**X86-64 Instruction Set Architecture**

**Instructor:** Sanjeev Setia

---

### Data Representations: IA32 + x86-64

#### Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<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>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>8</td>
<td>10/12</td>
<td>16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

*Or any other pointer*
x86-64 Integer Registers

- Extend existing registers. Add 8 new ones.
- Make %ebp/%rbp general purpose

Instructions

- Long word 1 (4 Bytes) ↔ Quad word q (8 Bytes)

- New instructions:
  - movl → movq
  - addl → addq
  - sall → salq
  - etc.

- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: addl
Swap in 32-bit Mode

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Swap:
```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

- **Setup**
  - pushl %ebp
  - movl %esp,%ebp
  - pushl %ebx

- **Body**
  - movl 12(%ebp),%ecx
  - movl 8(%ebp),%edx
  - movl (%ecx),%eax
  - movl (%edx),%ebx
  - movl %eax,(%edx)
  - movl %ebx,(%ecx)

- **Finish**
  - movl -4(%ebp),%ebx
  - movl %ebp,%esp
  - popl %ebp
  - ret

Swap in 64-bit Mode

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Swapping:
```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

- **Operands passed in registers (why useful?)**
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers

- **No stack operations required**

- **32-bit data**
  - Data held in registers %eax and %edx
  - movl operation

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```
Swap Long Ints in 64-bit Mode

```c
void swap_l
    (long int *xp, long int *yp)
{
    long int t0 = *xp;
    long int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

- **64-bit data**
  - Data held in registers `%rax` and `%rdx`
  - `movq` operation
  - "q" stands for quad-word

Reading Condition Codes: x86-64

- **SetX Instructions:**
  - Set single byte based on combination of condition codes
  - Does not alter remaining 3 bytes

```c
int gt (long x, long y)
{
    return x > y;
}
```

Body (same for both)

```c
xorl %eax, %eax
cmpq %rsi, %rdi
setg %al
```


Reading Condition Codes: x86-64

- **SetX Instructions:**
  - Set single byte based on combination of condition codes
  - Does not alter remaining 3 bytes

```c
int gt (long x, long y)
{
    return x > y;
}
long lgt (long x, long y)
{
    return x > y;
}
```

**Body (same for both)**

```
xorl %eax, %eax # eax = 0
cmpq %rsi, %rdi  # Compare x and y
setg %al # al = x > y
```

Is `%rax` zero?
Yes: 32-bit instructions set high order 32 bits to 0!

Conditionals: x86-64

```
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff: # x in %edi, y in %esi
        movl %edi, %eax
        movl %esi, %edx
        subl %esi, %eax
        subl %edi, %edx
        cmpl %esi, %edi
        cmovle %edx, %eax
        ret
```
Conditionals: x86-64

Conditionals:
- x86-64

Conditional move instruction
- `cmovC` src, dest
- Move value from src to dest if condition C holds
- More efficient than conditional branching (simple control flow)
- But overhead: both branches are evaluated

```c
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
        result = x - y;
    } else {
        result = y - x;
    }
    return result;
}
```

```assembly
absdiff: # x in %edi, y in %esi
movl %edi, %eax # eax = x
movl %esi, %edx # edx = y
subl %esi, %eax # eax = x - y
subl %edi, %edx # edx = y - x
cmpl %esi, %edi # x:y
cmovle %edx, %eax # eax=edx if <=
ret
```

General Form with Conditional Move

C Code
```
val = Test ? Then-Expr : Else-Expr;
```

Conditional Move Version
```
val1 = Then-Expr;
val2 = Else-Expr;
val1 = val2 if !Test;
```

- Both values get computed
- Overwrite then-value with else-value if condition doesn’t hold
- Don’t use when:
  - Then or else expression have side effects
  - Then and else expression are too expensive
New Style “While” Loop Translation

C Code
```c
int fact_while(int x) {
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

Goto Version
```c
int fact_while_goto3(int x) {
    int result = 1;
    goto middle;
    loop:
        result *= x;
        x = x-1;
    middle:
        if (x > 1)
            goto loop;
    return result;
}
```

- **Recent technique for GCC**
  - Both IA32 & x86-64
- **First iteration jumps over body computation within loop**

Jump-to-Middle While Translation

C Code
```
while (Test) {
    Body
}
```

Goto Version
```
goto middle;
loop:
    Body
middle:
    if (Test)
        goto loop;
```

- **Avoids duplicating test code**
- **Unconditional goto incurs no performance penalty**
- **for loops compiled in similar fashion**

Goto (Previous) Version
```
if (!Test)
    goto done;
loop:
    Body
middle:
    if (Test)
        goto loop;
done:
```
Jump-to-Middle Example

```c
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x--;  
    }
    return result;
}
```

```assembly
# x in %edx, result in %eax
jmp .L34  # goto Middle
.L35:     # Loop:
imull %edx, %eax  # result *= x
decl %edx       # x--
.L34:      # Middle:
cmpl $1, %edx  # x:1
jg .L35     # if >, goto Loop
```

Implementing Loops

- **IA32**
  - All loops translated into form based on “do-while”

- **x86-64**
  - Also make use of “jump to middle”

**Why the difference**

- IA32 compiler developed for machine where all operations costly
- x86-64 compiler developed for machine where unconditional branches incur (almost) no overhead
x86-64 Integer Registers

- Twice the number of registers
- Accessible as 8, 16, 32, 64 bits
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 or 3 have special uses

x86-64 Long Swap

```c
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

- Operands passed in registers
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- No stack operations required (except ret)
- Avoiding stack
  - Can hold all local information in registers
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

void 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]

x86-64 NonLeaf without Stack Frame

long scount = 0;

/* Swap a[i] & a[i+1] */
void swap_ele_se
  (long a[], int i)
{
  swap(&a[i], &a[i+1]);
  scount++;
}

swap_ele_se:
  movslq %esi,%rsi  # Sign extend i
  leaq (%rdi,%rsi,8), %rdi # &a[i]
  leaq 8(%rdi), %rsi  # &a[i+1]
  call swap          # swap()
  incq scount(%rip)  # scount++;
ret

- No values held while swap being invoked
- No callee save registers needed
x86-64 Call using Jump

long scount = 0;
/* Swap a[i] & a[i+1] */
void swap_ele(long a[], int i) {
    swap(&a[i], &a[i+1]);
}

swap_ele:
movslq %esi,%rsi           # Sign extend i
leaq (%rdi,%rsi,8), %rdi
leaq 8(%rdi), %rsi
jmp swap

- When swap executes ret, it will return from swap_ele
- Possible since swap is a "tail call" (no instructions afterwards)
**x86-64 Stack Frame Example**

```c
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
    (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += a[i];
}
```

- Keeps values of `a` and `i` in callee save registers
- Must set up stack frame to save these registers

```assembly
swap_ele_su:
    movq %rbx, -16(%rsp)       # Save %rbx
    movslq %esi,%rbx           # Extend & save i
    movq %r12, -8(%rsp)        # Save %r12
    movq %rdi, %r12            # Save a
    leaq (%rdi,%rbx,8), %rdi   # &a[i]
    subq $16, %rsp             # Allocate stack frame
    leaq 8(%rdi), %rsi         # &a[i+1]
    call swap                  # swap()
    movq (%r12,%rbx,8), %rax  # a[i]
    addq %rax, sum(%rip)       # sum += a[i]
    movq (%rsp), %rbx          # Restore %rbx
    movq 8(%rsp), %r12         # Restore %r12
    addq $16, %rsp             # Deallocate stack frame
    ret
```

---

**Understanding x86-64 Stack Frame**

```assembly
swap_ele_su:
    movq %rbx, -16(%rsp)       # Save %rbx
    movslq %esi,%rbx           # Extend & save i
    movq %r12, -8(%rsp)        # Save %r12
    movq %rdi, %r12            # Save a
    leaq (%rdi,%rbx,8), %rdi   # &a[i]
    subq $16, %rsp             # Allocate stack frame
    leaq 8(%rdi), %rsi         # &a[i+1]
    call swap                  # swap()
    movq (%r12,%rbx,8), %rax  # a[i]
    addq %rax, sum(%rip)       # sum += a[i]
    movq (%rsp), %rbx          # Restore %rbx
    movq 8(%rsp), %r12         # Restore %r12
    addq $16, %rsp             # Deallocate stack frame
    ret
```
Understanding x86-64 Stack Frame

swap_ele_su:

```
movq (%rsp), %rbx  # Save %rbx
movq %r12, -8(%rsp) # Save %r12
subq $16, %rsp  # Allocate stack frame
movq (%rsp), %rbx # Restore %rbx
movq 8(%rsp), %r12 # Restore %r12
addq $16, %rsp  # Deallocate stack frame
```

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
x86-64 Procedure Summary

- **Heavy use of registers**
  - Parameter passing
  - More temporaries since more registers

- **Minimal use of stack**
  - Sometimes none
  - Allocate/deallocate entire block

- **Many tricky optimizations**
  - What kind of stack frame to use
  - Calling with jump
  - Various allocation techniques

Specific Cases of Alignment (x86-64)

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

- **2 bytes:** short, ...
  - lowest 1 bit of address must be 0₂

- **4 bytes:** int, float, ...
  - lowest 2 bits of address must be 00₂

- **8 bytes:** double, char *, ...
  - Windows & Linux:
    - lowest 3 bits of address must be 000₂

- **16 bytes:** long double
  - Linux:
    - lowest 3 bits of address must be 000₂
    - i.e., treated the same as a 8-byte primitive data type