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

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

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

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

Arithmetic/Logical operations – 2 operands

\[ \text{addl \ Src,\ Dest} \]
\[ \text{subl \ Src,\ Dest} \]
\[ \text{imull \ Src,\ Dest} \]
\[ \text{shrl \ Src,\ Dest} \]
\[ \text{sarh \ Src,\ Dest} \]
\[ \text{shll \ Src,\ Dest} \]
\[ \text{sall \ Src,\ Dest} \]
\[ \text{shrl \ Src,\ Dest} \]
\[ \text{xorl \ Src,\ Dest} \]
\[ \text{andl \ Src,\ Dest} \]
\[ \text{orl \ Src,\ Dest} \]

\[ \text{mull \ Src,\ Dest} \]
\[ \text{imull \ Src,\ Dest} \]
\[ \text{divl \ Src,\ Dest} \]
\[ \text{idivl \ Src,\ Dest} \]
Arithmetic/Logical operations – 1 operand

incl Dest
dcl 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:

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

- **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

- (Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]] \quad (S=1, D=0)
- D(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D] \quad (S=1)
- (Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + S \times \text{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>jns</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>

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)
• 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.

<table>
<thead>
<tr>
<th>Object</th>
<th>Disassembled by GDB</th>
</tr>
</thead>
</table>
| 0x00400536: 0x48 0x01 0xfe 0x48 0x89 0x32 0xc3 | 0x000000000400536 <+0>: add %rdi,%rsi  
0x000000000400539 <+3>: mov %rsi,(%rdx)  
0x00000000040053c <+6>: retq |

**objdump**
Can also be used to disassemble and display information

```
$ gdb sum
(gdb) disassemble sumstore
   (disassemble function)
(gdb) x/7b sum
   (examine the 13 bytes starting at sum)
```

**$ 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.
```plaintext
$ objdump -d p

Object Code
0x401040 <sum>:
  0x55
  0x89
  0xe5
  0x8b 0x0c
  0x03 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.
```