3_4 Accessing Information

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

- Details:
  - The original 8086 had eight 16-bit registers `%ax` through `%bp`.
  - With the extension to IA32, these registers were expanded to 32-bit registers, labeled `%eax` through `%ebp`.
  - In the extension to x86-64, the original eight registers were expanded to 64 bits, labeled `%rax` through `%rbp`.
    - 8 new registers were added, and these were given labels according to a new naming convention: `%r8` through `%r15`.
- When these instructions have registers as destinations, two conventions arise for what happens to the remaining bytes in the register for instructions that generate less than

8 bytes:

- Those that generate 1- or 2-byte quantities leave the remaining bytes unchanged.
- Those that generate 4- byte quantities set the upper 4 bytes of the register to zero (expansion from IA32 to x86-64).

- Different registers serve different roles:
  - `%rsp` - the stack pointer used to indicate the end position in the run-time stack.

3.4.1 Operand Specifiers

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$\mathtt{r}_a$	$\mathtt{R}[\mathtt{r}_a]$	Register
Memory	Imm	$\mathtt{M}[Imm]$	Absolute
Memory	$(\mathtt{r}_a)$	$\mathtt{M}[\mathtt{R}[\mathtt{r}_a]]$	Indirect
Memory	$Imm(\mathtt{r}_b)$	$\mathtt{M}[Imm + \mathtt{R}[\mathtt{r}_b]]$	Base + displacement
Memory	$(\mathtt{r}_b,\mathtt{r}_i)$	$\mathtt{M}[\mathtt{R}[\mathtt{r}_b] + \mathtt{R}[\mathtt{r}_i]]$	Indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i)$	$\mathtt{M}[Imm + \mathtt{R}[\mathtt{r}_b] + \mathtt{R}[\mathtt{r}_i]]$	Indexed
Memory	$(,\mathtt{r}_i,s)$	$\mathtt{M}[\mathtt{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(,\mathtt{r}_i,s)$	$\mathtt{M}[Imm + \mathtt{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$(\mathtt{r}_b,\mathtt{r}_i,s)$	$\mathtt{M}[\mathtt{R}[\mathtt{r}_b] + \mathtt{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i,s)$	$\mathtt{M}[Imm + \mathtt{R}[\mathtt{r}_b] + \mathtt{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed

- Most instructions have one or more operands specifying the source values to use in performing an operation and the destination location into which to place the result.
  - Source values can be given as constants or read from registers or memory.
  - Results can be stored in either registers or memory.
- 3 types of operand:
  - immediate - constant values;
    - in ATT format - `'$'` is the suffix, for example, `$-577`, `$0x1F`.
  - register - the content of the registers;
    - one of the sixteen 8-, 4-, 2-, or 1-byte low-order portions of the registers for operands having 64, 32, 16, or 8 bits;
    - the notation r_a to denote an arbitrary register a;
    - the value of r_a is the reference $R[r_a]$;
  - memory reference - also called effective address
    - $M_b[Addr]$ to denote a reference to the b-byte value stored in memory starting at address $Addr$.
      - Generally, we just drop the subscript b.
- Addressing modes:
  - the most general form: $Imm(r_b, r_i, s)$ - often seen when referencing elements of arrays.
    - Imm - the immediate offset;
    - r_b - the base register and must be 64-bit registers;
    - r_i - the index register and must be 64-bit registers;
    - s - the scale factor (s must be 1, 2, 4 or 8);

Practice Problem 3.1

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Fill in the following table showing the values for the indicated operands:

Operand	Value
%rax	
0x104	
$0x108	
(%rax)	
4(%rax)	
9(%rax,%rdx)	
260(%rcx,%rdx)	
0xFC(,%rcx,4)	
(%rax, %rdx, 4)	

Solution:

Operand	Value	Comments
%rax	0x100	the value in rax register
0x104	0xAB	the value stored in address 0x104
$0x108	0x108	immediate
(%rax)	0xFF	the value of the address stored in rax
4(%rax)	0xAB	the value stored in the address rax+4
9(%rax,%rdx)	0x11	the value stored in the address 9+rax+rdx
260(%rcx,%rdx)	0x13	the value stored in the address 260+rcx+rdx
0xFC(,%rcx,4)	0xFF	the value stored in the address rcx*4+0xFC
(%rax, %rdx, 4)	0x11	the value stored in the address rax+rdx*4

3.4.2 Data Movement Instructions - MOV

- We group the many different instructions into instruction classes, where the instructions in a class perform the same operation but with different operand sizes.

Simple data movement instructions

- `MOV` class - copy data from a source location to a destination location, without any transformation.

Instruction		Effect	Description
MOV	S, D	$D \leftarrow S$	Move
movb			Move byte
movw			Move word
movl			Move double word
movq			Move quad word
movabsq	I, R	$R \leftarrow I$	Move absolute quad word

- Details:
  - All 4 instructions have similar effects; they differ primarily in that they operate on data of different sizes: 1, 2, 4, and 8 bytes, respectively.
  - S - the source operand designates a value that is immediate, stored in a register, or stored in memory.
  - D - the destination operand designates a location that is either a register or a memory address.
  - Exception: copying a value from one memory location to another requires two instructions:
    - the first to load the source value into a register;
    - the second to write this register value to the destination.
  - For most cases, the mov instructions will only update the specific register bytes or memory locations indicated by the destination operand. The only exception is that when `movl` has a register as the destination, it will also set the high-order 4 bytes of the register to 0.
  - Examples:

```
1    movl $0x4050,%eax.   ; immediate -> register, 4 bytes
2    movw %bp,%sp.        ; register -> register,  2 bytes
3    movb (%rdi,%rcx),%al ; memory -> register,    1 byte
4    movb $-17,(%esp)     ; immediate -> memory,   1 byte
5    movq %rax,-12(%rbp)  ; register -> memory,    8 bytes
```

  - The final instruction: `movabsq I, R`
    - `movabsq` is for dealing with 64-bit immediate data;
    - `movq` can only have immediate source operands that can be represented as **32-bit** two's-complement numbers, then extend to produce the 64-bit value for the destination;
    - `movabsq` can have an arbitrary 64-bit immediate value as its source operand and can only have a register as a destination.

Zero-extending data movement instructions

- `MOVZ` class - copy a smaller source value to a larger destination and fill out the remaining bytes of the destination with zeros.

Instruction	Effect	Description
MOVZ S, R	$R \leftarrow$ ZeroExtend(S)	Move with zero extension
movzbw		Move zero-extended byte to word
movzbl		Move zero-extended byte to double word
movzwl		Move zero-extended word to double word
movzbq		Move zero-extended byte to quad word
movzwq		Move zero-extended word to quad word

- Observe that each instruction name has size designators as its final two characters—the first specifying the source size, and the second specifying the destination size.

- The absence of an explicit instruction to zero-extend a 4-byte source value to an 8-byte destination, why?
  - This type of data movement can be implemented using a `movl` instruction having a register as the destination.
  - This technique takes advantage of the property that an instruction generating a 4-byte value with a register as the destination will fill the upper 4 bytes with zeros.
- For 64-bit destinations, moving with sign extension is supported for all three source types, and moving with zero extension is supported for the two smaller source types.

Sign-extending data movement instructions

- `MOVS` class - copy a smaller source value to a larger destination and fill out the remaining bytes of the destination with sign extension.

Instruction	Effect	Description
MOVS S, R	$R \leftarrow$ SignExtend(S)	Move with sign extension
movsbw		Move sign-extended byte to word
movsbl		Move sign-extended byte to double word
movswl		Move sign-extended word to double word
movsbq		Move sign-extended byte to quad word
movswq		Move sign-extended word to quad word
movslq		Move sign-extended double word to quad word
cltq	%rax $\leftarrow$ SignExtend(%eax)	Sign-extend %eax to %rax

- `cltq` - no operands. uses register `%eax` as its source and `%rax` as the destination for the sign-extended result.
  - exactly same function as `movslq %eax, %rax`.
  - `cltq` - convert long to quad.
  - `cltq` has a shorter encoding than `movslq %eax, %rax`
    - `cltq` - 4898 in raw hex
    - `movslq %eax, %rax` - $4863C0$ in raw hex

Practice Problem 3.2

For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands. (For example, mov can be rewritten as `movb, movw, movl, or movq`.)

`mov__ %eax, (%rsp)`
`mov__ (%rax), %dx`
`mov__ $0xFF, %bl`
`mov__ (%rsp,%rdx,4), %dl`
`mov__ (%rdx), %rax` `mov__ %dx, (%rax)`

Solution:

`movl %eax, (%rsp)`
`movw (%rax), %dx`
`movb $0xFF, %bl`
`movb (%rsp,%rdx,4), %dl`
`movq (%rdx), %rax` `movw %dx, (%rax)`

Aside - Understanding how data movement changes a destination register

- 2 different conventions regarding whether and how data movement instructions modify the upper bytes of a destination register:

```
1           movabsq $0x0011223344556677, %rax ; %rax = 0011223344556677
2           movb $-1, %al                     ; %rax = 00112233445566FF
3           movw $-1, %ax                     ; %rax = 001122334455FFFF
```

- Convention 1:
  - The `movb` instruction therefore sets the low-order byte of %rax to FF with the remaining bytes unchanged;
  - The `movw` instruction sets the low-order 2 bytes to FFFF, with the remaining bytes unchanged.

```
1           movl $-1, %eax                    ; %rax = 00000000FFFFFFFF
2           movq $-1, %rax                    ; %rax = FFFFFFFFFFFFFFFF
```

- Convention 2:
  - The `movl` instruction sets the low-order 4 bytes to FFFFFFFF, but it also **sets the high-order 4 bytes to 00000000;**
  - The `movq` instruction sets the complete register to FFFFFFFFFFFFFFFF.

Aside - Comparing byte movement instructions

```
1   movabsq $0x0011223344556677, %rax ; %rax = 0011223344556677
2   movb $0xAA, %dl                   ; %dl  = AA
3   movb %dl,%al                      ; %rax = 00112233445566AA
4   movsbq %dl,%rax                   ; %rax = FFFFFFFFFFFFFFAA
5   movzbq %dl,%rax                   ; %rax = 00000000000000AA
```

- The `movsbq` instruction sets the other 7 bytes to either all ones or all zeros depending on the high-order bit of the source byte, which is sign bit;
- The `movzbq` instruction always sets the other 7 bytes to zero.

Practice Problem 3.3

Each of the following lines of code generates an error message when we invoke the assembler. Explain what is wrong with each line.

```
1           movb $0xF, (%ebx)
2           movl %rax, (%rsp)
3       movw (%rax),4(%rsp)
4       movb %al,%sl
5       movq %rax,$0x123
6       movl %eax,%rdx
7       movb %si, 8(%rbp)
```

Solution:

`movb $0xF, (%ebx)` - the address is 64bits while ebx is only 32bits.

`movl %rax, (%rsp)` - movl is too small.

`movw (%rax),4(%rsp)` - Source and dest cannot be memory reference at the same time.

`movb %al,%sl` - %sl is not a register.

`movq %rax,$0x123` - immediate cannot be the dest.

`movl %eax,%rdx` - should be movq
`movb %si, 8(%rbp)` - movb is wrong

3.4.3 Data Movement Example

```
long exchange(long *xp, long y)
{
    long x = *xp;
    *xp = y;
    return x;
}
```

- Data exchange routine - C: common data exchange routine.

```
    ;long exchange(long *xp, long y)
        ; xp in %rdi, y in %rsi
exchange:
        movq  (%rdi), %rax    ; Get x at xp. Set as return value.
        movq  %rsi, (%rdi)    ; Store y at xp.
        ret
```

- Data exchange routine - asm:
  - Function exchange is implemented with just three instructions:
    - two data movements (`movq`);
    - an instruction to return back to the point from which the function was called (`ret`).
  - instructions details:
    - Procedure parameters `xp` and `y` are stored in registers `%rdi` and `%rsi`;
    - `movq (%rdi), %rax` - reads x from memory and stores the value in register `%rax`, equal to `x = *xp` in C;
      - register `%rax` will be used to return a value from the function, and so the return value will be x;
    - `movq %rsi, (%rdi)` - writes `y` to the memory location designated by `xp` in register `%rdi`, equal to `*xp = y` in C;
    - `ret` - A function returns a value by storing it in register `%rax`, or in one of the low-order portions of this register.
  - Some notes:
    - What we call “pointers” in C are simply addresses;
      - Dereferencing a pointer involves copying that pointer into a register, and then using this register in a memory reference.
    - Local variables such as x are often kept in **registers** rather than stored in memory locations;
      - Register access is much faster than memory access.

Practice Problem 3.4

Assume variables `sp` and `dp` are declared with types

```
        src_t *sp;
        dest_t *dp;
```

where src_t and dest_t are data types declared with typedef. We wish to use the appropriate pair of data movement instructions to implement the operation

```
*dp = (dest_t) *sp;
```

Assume that the values of `sp` and `dp` are stored in registers `%rdi` and `%rsi`, respectively. For each entry in the table, show the two instructions that implement the specified data movement. The first instruction in the sequence should read from memory, do the appropriate conversion, and set the appropriate portion of register `%rax`. The second instruction should then write the appropriate portion of `%rax` to memory. In both cases, the portions may be `%rax, %eax, %ax,` or `%al`, and they may differ from one another.

Recall that when performing a cast that involves both a size change and a change of "signedness" in C, the operation should change the size first.

src_t	dest_t	Instruction
long	long	`movq (%rdi), %rax` `movq %rax, (%rsi)`
char	int	
char	unsigned	
unsigned char	long	
int	char	
unsigned	unsigned char	
char	short	

Solution:
We know that the values of `sp` and `dp` are stored in registers `%rdi` and `%rsi`

src_t	dest_t	Instruction
long	long	`movq (%rdi), %rax` `movq %rax, (%rsi)`
char	int	`movsbl (%rdi), %eax` `movl %eax, (%rsi)`
char	unsigned	`movsbl (%rdi), %eax` `movl %eax, (%rsi)`
unsigned char	long	`movzbq (%rdi), %rax` `movq %rax, (%rsi)`
int	char	`movl (%rdi), %eax` `movb al, (%rsi)`
unsigned	unsigned char	`movl (%rdi), %eax` `movb al, (%rsi)`
char	short	`movsbw (%rdi), %ax` `movw %ax, (%rsi)`

Practice Problem 3.5

You are given the following information. A function with prototype

```
void decode1(long *xp, long *yp, long *zp);
```

is compiled into assembly code, yielding the following:

```
  ; void decode1(long *xp, long *yp, long *zp)
  ; xp in %rdi, yp in %rsi, zp in %rdx

decode1:
   movq    (%rdi), %r8
   movq    (%rsi), %rcx
```

```
    movq    (%rdx), %rax
    movq    %r8, (%rsi)
    movq    %rcx, (%rdx)
    movq    %rax, (%rdi)
    ret
```

Parameters `xp`, `yp`, and `zp` are stored in registers `%rdi`, `%rsi`, and `%rdx`, respectively.

Write C code for decode1 that will have an effect equivalent to the assembly code shown.

Solution:

```
; void decode1(long *xp, long *yp, long *zp)
; xp in %rdi, yp in %rsi, zp in %rdx

decode1:
    movq    (%rdi), %r8     # r8  = M(rdi): r8  = *xp
    movq    (%rsi), %rcx    # rcx = M(rsi): rcx = *yp
    movq    (%rdx), %rax    # rax = M(rdx): rax = *zp
    movq    %r8, (%rsi)     # M(rsi) = r8:  *yp = r8  = original *xp
    movq    %rcx, (%rdx)    # M(rdx) = rcx: *zp = rcx = original *yp
    movq    %rax, (%rdi)    # M(rdi) = rax: *xp = rax = original *zp
    ret
```

```
void decode1(long *xp, long *yp, long *zp)
{
        long x = *xp; // r8
        long y = *yp; // rcx
        long z = *zp; // rax

        *yp = x;
        *zp = y;
        *xp = z;
}
```

3.4.4 Pushing and Popping Stack Data

Instruction	Effect	Description
pushq S	R[%rsp] ← R[%rsp] − 8; M[R[%rsp]] ← S	Push quad word
popq D	D ← M[R[%rsp]]; R[%rsp] ← R[%rsp] + 8	Pop quad word

- The final two data movement operations are used to push data onto and pop data from the program stack;
- A stack is a data structure where values can be added or deleted, but only according to a “last-in, first-out” discipline.
  - push operation - add data to a stack;
  - pop operation - pop data from a stack to an operand;
- A stack can be implemented as an array, where we always insert and remove elements from one end of the array.
  - The top of the stack - the end of the array;
  - The stack pointer `%rsp` holds the address of the top stack element.
- The stack is contained in the same memory as the program code and other forms of program data, programs can access arbitrary positions within the stack using the standard memory addressing methods.
  - For example, assuming the topmost element of the stack is a quad word, the instruction `movq 8(%rsp),%rdx` will copy the second quad word from the stack to register `%rdx`.

pushq instruction

- `pushq %rbp` is equivalent to the instructions:

```
1           subq $8,%rsp     ; Expand the stack by descreasing %rsp
2           movq %rbp,(%rsp) ; Store %rbp on stack
```

- The pushq instruction is encoded in the machine code as a single byte

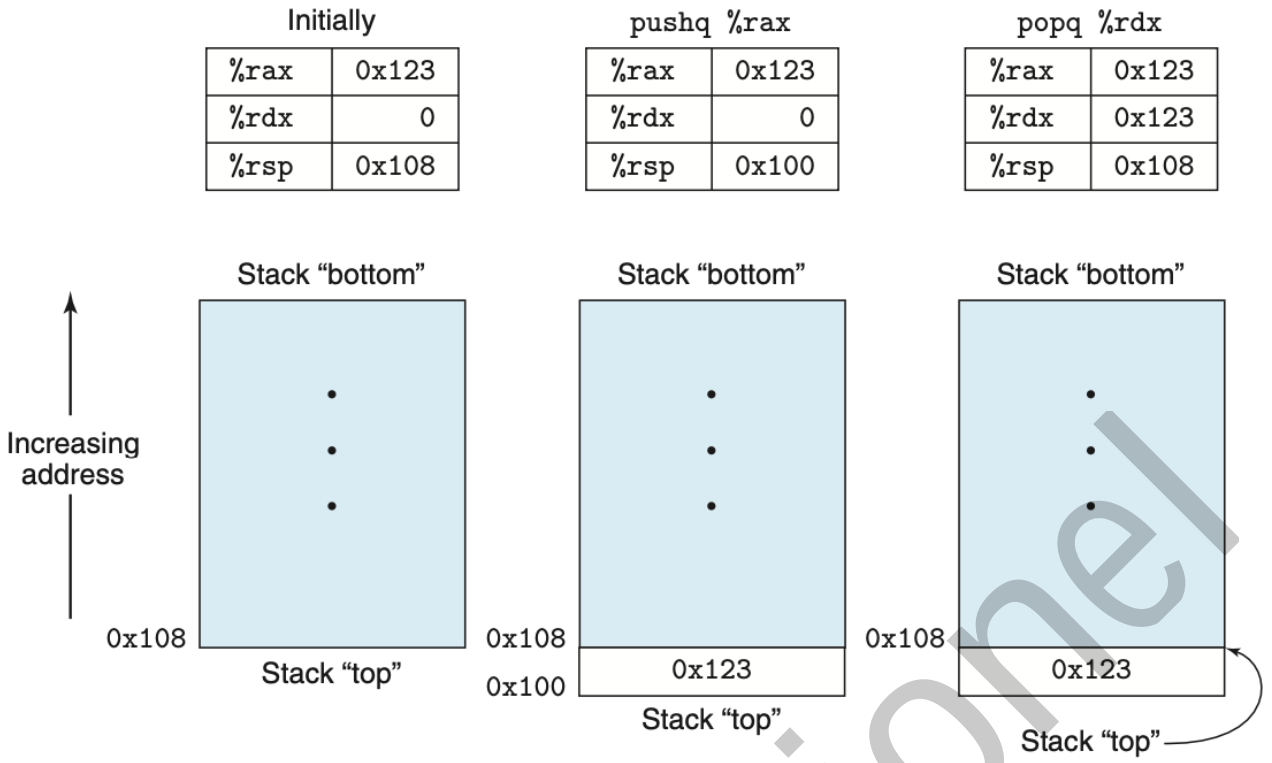
PUSH—Push Word, Doubleword, or Quadword Onto the Stack

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH r/m16	M	Valid	Valid	Push r/m16.
FF /6	PUSH r/m32	M	N.E.	Valid	Push r/m32.
FF /6	PUSH r/m64	M	Valid	N.E.	Push r/m64.
50+rw	PUSH r16	O	Valid	Valid	Push r16.
50+rd	PUSH r32	O	N.E.	Valid	Push r32.
50+rd	PUSH r64	O	Valid	N.E.	Push r64.
6A ib	PUSH imm8	I	Valid	Valid	Push imm8.
68 iw	PUSH imm16	I	Valid	Valid	Push imm16.
68 id	PUSH imm32	I	Valid	Valid	Push imm32.
0E	PUSH CS	ZO	Invalid	Valid	Push CS.
16	PUSH SS	ZO	Invalid	Valid	Push SS.
1E	PUSH DS	ZO	Invalid	Valid	Push DS.
06	PUSH ES	ZO	Invalid	Valid	Push ES.
0F A0	PUSH FS	ZO	Valid	Valid	Push FS.
0F A8	PUSH GS	ZO	Valid	Valid	Push GS.

popq instruction

- `popq %rax` is equivalent to the instructions:

```
1           movq (%rsp),%rax ; Pop the data on topmost stack to %rax
2           addq $8,%rsp     ; Shrink stack by increasing stack pointer
```



- The first two columns illustrate the effect of executing the instruction `pushq %rax` when `%rsp` is 0x108 and `%rax` is 0x123.
- Firstly `%rsp` is decremented by 8, giving 0x100, and then 0x123 is stored at memory address 0x100, by executing `pushq %rax`.
- Secondly, value 0x123 is read from memory and written to register `%rdx`. Register `%rsp` is incremented back to 0x108, by executing `popq %rdx`.