Teaching materials for

Assembly programming

subject of CSE BSc

Imre VARGA PhD

Faculty of Informatics
University of Debrecen

2024

Last update: 2 Sept 2024

1. Introduction

The assembly is a hardware-close way of programming. It is below the high-level programming languages (e.g.: C, Java, Python) but more readable for humans than binary machine codes. Assembly is **ISA** (Instruction-Set Architecture) dependent. This document is restricted to only **x86-64** architecture which is the base of almost all Intel and AMD processors of modern computers. Two dialects of this assembly language can be used: Intel and AT&T syntax. The author focuses only on **Intel syntax**. Understanding assembly programming requires hardware knowledge. Familiarity with x86-64 ISA is required (only partially included here).

The assembly source code in the Linux operating system is a simple text file(s) with "`.s`" extension. One of the instructions must be labeled by the `main` name, this will be the entry point of the execution. The `gcc` command can compile the assembly codes to executable files. For example, the `test.s` assembly source file can be compiled to an executable called `run` using the following command:

```
gcc test.s -o run
```

In some cases, the `-no-pie` switch is also required in the compilation command to create position-independent executables. In case of successful compilation, the program can be executed in a terminal by the

```
./run
```

command. (The source and the executable are in the same current folder.) The return value of the program (the `main` function) is stored in an environment variable. Its value can be seen by executing the

```
echo $?
```

command immediately after the termination of the program. The return value in Linux is interpreted as an unsigned char value, so only the last byte is considered as a non-negative whole number. Conventionally (but not always), the `0` return value indicates successful termination.

2. Registers and memory

Register set

In the CPU all data and instructions are stored in registers. These are some sequential digital circuits in the processor with very short access time and a few bytes of storage capacity. The programmer can use several different-size **general-purpose fixed-point register**s.

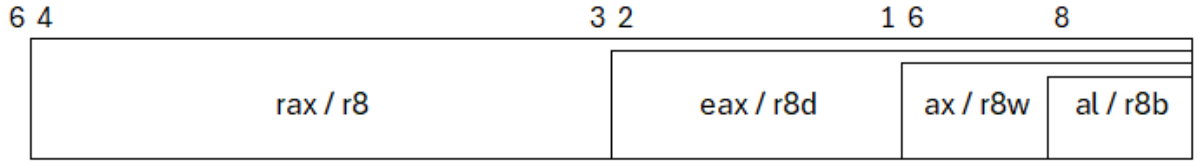
64-bit (quad word) registers:
`rax, rbx, rcx, rdx, rbp, rsp, rdi, rsi, r8, r9, r10, r11, r12, r13, r14, r15`

32-bit (double word) registers:
`eax, ebx, ecx, edx, ebp, esp, edi, esi, r8d, r9d, r10d, r11d, r12d, r13d, r14d, r15d`

16-bit (word) registers:
`ax, bx, cx, dx, bp, sp, di, si, r8w, r9w, r10w, r11w, r12w, r13w, r14w, r15w`

8-bit (byte) registers:
`al, ah, bl, bh, cl, ch, dl, dh, bpl, spl, dil, sil, r8b, r9b, r10b, r11b, r12b, r13b, r14b, r15b`

Small registers are always parts of larger ones (aliases to the fragments of the whole registers) as it is illustrated in the following figure.



When a 32-bit value is stored in a 64-bit register, the most significant 32 bits becomes 0s, in any other cases when a part of a register is overwritten the most significant bits do not change.

All the above registers use the fixed-point representation of the stored values, so only integer values (`signed/unsigned int/short/long/char`) can be stored in them. Floating point values (i.e. real numbers) can be stored in other registers (see x87 architecture or MMX vector registers).

Besides the general-purpose registers, some special registers are also important. The `rip` (instruction pointer) register plays the role of the **program counter**, so it always stores the 64-bit memory address of the next binary instruction (within the code segment of the RAM). It is essential to control flow. During normal sequential execution, its value is incremented by the size of the current machine code instruction.

The **status register** is called `rflags`. Its bits separately represent different states/settings of processor operation. Four of these bits have large importance even in simple codes, these are the following.

`CF` (carry flag): it is 1, if there was a carry in the last arithmetic operation, so when x+1 bits are necessary to represent the results of the operation having x-bit operands, otherwise it is 0.

`OF` (overflow flag): it is 1, if the result of an arithmetic operation is wrong in the case of signed fixed-point representation, else it is 0. It indicates an unexpected sign change.

`ZF` (zero flag): it is 1, if the result of the last arithmetic operation is zero, otherwise it is a 0 bit.

`SF` (sign flag): it is 1, if the result of the last operation is negative in the case of signed fixed-point representation. Practically speaking it is the most significant bit of the result.

These bits change during comparisons as well, so they are essential in the case of condition evaluations.

There are so-called segment selector registers in x86-64 ISA, but they are used only for legacy reasons. Here is the list of them: `CS` (code segment), `SS` (stack segment), `DS` (Data segment), `ES`, `FS`, `GS`.

Memory

Variables are stored in the RAM. Their locations are referred by memory **address**es (thus a kind of memory cell identifiers). Each byte is addressable, i.e. has its own address, but bits inside a byte cannot be addressed. These addresses can be imagined as 64-bit unsigned integers. When the processor reads or writes a value, not just its address must be specified, but also its representation length as well (`byte`: 8 bits, `word`: 16 bits, `double word`: 32 bits, `quad word`: 64 bits).

More than one byte-long values are stored according to the **little-endian byte order**, so the least significant byte is located in the lowest address byte and the most significant byte is stored in the last byte of the memory space of the variable. Thus, reverse byte order is used, however the order of bits inside the bytes does not change. For example, the decimal value 27 970 800 (which in binary notation looks like 1 10101010 11001100 11110000) is stored as the following 32-bit bit sequence: 11110000 11001100 10101010 00000001.

A part of the RAM is managed specially. It is the so-called **stack**, which implements LIFO operation, by special assembly instructions (however the content can be still accessed directly as well). The top of the stack is the only important part of it. The `rsp` register stores the address of the top of the stack (i.e. the starting address of the last data). The value of the `rsp` content is decreasing automatically by the size of the value **push**ed in or decreased by the number of bytes **pop**ped out from the stack. In this way the stack is growing towards the lower addresses.

3. Assembly instructions in general

Each line of the assembly source files contains a maximum of one instruction. Each instruction can have 4 components: label, mnemonic, operand(s), comment.

Label

It is a kind of identifier of the given instruction. Usually, programmer must refer to another existing instruction, this can be done by the unique label of the given instruction. A label is an optional component so it can be skipped, but if a label is necessary, it must be at the beginning of the line.

Formally, a label is a character sequence containing alphanumeric characters (English letters and/or digit characters) and/or underscore (‘`_`’) and/or dot (‘`.`’) characters finished by a colon (‘`:`’) character. This assembly language is not case-sensitive, so upper- and lower-case letter characters mean the same. Labels started by a dot character mean local names/identifiers valid only in the given source file. It is a readable text for the programmer, but in the background, it means the memory address of the given instruction (which is useful for control flow) or the address of a statically allocated memory location. More than one different labels can refer to the same memory address.

Mnemonic

It is the name of an assembly instruction/operation, usually a kind of abbreviation or short form referring to the meaning of an operation, for example, the mnemonic for the shift arithmetic right operation is ‘`sar`’. Each instruction line must contain a mnemonic. The x86-64 ISA provides hundreds of mnemonics, but this document covers only the most frequently

used ones (see Chapter 4). Mnemonics started with a dot (‘.’) character are directives, so instructions that influence the compilation process.

Operand

These are values on which the given operations will be performed. A space or tab character is needed between the mnemonic and the first operand. An instruction can have 0, 1, 2, or 3 operands. Three operands are very rare in x86-84 architecture. Zero operand means that an operation can be executed alone without the specification of operands. Sometimes specification of an operand is not necessary (however the operand itself is needed) because in this case, the required value can be found in a fixed location. If there is more than one operand they are separated by a comma (‘,’) character. According to the Intel syntax, the first operand is the **destination operand** where the result is usually stored overwriting the old content.

Operands can be constants, register contents, or memory contents. Numeric constants (literals) can be given using simple decimal notation (e.g. `196`) or in hexadecimal form using “`0x`” prefix (e.g. `0xC4`). Both negative and positive integers can be used in the assembly code. (In the case of an 8-bit operand both `-1` and `255` result in the `11111111` bit sequence as a value.) The operation itself will define which interpretation is used during calculations. The destination operand cannot be a constant.

Only one of the operands can be memory content due to technical limitations. If an operand is a memory content, both its **memory address** and its **representation length** must be specified. The starting address must be given between square brackets (“`[ ]`”) according to the possible **addressing mode**s, while the length (i.e. the number of bytes from the starting address) must be specified before the square bracket by one of the following 4 phrases: `BYTE PTR`, `WORD PTR`, `DWORD PTR`, `QWORD PTR` (1, 2, 4 and 8 bytes respectively). The address can be given in a complex way summarized by the following general form: `[SS: B + I * SF + C]`, where `SS` is a segment selector register, `B` is a register containing a base address, `I` is playing the role of the index register (how many units far from the base address), `SF` is a scale factor (unit size), and `C` is a constant offset. Most components of the address calculation can be omitted. If `I` is not present, then `SF` must be also skipped. The `SF` can be 1, 2, 4, or 8. The `C` can be given as a literal or one can use the label of memory allocation directives as well. To understand the address calculation, an example is explained. A programmer wants to refer to the 5th element of an integer array. In a high-level programming language (e.g. in C language), it is written as `MyArray[4]`. Assuming that the array index has already stored in the `rcx` register, the beginning of the array is 48 bytes ahead of the address stored in the `rbp` register and the integer type is represented on 4 bytes. In assembly instruction, `DWORD PTR [rbp+rcx*4-48]` is written as an operand.

One of the operands or both of them can be specified in register(s). In most cases, the (register or memory) operands of a given instruction must be the same size. When the **source operand** (so the second operand in case of Intel syntax) is a constant value its representation length is determined by the destination operand.

Comment

From a hashtag (‘#’) character until the end of the given line all content is ignored by the assembler (assembly compiler), thus programmers can use end-of-line comments. Comments are very useful in assembly because the code is not as readable as in high-level languages.

4. Frequent instructions in detail

A short description of the most often used assembly directives and executable instructions are given below. For a more detailed description please, read [1, 2].

Directives

`.2byte`

Allocates 2 bytes (e.g. an initialized `static short int` variable). The address of the allocated memory field can be referred to by the label written at the beginning of the line. It has only one operand, a constant which is stored in the allocated field as an initialization.

`.4byte`

Allocates 4 bytes (e.g. an initialized `static int` variable). The address of the allocated memory field can be referred to by the label written at the beginning of the line. It has only one operand, a constant which is stored in the allocated field as an initialization. (Static allocation.)

`.bss`

It indicates the beginning of the uninitialized data section. No operand is needed.

`.byte`

Allocates 1 bytes (e.g. an initialized `static char` variable). The address of the allocated memory field can be referred to by the label written at the beginning of the line. It has only one operand, a constant which is stored in the allocated field as an initialization.

`.comm`

Allocates uninitialized memory space in the data section. It has 2 operands: the first one behaves as a symbolic name of the starting address of the allocated area, while the second is the number of bytes that must be allocated. Creation of uninitialized static scalar or array variables.

`.data`

It indicates the beginning of the data section. Static memory allocation can happen after this. No operands.

`.globl`

It is followed by one or more symbolic names (later labels) that must be handled as global names. For example, the `main` must be a global name accessible anywhere in the code because it will be the address of the entry point for execution. If multiple symbolic names are given, they must be separated by comma (‘`,`’) characters.

`.intel_syntax`

Usually, the first instruction of the source code, when the programmer will use Intel syntax in the remaining part of the code. It has an operand: `noprefix`.

`.section`

A read-only data segment must start with this instruction followed by the `.rodata` operand. For example, string constants are stored here.

`.string`

Memory allocation for a constant string. The starting address of the allocated memory field can be referred to by the label written at the beginning of the line. It has an operand: a character sequence surrounded by double quotes like `"Hello World!\n"`. The number of the allocated bytes is determined by the length of the character given sequence including the termination zero (‘`\0`’) character as well (e.g. 14 bytes in the previous case). This allocation must be in a read-only part of the RAM, thus after the `.section .rodata` instruction.

`.text`

It indicates the beginning of the code segment (where the executable instructions are). It can be ignored if no other segments (e.g. `.data`, `.bss`) in the code.

`.zero`

Zero initialized memory allocation for static scalar or array variables. The starting address of the allocated memory field can be referred to by the label written at the beginning of the line. It has only one operand, the number of bytes to be allocated and initialized by only `0` bits.

Executable instructions

`add`

Addition of the values of the operands. The sum overwrites the destination operand (so the first operand according to the Intel syntax). The operands and the result can be interpreted as both signed and unsigned fixed-point values. The `SF`, `ZF`, `CF` and `OF` status register bits are updated according to the result.

`and`

Bitwise AND operation on the operands. All bit positions of the operands are considered simultaneously according to the truth table of the logical AND operation. The result overwrites the destination operand. The `OF` and `CF` flags are cleared; the `SF` and `ZF` flags are set according to the result.

`call`

Subroutine invocation/call. It has only one operand (usually a symbolic name) referring to the address of the first instruction of the subroutine. The current value of the instruction pointer (`rip`) is stored to the top of the stack, the stack pointer (`rsp`) is decreased by 8, and then the `rip` is overwritten by the address given as the operand.

`cbw`

Convert byte (8-bit value) to word (16-bit value) using sign extension. No operands are explicitly given. The `ah` register is overwritten by the copies of the most significant bit of `al` register. It is a kind of type conversion from `char` to `short`.

`cdq`

Convert double word (32-bit value) to quad word (practically two 32-bit value) using sign extension. No operands are explicitly given. The `edx` register is overwritten by the copies of the most significant bit of `eax` register, so the most significant 4 bytes of the result are stored

in `edx` and the least significant 4 bytes of the result remain in the `eax` register. It is a kind of type conversion from `int` to `long`.

`cdqe`

Convert double word (32-bit value) to quad word (64-bit value) using sign extension. No operands are explicitly given. The most significant 4 bytes of `rax` register are overwritten by the copies of the most significant bit of `eax` register. It is a kind of type conversion from `int` to `long`.

`cmp`

Compares the values of the two operands, so it checks whether they are equal or the first operand is greater or less than the second in both interpretations (signed and unsigned fixed-point) separately. (Practically the second operand is subtracted from the first, but the difference is discarded.) None of the operands change. Based on the comparison/subtraction the `SF`, `ZF`, `CF` and `OF` status register bits are updated according to the following table.

cmp Op1, Op2		ZF	CF	SF	OF
unsigned	Op1 < Op2	0	1		
signed				1	0
Op1 == Op2		1			
unsigned	Op1 > Op2	0	0		
signed				0	0

All conditional jump (j_{cc}) instructions use these status register bits (usually set by the `cmp` instruction).

`cwd`

Convert word (16-bit value) to double word (32-bit value) using sign extension. No operands are explicitly given. The most significant 2 bytes of the `eax` register are overwritten by the copies of the most significant bit of `ax` register. It is a kind of type conversion from `short` to `int`.

`dec`

Decrementation. It has only one operand and its value is decreased by 1 during the execution. The `SF`, `ZF` and `OF` status register bits are updated according to the result.

`div`

Unsigned integer division. It has only one (!) operand the divisor, the dividend is not given explicitly. If the divisor/operand is a 32-bit value, then the dividend is the `edx:eax` register combination (i.e. it is only one 64-bit value, where the most significant 4 bytes are stored in the `edx` register and the least significant 4 bytes are in the `eax` register separately). After the integer division, the 32-bit quotient (truncated toward 0) overwrites the `eax` register content and the 32-bit reminder updates the `edx` register. Unsigned fixed-point interpretation is used, so the values cannot be interpreted as negative. The `SF`, `ZF`, `CF` and `OF` status register bits are unaffected.

`idiv`

Signed integer division. It has only one (!) operand the divisor, the dividend is not given explicitly. If the divisor/operand is a 32-bit value, then the dividend is the `rdx:rax` register

combination (i.e. it is only one 64-bit value, where the most significant 4 bytes are stored in the `edx` register and the least significant 4 bytes are in the `eax` register separately). After the integer division, the 32-bit quotient (truncated toward 0) overwrites the `eax` register content and the 32-bit reminder updates the `edx` register. Signed fixed-point interpretation is used, so the values can also be interpreted negative. The `SF`, `ZF`, `CF` and `OF` status register bits are unaffected.

`imul`

Signed multiplication of two values. Now only the 2-operand form is explained (however there are 1-operand and 3-operand forms as well). The product overwrites the destination operand. The `CF` and `OF` flags are set when the result must be truncated to fit in the destination operand size.

`inc`

Incrementation. It has only one operand and its value is increased by 1 during the execution. The `SF`, `ZF` and `OF` status register bits are updated according to the result.

`ja`

Jump if above. It has an operand which is a memory address (practically a symbolic name used somewhere as a label of an instruction). It checks the status register bits (which are probably updated by the previous `cmp` instruction). If `CF` is 0 and `ZF` is 0, then it overwrites the content of the instruction pointer (`rip`) by the address specified by the operand, else the `rip` just simply incremented by the length of this instruction. Thus, the execution either jumps to another part of the code or just goes on to the next instruction. The condition is equivalent to the status register content if a `cmp` instruction realizes that its first operand is above the second one using unsigned interpretation. Status register bits are not changed. (It has an alias: `jnbe`.)

`jae`

Jump if above or equal. It has an operand which is a memory address (practically a symbolic name used somewhere as a label of an instruction). It checks the status register bits (which are probably updated by the previous `cmp` instruction). If `CF` is 0, then it overwrites the content of the instruction pointer (`rip`) by the address specified by the operand, else the `rip` just simply incremented by the length of this instruction. Thus, the execution either jumps to another part of the code or just goes on to the next instruction. The condition is equivalent to the status register content if a `cmp` instruction realizes that its first operand is above or equal to the second one using unsigned interpretation. Status register bits are not changed. (It has an alias: `jnb`.)

`jb`

Jump if below. It has an operand which is a memory address (practically a symbolic name used somewhere as a label of an instruction). It checks the status register bits (which are probably updated by the previous `cmp` instruction). If `CF` is 1, then it overwrites the content of the instruction pointer (`rip`) by the address specified by the operand, else the `rip` just simply incremented by the length of this instruction. Thus, the execution either jumps to another part of the code or just goes on to the next instruction. The condition is equivalent to the status register content if a `cmp` instruction realizes that its first operand is below the second one using unsigned interpretation. Status register bits are not changed. (It has an alias: `jnae`.)

`jbe`

Jump if below or equal. It has an operand which is a memory address (practically a symbolic name used somewhere as a label of an instruction). It checks the status register bits (which are probably updated by the previous `cmp` instruction). If `CF` is 1 or `ZF` is 1, then it overwrites the content of the instruction pointer (`rip`) by the address specified by the operand, else the `rip` just simply incremented by the length of this instruction. Thus, the execution either jumps to another part of the code or just goes on to the next instruction. The condition is equivalent to the status register content if a `cmp` instruction realizes that its first operand is below or equal to the second one using unsigned interpretation. Status register bits are not changed. (It has an alias: `jna`.)

`jc`

Jump if carry. It has an operand which is a memory address (practically a symbolic name used somewhere as a label of an instruction). It checks the status register bits. If `CF` is 1, then it overwrites the content of the instruction pointer (`rip`) by the address specified by the operand, else the `rip` just simply incremented by the length of this instruction. It is an alias for the `jb` instruction. Status register bits are not changed. (The `jz`, `jo` and `js` conditional jump instructions are similar to this just the `ZF`, `OF` and `SF` are checked respectively.)

`je`

Jump if equal. It has an operand which is a memory address (practically a symbolic name used somewhere as a label of an instruction). It checks the status register bits (which are probably updated by the previous `cmp` instruction). If `ZF` is 1, then it overwrites the content of the instruction pointer (`rip`) by the address specified by the operand, else the `rip` just simply incremented by the length of this instruction. Thus, the execution either jumps to another part of the code or just goes on to the next instruction. The condition is equivalent to the status register content if a `cmp` instruction realizes that its first operand is equal to the second one. Status register bits are not changed.

`jg`

Jump if greater. It has an operand which is a memory address (practically a symbolic name used somewhere as a label of an instruction). It checks the status register bits (which are probably updated by the previous `cmp` instruction). If `ZF` is 0 and `OF` is equal to `SF`, then it overwrites the content of the instruction pointer (`rip`) by the address specified by the operand, else the `rip` just simply incremented by the length of this instruction. Thus, the execution either jumps to another part of the code or just goes on to the next instruction. The condition is equivalent to the status register content if a `cmp` instruction realizes that its first operand is greater than the second one using signed interpretation. Status register bits are not changed. (It has an alias: `jnle`.)

`jge`

Jump if greater or equal. It has an operand which is a memory address (practically a symbolic name used somewhere as a label of an instruction). It checks the status register bits (which are probably updated by the previous `cmp` instruction). If `OF` is equal to `SF`, then it overwrites the content of the instruction pointer (`rip`) by the address specified by the operand, else the `rip` just simply incremented by the length of this instruction. Thus, the execution either jumps to another part of the code or just goes on to the next instruction. The condition is

equivalent to the status register content if a `cmp` instruction realizes that its first operand is greater than or equal to the second one using signed interpretation. Status register bits are not changed. (It has an alias: `jnl`.)

`jl`

Jump if less. It has an operand which is a memory address (practically a symbolic name used somewhere as a label of an instruction). It checks the status register bits (which are probably updated by the previous `cmp` instruction). If `OF` is not equal to `SF`, then it overwrites the content of the instruction pointer (`rip`) by the address specified by the operand, else the `rip` just simply incremented by the length of this instruction. Thus, the execution either jumps to another part of the code or just goes on to the next instruction. The condition is equivalent to the status register content if a `cmp` instruction realizes that its first operand is less than the second one using signed interpretation. Status register bits are not changed. (It has an alias: `jnge`.)

`jle`

Jump if less or equal. It has an operand which is a memory address (practically a symbolic name used somewhere as a label of an instruction). It checks the status register bits (which are probably updated by the previous `cmp` instruction). If `ZF` is 1 and `OF` is not equal to `SF`, then it overwrites the content of the instruction pointer (`rip`) by the address specified by the operand, else the `rip` just simply incremented by the length of this instruction. Thus, the execution either jumps to another part of the code or just goes on to the next instruction. The condition is equivalent to the status register content if a `cmp` instruction realizes that its first operand is less than or equal to the second one using signed interpretation. Status register bits are not changed. (It has an alias: `jng`.)

`jmp`

Unconditional jump. Transfers program control to a different point in the instruction stream, so it overwrites the content of the instruction pointer (`rip`) by the address specified by the operand (which is practically a symbolic name used somewhere as a label of an instruction).

`jnc`

Jump if not carry. It has an operand which is a memory address (practically a symbolic name used somewhere as a label of an instruction). It checks the status register bits. If `CF` is 0, then it overwrites the content of the instruction pointer (`rip`) by the address specified by the operand, else the `rip` just simply incremented by the length of this instruction. It is an alias for the `jnb` instruction. Status register bits are not changed. (The `jnz`, `jno` and `jns` conditional jump instructions are similar to this just the `ZF`, `OF` and `SF` are checked respectively.)

`jne`

Jump if not equal. It has an operand which is a memory address (practically a symbolic name used somewhere as a label of an instruction). It checks the status register bits (which are probably updated by the previous `cmp` instruction). If `ZF` is 0, then it overwrites the content of the instruction pointer (`rip`) by the address specified by the operand, else the `rip` just simply incremented by the length of this instruction. Thus, the execution either jumps to another part of the code or just goes on to the next instruction. The condition is equivalent to the status register content if a `cmp` instruction realizes that its first operand is not equal to the second one. Status register bits are not changed.

`lea`

Load effective address instruction which computes the address given in the second operand (specified with one of the addressing modes) and stores it in the first operand (a register). Formally, the second operand looks like a memory content operand, but the representation length is missing so only the address part is present between square brackets. Status register bits are not affected.

`leave`

It is used at the end of subroutines. It has no operands. It is equivalent to the following two consecutive instructions: `mov rbp, rsp` and `pop rbp`.

`loop`

It organizes an iteration using the `rcx` register as a counter. It has one operand, a memory address (practically given as symbolic name which is used as a label somewhere else). Each time the `loop` instruction is executed, the `rcx` register is decremented and then checked for 0. If the count is 0, the loop is terminated, and program execution continues with the instruction following the `loop` instruction. If the count is not zero, a jump is performed to the address specified in the destination operand, which is presumably the instruction at the beginning of the loop.

`mov`

Data transfer instruction, which copies the second operand (source operand) to the first operand (destination operand). The operands must have the same size. Status register bits are not affected.

`movsx`

Move with sign extension. This instruction has two operands, but their size is not the same namely the destination operand has more bits than the source operand. It copies the content of the second operand to the least significant part of the first operand and the remaining bits of the destination register are filled by the copies of the most significant bit (sign bit) of the source operand. Practically it results in the same signed value represented on more bits. Status register bits are not affected.

`movzx`

Move with zero extension. This instruction has two operands, but their size is not the same namely the destination operand has more bits than the source operand. It copies the content of the second operand to the least significant part of the first operand and the remaining bits of the destination register are filled by 0 bits. Practically it results in the same unsigned value represented on more bits. Status register bits are not affected.

`mul`

Unsigned multiplication of two values. It has only 1 operand (!). If the operand is a 32-bit value, then the value of the `eax` register is multiplied by the value of the explicit operand and the 64-bit product is stored into the `edx`**:**`eax` register combination (i.e. the most significant 4 bytes of the product are stored in the `edx` register and the least significant 4 bytes of the result go into the `eax` register separately). The `CF` and `OF` flags are set to 0 if the upper half of the result is 0 (so when the product is no longer than 32 bits, so `edx` is not really used) ; otherwise, they are set to 1.

`neg`

Negation of the only one operand. It replaces the value of the operand (the destination operand) with its two's complement, so its effect is similar to a multiplication by -1.

`nop`

No operation. It does not do any useful thing (just cause some delay). It can be used as a placeholder as well. It has no effect to the general-purpose registers or to the status register.

`not`

Bitwise negation (ones-complement). It has one operand, and it changes all the bits of the operand to the opposite.

`or`

Bitwise OR operation on the operands. All bit positions of the operands are considered simultaneously according to the truth table of the logical OR operation. The result overwrites the destination operand. The `OF` and `CF` flags are cleared; the `SF` and `ZF` flags are set according to the result.

`pop`

Moves the top value out of the stack. It reads the memory address referred by the `rsp` register and this data is stored in the only one 64-bit operand then the value of the `rsp` register is incremented by 8.

`push`

Moves the 64-bit value of the only one operand to the top of the stack, i.e. the value of the `rsp` register is first decremented by 8 and then to this address of the memory the system writes the content of the operand. The operand itself does not change.

`ret`

Return from subroutines. It has no operand (so the return value is not an operand). From the memory location addressed by current value of the `rsp` register, 64-bit data (a return address) is loaded to the program counter (`rip`) and then the stack pointer is increased by 8.

`rol`

Rotate left. It has 2 operands. It behaves similarly to the `shl` instruction, but the shifted out most significant bits periodically go to the “empty” least significant bit positions of the destination register.

`ror`

Rotate right. It has 2 operands. It behaves similarly to the `shr` instruction, but the shifted out least significant bits periodically go to the “empty” most significant bit positions of the destination register.

`sal`

Shift arithmetic left. The bits of the first operand are moving toward left by the number of positions specified by the second operand. The new “empty” positions of the result are filled by 0 bits. The previously most significant bits of the first operand are lost except the last (smallest position) one which is stored into the `CF` status register bits. It is the same as the `shl` instruction.

`sar`

Shift arithmetic right (sign-keeping right shift). The bits of the first operand are moving toward right by the number of positions specified by the second operand. The new "empty" positions of the result are filled by copies of the most significant bit of the original value (i.e. in case of negative value 1 bits are set else 0 bits). The previously least significant bits of the first operand are lost except the last (largest position) one which is stored into the `CF` status register bits.

`seta, setb, setbe, setc, sete, setea, setg, setge, setl, setle, setnc, setne, setnz, setz`

Set byte on condition. This group of instructions (often referred as $\texttt{set}_{\texttt{cc}}$) has only one 8-bit operand. The operand is set to either `00000000` (false) or `00000001` (true) bit sequence based on a condition. The type of condition is indicated by the end of the mnemonic like the conditional jump ($\texttt{j}_{\texttt{cc}}$) instructions. The decision is made according to the `SF`, `ZF`, `CF`, and `OF` status register bits (probably set by the previous `cmp` instruction). Flags are not modified.

`shl`

Shift (logic) left. The bits of the first operand are moving toward left by the number of positions specified by the second operand. The new "empty" positions of the result are filled by 0 bits. The previously most significant bits of the first operand are lost except the last (smallest) one which is stored into the `CF` status register bits. It is the same as the `sal` instruction.

`shr`

Shift (logic) right. The bits of the first operand are moving toward right by the number of positions specified by the second operand. The new "empty" positions of the result are filled by 0 bits. The previously least significant bits of the first operand are lost except the last (largest position) one which is stored into the `CF` status register bits.

`sub`

Subtraction of the values of the operands (i.e. the value of the second operand is subtracted from the value of the first operand. The difference overwrites the destination operand. The operands and the result can be interpreted as both signed and unsigned fixed-point values. The `SF`, `ZF`, `CF` and `OF` status register bits are updated according to the result.

`test`

Logical compare. It computes the bitwise logical AND of its two operands and sets the `SF` and `ZF` status flags according to the result. The result itself of the AND operation is discarded.

`xchg`

Exchange operands. It has 2 operands which are swapped, so the value of the first operand is stored in the second operand and simultaneously the original second operand overwrites the first operand.

`xor`

Bitwise XOR operation on the operands. All bit positions of the operands are considered simultaneously according to the truth table of the logical XOR operation. The result overwrites

the destination operand. The `OF` and `CF` flags are cleared; the `SF` and `ZF` flags are set according to the result.

5. Calling conventions

Programmers must use some conventions to be able to write codes compatible with the codes of other programmers. It describes how to call subroutines, how to pass parameters, how to give back a return value to the caller, how to implement local (dynamic lifetime) variable or how to implement a recursion.

First, the values of the parameters must be prepared to pass before the invocation of a subroutine. If there are not more than 6 parameters, all of them are passed via registers. The value of the first fixed-point parameter must be loaded to the `rdi`, the second goes to the `rsi`, the third goes to the `rdx`, the fourth goes to the `rcx`, the fifth to the `r8,` and the sixth to `r9`. If there are further fixed-point parameters, they must be stored in the stack. If the subroutine has real number parameters the first floating point parameter must be stored into the least significant single part of the `xmm0` vector register, the second real parameter to the `xmm1`, and so forth. For those subroutines where the parameter list is not fixed, the number of floating point parameters must be saved into the `eax` register before the call. Then the caller can invoke the subroutine (procedure of function) by the `call` assembly instruction. This saves the return address (i.e. the current content of the rip) to the stack (rsp is decremented by 8) then it sets the `rip` to the address of the first instruction of the callee. At the beginning of the subroutine, the content of the `rbp` register must be pushed to the stack, then the content of the stack pointer (`rsp`) is copied to the base pointer (`rbp`) and after this, the `rsp` is decreased by the total number of bytes needed as local variables. Then the values of parameters (if any) are copied from an intermediate register (and stack) to local variables. After all the necessary instructions of the subroutine are done a leaving process is started. If the subroutine is a function with a fixed-point return value, then it must be stored in the proper part of the `rax` register before return. When the return value is a floating point value the `xmm0` register must store the return value. The recovery of the stack and registers starts by the `mov rsp, rbp` instruction which is followed by a `pop rbp`. The last instruction of the callee is a ret, which pops the return address from the stack to the `rip` (and increments the `rsp` by 8). After the control is given back to the caller it removes parameters from the stack if any.

6. Example

There are two sample codes below. One of them is a C language program code, the other is an x86-64 assembly code (for Linux, gcc). They do the same. Comparing them can help to understand how to write real assembly code if you know how to write it in a high-level language. Please, read them carefully.

The program contains 2 functions. The `square` function has an integer parameter (which is one of the local variables), and it returns by the square of this value. The `main` program unit has a loop to initialize an array with small integer square numbers and then it prints the last one to the screen. The assembly code can be much more optimized but in this form they are totally equivalent.

```
/* C program: loop, array, function, etc. */
#include<stdio.h>
int square(int Num){
  int N2;
  N2 = Num*Num;
  return N2;
}
int main(){
  int i = 0, S[10];
  while(i<10){
    S[i] = square(i);
    i++;
    }
  printf("Last: %d\n",S[9]);
  return 0;
}
```

```
# x86-64 assembly program: loop, array, function, etc.
        .intel_syntax noprefix
        .globl    square
square: push      rbp
        mov       rbp, rsp
        sub       rsp, 4              # Space for local variables
        mov       DWORD PTR [rbp-20], edi  # Save parameter to RAM
        mov       eax, DWORD PTR [rbp-20]
        imul      eax, eax                 # Calculate the square
        mov       DWORD PTR [rbp-4], eax   # Save into variable
        mov       eax, DWORD PTR [rbp-4]   # Prepare return value
        mov       rsp, rbp
        pop       rbp
        ret
        .globl    main
main:   push      rbp
        mov       rbp, rsp
        sub       rsp, 64             # Space for local variables
        mov       DWORD PTR [rbp-44], 0
        jmp       .L4
.L5:    mov       edi, DWORD PTR [rbp-44]  #Passing the parameter
        call      square                   # Invocation
        movsx     rdx, DWORD PTR [rbp-44]
        mov       DWORD PTR [rbp+rdx*4-40], eax
        add       DWORD PTR [rbp-44], 1    # Incrementation of i
.L4:    cmp       DWORD PTR [rbp-44], 9    # Condition evaluation
        jle       .L5                      # Go on or go back
        mov       esi, DWORD PTR [rbp-4]   # First parameter
        lea       rdi, [.LC0]              # Second parameter
        mov       eax, 0                   # No float parameters
        call      printf                   # Built-in output
        mov       eax, 0                   # EXIT_SUCCESS
.L7:    mov       rsp, rbp
        pop       rbp
        ret
        .section  .rodata
.LC0:   .string   "Last: %d\n"             # Format string
```

References

1. *x86 and amd64 instruction reference* https://www.felixcloutier.com/x86/ (2023).
2. *Intel® 64 and IA-32 Architectures Software Developer's Manual* https://irh.inf.unideb.hu/~vargai/download/assembly/Intel.pdf (Intel, 2021).
3. Ray Seyfarth: *Introduction to 64 bit assembly programming for Linux and OS X*, (Amazon, 2013).
4. R. E. Bryant, D. R. O'Hallaron: *Computer Systems - A programmer's perspective* (Pearson, 2016).
5. Joseph Cavanagh: *X86 Assembly Language and C Fundamentals*, (CRC Press, 2013).
6. Richard Blum: *Professional Assemby Language*, (Wiley, 2005).