X64 Assembly Language

September 12, 2018

CS 105, Computer Systems Pomona College

Today

- We'll get a good start on Process Lab, next week
- X64 Assembly Language

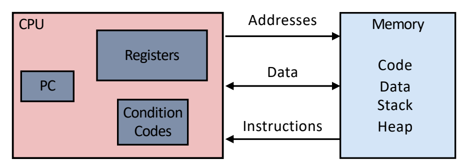
CS 105, Computer Systems Pomona College

X64 Assembly Language

- Intel Pentium: 64 bit instruction set
- Evolutionary design, going back to 8086 in 1978
  - Basis for original IBM Personal Computer, 16-bits
- Other languages are translated into X64 instructions and then executed on the CPU
  - Actual instructions are sequences of bytes
  - We give them mnemonic names

CS 105, Computer Systems Pomona College

Assembly/Machine Code View



Programmer-Visible State

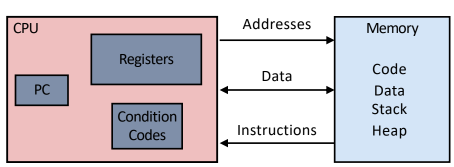
- PC: Program counter
- 16 Registers
- Condition codes

Memory

- Byte addressable array
- Code and user data
- Stack to support procedures

CS 105, Computer Systems Pomona College

Assembly/Machine Code View



Repeat forever:

- Fetch instruction at address in PC
- Execute the instruction
- Update PC

CS 105, Computer Systems Pomona College

X86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

X86-64 Register Usage Conventions

%rax, function result	%r8
%rbx	%r9
%rcx, fourth argument	%r10
%rdx, third argument	%r11
%rsi, second argument	%r12
%rdi, first argument	%r13
%rsp, stack pointer	%r14
%rbp	%r15

Callee-saved registers are in yellow

Assembly Characteristics: Data Types

- "Integer" data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- Code: Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Assembly Characteristics: Operations

- Perform arithmetic function on register or memory data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Compiling into Assembly

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest) {

    long t = plus(x, y);
    *dest = t;

}
```

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

Obtain assembly listing (on project5) with command

`gcc -Og -S sum.c`

Produces the file `sum.s`

May get very different results on different machines!

Machine Instruction Example

```
*dest = t;
```

- C Code
  - Store value `t` where designated by `dest`

```
movq %rax, (%rbx)
```

- Assembly
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - `t:` Register `%rax`
    - `dest:` Register `%rbx`
    - `*dest:` Memory `M[%rbx]`

```
0x40059e:   48 89 03
```

- Object Code
  - 3-byte instruction
  - at address `0x40059e`

Disassembling Object Code

```
0000000000400595 <sumstore>:
  400595:  53              push   %rbx
  400596:  48 89 d3        mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff  callq  400590 <plus>
  40059e:  48 89 03        mov    %rax,(%rbx)
  4005a1:  5b              pop    %rbx
  4005a2:  c3              retq
```

- Disassembler
  - `$ objdump -d sum`
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

Alternate Disassembly

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- Using the gdb Debugger

```
$ gdb sum
(gdb) disassemble sumstore
(gdb) x/14xb sumstore
```

  - Examine the 14 bytes starting at **sumstore**

Moving Data

movq ***Source, Dest*****:**

- Operand Types
  - ***Immediate:*** Constant integer data
    - Example: **$0x400, $-533**
    - Like C constant, but prefixed with '**$**'
    - Encoded with 1, 2, 4, or 8 bytes
  - ***Register:*** One of 16 integer registers
    - Example: **%rax, %r13**
    - But **%rsp** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** up to 8 consecutive bytes of memory at address given by register
    - Simplest example: **(%rax)**
    - Various other "address modes"

%rax
%rcx
%rdx
%rbx
%rsi
%rdi
%rsp
%rbp
%rN …

movq Operand Combinations

	Source	Dest	Src,Dest	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

swap - Example of Simple Addressing Modes

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

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

- First argument, **xp**, is in **%rdi**
- Second argument, **yp** is in **%rsi**
- Compiler decides to use **%rax** and **%rdx** for **t0** and **t1**
  No need to allocate memory

Memory Addressing—General Case

- Most General Form

  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]

  - D: Constant "displacement"
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for **%rsp**
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- Special Cases

  (Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]

  D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D]

  (Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Address Computation Instruction

- **leaq** Src, Dst
  - Src is address mode expression
  - Set Dst to address denoted by expression
- Uses
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8
- Example

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```


Some Arithmetic Operations

- Two Operand Instructions:

Format		Computation	
addq	Src,Dest	Dest = Dest + Src	
subq	Src,Dest	Dest = Dest – Src	
imulq	Src,Dest	Dest = Dest * Src	Signed multiply
salq	Src,Dest	Dest = Dest << Src	Also called shlq
sarq	Src,Dest	Dest = Dest >> Src	Arithmetic
shrq	Src,Dest	Dest = Dest >> Src	Logical
xorq	Src,Dest	Dest = Dest ^ Src	
andq	Src,Dest	Dest = Dest & Src	
orq	Src,Dest	Dest = Dest | Src	

- Watch out for argument order!
- Different instructions for signed/unsigned multiply and divide
- Otherwise, no distinction between signed and unsigned int (why?)

Some Arithmetic Operations

- One Operand Instructions

incq	Dest	Dest = Dest + 1
decq	Dest	Dest = Dest – 1
negq	Dest	Dest = – Dest
notq	Dest	Dest = ~Dest

- See text for more instructions

Assembly Operations

- addq $47, %rax
- addq %rbx, %rax
- addq (%rbx), %rax
- addq %rbx, (%rax)
- addq 12(%rbx,%rdi,2), %rax

Suffixes

char	b	1
short	w	2
int	l	4
long	q	8
pointer	q	8

- Also movq, subq, andq, …
- leaq 12(%rbx,%rdi,2), %rax

Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

Interesting Instructions
- **leaq**: address computation
- **salq**: shift
- **imulq**: multiplication
  - But, only used once

Understanding Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx   # t5
  imulq   %rcx, %rax           # rval
  ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z, t4
%rax	t1, t2, rval
%rcx	t5

Preview: Conditional Branching

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

```
absdiff:
   cmpq    %rsi, %rdi  # x:y
   jle     .L4
   movq    %rdi, %rax
   subq    %rsi, %rax
   ret
.L4:       # x <= y
   movq    %rsi, %rax
   subq    %rdi, %rax
   ret
```

Register	Use
%rdi	x
%rsi	y
%rax	return value