x86-64 Programming I
CSE 351 Summer 2020

Instructor:
Porter Jones

Teaching Assistants:
Amy Xu
Callum Walker
Sam Wolfson
Tim Mandzyuk

http://www.smbc-comics.com/?id=2999
Administrivia

❖ Questions doc: https://tinyurl.com/CSE351-7-8

❖ hw6 & hw7 due Friday (7/10) – 10:30am
❖ hw8 due Monday (7/13) – 10:30am

❖ Lab 1b due Friday at 11:59pm (7/10)
  ▪ Submit aisle_manager.c, store_client.c, and lab1Breflect.txt
Administrivia

❖ Unit Summary 1 Due Wednesday 7/15
  ▪ Submitted via Gradescope

❖ Unit Summaries are meant to encourage review/reflection of material in place of exams
  ▪ See course website for specification and instructions, including small examples

❖ Grading very lenient and forgiving, mostly based on effort! If you put in a solid effort you will likely get full credit
Floating point topics

- Fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- **Floating-point in C**

- There are many more details that we won’t cover
  - It’s a 58-page standard...
Floating Point in C

- Two common levels of precision:
  - float 1.0f single precision (32-bit)
  - double 1.0 double precision (64-bit)

- #include <math.h> to get INFINITY and NAN constants
  <float.h> for additional constants

- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results, so just avoid them!

  instead use \( \text{abs}(f1 - f2) < 2^{-20} \)

  \( \uparrow \text{some arbitrary threshold} \)
Floating Point Conversions in C

- **Casting between int, float, and double changes the bit representation**
  - int → float
    - May be rounded (not enough bits in mantissa: 23)
    - Overflow impossible
  - int or float → double
    - Exact conversion (all 32-bit ints representable)
  - long → double
    - Depends on word size (32-bit is exact, 64-bit may be rounded)
  - double or float → int
    - Truncates fractional part (rounded toward zero)
    - “Not defined” when out of range or NaN: generally sets to Tmin (even if the value is a very big positive)
#include <stdio.h>

int main(int argc, char* argv[]) {
    float f1 = 1.0;
    float f2 = 0.0;
    int i;
    for (i = 0; i < 10; i++)
        f2 += 1.0/10.0;

    printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
    printf("f1 = %10.9f\n", f1);
    printf("f2 = %10.9f\n\n", f2);

    f1 = 1E30;
    f2 = 1E-30;
    float f3 = f1 + f2;
    printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no");

    return 0;
}
Floating Point Summary

❖ Floats also suffer from the fixed number of bits available to represent them
  ▪ Can get overflow/underflow
  ▪ “Gaps” produced in representable numbers means we can lose precision, unlike ints
    • Some “simple fractions” have no exact representation (e.g. 0.2)
    • “Every operation gets a slightly wrong result”

❖ Floating point arithmetic not associative or distributive
  ▪ Mathematically equivalent ways of writing an expression may compute different results

❖ Never test floating point values for equality!
❖ Careful when converting between ints and floats!
Number Representation Really Matters

- **1991**: Patriot missile targeting error
  - clock skew due to conversion from integer to floating point
- **1996**: Ariane 5 rocket exploded ($1 billion)
  - overflow converting 64-bit floating point to 16-bit integer
- **2000**: Y2K problem
  - limited (decimal) representation: overflow, wrap-around
- **2038**: Unix epoch rollover
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to TMin in 2038
- **Other related bugs:**
  - 1982: Vancouver Stock Exchange 10% error in less than 2 years
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown “smart” warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)
Summary

❖ Floating point approximates real numbers:

- Handles large numbers, small numbers, special numbers
- Exponent in biased notation (bias = $2^{w-1}-1$)
  - Size of exponent field determines our representable range
  - Outside of representable exponents is overflow and underflow
- Mantissa approximates fractional portion of binary point
  - Size of mantissa field determines our representable precision
  - Implicit leading 1 (normalized) except in special cases
  - Exceeding length causes rounding
Summary

Floating point encoding has many limitations

- Overflow, underflow, rounding
- Rounding is a HUGE issue due to limited mantissa bits and gaps that are scaled by the value of the exponent
- Floating point arithmetic is NOT associative or distributive

Converting between integral and floating point data types *does* change the bits

<table>
<thead>
<tr>
<th>E</th>
<th>M</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0</td>
<td>± 0</td>
</tr>
<tr>
<td>0x00</td>
<td>non-zero</td>
<td>± denorm num</td>
</tr>
<tr>
<td>0x01 – 0xFE</td>
<td>anything</td>
<td>± norm num</td>
</tr>
<tr>
<td>0xFF</td>
<td>0</td>
<td>± ∞</td>
</tr>
<tr>
<td>0xFF</td>
<td>non-zero</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Roadmap

C:
```c
#include <stdlib.h>

#define CAR_STRUCT 
struct car {
    int miles;
    int gals;
} 

int get_mpg(car *c) {
    // Implementation
    return mpg;
}

int main() {
    car *c = malloc(sizeof(car));
    c->miles = 100;
    c->gals = 17;
    float mpg = get_mpg(c);
    free(c);
    return 0;
}
```

Java:
```java
public class Car {
    private int miles;
    private int gals;

    public Car() {
    }

    public void setMiles(int miles) {
        this.miles = miles;
    }

    public void setGals(int gals) {
        this.gals = gals;
    }

    public float getMPG() {
        // Implementation
        return mpg;
    }
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car();
        c.setMiles(100);
        c.setGals(17);
        float mpg = c.getMPG();
    }
}
```

Assembly language:
```
get_mpg:
    pushq  %rbp
    movq   %rsp, %rbp
    ...
    popq   %rbp
    ret
```

Machine code:
```
0111010000110000
1000110100000100
1000100111000010
1100000111111101
1000000111111010
```

Computer system:
- OS:
  - Windows 10
  - OS X Yosemite
- Hardware: Intel Core i5, RAM, SSD

Memory & data
- Integers & floats

x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

Java vs. C
- Differences in memory management
- Java's automatic memory management
- C's manual memory management

Operating System
- Comparison of OS X Yosemite and Windows 10
- Features of each OS

Virtual memory
- Memory management techniques
- Page replacement algorithms

Memory allocation
-分配内存
- Heap vs. stack
- Dynamic allocation

Arrays & structs
- Data structures in C
- Arrays vs. Java's Collections

Executables
- Library dependencies
- Compilation process

Processes
- Process creation
- Process states
- Process scheduling

Procedure & stacks
- Function call stack
- Recursion

Memory & caches
- Cache hierarchy
- LRU vs. FIFO
- Cache coherence

Integers & floats
- Arithmetic operations
- Floating-point operations

Computer system
- Hardware components
- CPU architecture
- Memory system

OS:
- Windows 10
- OS X Yosemite
- Features of each OS

Memory & data
- Integers & floats

x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

Java vs. C
- Differences in memory management
- Java's automatic memory management
- C's manual memory management

Operating System
- Comparison of OS X Yosemite and Windows 10
- Features of each OS

Virtual memory
- Memory management techniques
- Page replacement algorithms

Memory allocation
-分配内存
- Heap vs. stack
- Dynamic allocation

Arrays & structs
- Data structures in C
- Arrays vs. Java's Collections

Executables
- Library dependencies
- Compilation process

Processes
- Process creation
- Process states
- Process scheduling

Procedure & stacks
- Function call stack
- Recursion

Memory & caches
- Cache hierarchy
- LRU vs. FIFO
- Cache coherence

Integers & floats
- Arithmetic operations
- Floating-point operations

Computer system
- Hardware components
- CPU architecture
- Memory system
Architecture Sits at the Hardware Interface

Source code
Different applications or algorithms

Compiler
Perform optimizations, generate instructions

Architecture
Instruction set

Hardware
Different implementations

- Intel Pentium 4
- Intel Core 2
- Intel Core i7
- AMD Opteron
- AMD Athlon
- ARM Cortex-A53
- Apple A7

C Language

Program A

Program B

Your program

Compiler

GCC

Clang

x86-64

ARMv8 (AArch64/A64)
Definitions

❖ **Architecture (ISA):** The parts of a processor design that one needs to understand to write assembly code
  - “What is directly visible to software”

❖ **Microarchitecture:** Implementation of the architecture
  - CSE/EE 469
Instruction Set Architectures

❖ The ISA defines:
  ▪ The system’s state (e.g. registers, memory, program counter)
  ▪ The instructions the CPU can execute
  ▪ The effect that each of these instructions will have on the system state
Instruction Set Philosophies

❖ **Complex Instruction Set Computing (CISC):** Add more and more elaborate and specialized instructions as needed
  - Lots of tools for programmers to use, but hardware must be able to handle all instructions
  - x86-64 is CISC, but only a small subset of instructions encountered with Linux programs

❖ **Reduced Instruction Set Computing (RISC):** Keep instruction set small and regular
  - Easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones
General ISA Design Decisions

❖ Instructions
  ▪ What instructions are available? What do they do?
  ▪ How are they encoded?

❖ Registers
  ▪ How many registers are there?
  ▪ How wide are they?

❖ Memory
  ▪ How do you specify a memory location?
Mainstream ISAs

**x86**
- Designer: Intel, AMD
- Bits: 16-bit, 32-bit and 64-bit
- Design: CISC
- Type: Register-memory
- Encoding: Variable (1 to 15 bytes)
- Endianness: Little

**ARM architectures**
- Designer: ARM Holdings
- Bits: 32-bit, 64-bit
- Introduced: 1985; 31 years ago
- Design: RISC
- Type: Register-Register
- Encoding: AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility
- Endianness: Little

**MIPS**
- Designer: MIPS Technologies, Inc.
- Bits: 64-bit (32→64)
- Introduced: 1981; 35 years ago
- Design: RISC
- Type: Register-Register
- Encoding: Fixed
- Endianness: Bi

---

Macbooks & PCs
(Core i3, i5, i7, M)
**x86-64 Instruction Set**

Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
**ARM Instruction Set**

Digital home & networking equipment
(Blu-ray, PlayStation 2)
**MIPS Instruction Set**
Writing Assembly Code? In 2020???

- Chances are, you’ll never write a program in assembly, but understanding assembly is the key to the machine-level execution model:
  - Behavior of programs in the presence of bugs
    - When high-level language model breaks down
  - Tuning program performance
    - Understand optimizations done/not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing systems software
    - What are the “states” of processes that the OS must manage
    - Using special units (timers, I/O co-processors, etc.) inside processor!
  - Fighting malicious software
    - Distributed software is in binary form
Assembly Programmer’s View

❖ Programmer-visible state
  ▪ PC: the Program Counter ($rip$ in x86-64)
    • Address of next instruction
  ▪ Named registers
    • Together in “register file”
    • Heavily used program data
  ▪ Condition codes
    • Store status information about most recent arithmetic operation
    • Used for conditional branching

❖ Memory
  ▪ Byte-addressable array
  ▪ Code and user data
  ▪ Includes the Stack (for supporting procedures)
x86-64 Assembly “Data Types”

- Integral data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses
- Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  - Different registers for those (e.g. %xmm1, %ymm2)
  - Come from extensions to x86 (SSE, AVX, …)
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
- Two common syntaxes
  - “AT&T”: used by our course, slides, textbook, gnu tools, …
  - “Intel”: used by Intel documentation, Intel tools, …
  - Must know which you’re reading

Not covered in 351
What is a Register?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)

- Registers have *names*, not *addresses*
  - In assembly, they start with `%` (*e.g.* `%rsi`)

- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but especially x86
x86-64 Integer Registers – 64 bits wide

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Can reference low-order 4 bytes (also low-order 2 & 1 bytes)
### Some History: IA32 Registers – 32 bits wide

<table>
<thead>
<tr>
<th>Register</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>accumulate</td>
</tr>
<tr>
<td>%cx</td>
<td>counter</td>
</tr>
<tr>
<td>%edx</td>
<td>data</td>
</tr>
<tr>
<td>%ebx</td>
<td>base</td>
</tr>
<tr>
<td>%esi</td>
<td>source_index</td>
</tr>
<tr>
<td>%edi</td>
<td>destination_index</td>
</tr>
<tr>
<td>%esp</td>
<td>stack_pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>base_pointer</td>
</tr>
</tbody>
</table>

- %eax: 16-bit virtual registers (backwards compatibility)
- %eax: Name Origin (mostly obsolete)

- %esi: 16 bits
- %esi: 8 bits

- %eax: 32 bits (same as last slide)
## Memory vs. Registers

- **Addresses**
  - 0x7FFFD024C3DC

- **Big**
  - ~8 GiB

- **Slow**
  - ~50-100 ns

- **Dynamic**
  - Can “grow” as needed while program runs

- **Names**
  - %rdi

- **Small**
  - (16 x 8 B) = 128 B

- **Fast**
  - sub-nanosecond timescale

- **Static**
  - fixed number in hardware
Three Basic Kinds of Instructions

1) Transfer data between memory and register
   - Load data from memory into register
     - $%\text{reg} = \text{Mem}[\text{address}]$
   - Store register data into memory
     - $\text{Mem}[\text{address}] = %\text{reg}$

2) Perform arithmetic operation on register or memory data
   - $c = a + b; \quad z = x \ll y; \quad i = h \& g;$

3) Control flow: what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches

Remember: Memory is indexed just like an array of bytes!
Operand types

❖ **Immediate:** Constant integer data
  - Examples: $0x400, -533
  - Like C literal, but prefixed with ‘$’
  - Encoded with 1, 2, 4, or 8 bytes depending on the instruction

❖ **Register:** 1 of 16 integer registers
  - Examples: %rax, %r13
  - But %rsp reserved for special use
  - Others have special uses for particular instructions

❖ **Memory:** Consecutive bytes of memory at a computed address
  - Simplest example: (%rax)
  - Various other “address modes”
x86-64 Introduction

❖ Data transfer instruction (*mov*)
❖ Arithmetic operations
❖ Memory addressing modes
  ▪ *swap* example
❖ Address computation instruction (*lea*)
Moving Data

❖ General form: `mov_ source, destination`
  ▪ Missing letter (_) specifies size of operands
  ▪ Note that due to backwards-compatible support for 8086 programs (16-bit machines!), “word” means 16 bits = 2 bytes in x86 instruction names
  ▪ Lots of these in typical code

❖ `movb src, dst`
  ▪ Move 1-byte “byte”

❖ `movw src, dst`
  ▪ Move 2-byte “word”

❖ `movl src, dst`
  ▪ Move 4-byte “long word”

❖ `movq src, dst`
  ▪ Move 8-byte “quad word”
# Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movq $0x4, %rax</td>
<td>var_a = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq $-147, (%rax)</td>
<td>*p_a = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>Mem</td>
<td>movq %rax, (%rdx)</td>
<td>var_d = var_a;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax), %rdx</td>
<td>var_d = *p_a;</td>
</tr>
</tbody>
</table>

- Cannot do memory-memory transfer with a single instruction
  - How would you do it?
Some Arithmetic Operations

- **Binary (two-operand) Instructions:**
  - **Maximum of one memory operand**
  - Beware argument order!
  - No distinction between signed and unsigned
    - Only arithmetic vs. logical shifts
  - How do you implement “r3 = r1 + r2”?

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>operand size specifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq src, dst</td>
<td>dst = dst + src</td>
<td>(dst += src)</td>
</tr>
<tr>
<td>subq src, dst</td>
<td>dst = dst - src</td>
<td>signed mult</td>
</tr>
<tr>
<td>imulq src, dst</td>
<td>dst = dst * src</td>
<td>Arithmetic logical</td>
</tr>
<tr>
<td>sarq src, dst</td>
<td>dst = dst &gt;&gt; src</td>
<td>(same as salq)</td>
</tr>
<tr>
<td>shrq src, dst</td>
<td>dst = dst &gt;&gt; src</td>
<td></td>
</tr>
<tr>
<td>shlq src, dst</td>
<td>dst = dst &lt;&lt; src</td>
<td></td>
</tr>
<tr>
<td>xorq src, dst</td>
<td>dst = dst ^ src</td>
<td></td>
</tr>
<tr>
<td>andq src, dst</td>
<td>dst = dst &amp; src</td>
<td></td>
</tr>
<tr>
<td>orq src, dst</td>
<td>dst = dst</td>
<td></td>
</tr>
</tbody>
</table>

Imm, Reg, or Mem
Polling Question [Asm I – a]

❖ Assume: r3 is in %rcx, r1 is in %rax, and r2 is in %rbx
which of the following would implement:

\[ r3 = r1 + r2 \]

✦ Vote at [http://pollev.com/pbjones](http://pollev.com/pbjones)

A. `addq %rax, %rbx, %rcx`
B. `addq %rcx, %rax, %rbx`
C. `movq %rax, %rcx
   addq %rbx, %rcx`
D. `movq (%rbx), %rcx
   addq (%rax), %rcx`
E. We’re lost…
Some Arithmetic Operations

- **Unary (one-operand) Instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>incq dst</code></td>
<td>$dst = dst + 1$</td>
</tr>
<tr>
<td><code>decq dst</code></td>
<td>$dst = dst - 1$</td>
</tr>
<tr>
<td><code>negq dst</code></td>
<td>$dst = -dst$</td>
</tr>
<tr>
<td><code>notq dst</code></td>
<td>$dst = \sim dst$</td>
</tr>
</tbody>
</table>

- See CSPP Section 3.5.5 for more instructions: `mulq`, `cqto`, `idivq`, `divq`
## Arithmetic Example

```c
long simple_arith(long x, long y) {
    long t1 = x + y;
    long t2 = t1 * 3;
    return t2;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

```assembly
.simple_arith:
    addq  %rdi, %rsi  ; y += x;
    imulq 3, %rsi      ; y *= 3;
    movq  %rsi, %rax   ; long r = y;
    ret
```

Example of Basic Addressing Modes

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

```assembly
swap:
    movq (%rdi), %rax
    movq (%rsi), %rdx
    movq %rdx, (%rdi)
    movq %rax, (%rsi)
    ret
```
Understanding `swap()`

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

**Register**  | **Variable**
---|---
%rdi  | xp
%rsi  | yp
%rax  | t0
%rdx  | t1

**Registers**

`movq (%rdi), %rax`
`movq (%rsi), %rdx`
`movq %rdx, (%rdi)`
`movq %rax, (%rsi)`
`ret`
Understanding `swap()`

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>0x120</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>0x100</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td></td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td></td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

### swap:

- `movq (%rdi), %rax`  # `t0 = *xp`
- `movq (%rsi), %rdx`  # `t1 = *yp`
- `movq %rdx, (%rdi)`  # `*xp = t1`
- `movq %rax, (%rsi)`  # `*yp = t0`
- `ret`
Understanding `swap()`

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x100</td>
</tr>
<tr>
<td>123</td>
</tr>
<tr>
<td>456</td>
</tr>
</tbody>
</table>

### swap:

- `movq (%rdi), %rax` # t0 = *xp
- `movq (%rsi), %rdx` # t1 = *yp
- `movq %rdx, (%rdi)` # *xp = t1
- `movq %rax, (%rsi)` # *yp = t0

- `ret`
Understanding `swap()`

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>0x120</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>0x100</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>123</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>456</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>123</th>
<th>456</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### `swap`:

```assembly
movq (%rdi), %rax  # t0 = *xp
movq (%rsi), %rdx  # t1 = *yp
movq %rdx, (%rdi)  # *xp = t1
movq %rax, (%rsi)  # *yp = t0
ret
```
### Understanding `swap()`

#### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

#### Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

#### swap:

```assembly
swap:
  movq (%rdi), %rax  # t0 = *xp
  movq (%rsi), %rdx  # t1 = *yp
  movq %rdx, (%rdi)  # *xp = t1
  movq %rax, (%rsi)  # *yp = t0
  ret
```
Understanding `swap()`

**Registers**

| %rdi | 0x120 |
| %rsi | 0x100 |
| %rax | 123   |
| %rdx | 456   |

**Memory**

<table>
<thead>
<tr>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>456</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

**swap:**

```assembly
movq (%rdi), %rax  # t0 = *xp
movq (%rsi), %rdx  # t1 = *yp
movq %rdx, (%rdi)  # *xp = t1
movq %rax, (%rsi)  # *yp = t0
ret
```
Memory Addressing Modes: Basic

❖ **Indirect:** \((R)\) \text{Mem}[	ext{Reg}[R]]
  - Data in register \(R\) specifies the memory address
  - Like pointer dereference in C
  - **Example:** \text{movq} (\%rcx), \%rax

❖ **Displacement:** \(D(R)\) \text{Mem}[	ext{Reg}[R]+D]
  - Data in register \(R\) specifies the \textit{start} of some memory region
  - Constant displacement \(D\) specifies the offset from that address
  - **Example:** \text{movq} 8(\%rbp), \%rdx
Complete Memory Addressing Modes

❖ **General:**

- \( D(Rb, Ri, S) \) \( \text{Mem[Reg[Rb]+Reg[Ri]*S+D]} \)
  - \( Rb \): Base register (any register)
  - \( Ri \): Index register (any register except \( \%rsp \))
  - \( S \): Scale factor (1, 2, 4, 8) – *why these numbers?*
  - \( D \): Constant displacement value (a.k.a. immediate)

❖ **Special cases** (see CSPP Figure 3.3 on p.181)

- \( D(Rb, Ri) \) \( \text{Mem[Reg[Rb]+Reg[Ri]+D]} \) (\( S=1 \))
- \( (Rb, Ri, S) \) \( \text{Mem[Reg[Rb]+Reg[Ri]*S]} \) (\( D=0 \))
- \( (Rb, Ri) \) \( \text{Mem[Reg[Rb]+Reg[Ri]]} \) (\( S=1, D=0 \))
- \( (, Ri, S) \) \( \text{Mem[Reg[Ri]*S]} \) (\( Rb=0, D=0 \))
Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ D(Rb, Ri, S) \rightarrow Mem[Reg[Rb] + Reg[Ri] \times S + D] \]
Summary

❖ x86-64 is a complex instruction set computing (CISC) architecture
  ▪ There are 3 types of operands in x86-64
    • Immediate, Register, Memory
  ▪ There are 3 types of instructions in x86-64
    • Data transfer, Arithmetic, Control Flow

❖ Memory Addressing Modes: The addresses used for accessing memory in mov (and other) instructions can be computed in several different ways
  ▪ Base register, index register, scale factor, and displacement map well to pointer arithmetic operations