CSCI-UA.0201

Computer Systems Organization

Machine-Level Programming V

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
Manipulating Data

How are data structures, like arrays, presented and manipulated in assembly?
Array Allocation

• Basic Principle

\( T \text{ A}[L]; \)

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

• Basic Principle

\[ T \ A[L] ; \]
– Array of data type \( T \) and length \( L \)
– Identifier \( A \) used as a pointer to array element 0: Type \( T^* \)

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

• Reference Type Value

<table>
<thead>
<tr>
<th>Identifier</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>
#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 nyu = { 9, 4, 7, 2, 0 };

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

zip_dig nyu;

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

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

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

```asm
# %rdi = z
# ZLEN is 5
movl $0, %eax                # i = 0
jmp .L3                       # goto middle
.L4:
    addl $1, (%rdi,%rax,4)     # z[i]++
    addl $1, %eax              # i++
.L3:
    cmpl $4, %eax              # i:4
    jbe .L4                    # if <=, goto loop
ret
```
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 in memory
  - Row-Major Ordering

\[
\begin{array}{c}
A[0][0] \quad \cdots \quad A[0][C-1] \\
\quad \quad \ddots \quad \quad \ddots \\
A[R-1][0] \quad \cdots \quad A[R-1][C-1]
\end{array}
\]

\[
\begin{array}{cccc}
A[0][0] & \cdots & A[0][C-1] \\
\vdots & \ddots & \vdots \\
A[R-1][0] & \cdots & A[R-1][C-1]
\end{array}
\]

\[
\text{int } A[R][C];
\]

<table>
<thead>
<tr>
<th>A[0][0]</th>
<th>\cdots</th>
<th>A[0][C-1]</th>
<th>A[1][0]</th>
<th>\cdots</th>
<th>A[1][C-1]</th>
<th>\cdots</th>
<th>A[R-1][0]</th>
<th>\cdots</th>
<th>A[R-1][C-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>| |</td>
<td>| |</td>
<td>| |</td>
<td>| |</td>
<td>| |</td>
<td>| |</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
4 \times R \times C \text{ Bytes}
\]
Nested Array Example

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

- Variable `pgh`: array of 4 elements, allocated contiguously
- Each element is an array of 5 `int`'s, allocated contiguously

- “Row-Major” ordering of all elements in memory
Nested Array Element Access

• Array Elements
  – address of $A[i][j]$:
    Address $A + i \times (C \times K) + j \times K = A + (i \times C + j) \times K$

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

![Diagram of nested array element access](image)
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
int cmu[5] = { 1, 5, 2, 1, 3 };
int mit[5] = { 0, 2, 1, 3, 9 };
int nyu[5] = { 9, 4, 7, 2, 0 };

#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, nyu};
```
Element Access in Multi-Level Array

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

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

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

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

\[
\text{Mem}[\text{pgh} + 20 \times \text{index} + 4 \times \text{digit}] \quad \text{Mem}[\text{Mem}[\text{univ} + 8 \times \text{index}] + 4 \times \text{digit}]
\]
How about structures?
• 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
  - Offset of each structure member determined at compile time
  - Compute as `r + 4*idx`

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

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

• C Code

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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>r</td>
</tr>
<tr>
<td>%rsi</td>
<td>val</td>
</tr>
</tbody>
</table>

```
.L11:
   movslq 16(%rdi), %rax # i = M[r+16]  
   movl  %esi, (%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
Alignment
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 (i.e. 8 bytes boundaries)

• Compiler
  – Inserts gaps in structure to ensure correct alignment of fields
Structures & Alignment

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

  ```
  struct S1 {
    char c;
    int i[2];
    double v;
  } *p;
  p is multiple of 8
  ```

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

  ```
  struct S1 {
    char c;
    int i[2];
    double v;
  } *p;
  ```
Specific Cases of Alignment (x86-64)

- 1 byte: char, ...
  - no restrictions on address
- 2 bytes: short, ...
  - address must be multiple of 2
- 4 bytes: int, float, ...
  - address must be multiple of 4
- 8 bytes: double, long, char *, ...
  - address must be multiple of 8
- 16 bytes: long double (GCC on Linux)
  - address must be multiple of 16
How about structures?

- **Within structure:**
  - Must satisfy each element’s alignment requirement

- **Overall structure placement**
  - Each structure has alignment requirement $K$
    - $K = \text{Largest alignment of any element}$
  - Initial address & structure length must be multiples of $K$

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

```c
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;
```
Saving Space

• Put large data types first

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

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

• Effect (K=4)

```
c   3 bytes  i  d  3 bytes
i   2 bytes  c  d
```
Final Look at Memory Layout
x86-64 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)

- **Heap**
  - Dynamically allocated as needed
  - When call `malloc()`, `calloc()`, `new()`

- **Data**
  - Statically allocated data
  - E.g., global vars, static vars, string constants

- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only

Hex Address: 0000000000000000

8MB
Conclusions

• We have not covered everything in x86-64, just gave you a glimpse and a feel for it.

• Compiler does more than blind translating your HLL code:
  – It manages the stack.
  – It translates the sophisticated data structure access to assembly
  – It optimizes your code

• No matter how sophisticated your HLL language code, it will be translated to assembly with 16 registers and basic data types!