# 9 x86-64 ASSEMBLY LANGUAGE

A very small subset of the x86-64 instruction set will be used to provide examples of various programming concepts using assembly language. Although small, this set is nevertheless sufficient to implement many algorithms that employ control structures and subprograms.

The core set of instructions to be used are as follows:

| SYNTAX | | SEMANTICS |
|---|---|---|
| INST | OPNDS | |
| mov | S, D | D $\leftarrow$ S |
| add | S, D | D $\leftarrow$ D + S |
| sub | S, D | D $\leftarrow$ D $-$ S |
| imul | S, D | D $\leftarrow$ D * S |
| cmp | $S_1$, $S_2$ | SF $\leftarrow$ ($S_2$ - $S_1$) $< 0$ |
| | | ZF $\leftarrow$ ($S_2$ - $S_1$) $= 0$ |
| jmp | addr | PC $\leftarrow$ PC + (addr - (*+2)) |
| jl | addr | if SF ^ OF then PC $\leftarrow$ PC + (addr - (*+2)) |
| je | addr | if ZF then PC $\leftarrow$ PC + (addr - (*+2)) |

The symbols S, $S_1$, $S_2$, and D represent operands that can employ (with some exceptions) any of the following addressing modes:

1. **Immediate mode**: Operand is a value. Value is preceded by "$".
2. **Register mode**: Operand is a register. Register name is preceded by "%".
3. **Base+Displacement mode**: Operand is a register and value. Register name is preceded by "%", enclosed in perntheses, and preceded by the value (no "$").

NOTE: This is only a small subset of the possible addressing modes. See the textbook, figure 3.3, for a full list.

As mentioned there are some restrictions regarding the modes of S and D:

1. Source operands, S, $S_1$, $S_2$, can be any of the three modes.
2. Destination operands, D, cannot be immediate mode.
3. Both operands cannot be base+displacement mode.
4. Both operands should specify values of the same size.
5. Immediate mode operands must be less than $2^{31} - 1$.

### 9.1 Instruction Families

The data transfer, arithmetic/logic/shift, and test instruction mnemonics provided in the table actually define a family of instructions, depending on the size of the operands. The following table identifies the actual instruction from each family for an operand of a given size:

| size | MOV | ADD | SUB | IMUL | CMP |
|---|---|---|---|---|---|
| byte | movb | addb | subb | imulb | cmpb |
| word | movw | addw | subw | imulw | cmpw |
| double | movl | addl | subl | imull | cmpl |
| quad | movq | addq | subq | imulq | cmpq |

As can be seen from the mnemonics provided, the suffix is based on the size of the operand. If the suffix is omitted, the assembler will attempt to determine the correct size and instruction. So, providing suffixes is not required. However, general program practice dictates that is often safer to include the size suffix. Sometimes the assembler will be unable to infer the appropriate choice of instruction from the operands and this will generate a warning message during the translation of the assembly program to a machne language program.

### 9.2 Assembler Directives

In addition to instructions, most assembly languages include assembler directives (also called "peudo-ops") that provide guidance to the assembler on how a program should be translated into machine code and where program and data should be stored. All pseudo-op names begin with a period .

1. External memory is partitioned into a number of sections and two of the most important are the data section and the program section. In order to identify to the assembler code that is to be placed in either section, two pseudo-ops are provided:

   **.data** identifies the start of the section of memory where data will be placed for access by the program. All statements that follow this pseudo-op will be placed in consecutive locations of memory according to specifications given.

   **.text** identifies the program section of memory. This is where executable instructions that define a program, such as those described above, will be placed.

   **.globl <starting address>** : identifies the instruction that will be the first to be executed by the CPU. This pseudo-op is required in every file that defines an assmbly language program.

2. When allocating storage for memory, the primary consideration is the number of bytes that will be required for each item of data. The first byte of each data value is usually assigned a label which acts as a symbol for the actual address where the data is stored. Following each label, a pseudo-op defines the number of bytes to be allocated:

   **.byte N** : Allocates one byte of storage for the value N. Any address can specify the location of a byte.

**.word N** : Allocates two bytes of storage for the value N. Addresses begin on "even" addresses.

**.long N** : Allocates four bytes of storage for the value N. Addresses begin on an address divisible by 4.

**.quad N** : Allocates eight bytes of storage for the value N. Addresses begin on an address ending in 0 or 8 (i.e., divisible by 8).

**.space N** Allocates N bytes of storage without assigning any values.

**.string "<characters>"** : Allocates sufficient storage to store the characters of the string as 8-bit ASCII character codewords.

NOTE: The value, N, can be expressed as an integer value or as a hexadecimal value. The latter must be prefixed with "0x".

As a first example of an x-86-64 program consider again the problem of writing an assembly language program to compute z = (p + q) * (p − q):

```
        .data
P:      .quad   6
Q:      .quad   4
Z:      .quad

        .text
        .globl main
main:   mov     $P, %rbx
        mov     0(%rbx) ,%rax
        mov     8(%rbx), %rcx
        mov     %rax, %rsi
        add     %rcx, %rax
        sub     %rcx, %rsi
        imul    %rsi
        mov     %rax, 16(%rbx)
        ret
```

Note that this program provides no means of enterering a value or displaying a result during execution. To do so requires that the program use the operating system. This is usually achieved using function calls to system defined input and output functions such as `scanf` and `printf`. For now input and output will be performed by using the data area of the program to provide values and store results. These locations can be examined using `gdb`.

## 9.3 Machine Language Program

The assembly language program is really just a symbolic representation of the actual program stored in external memory. The linux command "`objdump`" can display what the machine language program looks like

For example, if the loaded, runtime file for the example above s called "`runpq`" then:

```
objdump runpq
```

will display the output shown on the next page.

```
00000000004004ed <main>:
  4004ed: 48 c7 c3 38 10 60 00  mov    $0x601038,%rbx
  4004f4: 48 8b 03              mov    (%rbx),%rax
  4004f7: 48 8b 4b 08           mov    0x8(%rbx),%rcx
  4004fb: 48 89 c2              mov    %rax,%rdx
  4004fe: 48 01 c8              add    %rcx,%rax
  400501: 48 29 ca              sub    %rcx,%rdx
  400504: 48 f7 ea              imul   %rdx
  400507: 48 89 43 10           mov    %rax,0x10(%rbx)
  40050b: c3                    retq
  40050c: 0f 1f 40 00           nopl   0x0(%rax)
```

By executing the `runpq` using `gdb`, one can examine the contents of internal or external memory:

**Step 1** : Dump to determine address of the return statement:

```
(gdb) disassemble runpq

Dump of assembler code for function main:
   0x00000000004004ed <+0>:  mov    $0x601038,%rbx
   0x00000000004004f4 <+7>:  mov    (%rbx),%rax
   0x00000000004004f7 <+10>: mov    0x8(%rbx),%rcx
   0x00000000004004fb <+14>: mov    %rax,%rdx
   0x00000000004004fe <+17>: add    %rcx,%rax
   0x0000000000400501 <+20>: sub    %rcx,%rdx
   0x0000000000400504 <+23>: imul   %rdx
   0x0000000000400507 <+26>: mov    %rax,0x10(%rbx)
   0x000000000040050b <+30>: retq
   0x000000000040050c <+31>: nopl   0x0(%rax)
End of assembler dump.
```

**Step 2** : Break inserted on the return statement

```
(gdb) break *main+30
```

**Step3** : Display inputs:

```
(gdb) x/16xb &P
0x601038: 0x06 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x601040: 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

NOTE that since `Q` is stored 8 bytes after `P`, by printing 16 bytes beginning at address `&P`, the values of both `P` and `Q` can be observed.

**Step 4** : Display output:

```
(gdb) x/8xb &Z
0x601048: 0x14 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

# 10 CONTROL STRUCTURES

Testing and branching instructions are used in assembly language to implement programming structures called "control structures" commonly found in high-level languages. Two such control structures are described here.

## 10.1 Conditional Branching

"Conditional branching," more commonly known as "IF-THEN-ELSE," is a control structure that can be represented in pseudo-code as follows:

IF *conditional expression* THEN
    *Instructions to execute if condition is true*
ELSE
    *Instructions to execute if condition is false*
END IF
Steps that follow the branch control structure

This control structure requires careful choice of testing and branching instructions to insure that it is implemented correctly. Common mistakes include executing the wrong program segment for a given condition and execute both cases of the given control structure. In order to avoid these hazards, a programming technique called the "Method of Coding the False Condition First" can be used:

1. A test instruction for the condition (cmp),
2. Branch instruction to step 5 if condition is TRUE,
3. Instructions to execute if condition is FALSE,
4. An unconditional branch instruction to step 6 (jmp),
5. Instructions to execute if condition is TRUE,
6. Instructions to execute following the control structure.

EXAMPLE: Round an amount between 0 and 5 cents to the nearest 5 cents

ALGORITHM:

IF AMT $<$ 3 THEN
    CHGE = 0
ELSE
    CHGE = 5
END IF

IMPLEMENTATION:

```
        .data
AMT:    .quad  4
CHGE:   .quad

        .text
        .globl main
main:   mov    $AMT, %rbx
        mov    0(%rbx), %rax   # R[rax] = M[AMT]
        cmp    $3, %rax        # M[AMT] < 3?
        jl     NO_CHG
        #condition is false
MK_CHG: mov    $5, 8(%rbx)     # M[8+AMT] = 5
        jmp    END_IF
        #condition is true
NO_CHG: mov    $0, 8(%rbx)     # M[8+AMT] = 0
END_IF:  . . .
```

### 10.2 Iteration

An iteration control structure can be any of the following types of "loop control structures":

WHILE *conditional expression* DO
    *Steps to repeat until conditional expr. false*
END WHILE

While there are several way to encode a WHILE loop in assembly language, the following structure minimizes the possibility of an incorrect implementation. This method, called "Complement the Conditional Expression," is described as follows:

1. A test instruction for the complement of the condition, (cmp)
2. A conditional branch instruction to step 5 if the complemented condition is true (i.e., when the condition is false)
3. Instructions to execute while the original condition is true,
4. An unconditional branch instruction to step 1 (jmp),
5. Instructions to execute once the original condition is false.

EXAMPLE: Multiply M by N using repeated addition:

PROD = M + M + ... + M (N times)

ALGORITHM:

PROD = 0
WHILE N $\neq$ 0 DO
    PROD = PROD + M
    N = N - 1
END WHILE

PROGRAM:

```
        .data
N:      .quad  2
M:      .quad  3
PROD:   .quad

        .text
        .globl  main
main:   mov    $N, %rbx
        mov    0(%rbx), %rax #R[rax] is loop counter
        mov    $0, %rcx      #R[rcx] is the product
LOOP:   cmp    $0, %rax      #R[rax] = 0?
        je     END_LP
        sub    $1, %rax
        add    8(%rbx), %rcx #R[rcx] = R[rcx] + M[8+N]
        jmp    LOOP
END_LP: mov    %rcx,16(%rbx) #M[16+N] = R[rcx]
```

Previously, it was shown that a good measure of performance is to count the number of external memory accesses required to execute the program. The fewer the number of accesses, the less the von Neumann bottleneck degrades performance. The syntax of x86-64 programs lends itself to calculating the number of memory accesses that occur in the **execution phase** of the instruction execution algorithm.

In the previous example, every base+displacement operand represents an external memory access. Since the number of iterations that occur is determined by the value of N. Therefore the number of memory accesses during the execution phase equals N + 2. Effectively this means that this programs performance is dramatically affected by the von Neumann bottleneck.

The most effective way to improve performance is to eliminate the external memory accesses within the loop, if possible. The following solution does this:

```
        .data
N:      .quad  2
M:      .quad  3
PROD:   .quad

        .text
        .globl main
main:   mov    $N, %rbx
        mov    0(%rbx), %rax  # R[rax] is loop counter
        mov    $0, %rcx       # R[rcx] is the product
        mov    8(%rbx), %rdx  # R[rdx] <- 3.
LOOP:   cmp    $0, %rax
        je     END_LP
        sub    $1, %rax
        add    %rdx, %rcx     # R[rcx] <- R[rcx] + 3
        jmp    LOOP
END_LP: mov    %rcx, 16(%rbx)
```

In this case the total number of memory accesses that occur during exectuion is 3: that is, it is constant. So performance no longer depends on the value of N.

# 11 1-DIMENSIONAL ARRAYS

A *1-dimensional array* in assembly language programming refers to a sequence of values stored in consecutive locations of memory. Only the first location is labelled, and all other locations are referenced with respect to the first location. The programming technique used to access each of the values in an array is called indexing. An "index" is an address to a specific location in an array to access the value stored there. Each index is obtained by adding the address of the first location of the array, called the "base address" to a value called the "displacement".

Sufficient storage must be allocated to accommodate all the values in the array. For example, consider a program to sum the values in an array of four unsigned integers. Four 8-byte locations will be required to hold the all the values of the array. Therefore a sequence of four declaration statements is required:

```
        .data
NMBR    .quad   20
        .quad    31
        .quad    4
        .quad    75
```

This can be expressed more succinctly in HC-12 assembler as:

```
NMBR    .quad    20, 31, 4, 75
```

EXAMPLE: Sum the elements in a list of integers

Algorithm:

```
POSN = 0
SUM = 0
WHILE POSN < SIZE DO
     SUM = SUM + LST[POSN]
     POSN = POSN + 1
END WHILE
```

Implementation:

```
            .data
SUM:        .quad
SIZE:       .quad   4
LST:        .quad   20, 31, 4, -10

            .text
            .globl  main
main:       mov     $LST, %rbx      #R[rbx] holds address “LST”
            mov     -8(%rbx), %rsi  #R[rsi] ← M[SIZE]
            mov     $0, %rax        #R[rax] is the loop counter
            mov     $0, %rcx        #R[rcx] accumulates the SUM
LOOP:       cmp     %rsi,%rax
            je      DONE
            add     0(%rbx), %rcx   #R[rcx] = R[rcx] + M[0+R[rbx]]
            add     $1, %rax        #R[rax] = R[rax] + 1
            add     $8, %rbx        #increment the base register
            jmp     LOOP
DONE:       mov     $SUM, %rbx      #R[rbx] holds the address "SUM"
            mov     %rcx, 0(%rbx)   #Store R[rcx] in M[SUM]
```

This implementation controls the number of iterations of the loop by using the fact that the size of the array is known. However, when this information is not provided, an alternate strategy can be used:

```
            .data
SUM:        .quad
LST:        .quad   20, 31, 4, -10
LSTEND:     .byte

            .text
            .globl  main
main:       mov     $LST, %rbx      #R[rbx] holds address “LST”
            mov     $0, %rcx        #R[rcx] accumulates SUM
LOOP:       cmp     $LSTEND, %rbx
            je      DONE
            add     0(%rbx), %rcx   #SUM = SUM + M[0+R[rbx]]
            add     $8,%rbx         #increment the base register
            jmp     LOOP
DONE:       mov     $SUM, %rbx      #R[rbx] holds address "SUM"
            mov     %rcx, 0(%rbx)   #Store R[rcx] in M[SUM]
```

In this case, a “sentinel” is used to terminate the loop. A sentinel is a value that can be used to flag the end of the array in memory. Sometimes this can be a value, such as the null character used to terminate the end of a string. In this example, the address of the first location following the last value in the array is used. A test is then made comparing the value of the array base address + index with the value of `LSTEND`.

## 11.1 Dynamic Arrays

Many programs provide memory for their data values in two ways:

1. **Static memory** is memory located in the data area section of external memory:
   (a) Space is allocated prior to execution by the assembly as defined by pseudo-ops.
   (b) This space is called "static" because it remains allocated during the execution of the entire program.
2. **Dynamic memory** is memory located in the stack area of external memory:
   (a) Space is allocated by the program during execution.
   (b) Space is recoverable by the program during execution.

In the x86-64 CPU, the "user-managed" dynamic memory is called "stack memory"

### 11.1.1 Properties of Stack Memory

1. Memory is accessed via the stack pointer register, `rsp`:
   - The value in the stack pointer is an address, pointing to the last value stored in stack memory.
   - The first value removed from stack memory is the last value stored.
2. Memory is allocated "backwards" :
   - Stack memory is allocated by decrementing the stack pointer (by 8).
   - Stack memory is recovered by incrementing the stack pointer (by 8).
3. To store a value in stack memory:

| SYNTAX | SEMANTICS |
|---|---|
| pushq S | 1. R[rsp] ← R[rsp] − 8<br>2. M[R[rsp]] ← S |

4. To remove a value from stack memory:

| SYNTAX | SEMANTICS |
|---|---|
| popq D | 1. D ← M[R[rsp]]<br>2. R[rsp] ← R[rsp] + 8 |