### Memory Layout

#### Kernel
- Above 0x7fffffff

#### Stack
- Below 0xffffffff
- Grows down

#### Heap
- Above Data segment

#### Data segment
- Statics and literals

#### Text segment
- Starts at 0x400000

<table>
<thead>
<tr>
<th>Address</th>
<th>Permissions</th>
<th>Contents</th>
<th>Managed by</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>2ᴺ⁻¹</td>
<td>RW</td>
<td>Procedure context</td>
<td>Compiler</td>
<td>Run-time</td>
</tr>
<tr>
<td></td>
<td>RW</td>
<td>Dynamic data structures</td>
<td>Programmer, malloc/free, new/GC</td>
<td>Run-time</td>
</tr>
<tr>
<td></td>
<td>RW</td>
<td>Global variables/static data structures</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>String literals</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>Instructions</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
</tbody>
</table>
Instructions

**Moving Data**

\[ \text{mov } \text{Src, Dest} \]

Note: 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).

**Arithmetic/Logical operations** – 2 operands

- \( \text{add } \text{Src, Dest} \)
- \( \text{sub } \text{Src, Dest} \)
- \( \text{imul } \text{Src, Dest} \)
- \( \text{shr } \text{Src, Dest} \)
- \( \text{sar } \text{Src, Dest} \)
- \( \text{shl } \text{Src, Dest} \)
- \( \text{sal } \text{Src, Dest} \)
- \( \text{shr } \text{Src, Dest} \)
- \( \text{xor } \text{Src, Dest} \)
- \( \text{and } \text{Src, Dest} \)
- \( \text{or } \text{Src, Dest} \)
mul \textit{Src,Dest}
imul \textit{Src,Dest}
div \textit{Src,Dest}
idiv \textit{Src,Dest}

\textbf{Arithmetic/Logical operations} – 1 operand

\textit{inc} \textit{Dest}
\textit{del} \textit{Dest}
\textit{neg} \textit{Dest}
\textit{not} \textit{Dest}

\textbf{Setting Condition Codes Explicitly} – used for control flow

\textit{cmp} \textit{Src2,Src1} \quad \text{sets flags based on value of Src2 – Src1, discards result}
\textit{test} \textit{Src2,Src1} \quad \text{sets flags based on a \& b, discards result}

\textbf{Operand Types}

\textbf{Immediate} \quad $0x400, \$-533$

\textbf{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:

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

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

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

<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>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>¬CF &amp; ¬ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

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)
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
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>0x000000000000400536 &lt;+0&gt;: add %rdi,%rsi 0x0000000000400539 &lt;+3&gt;: mov %rsi,(%rdx) 0x0000000000040053c &lt;+6&gt;: retq</td>
</tr>
</tbody>
</table>

**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
- 0x0c
- 0x45
- 0x08
- 0x89
- 0xec
- 0x5d

### 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.