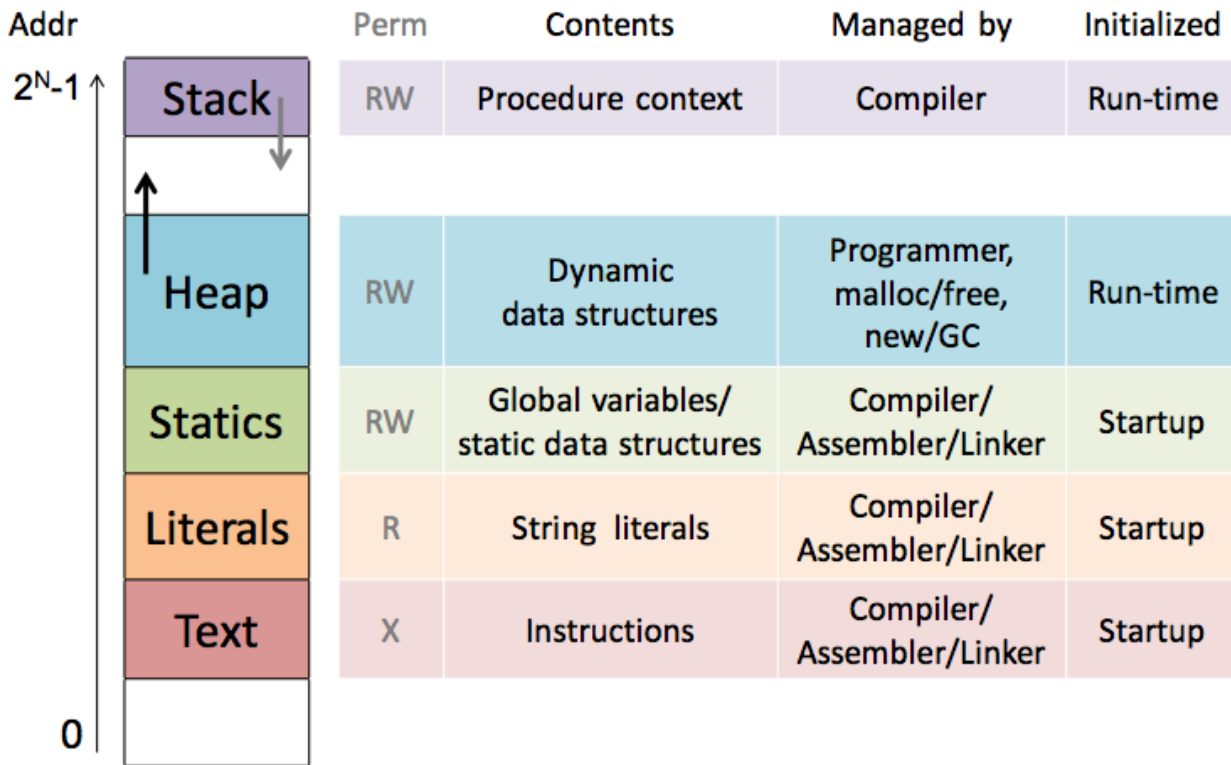


# CS240 Laboratory 8

## Disassembly and Reverse Engineering

### Memory Layout



#### Text Segment

Program instructions can be stored starting at 0x400000 in memory  
Grows up into higher addresses in memory with longer programs.

#### Stack Segment

Top of stack is initially 0x7fffffff ( $2^{47} - 1$ ).

Grows down into lower addresses in memory as stack fills.

When examining X86 code, addresses or numbers used as displacements or pointers/addresses will have values in the range of the text or stack segments.

# Instructions

## Moving Data

*movl Src, Dest*

**Load Effective Address** - compute address or arithmetic expression of the form  $x + k * I$  (does not set the condition flags!)

*leal Src, Dest*

## Arithmetic/Logical operations – 2 operands

*addl Src, Dest*

*subl Src, Dest*

*imull Src, Dest*

*shrl Src, Dest*

*sarl Src, Dest*

*shll Src, Dest*

*sall Src, Dest*

*shrl Src, Dest*

*xorl Src, Dest*

*andl Src, Dest*

*orl Src, Dest*

*mull Src, Dest*

*imull Src, Dest*

*divl Src, Dest*

*idivl Src, Dest*

## **Arithmetic/Logical operations – 1 operand**

*incl Dest*

*decl Dest*

*negl Dest*

*notl Dest*

## **Zero Extend from Byte to Quad Word**

*movzbq Src, Dest*

## **Setting Condition Codes Explicitly – used for control flow**

*cmpl/cmpq Src2, Src1*      sets flags based on value of Src2 – Src1,  
discards result

*testl/testq Src2, Src1*      sets flags based on a & b, discards result

# Operand Types

## Immediate

\$0x400, \$-533

**Register:** 16 general purpose

%rax,%rbx,%rcx,%rdx,%rsi,%rdi,%rbp,%rsp,  
%r8,%r9,%r10,%r11,%r12,%r13,%r14,%r15

**Memory:**  
(%rsp)

## Most General Form:

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

D:        Constant "displacement" value represented in 1, 2, or 4 bytes

Rb:       Base register: Any register

Ri:       Index register: Any except %esp (or %rsp if 64-bit); %ebp unlikely

S:        Scale: 1, 2, 4, or 8 (*why these numbers?*)

**Special Cases:** can use any combination of D, Rb, Ri and S

**(Rb,Ri)                      Mem[Reg[Rb]+Reg[Ri]]        (S=1, D=0)**

**D(Rb,Ri)                      Mem[Reg[Rb]+Reg[Ri]+D]    (S=1)**

**(Rb,Ri,S)                      Mem[Reg[Rb]+S\*Reg[Ri]]    (D=0)**

## Control Flow

Conditional jump instructions in X86 implement the following high-level constructs:

- if (condition) then {...} else {...}
- while (condition) {...}
- do {...} while (condition)
- for (initialization; condition; iterative) {...}

Unconditional jumps are used for high-level constructs such as:

- break
- continue

<b>JX</b>	<b>Condition</b>	<b>Description</b>
<b>jmp</b>	<b>1</b>	Unconditional
<b>je</b>	<b>ZF</b>	Equal / Zero
<b>jne</b>	<b>~ZF</b>	Not Equal / Not Zero
<b>ja</b>	<b>SF</b>	Negative
<b>jns</b>	<b>~SF</b>	Nonnegative
<b>jg</b>	<b>~ (SF^OF) &amp; ~ZF</b>	Greater (Signed)
<b>jge</b>	<b>~ (SF^OF)</b>	Greater or Equal (Signed)
<b>jl</b>	<b>(SF^OF)</b>	Less (Signed)
<b>jle</b>	<b>(SF^OF)   ZF</b>	Less or Equal (Signed)
<b>ja</b>	<b>~CF &amp; ~ZF</b>	Above (unsigned)
<b>jb</b>	<b>CF</b>	Below (unsigned)

## PC-relative Addressing

Jump instructions encode the offset from next instruction to destination PC, instead of the absolute address of the destination (makes it easier to relocate the code)

# Turning C into Machine Code

## C Code

```
void sumstore(long x, long y,  
              long *dest) {  
    long t = x + y;  
    *dest = t;  
}
```

sum.c

**compiler (CS 301)**

gcc -Og -S sum.c

## Generated x86 Assembly Code

Human-readable language close to machine code.

```
sum:  
    addq %rdi,%rsi  
    movq %rsi,(%rdx)  
    retq
```

sum.s

**assembler**

## Object Code

```
01010101100010011110010110  
00101101000101000011000000  
00110100010100001000100010  
01111011000101110111000011
```

sum.o

**linker**

**Executable: sum**

Resolve references between object files,  
libraries, (re)locate data

- X86 instructions can be in different order from C code
- Some C expressions require multiple X86 instructions
- Some X86 instructions can cover multiple C expressions
- Compiler optimization can do some surprising things!
- Local or temporary variables can be stored in registers or on the stack

## Function Calling Conventions

- Arguments for functions are stored in registers, in the following order: arg1 – arg6: `%rdi,%rsi,%rdx,%rcx,%r8,%r9`
- Return value from function always in `%rax`

# Tools

Tools can be used to examine bytes of object code (executable program) and reconstruct (reverse engineer) the assembly source.

**gdb** – disassembles an executable file into the associated assembly language representation, and provides tools for memory and register examination, single step execution, breakpoints, etc.

Object	Disassembled by GDB
0x00400536:	0x0000000000400536 <+0>: add %rdi,%rsi
0x48	0x0000000000400539 <+3>: mov %rsi,(%rdx)
0x01	0x000000000040053c <+6>: retq
0xfe	
0x48	\$ gdb sum
0x89	(gdb) disassemble sumstore
0x32	(disassemble function)
0xc3	(gdb) x/7b sum
	(examine the 13 bytes starting at sum)

## objdump

can also be used to disassemble and display information

`$ objdump -t p`

Prints out the program's symbol table. The symbol table includes the names of all functions and global variables, the names of all the functions the called, and their addresses.



`$ objdump -d p`

## Object Code

0x401040 <sum>:

0x55

0x89

0xe5

0x8b

0x45

0x0c

0x03

0x45

0x08

0x89

0xec

0x5d

0xc3

## Disassembled version

00401040 <\_sum>:

0: 55 push %ebp

1: 89 e5 mov %esp,%ebp

3: 8b 45 0c mov 0xc(%ebp),%eax

6: 03 45 08 add 0x8(%ebp),%eax

9: 89 ec mov %ebp,%esp

b: 5d pop %ebp

c: c3 ret

## strings

`$ strings -t x p`

Displays the printable strings in your program.

## Lab Assignment: Disassembled Version of test\_prime

```
0x400478 <+0>: mov    %rdi,-0x18(%rsp)
0x40047c <+4>: movq   $0x2,-0x8(%rsp)
0x400484 <+12>: jmp    0x4004a9 <test_prime+49>
0x400486 <+14>: mov    -0x18(%rsp),%rax
0x40048a <+18>: mov    %rax,%rdx
0x40048d <+21>: sar   $0x3f,%rdx
0x400491 <+25>: idivq -0x8(%rsp)
0x400495 <+29>: mov    %rdx,%rax
0x400498 <+32>: test   %rax,%rax
0x40049b <+35>: jne   0x4004a4 <test_prime+44>
0x40049d <+37>: mov    $0x0,%eax
0x4004a2 <+42>: jmp    0x4004c6 <test_prime+78>
0x4004a4 <+44>: addq   $0x1,-0x8(%rsp)
0x4004a9 <+49>: mov    -0x18(%rsp),%rax
0x4004ad <+53>: mov    %rax,%rdx
0x4004b0 <+56>: shr   $0x3f,%rdx
0x4004b4 <+60>: lea   (%rdx,%rax,1),%rax
0x4004b8 <+64>: sar   %rax
0x4004bb <+67>: cmp   -0x8(%rsp),%rax
0x4004bf <+71>: jge   0x400486 <test_prime+14>
0x4004c1 <+73>: mov    $0x1,%eax
0x4004c6 <+78>: retq
```