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

\[ T \ A[L]; \]

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

char string[12];

\[
\begin{align*}
\text{x} & \quad \text{x+12} \\
\end{align*}
\]

int val[5];

\[
\begin{align*}
\text{x} & \quad \text{x+4} & \quad \text{x+8} & \quad \text{x+12} & \quad \text{x+16} & \quad \text{x+20} \\
\end{align*}
\]

double a[3];

\[
\begin{align*}
\text{x} & \quad \text{x+8} & \quad \text{x+16} & \quad \text{x+24} \\
\end{align*}
\]

cchar *p[3];

\[
\begin{align*}
\text{x} & \quad \text{x+8} & \quad \text{x+16} & \quad \text{x+24} \\
\end{align*}
\]
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 \( T^* \)

```
int val[5];
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x + 4</td>
<td>x + 8</td>
<td>x + 12</td>
<td>x + 16</td>
</tr>
</tbody>
</table>

• Reference

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

```c
#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

```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

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

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

```assembly
# %rdi = z
# ZLEN is 5
movl    $0, %eax          #   i = 0
jmp     .L3               #   goto middle
.L4:                        # loop:
    addl    $1, (%rdi,%rax,4) #   z[i]++
    addl    $1, %eax          #   i++
.L3:                        # middle
    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**

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

![Array Diagram]

- **Diagram**: Arrangement of elements in memory for a 2D array $A[R][C]$. The elements are arranged in row-major order, indicating that elements in each row are stored contiguously in memory before moving to the next row. The diagram shows the allocation of 4 * $R$ * $C$ bytes in memory.
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 showing array access with indices and offsets](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];
}
```

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

- Computation
  - Element access `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:

```c
Mem[pgh+20*index+4*digit] Mem[Mem[univ+8*index]+4*digit]
```
How about structures?
• Structure represented as a 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 Array Element
  – Offset of each structure member determined at compile time
  – Compute as $r + 4 \times idx$

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

```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;
    }
}
```

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

### Register Value

<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>

### Element i

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>0</td>
</tr>
<tr>
<td>r</td>
<td>16</td>
</tr>
<tr>
<td>next</td>
<td>24</td>
</tr>
<tr>
<td>a</td>
<td>32</td>
</tr>
</tbody>
</table>

### Instructions

```
.L11:     # loop:
    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
```
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
• **Unaligned Data**

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

- 3 bytes
- 4 bytes

```
c i[0] i[1] v
p p+1 p+5 p+9 p+17
```

- `p` is multiple of 8

• **Aligned Data**

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

```
c i[0] i[1] v
p+0 p+4 p+8 p+16 p+24
```

- Multiple of 4
- Multiple of 8

- Multiple of 8
- Multiple of 8

- Multiple of 8
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

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
• For largest alignment requirement $K$
• Overall structure must be multiple of $K$

```
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)

<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>c</td>
<td>d</td>
<td>2 bytes</td>
<td></td>
</tr>
</tbody>
</table>
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: 40000000

00007FFFFFFF

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!