**Memory Layout**

- **Kernel**
  - above 0x7fffffff

- **Stack**
  - below 0x7fffffff
grows down

- **Heap**
  - above Data segment

- **Data segment**
  - statics and literals

- **Text segment**
  - starts at 0x400000
  - addresses below 0x400000 reserved for operating system

**Addr**

<table>
<thead>
<tr>
<th>Perm</th>
<th>Contents</th>
<th>Managed by</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>RW</td>
<td>Procedure context</td>
<td>Compiler</td>
<td>Run-time</td>
</tr>
<tr>
<td>RW</td>
<td>Dynamic data structures</td>
<td>Programmer, malloc/free, new/GC</td>
<td>Run-time</td>
</tr>
<tr>
<td>RW</td>
<td>Global variables/static data structures</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td>R</td>
<td>String literals</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td>X</td>
<td>Instructions</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
</tbody>
</table>

**Instructions**

The size of the data being referenced is often specified with an additional character:

- $b$ (byte)
- $w$ (2 bytes)
- $l$ (4 bytes), or
- $q$ (8 bytes).
Operand Types

**Immediate**  $0x400$, $-533$

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

Some have special purpose: `%rsp` is stack pointer, `%rax` always used to
return value from functions

**Memory**  -0x18(%rsp)

**Most General Form:**

\[
D(Rb,Ri,S) = \text{Mem}[\text{Reg}[Rb] + S*\text{Reg}[Ri] + D]
\]

- **D**: Constant "displacement" value represented in 1, 2, or 4 bytes
- **Rb**: Base register: Any register
- **Ri**: Index register: Any except `%esp` (or `%ebp` 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

\[
\begin{align*}
(Rb,Ri) & : \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] & (S=1, D=0) \\
D(Rb,Ri) & : \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] & (S=1) \\
(Rb,Ri,S) & : \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]] & (D=0)
\end{align*}
\]
Types of Instruction

**Moving Data**

\[ \text{movl} \ Src,\Dest \quad // \text{copy 4 bytes from source to destination} \]

**Arithmetic/Logical operations** – 2 operands

\[
\begin{align*}
\text{add} & \quad \text{Src, Dest} \\
\text{sub} & \quad \text{Src, Dest} \\
\text{imul} & \quad \text{Src, Dest} \\
\text{shr} & \quad \text{Src, Dest} \\
\text{sar} & \quad \text{Src, Dest} \\
\text{shl} & \quad \text{Src, Dest} \\
\text{sal} & \quad \text{Src, Dest} \\
\text{shr} & \quad \text{Src, Dest} \\
\text{xor} & \quad \text{Src, Dest} \\
\text{and} & \quad \text{Src, Dest} \\
\text{or} & \quad \text{Src, Dest} \\
\text{mul} & \quad \text{Src, Dest} \\
\text{imul} & \quad \text{Src, Dest} \\
\text{div} & \quad \text{Src, Dest} \\
\text{idiv} & \quad \text{Src, Dest}
\end{align*}
\]
**Arithmetic/Logical operations** – 1 operand

- `inc Dest`
- `del Dest`
- `neg Dest`
- `not Dest`

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

- `cmp Src2,Src1` sets flags based on value of Src2 – Src1, discards result
- `test Src2,Src1` sets flags based on a & b, discards result

**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
<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>ja</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jna</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>ja</td>
<td>~CF6-ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
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
- If there are more than 6 parameters for a function, the rest of the arguments are stored on the stack before the function is called
- Return value from function is always in %rax

The compiler will use only part of a register if the value stored there will fit in less than 64 bits (8 bytes). This is an optimization that makes instructions a bit shorter.

So, in the code, you may see register names of the following form, all of which refer to %rax:

%rax = 8 byte value
%eax = 4 byte value
%ax = 2 byte value
%al = 1 byte value
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.

<table>
<thead>
<tr>
<th>Object</th>
<th>Disassembled by GDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400536: 0x48 0x01 0xfe 0x48 0x89 0x32 0xc3</td>
<td>0x0000000000400536 &lt;+0&gt;: add %rdi,%rsi 0x0000000000400539 &lt;+3&gt;: mov %rsi,(%rdx) 0x000000000040053c &lt;+6&gt;: retq</td>
</tr>
</tbody>
</table>

$ gdb sum
(gdb) disassemble sumstore
   (disassemble function)
(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

<table>
<thead>
<tr>
<th><strong>Object Code</strong></th>
<th><strong>Disassembled version</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040 &lt;sum&gt;:</td>
<td>00401040 &lt;_sum&gt;:</td>
</tr>
<tr>
<td>0x55</td>
<td>0:  55   push %ebp</td>
</tr>
<tr>
<td>0x89</td>
<td>1:  89 e5 mov %esp,%ebp</td>
</tr>
<tr>
<td>0xe5</td>
<td>3:  8b 45 0c mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>0x8b</td>
<td>6:  03 45 08 add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>0x45</td>
<td>9:  89 ec mov %ebp,%esp</td>
</tr>
<tr>
<td>0x0c</td>
<td>b:  5d pop %ebp</td>
</tr>
<tr>
<td>0x03</td>
<td>c:  c3 ret</td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xec</td>
<td></td>
</tr>
<tr>
<td>0x5d</td>
<td></td>
</tr>
<tr>
<td>0xc3</td>
<td></td>
</tr>
</tbody>
</table>

**strings**

$ strings –t x p

Displays the printable strings in your program.