Machine Level Programming: Arithmetic and Control

CSCI 237: Computer Organization
13th Lecture, Friday, Oct. 4

Kelly Shaw

Administrative Details

- Lab #3 checkpoint due Tuesday at 11pm
  - Any questions?
  - Watch video about gdbomb lab to help with phase 1: https://courses.cs.washington.edu/courses/cse351/videos/tutorials/gdb.mp4
- Weekly quiz due today at 2:30pm
- Read CSAPP 3.7-3.8
- Watch short lecture video to be posted on Glow
- CS Colloquium talk on Friday at 2:35pm in Wege
  - What I did over the summer

Last Time: Machine-Level Programming: Basics

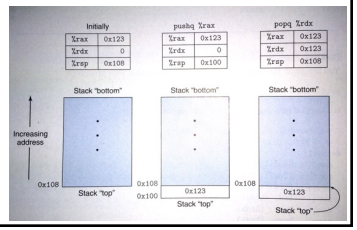
- Dynamic memory allocation in C
- Assembly instruction basics: registers, operands, move

Today: Machine-Level Programming: Control

- Arithmetic and Logic Instructions
- gdb commands for bomb portion of lab
- Intro to data-dependent control
  - Condition codes
  - Conditional branches
  - Conditional data
  - Loops
  - Switch Statements

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



5

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

6

6

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

7

7

Review: `movq` 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`

8

8

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
```

9

9

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

10

10

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

11

11

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

12

12

Today: Machine-Level Programming: Control

- Arithmetic and logic instructions
- gdb commands for bomb portion of lab
- Intro to data-dependent control
  - Condition codes
  - Conditional branches
  - Conditional data
  - Loops
  - Switch Statements

13

13

gdb commands for assembly

Description	C	Assembly
Display code snippet	list	disassemble
Specify language for display	layout src	layout asm
Step into next instruction including into fcn calls	s (or step)	si (or stepi)
Execute next instr but don't go into func calls	n (or next)	ni (or nexti)
Complete current func	f (or finish)	
Print value	p <variable or addr>	p <variable/addr/$reg>
Examine memory	x /nfu <addr or $reg> **/n** – num of items **/f** – format (i=instr, s=str, x-hex, t=bin, a=addr, c=char **/u** – size of data unit (b=byte, h=halfword, w=word, g=giant)	
Examine registers		info reg info reg $reg p $reg layout reg

14

14

Today: Machine-Level Programming: Control

- Arithmetic and logic instructions
- gdb commands for bomb portion of lab
- Intro to data-dependent control
  - Condition codes
  - Conditional branches
  - Conditional data
  - Loops
  - Switch Statements

15

15

Data-Dependent Control (Ch 3.6)

What about non-straight-line code?

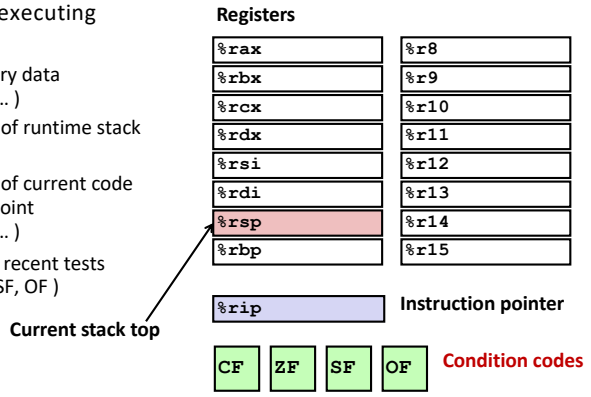
- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

16

16

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)



17

17

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)

18

18

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`!

19

19

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!

20

20

Reading Condition Codes 1: `setX` instrs

- Instructions of form: **`setX Dest`**
  - Set **low-order byte** of destination reg to 0 or 1 based on combinations of condition codes
    - Does not alter remaining 7 bytes!

SetX	Condition	Description	
`sete`	**`ZF`**	Equal / Zero	
`setne`	**`~ZF`**	Not Equal / Not Zero	
`sets`	**`SF`**	Negative	
`setns`	**`~SF`**	Nonnegative	
`setg`	**`~(SF^OF)&~ZF`**	Greater (Signed)	
`setge`	**`~(SF^OF)`**	Greater or Equal (Signed)	
`setl`	**`(SF^OF)`**	Less (Signed)	
`setle`	**`(SF^OF)|ZF`**	Less or Equal (Signed)	
`seta`	**`~CF&~ZF`**	Above (unsigned)	Set on unsigned greater
`setb`	**`CF`**	Below (unsigned)	Set on unsigned less

21

21

x86-64 Integer Registers

`%rax`	`%al`	`%r8`	`%r8b`
`%rbx`	`%bl`	`%r9`	`%r9b`
`%rcx`	`%cl`	`%r10`	`%r10b`
`%rdx`	`%dl`	`%r11`	`%r11b`
`%rsi`	`%sil`	`%r12`	`%r12b`
`%rdi`	`%dil`	`%r13`	`%r13b`
`%rsp`	`%spl`	`%r14`	`%r14b`
`%rbp`	`%bpl`	`%r15`	`%r15b`

- Recall: We can specifically reference the low-order byte of registers

22

22

Reading Condition Codes 1: `setX` (Cont.)

- **`setX D`**
  - Set single byte based on combination of condition codes
- **`D`** must be one of addressable byte registers
  - Does not alter remaining bytes
  - Typically use **`movzbl`** to finish job
    - 32-bit instructions also set upper 32 bits to 0

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

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rax`	Return value

```
cmpq   %rsi, %rdi   # Compare x:y
setg   %al          # Set to 1 when >
movzbl %al, %eax    # Zero rest of %rax
ret
```

23

23