# Accessing Information: Recap

## Registers

An x86-64 central processing unit (CPU) contains a set of 16 general-purpose registers storing 64-bit values.

Instructions can operate on data of different sizes stored in the low-order bytes of the 16 registers. Byte-level operations access the least significant byte, 16-bit operations access the least significant 2 bytes, and 32-bit operations access the least significant 4 bytes.

Different registers serve different roles in typical programs. Most unique among them is the stack pointer used to indicate the end position in the run-time stack. A set of standard programming conventions governs how the registers are to be used for managing the stack, passing function arguments, returning values from functions, and storing local (i.e., variables local to a function) and temporary data.

The following table lists these 16 registers, their names depending on the expected size of the data operands, and their roles in the standard programming convention used in this class.

| Byte (8 bits) | Word (16 bits) | Long (32 bits) | Quad (64 bits) | Role |
|---|---|---|---|---|
| `%al` | `%ax` | `%eax` | `%rax` | Return value |
| `%bl` | `%bx` | `%ebx` | `%rbx` | Callee saved |
| `%cl` | `%cx` | `%ecx` | `%rcx` | 4th argument |
| `%dl` | `%dx` | `%edx` | `%rdx` | 3rd argument |
| `%sil` | `%si` | `%esi` | `%rsi` | 2nd argument |
| `%dil` | `%di` | `%edi` | `%rdi` | 1st argument |
| `%bpl` | `%bp` | `%ebp` | `%rbp` | Callee saved |
| `%spl` | `%sp` | `%esp` | `%rsp` | Stack pointer |
| `%r8b` | `%r8w` | `%r8d` | `%r8` | 5th argument |

| | | | | |
|---|---|---|---|---|
| %r9b | %r9w | %r9d | %r9 | 6th argument |
| %r10b | %r10w | %r10d | %r10 | Caller saved |
| %r11b | %r11w | %r11d | %r11 | Caller saved |
| %r12b | %r12w | %r12d | %r12 | Callee saved |
| %r13b | %r13w | %r13d | %r13 | Callee saved |
| %r14b | %r14w | %r14d | %r14 | Callee saved |
| %r15b | %r15w | %r15d | %r15 | Callee saved |

Caller-saved registers are used to hold temporary values that need not be preserved across function calls. For this reason, it is the caller's responsibility to save the content of these registers (e.g., by pushing these registers onto the stack) if it needs said content after a function call.

Callee-saved registers are used to hold longer-lived values that should be preserved across function calls. When the caller makes a function call, it can expect that those registers will hold the same values after the function returns. For this reason, it is the callee's responsibility to save the content of these registers (e.g., by pushing these registers onto the stack) and to restore them before returning to the caller.

## Operand Specifiers

Most instructions have one or more operands specifying the source values to use in performing an operation and the destination into which to place the result. x86-64 supports three operand types.

| **Type** | **Form** | **Operand value** | **Name** |
|---|---|---|---|
| Immediate | $*Imm* | *Imm* | Immediate |
| Register | $r_a$ | $R[r_a]$ | Register |
| Memory | *Imm*($r_b$, $r_i$, *s*) | M[*Imm* + $R[r_b]$ + $R[r_i]$ · *s*] | |

With the Memory type, at least one of *Imm*, $r_b$, or $r_i$ must be provided; if not specified the associated quantities (*Imm*, $R[r_b]$, $R[r_i]$ are assigned value 0). This gives rise to the following valid combinations and names.

| Form | Operand value | Name |
|---|---|---|

| | | |
|---|---|---|
| *Imm* | M[*Imm*] | Absolute |
| ($r_b$) | M[R[$r_b$]] | Indirect |
| *Imm*($r_b$) | M[*Imm* + R[$r_b$]] | Base + displacement |
| ($r_b$, $r_i$) | M[R[$r_b$] + R[$r_i$]] | Indexed |
| *Imm*($r_b$, $r_i$) | M[*Imm* + R[$r_b$] + R[$r_i$]] | Indexed |
| (, $r_i$, *s*) | M[R[$r_i$] · *s*] | Scaled indexed |
| *Imm*(, $r_i$, *s*) | M[*Imm* + R[$r_i$] · *s*] | Scaled indexed |
| ($r_b$, $r_i$, *s*) | M[R[$r_b$] + R[$r_i$] · *s*] | Scaled indexed |
| *Imm*($r_b$, $r_i$, *s*) | M[*Imm* + R[$r_b$] + R[$r_i$] · *s*] | Scaled indexed |

The scaling factor *s*, if specified, must be 1, 2, 4, or 8.

## Data Movement Instructions

There are four simple move instructions: *movb*, *movw*, *movl*, and *movq*. They primarily differ in the amount of data that they transfer: 1, 2, 4, and 8 bytes, respectively. The source can be an Immediate, a Register, or Memory. The destination can be a Register or Memory. The sole restriction is that both cannot be Memory. When specifying a Register as a source or destination, its size designation (e.g., *%al*, *%ax*, *%eax*, or *%rax*) must match the type of the move instruction (i.e., match the suffix *b*, *w*, *l*, or *q* of the instruction). In most cases, the move instruction only updates the specific bytes of the register or memory destination. The only exception is when the destination is a 32-bit register; in that case the move instruction will also set the high-order 4 bytes of the register to 0.

The *movq* instruction can only have Immediate source operands that can be represented as 32-bit two's complement numbers. The value is then signed extended to produce the 64-bit value for the destination. The *movabsq* instruction can have an arbitrary Immediate value as its source operand and can only have a Register as a destination.

There are two groups of move instructions that deal with moving a value from a smaller size source to a larger size destination.

| Instruction | Effect | Description |
|---|---|---|
| `movzbw` | $R_{16} \leftarrow \text{ZeroExtend}(S_8)$ | Move zero-extended byte to word |
| `movzbl` | $R_{32} \leftarrow \text{ZeroExtend}(S_8)$ | Move zero-extended byte to double word |
| `movzbq` | $R_{64} \leftarrow \text{ZeroExtend}(S_8)$ | Move zero-extended byte to quad word |
| `movzwl` | $R_{32} \leftarrow \text{ZeroExtend}(S_{16})$ | Move zero-extended word to double word |

| | | |
|---|---|---|
| `movzwq` | $R_{64} \leftarrow \text{ZeroExtend}(S_{16})$ | Move zero-extended double word to quad word |
| `movsbw` | $R_{16} \leftarrow \text{SignExtend}(S_8)$ | Move sign-extended byte to word |
| `movsbl` | $R_{32} \leftarrow \text{SignExtend}(S_8)$ | Move sign-extended byte to double word |
| `movsbq` | $R_{64} \leftarrow \text{SignExtend}(S_8)$ | Move sign-extended byte to quad word |
| `movswl` | $R_{32} \leftarrow \text{SignExtend}(S_{16})$ | Move sign-extended word to double word |
| `movswq` | $R_{64} \leftarrow \text{SignExtend}(S_{16})$ | Move sign-extended word to quad word |
| `movslq` | $R_{64} \leftarrow \text{SignExtend}(S_{32})$ | Move sign-extended double word to quad word |
| `cltq` | *%rax* ← SignExtend(*%eax*) | Move sign-extended %eax to %rax |

The source operand must be either a Register or Memory; destination must be a Register. When specifying a Register as a source, its size designation must be appropriate for the instruction (i.e., must match the second to last suffix letter). Likewise the size designation of the destination Register must match the last suffix letter. In most cases the instruction performs the expected operation. The exception is when the destination is a 32-bit register; in that case will also set the high-order 4 bytes of the register to 0 (even in the sign-extending cases!).

The *cltq* instruction has no operands; it always uses *%eax* as its source and *%rax* as its destination. It performs the same operation as *movslq %eax, %rax* but it has a more compact encoding.

## Pushing and Popping Stack Data

The final two data movement operations push data onto and pop data from the program stack.

| Instruction | Effect | Description |
|---|---|---|
| `pushq S` | R[*%rsp*] ← R[*%rsp*] − 8<br>M[R[*%rsp*]] ← *S* | Push quad word |
| `popq D` | *D* ← M[R[*%rsp*]]<br>R[*%rsp*] ← R[*%rsp*] + 8 | Pop quad word |

The source operand for the *pushq* instruction can be an Immediate, a Register, or Memory; while the destination operand for the *popq* instruction can be a Register or Memory.