CS240 Laboratory 8
Disassembly and Reverse Engineering

Memory Layout

Addr		Perm	Contents	Managed by	Initialized
2^N-1	Stack	RW	Procedure context	Compiler	Run-time
	Heap	RW	Dynamic data structures	Programmer, malloc/free, new/GC	Run-time
	Statics	RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
	Literals	R	String literals	Compiler/ Assembler/Linker	Startup
	Text	X	Instructions	Compiler/ Assembler/Linker	Startup
0					

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 0x7fffffffffff ($2^{47} - 1$).
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:
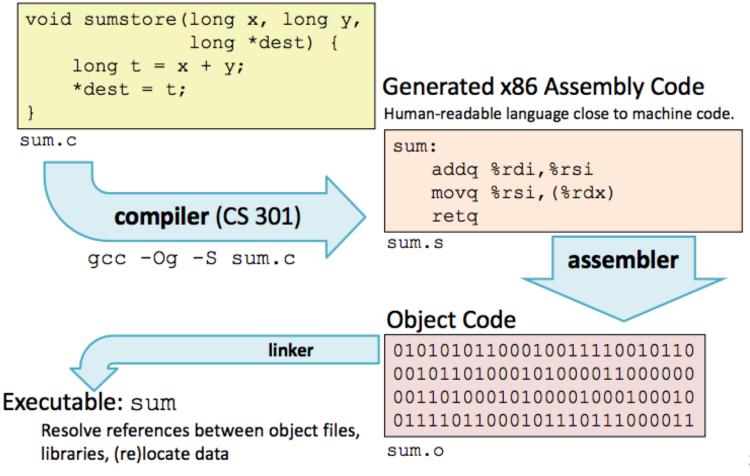
- break
- continue

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~ (SF^OF) &~ZF	Greater (Signed)
jge	~ (SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)	ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	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

```
0101010110001001111001 0110
0010110100010100001 1000000
0011010001010000100 0100010
0111101100010111011 1000011
```
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!
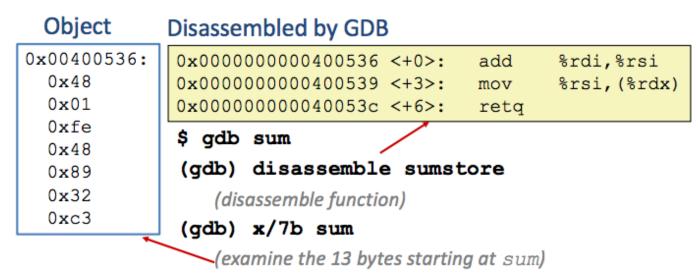- Local or temporary variables can be stored in registers or on the stack

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	**Disassembled version**

0x401040 <sum>:   00401040 <_sum>:

```
0x55            0:   55          push %ebp
0x89            1:   89 e5       mov %esp,%ebp
0xe5            3:   8b 45 0c    mov 0xc(%ebp),%eax
0x8b            6:   03 45 08    add 0x8(%ebp),%eax
0x45            9:   89 ec       mov %ebp,%esp
0x0c            b:   5d pop      %ebp
0x03            c:   c3          ret
0x45
0x08
0x89
0xec
0x5d
0xc3
```

strings

$ *strings –t x p*

Displays the printable strings in your program.