Machine Level Programming: Basics (III)

CSCI 237: Computer Organization
13th Lecture, Monday, October 6

Kelly Shaw

Administrative Details

- Lab #3 checkpoint due Tuesday at 11pm
  - Watch the video for getting started with the bomb
- Read CSAPP 3.5-3.6
- Colloquium Friday at 2:35pm in Wege
  - What I did this summer (industry)

Last Time: Machine-Level Programming: Basics

- Dynamic memory allocation
- Assembly instruction basics: registers, operands, move

Today: Machine-Level Programming: Basics

- Assembly instruction basics: registers, operands, move
- Arithmetic and logical operations
- Intro to data-dependent control
  - Condition codes
  - Conditional branches
  - Conditional data
  - Loops
  - Switch Statements

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset (which can be positive or negative)

```
movq 8(%rbp),%rdx
```

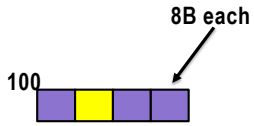
5

5

movq 8(%rbp), %rdx

Assume %rbp holds 100

```
long arr_long[4];
long tmp = arr_long[1];
```



6

6

Complete Memory Addressing Modes

- Most General Form

  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]
  - D: Constant "displacement" stored in 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
  - S: Scale: 1, 2, 4, or 8

- 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]]

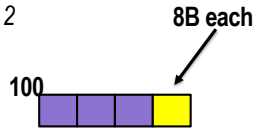
7

7

movq 8(%rbp, %rdi, 8), %rdx

Assume %rbp holds 100, %rdi holds 2

```
long arr_long[4];
long tmp = arr_long[3];
```



8

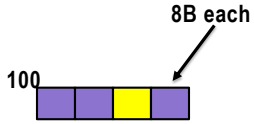
8

Practice On Your Own

Assume %rbp holds 100

```
long arr_long[4];

// Write code in assembly
arr_long[2] = 7;
```



9

9

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Most General Form
D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**
- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for %rsp
- S: Scale: 1, 2, 4, or 8

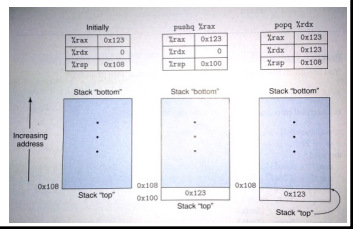
Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

10

10

Pushing and Popping Stack Data

- In addition to **mov**, can move data to and from program stack using **push** and **pop**
  - Review: Stacks are LIFO (Last In First Out)
  - Usually drawn upside down ("top" of stack is on bottom of pic)
  - The stack is part of memory
  - Registers are part of CPU
- %rsp holds address of top element
- **push**: Add data to top of stack
- **pop**: Remove data from stack



11

11

Today: Machine-Level Programming: Basics

- Assembly instruction basics: registers, operands, move
- Arithmetic and logical operations
- Intro to data-dependent control
  - Condition codes
  - Conditional branches
  - Conditional data
  - Loops
  - Switch Statements

12

12

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	
`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! *Src,Dest*
  (Warning (again): Intel docs use "op *Dest,Src*")
- No distinction between signed and unsigned int

13

13

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 book for more instructions

14

14

Address Computation Instruction

- **`leaq`** Src, Dst
  - "Load Effective Address" – copy memory address in src to dst
  - Src is an address
  - 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+2*x
salq $2, %rax             # return t<<2
```

16

16

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
```

Instructions

- **`leaq`**: address computation
- **`salq`**: left shift
- **`imulq`**: multiplication
  - Only used once

17

17

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

18

18

Practice on Your Own

- Rewrite the following C code in assembly:

```
void fcn(long *arrPtr)
{
   arrPtr[2] = arrPtr[1] + arrPtr[0];
}
```

- Rewrite the following assembly code in C code:

```
fcn:
    leaq (%rdi, %rsi, 8), %rax
    movq (%rax), %r8
    addq $16, %r8
    movq %r8, (%rax)
    ret
```

Note: Remember %rdi stores the first parameter and %rsi the second

19

19

Practice

- Rewrite the following C code in assembly:

```
long fcn(long *x, long *y, int num)
{
   long temp = x[num];
   long result = temp + y[num-temp];
   return result;
}
```

Note: Remember %rdi stores the first parameter and %rsi the second

20

20

Today: Machine-Level Programming: Basics

- Assembly instruction basics: registers, operands, move
- Arithmetic and logical operations
- Intro to data-dependent control
  - Condition codes
  - Conditional branches
  - Conditional data
  - Loops
  - Switch Statements

21

21

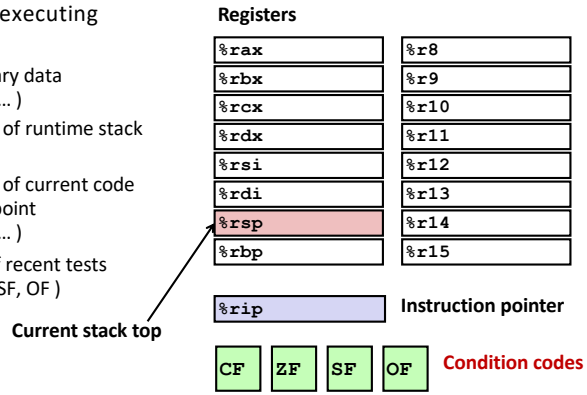
Data-Dependent Control (Ch 3.6)

What about non-straight-line code?

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

Processor State (x86-64, Partial)

- Information about currently executing program
  - Temporary data (**%rax**, …)
  - Location of runtime stack (**%rsp**)
  - Location of current code control point (**%rip**, …)
  - Status of recent tests (CF, ZF, SF, OF)



Condition Code Registers

We have several 1-bit condition registers.
The most useful ones are:

- **[Condition Code]**
  - Set to 1 if the most recent op …
- **[CF]** carry flag
  - Generated a carry out of most significant bit
- **[ZF]** Zero flag
  - Yielded 0
- **[SF]** Sign flag
  - Yielded a negative value
- **[OF]** Overflow flag
  - Caused a twos-complement overflow (positive *or* negative)

Setting Condition Code Registers

Condition codes are set after each arithmetic or logical op.
Condition codes are also set by compare and test instructions:

- **cmpX** S_1, S_2
  - Like **subX** S_1 S_2 : Calculates (S_2 – S_1), but does not overwrite S_2
- **testX**
  - Like **AND** S_1 S_2 : Calculates (S_1 & S_2), but does not overwrite S_2

Note: Condition codes are not altered by `leaq`!

Reading Condition Codes

- Three ways to "access" condition codes in assembly. We will go over them in detail:
  1. Operations that set a byte to 0/1 based on some combination of the condition codes
  2. Operations that "jump" to some part of program based on condition codes
  3. Operations that transfer data only if some condition codes are set

We are going to do a lot of conversion between C and assembly, and between assembly and C. The practice problems and examples in the textbook are really helpful!

26