
U.S. ARMY CYBER SCHOOL
DEFEND
ATTACK
EXPLOIT

x86 Assembly Guide

The stack

Memory stacks are locations also know as linear data structures that are used to place data into memory of the computer. Data is inserted into the stack in a linear order using Last in first out (LIFO), which is accomplished through operations. Memory that is allocated to a new process recieves a section of memory, which depending on the process could be in kernal or user space allocations. When data is added to the stack memory address goes down, where as if data is removed memory address goes up.

Instructions are used to manupilate, upload, download, or perform computations data into the memory locations. There are numerious types of architectures, common instructions are typically utilized but there are differences between architecutres through the use of specific instructions know as extensions.

For the purpose of this guide Intel x86_64 will be the basis.

As instructions are being called the data is being placed into registers inside the stack. Register is a small amount of memory in the stack. Think you have a big box that 10 other boxes of smaller size can be placed into, the smaller boxes are a register.

General Registers

X86_64 uses 16 general registers, however it can access smaller bit registers for 32, 16, and 8 bits.

- Accumulator register (AX) used for arithmetic operations
- Counter register (CX). Used ifor shift instructions and loops.
- Data register (DX). Used in arithmetic operations and I/O operations.
- Base register (BX). Used as a pointer to data
- Stack Pointer register (SP). Pointer for the top of the stack.
- Stack Base Pointer register (BP). pointer for base of the stack.
- Source Index register (SI). pointer to a source in stream operations.
- Destination Index register (DI). pointer to a destination in stream operations.

Understing difference registers by bits

To distingush the differences lets discuss the 64 register rax.

- Before 64 bit systems we had 32 bits, so the register was eax.
- Before 32 bit we had 16 bit and the register was ax.
- Than at 8 bit the register was al

The r8 thru r15 registers extensions happen from the end of the register. For example:

- r8 is 64 bit
- r8d is 32 bit
- r8w is 16 bit
- r8b is 8 bit.

As computing technology allowed for faster computing, registers where extended to accommindate those increases. However by extending the register the smaller registers are still able to be utilized, since backward capability is rather important so companies can continue to use older programs.

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
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
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

Reference: *https://stackoverflow.com/questions/20637569/assembly-registers-in-64-bit-architecture*

Byte position

bit Registers that contain a "H" or "L" identifies position of the bit as either high or low. In the chart above we see, for example, the al register in the lower 8 bits column. Should we see ah than we know that the register bits are set in the high postion.

Low order example:

```
00001111
```

Control Registers

Control registers which determine processor operating mode and executing task settings. These registers are used in kernel development and can be identified as %CR0 thru %CR15. For this course we will not work at the kernel but it important to know that control registers exist in case they are ever encounter in the future.

Segment Registers

These registers are utilized to logically divide memory into segments. Each segment will have its own base memory address, this allows processors to fetch and execute data quickly. Segment

registers are available in 16 values and can be set by general register or special instructions.

- Code segment register (CS): is used for addressing memory location in the code segment of the memory, where the executable program is stored.
- Data segment register (DS): points to the data segment of the memory where the data is stored.
- Extra Segment Register (ES): also refers to a segment in the memory which is another data segment in the memory.
- Stack Segment Register (SS): is used for addressing stack segment of the memory. The stack segment is that segment of memory which is used to store stack data.

Flags

Flags also known as condition codes represent the state of the processor. These flags are used to control the action when conditions are met which typically is used in conjunction with jump instruction to determine what the program will do next. To note flags also utilize extensions to identify how many bits the flag is.

- RFLAGS : 64 bit
- EFLAGS : 32 bit
- FLAGS : 16 bit

Bit	*Name*	*Symbol*
0	Carry flag	**cf**
2	Parity flag	**pf**
4	Auxiliary carry flag	**af**
6	Zero flag	**zf**
7	Sign flag	**sf**
8	Trace flag	**tf**
9	Interrupt flag	**if**
10	Direction flag	**df**
11	Overflow flag	**of**

Reference: *https://www.shsu.edu/~csc_tjm/fall2003/cs272/flags.html*

Common flags

- CF - carry flag : Set on high-order bit carry or borrow; cleared otherwise
- PF - parity flag : Set if low-order eight bits of result contain an even number of "1" bits; cleared otherwise
- ZF - zero flags : Set if result is zero; cleared otherwise
- SF - sign flag : Set equal to high-order bit of result (0 if positive 1 if negative)
- OF - overflow flag : Set if result is too large a positive number or too small a negative number (excluding sign bit) to fit in destination operand; cleared otherwise

Instructions

Instructions can be thought of as "commands" that the program calls to do something. This could be to push data into the memory stack or compare data to generate flags to name a few. The amount of possible instructions one can use is numerious, the 2019 Intel® 64 and IA-32 Architectures Software Developer's Manual shows over 1,000 instructions.

Common/basic instructions

Data movement

There are two syntaxes used for x86 ASM: Intel and AT&T. Intel is what most disassemblers use when disassembling a program and is what we teach.

Intel syntax is read from right to left. Data that is used located at the right, the destination is in the middle, and the instruction is to the left.

Instruction	Register	Data
mov	rbx,	10

- mov : moves data into destination

```
mov eax, ebx  # copy value in ebx into eax
```

- push : places data onto the stack

```
push[var] # push var into the stack
```

- pop : pulls data off the stack

```
pop [ebx] # pull top element of the stack into memory at EBX
```

- lea : loads data into identified address

```
lea eax,[var] # var is placed into EAX
```

Logical

- add : add items together
- sub : subtract items
- inc, dec : increment, decrement by 1
- imul : multiplication
- and, or, xor : logical operations

Control flow

- jmp : jump. move pointer to a memory location
- j<condition> : is a jump based upon a condition being meet
- cmp : compare.
- call : call a subroutine, which pushes current memory location onto the stack. Unlike a jmp call will save the location of memory which is returned to once the subroutine finishes.
- ret : return, pops the memory location of the stack and then execuates a unconditional jmp to subroutine location.

Jumps and flags

When j<condition> instruction are used it will check the current flags depending on the results of operation. For example if we had a JE (jump if equal) and the operation is true than the zero flag (ZF). If that flag is set than our code would conduct the jump to location in stack. If the condition is not met than program would move to the next instruction on the stack.

Instruction	Description	signed-ness	Flags	short jump opcodes	near jump opcodes
JO	Jump if overflow		OF = 1	70	0F 80
JNO	Jump if not overflow		OF = 0	71	0F 81
JS	Jump if sign		SF = 1	78	0F 88
JNS	Jump if not sign		SF = 0	79	0F 89
JE **JZ**	Jump if equal Jump if zero		ZF = 1	74	0F 84
JNE **JNZ**	Jump if not equal Jump if not zero		ZF = 0	75	0F 85
JB **JNAE** **JC**	Jump if below Jump if not above or equal Jump if carry	unsigned	CF = 1	72	0F 82
JNB **JAE** **JNC**	Jump if not below Jump if above or equal Jump if not carry	unsigned	CF = 0	73	0F 83
JBE **JNA**	Jump if below or equal Jump if not above	unsigned	CF = 1 or ZF = 1	76	0F 86
JA **JNBE**	Jump if above Jump if not below or equal	unsigned	CF = 0 and ZF = 0	77	0F 87
JL **JNGE**	Jump if less Jump if not greater or equal	signed	SF <> OF	7C	0F 8C
JGE **JNL**	Jump if greater or equal Jump if not less	signed	SF = OF	7D	0F 8D
JLE **JNG**	Jump if less or equal Jump if not greater	signed	ZF = 1 or SF <> OF	7E	0F 8E
JG **JNLE**	Jump if greater Jump if not less or equal	signed	ZF = 0 and SF = OF	7F	0F 8F
JP **JPE**	Jump if parity Jump if parity even		PF = 1	7A	0F 8A
JNP **JPO**	Jump if not parity Jump if parity odd		PF = 0	7B	0F 8B
JCXZ **JECXZ**	Jump if %CX register is 0 Jump if %ECX register is 0		%CX = 0 %ECX = 0	E3	

Reference: *http://www.unixwiz.net/techtips/x86-jumps.html*

Putting it together

- Example 1

```
main:
    mov rax, 16     //16 moved into rax
```

```
    push rax        //push value of rax (16) onto stack. RSP is pushed up 8 bytes (64
bits)
    jmp mem2        //jmp to mem2 memory location

mem1:           //Subroutine
    mov rax, 0      //move 0 (error free) exit code to rax
    ret             //return out of code

mem2:               //Subroutine
    pop r8          //pop value on the stack (16) into r8. RSP falls 8 bytes
    cmp rax, r8     //compare rax register value (16) to r8 register value (16)
    je mem1         //jump if comparison has zero bit set to mem1
```

- Example 2

```
main:
    mov rcx, 25     //store the value 25 in rcx register
    mov rbx, 62     //store the value 62 in rbx register
    jmp mem1        //jumps to mem1 location

mem1:               //Subroutine
    sub rbx, 40     //subtract 40 from rbx
    mov rsi, rbx    //copy rbx value to rsi
    cmp rcx, rsi    //compare the values in rcx and rsi
    jmple mem2      //jumps to mem2 location if value is less than or equal

mem2:               //Subroutine
    mov rax, 0      //store 0 in rax
    ret             //return out of code
```