Under The Covers: The Secret Life of Software (https://github.com/jappavoo/UndertheCovers)

INTEL Registers: Names, sizes and bit positions

63	31	16	7 0
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b/r8l
r9	r9d	r9w	r9b/r9l
r10	r10d	r10w	r10b/r10l
r11	r11d	r11w	r11b/r11l
r12	r12d	r12w	r12b/r12l
r13	r13d	r13w	r13b/r13l
r14	r14d	r14w	r14b/r14l
r15	r15d	r15w	r15b/r15l

INTEL Address modes: The ways you can specify the source or destinations for an instruction : Allowed combinations are:

Immediate: the source value is stored in the instruction
eg. ADD EAX, 14 # Add 14 into the 32 bit EAX register
MOV RAX, 0xdeadbeef # set RAX

Register to register:
eg. ADD R8B, AL # add 8 bit AL value to R8B register
MOV RAX, R8 # copy the value from R8 into RAX

Memory operands:
[BaseReg + scale * IndexReg + Displacement]
Where BaseReg and IndexReg can be any general purpose register
scale is a numeric value of 1,2,4,8
Displacement is 8, 16 or 32 bit value. Often this will be a symbolic label
MOV RAX, QWORD PTR [RBX + 8*RDI + XARRAY]
Note: In general you can omit various terms to meet your needs

When doing moves only one operand can be a memory operand.

INTEL EFLAGS Single bit flags that we are concerned with

- ZF Zero Flag set if result was zero
- SF Sign Flag set if result was negative (most significant bit set)
- CF Carry Flag set if operation generates a carry or borrow
- OF Overflow Flag set if overflow on signed operation

Typical Intel GNU Assembly Instruction formats:
mnemonic : mnemonic with no operands
eg. int3
mnemonic <dst> : mnemonic with one destination operand:
eg. inc RAX. # RAX=RAX+1
mnemonic <dst>, <src> : mnemonic with one destination and one src
eg. add RAX, RBX # RAX = RAX + RBX
mnemonic <dst>, <srcA>, <srcB> : mnemonic with one destination and two srcs
eg. imul rax, rbx, 42 # rax = rbx * 42

Common Intel mnemonics (instructions):

Data Transfer:
MOV : move to/from locations : mov dst, src : dst = src;
MOVZX/MOVSX: move smaller src to larger dst zero/sign extending respectively
CMOVcc: conditional mov : cmov<cc> dst, src: if cc then dst=src else do nothing : cmovg dst, src : dst = src if greater. See flow control instructions for example conditions

ALU:
ADD <dst>, <src>: add integers (signed or unsigned) : dst = dst + src;
SUB <dst>, <src> : subtract integers (signed or unsigned) : dst = dst - src;
IMUL <dst>,<srcA>,<srcB>: multiplies integers (signed): dst = srcA * srcB;
INC. : Increment : inc dst : dst = dst + 1
DEC. : Decrement : dec dst : dst = dst - 1
AND <dst>, <src> : Bitwise boolean and : dst = dst & src
OR <dst>, <src> : Bitwise boolean or : dst = dst | src
XOR <dst>, <src> : Bitwise boolean xor : dst = dst ^ src
NOT <dst>. : Bitwise boolean not : dst = ~dst
SHR <dst>, imm. : logical shift right : dst = dst >> imm (zero extends)
SAR <dst>, imm : arithmetic shift right : dst = dst >> imm (sign extends)
SHL <dst>, imm : logical shift left: dst = dst << imm (zero fill)
SAL <dst>, imm : arithmetic shift left: dst = dst << imm (zero fill same as SHL)
CMP <dst>, <src> : set eflags based on subtraction: dst - src
TEST <dst>, <src> : set eflags based on bitwise and of dst & src

Control flow:
JMP <dst> : Unconditional jump : jmp <dst> : pc = dst : dst is usually a label
eg. jmp loop_begin
but can also be indirect:
1) a register (which contains the address to jump to)
jmp rax
2) ptr to a memory location which contains the address to jump to
jmp qword ptr [myjmptable]
JE <dst> : jump if equal : jmp if zero eflag is set (1) same as JZ
JNE <dst> : jump if not equal: jmp if zero eflag is NOT set (0)
JZ <dst> : jump if zero : same as JE
JNZ <dst> : jump not zero: same as JNE
JG <dst> : jump if greater (signed)
JGE <dst> : jump if greater or equal (signed)
JL <dst> : jump if less (signed)
JLE <dst> : jump if less or equal (signed)
JA <dst> : jump if above (unsigned)
JAE <dst> : jump if above or equal (unsigned)
JB <dst> : jump if below (unsigned)
JBE <dst> : jump if below or equal (unsigned)

Stack:
PUSH : stack push : push src : rsp=rsp - len(src); M[rsp] = src;
POP : stack pop : pop dst : dst = M[rsp]; rsp = rsp + len(src);
CALL/RET : call and return from subroutine : call pushes address of the following instruction on the stack and then sets pc to the specified target address. ret pops the top value from the stack and sets the pc to this address

Misc:
NOP : no operation
INT3 : hand control back to debugger
SYSCALL : request operating system call routine

Byte Vector Sizes and Names
1 Byte : INTEL BYTE : GAS directive .byte. : C unsigned char and char (signed)
2 Bytes : INTEL WORD : GAS directive .short : C unsigned short and short (signed)
4 Bytes : INTEL DWORD : GAS directive .long : C unsigned int and int (signed)
8 Bytes : INTEL QWORD : GAS directive .quad : C unsigned long long and long long (signed)
NOTE: On INTEL 64 bit machines all pointer types (char *, short *, int *, long long * and void *) are 8 bytes in size

Two's Complement facts:
Value of bit vector $X_w = [b_{w-1} \ldots b_0]$ is
$-2^{w-1} b_{w-1} + \sum_{i=0}^{w-2} 2^i b^i$
Negation of a value: $-x = x + 1$
$-1 = [1.\,.\,.1]$
min = $-2^{w-1} = [10.\,.\,.0]$, max=$2^{w-1} - 1 = [01.\,.\,.1]$

```
GDB Commands:

file <binary>     : opens a new binary replacing the current one eg. file empty
run               : creates a process from the current open binary and initiates the cpu's
                    execution within it
b <symbol>        : sets a breakpoint to stop execution when the PC equals the address of
                    symbol eg. b _start
c                 : continue execution from current PC address until execution terminates
                    or a break point is hit
si                : single step a cpu instruction eg. unfreeze the cpu so it can do one
                    execution loop
p /x $<REG>       : print the current value of the specified register in hex
x/<n>bx <address> : print/examine n memory bytes sized values start at the specified address
                    in hex notation
x/<n>hx <address> : same as above but n memory 2-byte sized values
x/<n>wx <address> : same as above but n memory 4-byte sized values
x/<n>gx <address> : same as above but n memory 8-byte sized values
set $<REG>=<value>: sets the value of the specified registers.  Value can be specified in
                    notations by using the right prefix eg. 0x for hex, 0b for binary.  The
                    default is signed two-complement integers.
set {CType}(address)=<value>: set in memory at the address specified.  CType is one of the
                              C programming type names for bytes sized quantities.
                              See notes below for a list

NOTES:
      1. When using x to display multi-bytes sized (eg. x/1hx <addr>) gdb will reorder
      to account for endianness of the computer.  For example if the bytes at address _start,
      on a little endian computer, are 0xFA 0x10 and we use the command x/1hx & _start gdb
      will display something like
      (gdb) x/1xh &_start
      0x401000 <_start>:      0x10FA
      This is true for all the other multi-bytes sizes (h,w,g)

      2. For the p and x command the following Format letters can be used
      o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address),
      i(instruction), c(char), s(string) and z(hex, zero padded on the left).

      3. CType names: "unsigned char" : 1 byte, "unsigned short" : 2 byte,
                      "unsigned int" : 4 byte, "unsigned long long": 8 byte
```

INTEL C Linux Calling Conventions :

Defines how registers should be used by caller and callee code. It also defines how arguments and the return value for a C function should be assigned to registers and the stack. The First 6 integer arguments are passed in registers as follows
Argument 0 : rdi
Argument 1 : rsi
Argument 2 : rdx
Argument 3 : rcx
Argument 4 : r8
Argument 5 : r9
Return value : rax
If more than 6 arguments are required the remainder are pushed on the stack in reverse order (last pushed first). A functions return value must be place in rax.

The function code (callee) is free to overwrite any of the 7 above registers along with r10 and r11. Calling code (caller) needs to save and restore these registers if it wants to rely on their values. Thus they are called volatile and caller saved. The values of the remaining general purpose registers (rbx, rsp, rbp, r12-r15) must not be affected by a function as such they are called non-volatile and callee saved. Eg. if a function writes them it must restore their value before returning to the caller

```
ASCII Hex Table (Hex Character)

    00 nul   01 soh   02 stx   03 etx   04 eot   05 enq   06 ack   07 bel
    08 bs    09 ht    0a nl    0b vt    0c np    0d cr    0e so    0f si
    10 dle   11 dc1   12 dc2   13 dc3   14 dc4   15 nak   16 syn   17 etb
    18 can   19 em    1a sub   1b esc   1c fs    1d gs    1e rs    1f us
    20 sp    21  !    22  "    23  #    24  $    25  %    26  &    27  '
    28  (    29  )    2a  *    2b  +    2c  ,    2d  -    2e  .    2f  /
    30  0    31  1    32  2    33  3    34  4    35  5    36  6    37  7
    38  8    39  9    3a  :    3b  ;    3c  <    3d  =    3e  >    3f  ?
    40  @    41  A    42  B    43  C    44  D    45  E    46  F    47  G
    48  H    49  I    4a  J    4b  K    4c  L    4d  M    4e  N    4f  O
    50  P    51  Q    52  R    53  S    54  T    55  U    56  V    57  W
    58  X    59  Y    5a  Z    5b  [    5c  \    5d  ]    5e  ^    5f  _
    60  `    61  a    62  b    63  c    64  d    65  e    66  f    67  g
    68  h    69  i    6a  j    6b  k    6c  l    6d  m    6e  n    6f  o
    70  p    71  q    72  r    73  s    74  t    75  u    76  v    77  w
    78  x    79  y    7a  z    7b  {    7c  |    7d  }    7e  ~    7f del
```

Linux X86 64 Bit Alignment Rules: (type - alignment in bytes)
char - none, short - 2 , int - 4, long long - 8, Same for unsigned integers. Pointers - 8. long double - 16. Arrays aligned to alignment of element type. Structures aligned to max alignment of its fields, padding added in between fields as need and at end to ensure field and overall alignment.