A + i * (C * K) + j * 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
```
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
  
  - $$(R \times C) \times K$$ bytes

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

```c
int A[R][C]; /* Linearized */
```
Review: Nested vs. Multi-Level Arrays

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];
}
```

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

\[
\text{Mem}[\text{pgh}+20\times \text{index}+4\times \text{digit}] \quad \text{Mem}[\text{Mem}[\text{univ}+8\times \text{index}]+4\times \text{digit}]
\]
Outline

• Arrays
  • One-dimensional
  • Multi-dimensional (nested)
  • Multi-level

• Structures
  • Allocation
  • Access
  • Alignment

• Floating Point
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
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`

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

```assembly
# r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```
Following Linked List

C Code

void set_val
(struct rec *r, int val)
{
    while (r) {
        int i = r->i;  /* (*r).i */
        r->a[i] = val;  /* (*r).a[i]*/
        r = r->next;  /* (*r).next */
    }
}

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

.L11:
    # loop:
    movslq 16(%rdi), %rax  #  i = M[r+16]
    movl %rsi, (%rdi,%rax,4)  #  M[r+4*i] = val
    movq 24(%rdi), %rdi  #  r = M[r+24]
    testq %rdi, %rdi  #  Test r
    jne .L11  #  if !=0 goto loop

Element i

Register | Value
---|---
%rdi | r
%rsi | val

<table>
<thead>
<tr>
<th>a</th>
<th>i</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

Following a Linked List C Code

Register Value
%rdi r
%rsi val

Element i

void set_val
(struct rec *r, int val)
{
    while (r) {
        int i = r->i;  /* (*r).i */
        r->a[i] = val;  /* (*r).a[i]*/
        r = r->next;  /* (*r).next */
    }
}
### Structures & Alignment

- **Unaligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$

- **Aligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$
Alignment Principles

• **Aligned Data**
  • Primitive data type requires $K$ bytes
  • Address must be multiple of $K$
  • Required on some machines; advised on x86-64

• **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 $0_2$

• 4 bytes: int, float, ...
  • lowest 2 bits of address must be $00_2$

• 8 bytes: double, long, char *, ...
  • lowest 3 bits of address must be $000_2$

• 16 bytes: long double (GCC on Linux)
  • lowest 4 bits of address must be $0000_2$
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
  - Initial address & structure length must be multiples of $K$

- **Example:**
  - $K = 8$, due to double element

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Meeting Overall Alignment Requirement

- For largest alignment requirement $K$
- Overall structure must be multiple of $K$

```c
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```
Arrays of Structures

- Overall structure length multiple of $K$
- Satisfy alignment requirement for every element

```c
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```
Accessing Array Elements

- Compute array offset $12 \times \text{idx}$
- $\text{sizeof}(S3)$, including alignment spacers
- Element $j$ is at offset 8 within structure
- Assembler gives offset $a+8$
- Resolved during linking

```c
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```

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

```assembly
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(%rax,4),%eax
```
Saving Space

- Put large data types first

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

- Effect \((K=4)\)

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

<table>
<thead>
<tr>
<th>c</th>
<th>3 bytes</th>
<th>i</th>
<th>d</th>
<th>3 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td>c</td>
<td>d</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>
Outline

• Arrays
  • One-dimensional
  • Multi-dimensional (nested)
  • Multi-level

• Structures
  • Allocation
  • Access
  • Alignment

• Floating Point
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

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

# x in %xmm0, y in %xmm1
addss %xmm1, %xmm0
ret

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

# x in %xmm0, y in %xmm1
addsd %xmm1, %xmm0
ret
```
**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

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

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