Section 2: Computations & the stack

Address Computation Instruction

- **`leaq`** ***Src*, *Dst***
  - *Load Effective Address*
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression
  - Doesn’t affect condition codes (AGU op instead of an ALU op)
  - http://stackoverflow.com/questions/1658294/whats-the-purpose-of-the-lea-instruction

- 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
    - e. g. if %rdx contains a value x, then **leaq 7(%rdx, %rdx,4), %rax** sets %rax to 5x+7
- 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
```

~~Stupid math tricks~~ Computations with lea

- Consider:

```
leaq (%rdi, %rdi,1), %rax    =>   %rdi  + 1*%rdi = 2%rdi
leaq (%rdi, %rdi,2), %rax    =>   %rdi + 2*%rdi = 3%rdi
leaq (%rdi, %rdi,4), %rax    =>   %rdi + 4*%rdi = 5%rdi
leaq (%rdi, %rdi,8), %rax    =>   %rdi + 8*%rdi = 9%rdi
```

- What kind of multiplication problems can you come up with that might make these valuable?

```
leaq(%rdi, %rdi,2), %rax           # 3%rdi
leaq(%rdi, %rdi,8), %rbx           # 9%rdi
addq %rbx, %rax              # 12%rdi
```

Some Arithmetic Operations

- Two Operand Instructions:

Format		*Computation*	
`add`	*Src,Dest*	Dest = Dest + Src	
`sub`	*Src,Dest*	Dest = Dest – Src	
`imul`	*Src,Dest*	Dest = Dest * Src signed multiply	
`mul`	*Src,Dest*	Dest = Dest * Src unsigned multiply	
`idiv`	*Src,Dest*	Dest = Dest / Src signed divide	
`div`	*Src,Dest*	Dest = Dest / Src unsigned divide	
`sal`	*Src,Dest*	Dest = Dest << Src	***Also called shlq***
`sar`	*Src,Dest*	Dest = Dest >> Src	***Arithmetic (fills w/copy of sign bit)***
`shr`	*Src,Dest*	Dest = Dest >> Src	***Logical (fillls with 0s)***
`xor`	*Src,Dest*	Dest = Dest ^ Src	
`and`	*Src,Dest*	Dest = Dest & Src	
`or`	*Src,Dest*	Dest = Dest | Src	

- Watch out for argument order!
- Except for mul and div, no distinction between signed and unsigned int (why?)
- Don’t forget to include a suffix for each of these instructions.

Some Arithmetic Operations

- One Operand Instructions

`inc`	*Dest*	*Dest = Dest + 1*
`dec`	*Dest*	*Dest = Dest – 1*
`neg`	*Dest*	*Dest = – Dest*
`not`	*Dest*	*Dest = ~Dest*

- See book for more instructions (Figure 3.10)

- Obviously, each of these instructions must use the appropriate suffix based on the Destination size

Arithmetic Expression Example

(z+x+y)*((x+4)+(y*48))

```
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 = x+y
  addq    %rdx, %rax           # t2 = z + t1
  leaq    (%rsi,%rsi,2),%rdx   # %rdx = y+2y
  salq    $4, %rdx             # %rdx * 16
  leaq    4(%rdi,%rdx), %rcx   # x + t4 + 4
  imulq   %rcx, %rax           # t2=t2*t5
  ret
```

Interesting Instructions

- **leaq**: address computation
- **salq**: shift arithmetic left
- **imulq**: signed multiply
  - 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**
%rax	**t1**, **t2**, **rval**
%rdx	**t4**
%rcx	**t5**

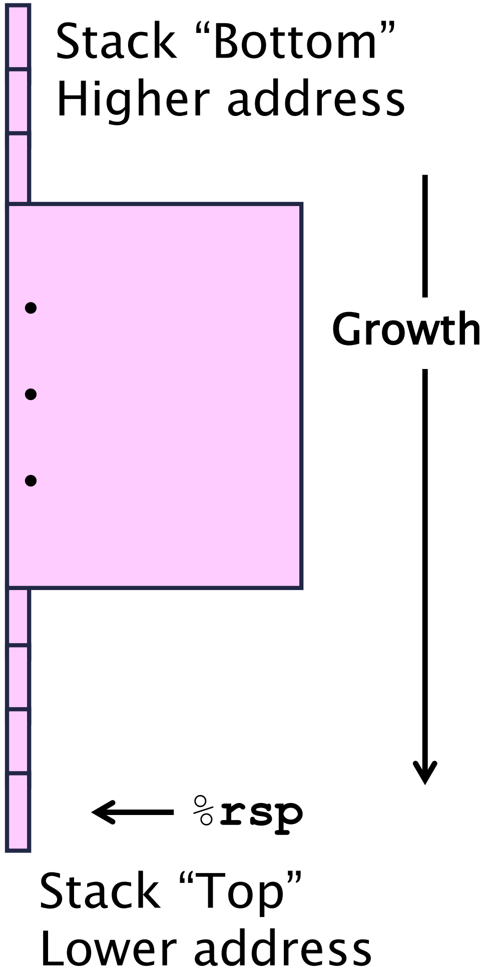
X86 program stack

- The program stack is actually divided conceptually into *frames*.
- Each procedure or function (main and any functions called from main or from another function) has its own part of the stack to use, which is called its frame.
- The frame goes from the stack address pointed to by %rbp in that procedure, this is called the frame (or base) pointer, to %rsp, which points to the top of the stack while the procedure is running.
- This implies that the address pointed to by %rbp is different in different procedures: %rbp must be set when the procedure is entered.

X86 Stack

- Stack top address always held in register %rsp
- Stack grows towards lower addresses

- Where is %rbp???
  - That depends…☺



Use of the stack in X86-64

- To save the caller's %rbp (frame pointer) before setting its own frame pointer;
- To preserve values needed after return before calling another function;
- To pass parameters to another function (if there are more than 6 parameters to pass);
- To store the return address when a call instruction is executed.
- If more data than registers, automatic variables

Procedure calls and returns

- To use procedure calls and returns in our X86 program, we have to manage the program stack and program registers correctly.
- Two different aspects to this:
  - Maintain the stack pointer and associated data in relation to each procedure call and return. (The OS initializes these values upon system start.)
  - Place appropriate values in “some” registers as expected by a calling or caller program. More on this later.

Setting up the program stack

- In X86 programs, you must set up the stack frame in your assembly language source code.
- There are three things to do:
  - At the start of a function:
    - Set %rbp to point to the bottom of the current stack frame.
    - Set %rsp to point to the top of the stack (the same address as the stack bottom initially).
  - At the end of a function:
    - Put them back

- The next slide shows a typical way of doing it.

Setting up the stack

Part 1:

pushq %rbp **# Save caller's base pointer**

movq %rsp, %rbp **# Set my base pointer**

Put these two instructions at the beginning of your function before any other statements!

* Notice that, since %rbp equals %rsp, the stack is empty.

* We are now ready to use the stack!

Part 2:

leave **# set caller's stack frame back up**

Put this statement directly before the **ret** instruction of your program.