Machine-Level Programming IV: x86-64 Procedures, Data
Today

- Procedures (x86-64)
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
# x86-64 Integer Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%r10</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r11</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%r12</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
x86-64 Registers

- **Arguments passed to functions via registers**
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well

- **All references to stack frame via stack pointer**
  - Eliminates need to update `%ebp`/ `%rbp`

- **Other Registers**
  - 6 callee saved
  - 2 caller saved
  - 1 return value (also usable as caller saved)
  - 1 special (stack pointer)
x86-64 Locals in the Red Zone

/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}

Avoiding Stack Pointer Change
- Can hold all information within small window beyond stack pointer

swap_a:
    movq (%rdi), %rax
    movq %rax, -24(%rsp)
    movq (%rsi), %rax
    movq %rax, -16(%rsp)
    movq -16(%rsp), %rax
    movq %rax, (%rdi)
    movq -24(%rsp), %rax
    movq %rax, (%rsi)
    ret

rtn Ptr

%rsp

-8 unused
-16 loc[1]
-24 loc[0]
Interesting Features of Stack Frame

- **Allocate entire frame at once**
  - All stack accesses can be relative to `%rsp`
  - Do by decrementing stack pointer
  - Can delay allocation, since safe to temporarily use red zone

- **Simple deallocation**
  - Increment stack pointer
  - No base/frame pointer needed
Today

- Procedures (x86-64)
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
Basic Data Types

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

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

```
char string[12];

int val[5];

double a[3];

char *p[3];
```
Array Access

Basic Principle

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

```
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>
Array Example

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

zip_dig ut = { 7, 8, 7, 1, 2 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };```

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

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

IA32

- Register %edx contains starting address of array
- Register %eax contains array index
- Desired digit at 4*%eax + %edx
- Use memory reference (%edx, %eax, 4)
Array Loop Example (IA32)

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

```asm
# edx = z
movl $0, %eax       # %eax = i
.L4:  # loop:
    addl $1, (%edx,%eax,4) # z[i]++
    addl $1, %eax         # i++
    cmpl $5, %eax         # i:5
    jne .L4              # if !=, goto loop
```
### Pointer Loop Example (IA32)

**void zincr_p(zip_dig z) {**
  ```c
  int *zend = z+ZLEN;
  do {
    (*z)++;
    z++;
  } while (z != zend);
  ```
**}**

**void zincr_v(zip_dig z) {**
  ```c
  void *vz = z;
  int i = 0;
  do {
    (*((int *) (vz+i)))++;
    i += ISIZE;
  } while (i != ISIZE*ZLEN);
  ```
**}**

```
# edx = z = vz
movl $0, %eax           # i = 0
.L8:
    addl $1, (%edx,%eax)  # Increment vz+i
    addl $4, %eax         # i += 4
    cmpl $20, %eax        # Compare i:20
    jne .L8               # if !=, goto loop
```
Nested Array Example

```
#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]” 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

- **“Row-Major” ordering of all elements guaranteed**
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**
  
  \[ R \times C \times K \text{ bytes} \]

- **Arrangement**
  
  - Row-Major Ordering

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

![Array Diagram]
Nested Array Row Access

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

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

```
| A[0] | | A[0][C-1] |
| A[0] | \( \cdots \) | A[i] |
| A[i] | \( \cdots \) | A[i][C-1] |
| A[R-1] | \( \cdots \) | A[R-1][C-1] |
```

\( A + i \times C \times 4 \)

\( A + (R-1) \times C \times 4 \)
Nested Array Row Access Code

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

```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}};
```

```asm
# %eax = index
leal (%eax,%eax,4),%eax  # 5 * index
leal pgh(%eax,4),%eax     # pgh + (20 * index)
```

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

- **IA32 Code**
  - Computes and returns address
  - Compute as `pgh + 4*(index+4*index)`
Nested Array Row Access

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

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

```
A[i][j]
A[i]
A[0]
A[R-1]
```

\(A + i \times C \times 4\)
Nested Array Element Access Code

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

```
movl 8(%ebp), %eax       # index
leal (%eax,%eax,4), %eax # 5*index
addl 12(%ebp), %eax     # 5*index+dig
movl pgh(%eax,4), %eax  # offset 4*(5*index+dig)
```

- **Array Elements**
  - `pgh[index][dig]` is int
  - Address: `pgh + 20*index + 4*dig`
    - `= pgh + 4*(5*index + dig)`

- **IA32 Code**
  - Computes address `pgh + 4*((index+4*index)+dig)`
Multi-Level Array Example

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

```c
zip_dig ut = { 7, 8, 7, 1, 2 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

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

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

- **Computation (IA32)**
  - Element access `Mem[Mem[univ+4*index]+4*dig]`
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

```asm
movl 8(%ebp), %eax    # index
movl univ(,%eax,4), %edx  # p = univ[index]
movl 12(%ebp), %eax    # dig
movl (%edx,%eax,4), %eax  # p[dig]
```
Array Element Accesses

Nested array

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

Multi-level array

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

Accesses looks similar in C, but addresses very different:

\[
\text{Mem}[\text{pgh}+20*\text{index}+4*\text{dig}] \quad \text{and} \quad \text{Mem}[(\text{Mem}[\text{univ}+4*\text{index}]+4*\text{dig})]
\]
**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, int i, int j)
{
    return a[i][j];
}

#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele
    (int n, int *a, int i, int j)
{
    return a[IDX(n,i,j)];
}

/* Get element a[i][j] */
int var_ele
    (int n, int a[n][n], int i, int 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, int i, int j) {
    return a[i][j];
}
```

```
movl 12(%ebp), %edx          # i
sall $6, %edx                 # i*64
movl 16(%ebp), %eax          # j
sall $2, %eax                 # j*4
addl 8(%ebp), %eax           # a + j*4
movl (%eax,%edx), %eax       # *(a + j*4 + i*64)
```
n X n Matrix Access

- **Array Elements**
  - Address $A + i \times (C \times K) + j \times K$
  - $C = n$, $K = 4$

```c
/* Get element a[i][j] */
int var_ele(int n, int a[n][n], int i, int j) {
    return a[i][j];
}
```

```assembly
movl 8(%ebp), %eax      # n
sall $2, %eax           # n*4
movl %eax, %edx         # n*4
imull 16(%ebp), %edx    # i*n*4
movl 20(%ebp), %eax    # j
sall $2, %eax           # j*4
addl 12(%ebp), %eax    # a + j*4
movl (%eax,%edx), %eax # *(a + j*4 + i*n*4)
```
Optimizing Fixed Array Access

- **Computation**
  - Step through all elements in column $j$

- **Optimization**
  - Retrieving successive elements from single column

```c
#define N 16
typedef int fix_matrix[N][N];

/* Retrieve column $j$ from array */
void fix_column
    (fix_matrix a, int j, int *dest)
{
    int i;
    for (i = 0; i < N; i++)
        dest[i] = a[i][j];
}
```
Optimizing Fixed Array Access

**Optimization**
- Compute $ajp = &a[i][j]$
  - Initially = $a + 4*j$
  - Increment by $4*N$

```c
/* Retrieve column j from array */
void fix_column(
    fix_matrix a, int j, int *dest)
{
    int i;
    for (i = 0; i < N; i++)
        dest[i] = a[i][j];
}
```

### Register Value Table

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>ajp</td>
</tr>
<tr>
<td>%ebx</td>
<td>dest</td>
</tr>
<tr>
<td>%edx</td>
<td>i</td>
</tr>
</tbody>
</table>

```asm
.L8:
    movl (%ecx), %eax      # Read *ajp
    movl %eax, (%ebx,%edx,4) # Save in dest[i]
    addl $1, %edx          # i++
    addl $64, %ecx         # ajp += 4*N
    cmpl $16, %edx         # i:N
    jne .L8               # if !==, goto loop
```

```asm
.L8:
    movl (%ecx), %eax      # Read *ajp
    movl %eax, (%ebx,%edx,4) # Save in dest[i]
    addl $1, %edx          # i++
    addl $64, %ecx         # ajp += 4*N
    cmpl $16, %edx         # i:N
    jne .L8               # if !==, goto loop
```
Optimizing Variable Array Access

- Compute $ajp = \&a[i][j]$
  - Initially = $a + 4\times j$
  - Increment by $4\times n$

```c
/* Retrieve column j from array */
void var_column
(int n, int a[n][n],
 int j, int *dest)
{
    int i;
    for (i = 0; i < n; i++)
        dest[i] = a[i][j];
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>ajp</td>
</tr>
<tr>
<td>%edi</td>
<td>dest</td>
</tr>
<tr>
<td>%edx</td>
<td>i</td>
</tr>
<tr>
<td>%ebx</td>
<td>4*n</td>
</tr>
<tr>
<td>%esi</td>
<td>n</td>
</tr>
</tbody>
</table>

.L18:
  movl  (%ecx), %eax       # Read *ajp
  movl  %eax, (%edi,%edx,4) # Save in dest[i]
  addl  $1, %edx           # i++
  addl  $ebx, %ecx         # ajp += 4*n
  cmpl  $edx, %esi         # n:i
  jg    .L18               # if >, goto loop
Today

- Procedures (x86-64)
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
Structure Allocation

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

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

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

- Accessing Structure Member
  - Pointer indicates first byte of structure
  - Access elements with offsets

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

**IA32 Assembly**

```assembly
# %edx = val
# %eax = r
movl %edx, 12(%eax) # Mem[r+12] = val
```
Generating Pointer to Structure Member

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

### Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Arguments
  - Mem[%ebp+8]: `r`
  - Mem[%ebp+12]: `idx`

```assembly
movl 12(%ebp), %eax  # Get idx
sall $2, %eax        # idx*4
addl 8(%ebp), %eax  # r+idx*4
```
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->n;
    }
}
```

### C Code

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

#### Element i

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>r</td>
</tr>
<tr>
<td>%ecx</td>
<td>val</td>
</tr>
</tbody>
</table>

---

.L17:  
  movl 12(%edx), %eax  # r->i
  movl %ecx, (%edx,%eax,4)  # r->a[i] = val
  movl 16(%edx), %edx  # r = r->n
  testl %edx, %edx  # Test r
  jne .L17  # If != 0 goto loop

---
Summary

- Procedures in x86-64
  - Stack frame is relative to stack pointer
  - Parameters passed in registers

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

- Structures
  - Allocation
  - Access