EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

VŠB - Technical University of Ostrava
Department of Computer Science, FEECS

x86 Assembly Language

Syllabus for Subject:

Assembly (Machine) Language

Ing. Petr Olivka, Ph.D.

2021

e-mail: petr.olivka@vsb.cz
http://poli.cs.vsb.cz

Contents

1 Processor Intel i486 and Higher – 32-bit Mode

Processors (CPU) Intel i486 and higher are in technical literature very well documented. The documentation is however very detailed and it contains many unnecessary information not only for experienced programmers, but especially for beginners.

From the historical point of view do not have sense to describe progressive evolution of processors before the i486. This processor is herein considered as the starting version.

1.1 Registers of i486

The overall overview of registers is visible in Figure 1.

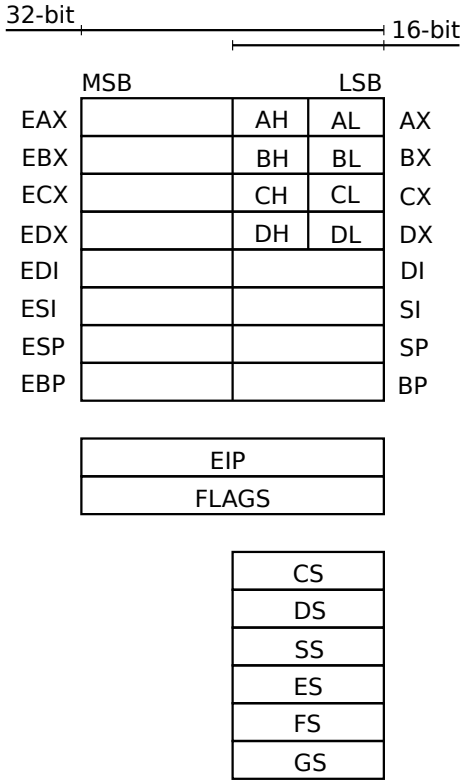

Figure 1: Registers of i486

The processor i486 contains eight basic registers of 32-bit for universal use. Furthermore, 6 segment registers, a status register (flags) and program counter (instruction pointer).

32-bit registers for the universal use:

`EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP`.

16-bit registers:

above mentioned registers allow access to its lower 16-bit part via: `AX, BX, CX, DX, SI, DI, BP, SP`.

8-bit registers:

first four 16-bit registers are split to the upper and the lower 8-bit parts: `AH` (high), `AL` (low), `BH, BL, CH, CL, DH, DL`.

16-bit segment registers:

DS (data), ES (extra), CS (code), SS (stack), FS, GS.

Program counter (instruction pointer) EIP (IP):

points to a currently executed instruction. The EIP register is never changed directly but only by jump instructions.

Status register FLAGS:

contains status bits. Bits, which a programmer should know, are: ZF (zero flag), CF (carry), OF (overflow), SF (sign), DF (direction). In addition to these bits also PF (parity flag) and AF (auxiliary flag) is implemented.

CF – Carry flag – Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for **unsigned-integer** arithmetic. It is also used in multiple-precision arithmetic.

OF – Overflow flag – Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for **signed-integer** (two's complement) arithmetic.

SF – Sign flag – Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. 0 indicates a positive value and 1 indicates a negative value.

ZF – Zero flag – Set if the result is zero; cleared otherwise.

Registers are in some cases bound to some purpose. A few examples for completeness:

- EAX, AX, AL is a accumulator. It is used for multiplication, division and in string instructions.

- ECX is used as a counter.
- CL is used as a counter in bit shift instructions.
- EDX, DX is the upper part of a dividend in division instructions.
- ESI and EDI are used as an index register in string instructions.

1.2 Addressing

The addressing is divided into 16-bit and 32-bit mode. The addressing is also divided to a direct and indirect addressing. The direct addressing is always provided by the specific direct (fixed) address. The example of direct address is an use of variable. But a programmer does not need in the practice only the direct addresses. In case when address is given by a variable or defined in run-time, it is necessary to use indirect addressing by registers.

The form of 16-bit address mode is:

`[ Base_Reg + Index_Reg + Constant ]`,

where SI and DI can be used as index register and BX and BP can be used as base register. It is possible to combine one index and one base register and it is also possible to use them individually.

In 32-bit mode is addressing mode versatile:

`[ Base_Reg + Index_Reg * Scale + Constant ]`.

where all eight 32-bit registers can be used as base and index register. The `Scale` can be one of four values: 1, 2, 4 and 8. It is used to simplify the addressing of array, where scale is size of an element type.

1.3 Assembly Language, Machine Code

The machine code is a binary representation of machine instructions executed by a processor. Nowadays nevertheless no programmer writes programs directly in machine code, but he writes them in "Assembly language" that is used to write a program in text form with symbolic names of individual instructions and registers. The Assembly language is then translated to the machine code by a compiler.

1.4 Data Types

Data types in Assembly language specify only the amount of allocated memory. Nowhere it says anything about type of variable neither it can be used for type checking (integer of float, signed or unsigned, ...). The programmer is responsible for everything.

- DB - data BYTE (8 bits)
- DW - data WORD (16 bits)
- DD - data DWORD (32 bits)
- DQ - data QWORD (64 bits)
- DT - data TBYTE (80 bits)

2 Linking Assembly and C Language Programs

Nowadays, it is not standard to write the entire programs in the Assembly language. The Assembly language is used to program only some important parts of a program. The main parts of a program is usually written in some high level programming language.

Herein will be used for programming the main program the C (later C++) language. This language will be used for data declaration and initialization and subsequent output of results.

2.1 Linking C and C Module

Before we show the linking of C and ASM modules, the example of linking of two C modules will be introduced. It is suitable at first to repeat basic principles of linking more modules in C language. The following example will be composed from two source modules: `main.c` and `modul.c`. At first `modul.c`:

```
// c-module.c
int  g_module_sum ;
static int  g_module_counter  = 0;

static int  inc_counter ()
{  m_counter ++; }

int  inc_sum ()
{  g_module_sum ++;  inc_counter (); }
```

The next module is the main program:

```
// c-main.c
// external  function  prototypes
int  inc_sum ();
int  inc_counter ();
// external  variables
extern  int  g_modul_sum ;
extern  int  g_modul_counter ;

int  main () {
     g_modul_sum  = 0;
     inc_sum ();
     // g_modul_counter  = 0; // impossible
     // inc_counter ();        // impossible
     printf ( "sum␣%d\n",  g_modul_sum  );
     //printf ( "counter  %d\n",  g_modul_counter  );
}
```

At first it is necessary to remember the difference between functions and variables which use in their declaration keyword `static`. In the file `modul.c` is in this way declared variable `m_sum` and function `inc_sum`. These declared functions and variables are valid (visible) only in the source file where they are declared. They are not published and they can not be used from another modules.

The rest of variable and function identifiers are public.

If there is a need to use functions and variables from another module, e.g. in our case `main.c` will use identifiers from `modul.c`, it is necessary to specify their prototype. The correct way is to define this prototypes in a header file. Herein in the simple example will be those prototypes defined at the beginning of the source code.

Prototypes of functions `inc_counter()` and `inc_sum()` are defined at begin of `main.c` source file. These prototypes say that these functions are external.

In the same way it is possible to define prototypes of external variables `m_counter` and `m_sum`. But the definition of external variable must start with keyword `extern`. This keyword is however optional for function prototypes. The compiler can easy recognize prototype of function and function implementation. After a close bracket in function head is the semicolon or the open curtly bracket.

The code of function `main` has access to all variables and it can call all functions that are known. The compiler will compile this code. The problem will be in next step in linking stage, when linker will try create an executable program. The linker will not found symbols which are defined as external, but nowhere they are defined as public. The final program can not be completed. All required symbols which are not public will be marked as unknown. Therefore a few lines in function `main` are commented out to avoid this problem.

Thus it is necessary to keep in mind three main rules for linking executable program from more modules. These three main rules are defined in every programming language for functions and variables:

- public symbols,
- local (private) symbols,
- external symbols.

The rules for C language were described above.

2.2 Linking C and ASM Module

The following code example `module_32.asm` is equivalent Assembly language module to previous module `module.c`.

```
    ; asm-module.asm
    bits 32                         ; 32 bit code
    section .data                   ; data section
    global g_module_counter         ; public symbol
g_module_counter    dd 0            ; variable g_module_couter
g_module_sum        dd 0            ; variable g_module_sum

    section .text                   ; code section
    global inc_sum                  ; public symbol

inc_counter:                        ; function inc_counter
    inc dword [ g_module_counter ]; g_module_counter ++
    ret

inc_sum:                            ; function inc_sum
    inc dword [ g_module_sum ]      ; g_module_sum ++
    call inc_counter                ; inc_counter ()
    ret
```

The every Assembly language code must be composed from two sections: the data and the code. In the data section is visible the use of data types from previous chapter for declaration of variables `m_counter` and `m_sum`. All symbols in Assembly language are automatically local. Thus to publish variable `m_counter` it is necessary to use keyword `global`.

The similar situation is in the code section. Functions are defined in code via their named labels. All symbols are automatically local and to publish them it is necessary to use keyword `global`.

The previous short Assembly language example shows that comments in assembly code begins with semicolon. The instructions are written in code indented from left edge and on the left edge are defined labels and names of variables.

More detail about writing assembly code and its syntax is described in technical documentation. The most important is to know correct form of numbers, characters and string constants.

More examples of linking Assembly language with C language are in archives `soj-c-c.tgz` and `soj-c-asm.tgz`. All programs can be build by command `make`.

2.3 Variables in Assembly Language

At least one instruction must be introduced to be possible explain the use of variables. The simplest instruction is MOV for moving data:

```
MOV destination, source          ; destination = source .
```

Register (R), memory (M) and constant (C) can be used as parameters of this instruction. These three types of parameters can be used in 5 different combinations:

```
MOV R, R                ; move data from register to register
MOV R, M                ; move content of memory to register
MOV M, R                ; move content of register to memory
MOV R, C                ; move constant to register
MOV M, C                ; move constant to memory
```

In all five cases the following rules must be observed:

- the size of both operands must be the same,
- never can be used two operands from memory,
- in first four cases where is one operand R, the size of operand is defined by an used register,
- in last case without register the size of operand must be defined manually by a programmer using known types (see chapter 1.4): byte, word, dword, etc.

The following example will demonstrate the use of global variables declared globally in C and in Assembly language. At first the main program in C language:

Listing 1: Main program in C

```
// c-main.c
// public global variables
int g_c_number;
char g_c_char;
int g_c_iarray[10];
char g_c_text[] = "String␣declared␣in␣C\n";

// external variables
extern int g_a_counter;
extern char g_a_byte;
extern int g_a_numbers[];
extern char g_a_str[];
```

```
// external function
void changes();

int main()
{
    changes();
    // printf selected variables ...
}
```

The example in Listing 2 shows the use of global variables in Assembly language. The access to variables declared in C or in Assembly language is exactly the same.

Listing 2: Usage of variables in Assembly language

```
    ; Example of using variables in Assembly language
    bits 32
    section .data
    ; external variables
    extern g_c_number, g_c_char, g_c_iarray, g_c_text

    ; list of public symbols
    global g_a_counter, g_a_byte, g_a_numbers, g_a_str

g_a_counter dd 0                    ; int
g_a_byte    db 0                    ; char
g_a_numbers dd 0,0,0,0,0            ; int[ 5 ]
    ; following string must be terminated by '\0'
g_a_str   db 'Text␣defined␣in␣ASM', 10, 0
    section .text
    global changes
changes:
    ; integer numbers
    mov eax, [ g_c_number ]         ; eax = g_c_number
    mov [ g_a_counter ], eax        ; g_a_counter = eax
    mov dword [ g_c_number ], 0     ; g_c_number = 0

    ; characters
    mov byte [ g_a_byte ], 13       ; g_a_byte = 13
    mov dl, [ g_c_char ]            ; dl = g_c_char

    ; array elements access
    mov ecx, [ g_c_iarray + 2 * 4]; ecx = g_c_iarray[ 2 ]
    mov edx, 3
    mov [ g_a_numbers + edx * 4 ], ecx
                                    ; g_a_numbers[ edx ] = ecx
    ; access to characters in string
    mov dh, [ g_c_text ]            ; dh = g_c_text[ 0 ]
```

```
    mov eax, 5
    mov [ g_a_str + eax ], dh      ; g_a_str[ eax ] = dh
    ret
```

In this example is also visible the use of addressing introduced in chapter 1.2. The every memory argument must be closed in square bracket `[]`, thus also the variables must be closed in brackets. The content of brackets is address of memory where is required value.

The every variable name is internally represented as a address, thus the use of variables is represented as variable closed in brackets.

According to information in chapter 1.2 it is also possible to use in brackets up to two registers which allow set address dynamically. In most cases registers are used to access elements of array. One register, formally called index register, can be extended by scale corresponding to size of element of array. It is visible in the example above when arrays `a_numbers` and `c_iarray` are used.

Results of executed code can be printed in function `main` using `printf`.

3 Instruction Set

The instruction set of i486 ALU is larger than the limited set of instruction used in practice, it is smaller then half of the whole. All instructions are very well documented in technical literature and in datasheets, but in most of them are instruction organized in alphabetical order.

Herein the first introduction will be organized thematically. Subset of instruction will be organized to a few groups:

- moving,
- logical and bitwise,
- arithmetical,
- jumps,
- strings,
- control and auxiliary.

Detailed description of instructions is in Intel's datasheets. They are in appendix as documents `intel-AZ.pdf`. This documentation has more than two thousands pages and for a routine work it is not very suitable. Much less for beginners.

A suitable documentation for learning is the document `nasm-0.98.pdf` and its appendix B. Its content is suitable for beginners. Instructions are there ordered in alphabetical order and every instruction is briefly and sufficiently described.

The following instruction description is for better clarity organized thematically. Main aim of following description is the selection of necessary instruction, not detailed description of them. So, the every instruction is described only shortly.

3.1 Moving Instruction

`MOV dest, src`

The instruction moves content of the `src` operand to the `dest` operand. The size of both operands must be the same. It is possible to use registers, memory and constant. See description at begin of this chapter 2.3.

`CMOVcc dest, src`

The instruction moves content of the `src` operand to the `dest` operand when the condition `cc` is true. The meaning of this abbreviation is explained below in conditional jumps. The size of both operands must

be the same. The argument `dest` must be register, the `src` can be register or memory.

`MOVZX dest, src`

This instruction also moves content of the `src` operand to the `dest`. But the size of `src` argument must be less then size of the `dest`. The `src` argument will be extended by zero bits.

`MOVSX dest, src`

This instruction works in a similar way as `MOVZX`. The only difference is that the signed extension of the operand `src` is stored in the `dest`.

`XCHG dest, src`

Instruction exchanges content of the both operands.

`BSWAP dest`

The instruction reorder bytes in operand. The little endian form of number is changed to the little endian form and vice versa.

`XLATB`

The instruction is used for translation table: `AL = [ EBX + AL ]`. Into register AL is moved AL^{th} byte from array at EBX address.

`LEA dest, src`

Into the `dest` operand will be stored addres of the operand `src`.

`LDS dest, src`

Content of memory at address `src` will be stored into operand `dest` and register DS.

`LES, LSS, LFS, LGS`

The same as LDS, but registers ES, SS, FS, GS will be used.

`PUSH src`

The instruction stores content of operand `src` onto the top of stack. Top of stack EPS is moved up.

`POP dest`

The value from the top of stack is stored into the operand `dest`. Top of stack EPS is moved down.

`PUSHA`

The instruction stores all 8 registers onto the top of stack.

`POPA`

The instruction load content of 8 registers from the top of stack.

`PUSHF`

The instruction stores content of FLAGS register onto the top of stack.

`POPF`

The value from the top of stack is loaded into FLAGS register.

`LAHF`

The instruction moves lower 8 bits from the FLAGS register into the AH register.

`SAHF`

The instruction moves content of the register AH into lower 8 bits of the FLAGS register.

`IN accumulator, address`

This instruction load data from port at an address `address` into accumulator AL, AX, EAX. For the address less then 8 bit can be used constant, otherwise address must be in register DX.

`OUT address, accumulator`

The complementary instruction to the instruction IN.

3.2 Logical and Bitwise Instruction

The every instruction in this group change some bits in FLAGS register. The logical instruction changes flags SF and ZF, the bit-shift instruction also flag CF.

`AND dest, src`

The instruction performs bitwise AND: `dest = dest and src`.

`TEST dest, src`

The instruction performs bitwise AND, but result is not stored: `dest and src`. The only FLAGS register is set.

`OR dest, src`

The instruction performs bitwise OR: `dest = dest or src`.

`XOR dest, src`

The instruction performs bitwise XOR: `dest = dest xor src`.

`NOT dest`

The instruction inverts all bits in the operand `dest`.

`SHL/SAL dest, num_bits`

This instruction is bitwise/arithmetical shift of bits to the left in operand `dest`. The operand `num_bits` can be constant or register CL. The last bit shifted out is stored in CF.

`SHR dest, num_bits`

This instruction is bitwise shift of bits to the right in operand `dest`. The operand `num_bits` can be constant or register CL. The last bit shifted out is stored in CF.

`SAR dest, num_bits`

This instruction is arithmetical shift of bits to the right in operand `dest`. The operand `num_bits` can be constant or register CL. The last bit shifted out is stored in CF.

`ROL dest, num_bits`

The instruction rotates all bits in operand `dest` to the left. The operand `num_bits` can be constant or register CL. The last rotated bit is stored in CF.

`ROR dest, num_bits`

The instruction rotates all bits in operand `dest` to the right. The operand `num_bits` can be constant or register CL. The last rotated bit is stored in CF.

`RCL dest, num_bits`

The instruction rotates all bits in operand `dest` through the CF to the left. The operand `num_bits` can be constant or register CL.

`RCR dest, num_bits`

The instruction rotates all bits in operand `dest` through the CF to the right. The operand `num_bits` can be constant or register CL.

`BT dest, bit_inx`

The instruction loads bit `bit_inx` from the operand `dest` and stores it into CF.

`BTR dest, bit_inx`

The instruction loads bit `bit_inx` from the operand `dest` and stores it into CF. The source bit in the operand `dest` is reset to 0.

`BTS dest, bit_inx`

The instruction loads bit `bit_inx` from the operand `dest` and stores it into CF. The source bit in the operand `dest` is set to 1.

`BTC dest, bit_inx`

The instruction loads bit `bit_inx` from the operand `dest` and stores it into CF. The source bit in the operand `dest` is inverted.

`SETcc dest`

This instruction set the operand `dest` to 0 or 1 according the condition `cc` (see conditional jumps below).

`SHRD/SHLD dest, src, num_bits`

The instruction shifts `num_bits` from the operand `src` into operand `dest`. For more, see documentation.

3.3 Arithmetical Instruction

All arithmetical instructions set bits in the FLAGS register.

`ADD dest, src`

The instruction performs arithmetical addition: `dest = dest + src`

`ADC dest, src`

The instruction performs arithmetical addition with carry (CF): `dest = dest + src + CF`

`SUB dest, src`

The instruction performs arithmetical subtraction: `dest = dest - src`

`CMP dest, src`

The instruction performs arithmetical subtraction: `dest - src`, but result is stored only to FLAGS.

`SBB dest, src`

The instruction perform arithmetical subtraction with borrow (CF): `dest = dest - src - CF`

`INC dest`

This instruction increment the operand `dest`. Instruction do not change CF.

`DEC dest`

This instruction decrement the operand `dest`. Instruction do not change CF.

`NEG dest`

This instruction invert (change sign) of the operand `dest`.

`MUL src`

The instruction perform arithmetical multiplication of two unsigned operands. The operand `src` is the second operand of multiplication. The size of operand `src` – (8, 16, 32)-bits – selects the proper first operand for multiplication from accumulator (AL, AX, EAX). The result is stored in registers (AX, AX-DX, EAX-EDX). See 3.7.

`IMUL src`

The instruction perform arithmetical multiplication of two signed operands. The operand `src` is the second operand of multiplication. The size of operand `src` – (8, 16, 32)-bits – selects proper first operand for multiplication from accumulator (AL, AX, EAX). The result is stored in registers (AX, AX-DX, EAX-EDX). See 3.7.

`DIV src`

This instruction performs arithmetical division of two unsigned numbers. The operand `src` is divisor. According the size of this operand – (8, 16, 32)-bits – must be prepared dividend in registers (AX, AX-DX, EAX-EDX). The result will be stored in registers (AL, AX, EAX) and remainder will be in (AH, DX, EDX). See 3.7.

`IDIV src`

This instruction performs arithmetical division of two signed numbers. The operand `src` is divisor. According the size of this operand – (8, 16, 32)-bits – must be prepared dividend in registers (AX, AX-DX, EAX-EDX). The result will be stored in registers (AL, AX, EAX) and remainder will be in (AH, DX, EDX). See 3.7.

`CBW`

This instruction performs the sign extension of the register AL into AX. It is used before IDIV.

`CWD`

This instruction performs the sign extension of the register AX into AX-DX. It is used before IDIV.

`CDQ`

This instruction performs the sign extension of the register EAX into EAX-EDX. It is used before IDIV.

`CQO` (64-bit mode)

This instruction performs the sign extension of the register RAX into RAX-RDX. It is used before IDIV.

3.4 Jump Instructions

`JMP dest`

The instruction perform jump to address specified by the operand `dest`. Usually this argument is label, but the address can be specified by register or by memory.

`CALL dest`

Call the function (subroutine). The behaviour of this instruction is similar to JMP, but this instruction stores on the top of stack address of next instruction.

`RET N`

This instruction perform return from function (subroutine). The instruction jumps to address stored on the top of stack and remove this address from stack.

The optional operand `N` specifies how many bytes will be removed from stack after return. It could be used for cleanup of arguments passed into a function.

`LOOP dest`

The behaviour of this instruction can be described by short C language code: `if ( --ECX ) goto dest;`. This instruction can be used to control loop and repeating of loop is specified by value in the register ECX. This register is decremented before testing!

`LOOPE/Z dest`

This instruction has similar behaviour as LOOP, but ZF flag is also tested: `if ( --ECX && ZF ) goto dest;`.

`LOOPNE/NZ dest`

This instruction is similar to LOOPE/Z: `if ( --ECX && !ZF ) goto dest;`.

`JCXZ dest`

This instruction test content of ECX register and perform jump to `dest` address when ECX is equal 0.

`Jcc dest`

Set of conditional jumps.

The first subset of conditional jumps directly test one flag in FLAGS:

Instructions `JZ/E, JNZ/NE, JS, JNS, JC, JNC, JO, JNO` test selected flag and jump to `dest` address when condition is true.

The second subset of conditional jumps is designed for a comparing of numbers:

`JB/JNAE/JC` - less than, not greater or equal,

`JNB/JAE/JNC` - not less, greater or equal,

`JBE/JNA` - less or equal, not greater,

`JNBE/JA` - not less or equal, greater,

`JL/JNGE` - less than, not greater or equal,

`JNL/JGE` - not less, greater or equal,

`JLE/JNG` - less or equal, not greater,

`JNLE/JG` - not less or equal, greater.

The letters in the instruction names **A-B-L-G-N-E** have the fixed meaning.

The comparison of unsigned numbers must use instructions with **A** (above) and **B** (below).

The comparison of two signed numbers must use instructions with **L** (less) and **G** (greater).

The negation is **N** (not) and equality is **E** (equal).

`INT number`

Interrupt number `number`.

`IRET`

Return from interrupt.

3.5 String Instructions

Every string instruction is tied to compulsory conventions use of index registers and the size of operands:

`ES:EDI` - destination operand.

`DS:ESI` - source operand.

`B/W/D` - operand size 1, 2 or 4 bytes. This value specifies change of index registers.

`DF` (direction flag) - 0 is up, 1 is down.

The every instruction can use one prefix to repeat itself (while condition is met):

```
REP: while (ECX) { ECX--; ... }
```

```
REPE/Z: while (ECX && ZF) { ECX--; ... }
```

```
REPNE/NZ: while (ECX && !ZF) { ECX--; ... }
```

The string instructions are only for moving and comparing.

`MOVSB/W/D`

This instruction move one element from source to destination. The use with a prefix can copy a block of memory.

`LODSB/W/D`

This instruction load one element from source into accumulator (AL, AX, EAX).

`STOSB/W/D`

This instruction stores content of accumulator (AL, AX, EAX) into destination. The prefix can be used to fill a block of memory.

`SCASB/W/D`

This instruction compares content of accumulator (AL, AX, EAX) with destination: `null=accumulator-ES:[EDI]`. The prefix can be used to search the required value or the first difference.

`CMPSB/W/D`

This instruction compares source and destination: `null=ES:[ESI]-DS:[EDI]`. The prefix can be used to search consensus or difference of strings or blocks of memory.

`INSB/W/D`

Instruction read data from port at address specified by DX into destination.

`OUTSB/W/D`

Instruction stores one element from source to port at address specified by DX.

3.6 Control and Auxiliary Instructions

`CLD`

The instruction reset flag DF to 0.

`STD`

The instruction set flag DF to 1.

`CLC`

The instruction reset CF to 0.

`STC`

The instruction set CF to 1.

`CMC`

The instruction invert (complement) CF.

`NOP`

No Operation.

`SETcc dest`

This instruction stores into operand `dest` value 0 or 1 according condition `cc`. Condition was explained in section with conditional jumps.

3.7 Multiplication and Division Instructions

The instructions for multiplication and division were described briefly in chapter 3.3. The principle of that instructions are depicted for better clearness in figure 2.

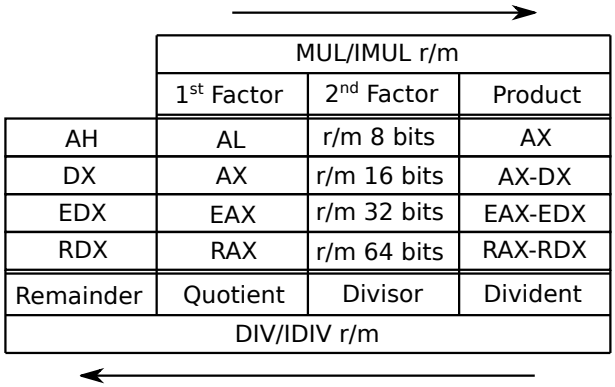

Figure 2: Description of multiplication MUL/IMUL and division DIV/IDIV

For the signed and unsigned multiplication is it necessary to look at the figure from the top and from left to right. The both instructions have only one operand that is register or memory. This operand is the 2nd Factor of multiplication. The 1st Factor is in accumulator and proper size of accumulator is specified by size of instruction operand. Product is then stored into corresponding registers.

Accumulator `AL/AX/EAX/RAX` contain LSB part of result and registers `DX/EDX/RDX` contain MSB part of result.

For the signed and unsigned division it is necessary to look at the figure from the bottom and from right to left. Also here it is important size of operand that is now Divisor. According its size it is necessary to prepare proper combination of registers for Divident.

The unsigned division `DIV` requires extension by left zeroes. The signed division `IDIV` requires signed extension using instructions `CBW`, `CWD`, `CDQ` a `CQO`, described in chapter 3.3.

4 32-bit Interfacing to C Language

The use of global variables, as noted in chapter 2.3, is neither normal nor comfortable, when programmers write programs. The parameters and return values from functions are passed in higher programming language by stack and registers. The standard describing the argument passing is in document "Application Binary Interface", in short ABI. The fourth edition of this document is in attached file `abi-32.pdf`.

Herein the selected most important principles will be described in more details. How return values are passed by registers and how to push data into stack to pass arguments to functions.

4.1 Return Values from Functions

Functions can return more types of value with size 8, 16, 32 and 64 bits. This size corresponds to the whole register, or to its part, or combination of registers, which are used for return value. According to the size of return value the following registers are used:

- 8 bits - register AL,
- 16 bits - register AX,
- 32 bits - register EAX,
- 64 bits - registers EAX-EDX,
- float/double - register FPU ST0.

4.2 Rules of Registers Usage

Inside functions a few following rules must be kept for registers usage.

- Registers EAX, ECX and EDX can be used in any way and it is not necessary to restore their value at the end of the function.
- Registers EBX, ESI and EDI can be also used in any way but at the end of the function their content must be restored. These registers can be used for local variables in functions.
- Registers ESP and EBP are used for the stack manipulation. This principle will be described below.
- The direction flag DF in the register FLAGS must be set at the end of the function to its default value 0.
- When a function uses FPU registers then at the end of the function all FPU registers must me released. Only a float or double return value can be returned in the register ST0.

4.3 Calling Function with Arguments

The passing arguments to functions is more complicated than the passing of a return value. Thus the principle will be described in a few steps.

4.3.1 Order of Passed Arguments

Let us have the following prototype of function which will compute sum of two numbers. The implementation in Assembly language is in Listing 3.

```
int sum( int a, int b );
```

Parameters can be passed from the left and from right side. Passing arguments from left is typical convention in Pascal language. The C language uses convention, where parameters are passed from the right. The passing of arguments is provided by stack using instruction PUSH. The overall view of this situation is depicted in Figure 3.

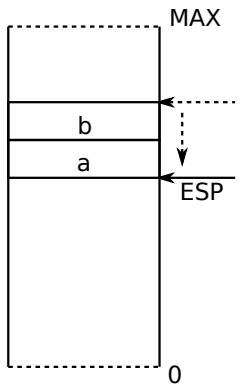

Figure 3: Arguments stored in the stack

To correctly understand the principle of passing parameters, it is necessary to keep in mind that the instruction PUSH puts data into the top of the stack, defined by the register `ESP`, and moves it toward to the 0. The top of the stack before putting arguments `a` and `b` is in Figure 3 marked by the dotted horizontal line. The position of the top of the stack, defined by `ESP`, is moved after inserting two arguments to the position marked in Figure 3 by horizontal solid line.

The stack is in this situation prepared to be possible to call the function `sum` using the instruction `CALL`.

4.3.2 Calling the Function and Set Register EBP

The execution of a function code is started after the instruction `CALL` jumps to the first instruction of a function. The instruction `CALL` before jumps

inserts into the stack the address of its subsequent instruction. This address will be used later by the instruction `RET` to return from function to address of a caller.

When the code execution is moved into a function it is necessary to set register `EBP` and allocate a space for local variables. This register `EBP` will be after that used to access arguments and local variables. The settings of `EBP` is provided by the instruction `ENTER N,0`. This instruction can be decomposed for clarity to three instructions:

```
; decomposition of instruction  ENTER N,0
push ebp            ; insert current EBP into stack
mov ebp, esp        ; EBP is set to current ESP
sub esp, N          ; the subtraction of ESP allocates
                    ; space for local variables
```

The state of the stack after instruction `ENTER N,0` is visible in Figure 4.

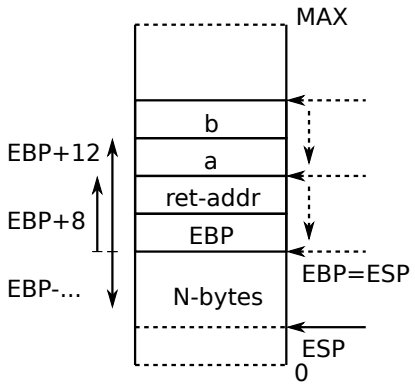

Figure 4: State of stack after instruction ENTER

In Figure 4 is visible that under parameters `a` and `b` was stored a return address by the instruction `CALL`. The subsequently the instruction `ENTER` inserts into the stack previous value of `EBP` and then it moves `ESP` down to allocate the space with size `N` for local variables.

The most important in this state is position specified by the register `EBP`. The settings of this register is in every function the same irrespective of number of passed arguments and size of local variables!

4.3.3 Access to Arguments and Local Variables

According the description in previous section it is known that the register `EBP` is set always to the same position. This position is in stack 4 bytes

under return address. This position allows every time to access arguments and local variables in the same way. Figure 4 depicts that first argument passed to function can be accessed by `[EBP+8]`, the second `[EBP+12]` and following arguments can be accessed in the same manner.

The access to local variables is in opposite direction. For example the first 32-bit local variable is at position `[EBP-4]`.

4.3.4 Return from Function, the Stack Cleanup

Every function must before return to caller cleans the stack. This step is performed in every function using instruction `LEAVE`. Instruction `LEAVE` can be decomposed to two instructions:

```
; decomposition of instruction LEAVE
mov esp, ebp        ; release local variables
pop ebp             ; set EBP back to previous value
```

The top of the stack is after instruction `LEAVE` set to return address of caller, see 4. Therefore after `LEAVE` can be executed instruction `RET` and the execution is moved back to caller.

The calling function is responsible to finish the cleanup of stack: the arguments inserted into stack must be removed. When calling function passed only a few arguments, it can clean up them using instruction `POP Register` (usually `ECX`). When number of arguments is 3 and more, the top of the stack `ESP` can be moved up by instruction `ADD`.

4.3.5 Function Example

The prototype of function `sum` was introduced at begin of this chapter:

```
int sum( int a, int b );
```

The code of this function in Assembly language is visible in Listing 3.

Listing 3: Simple function implementation and calling in Assembly language

```
    ; function sum
sum:
    enter 0,0

    mov eax, [ ebp + 8 ]          ; parameter t_a
    add eax, [ ebp + 12 ]         ; t_a += t_b
                                  ; return value is in eax
    leave
    ret

    ......
    ; function call: sum( ecx, 10 )
    push dword 10                 ; parameter t_b
    push ecx                      ; parameter t_a
    call sum                      ; function call
    add esp, 8                    ; remove t_a and t_b from the stack
    ; result is in eax
    .......
```

In Listing 3 is visible that two argument are passed from the right side. At the begin of function is instruction `ENTER 0,0` which do not allocate local variables. The first argument is loaded into register `EAX` from address `[EBP+8]` and the second argument is subsequently added. Thus the return value is prepared in the register `EAX`.

Finally the instruction `LEAVE` sets the top of stack to the return address and the instruction `RET` returns execution back to caller.

The code of the calling function must clean all arguments. In this example is the top of the stack moved 8 bytes by instruction `ADD`. When the function cleans the stack it can use the return value in register `EAX`.

4.4 Typical Examples of Arguments Passed to Functions

All functions described below are in the archive `soj-fun32.tgz`.

The first example with two integer arguments was introduced in the previous section.

The following example will demonstrate the passing and using of an integer array. This example will be function `average` which will compute the average value of all elements of array. The prototype of this function in C language is:

```
int average( int *array, int N );
```

The first argument is pointer to integer array and the second argument is the length of array. The implementation in Assembly language is visible in Listing 4.

Listing 4: Implementation of function `average`

```
    ; function average
average:
    enter 0,0
    mov ecx, [ ebp + 12 ]       ; length of t_array
    mov edx, [ ebp + 8 ]        ; *t_array
    mov eax, 0                  ; l_sum = 0
.back:
    add eax, [ edx + ecx * 4 - 4 ]; l_sum += t_array[ecx-1]
    loop .back
    cdq                         ; extension of eax to edx
    idiv dword [ ebp + 12 ]     ; l_sum /= t_N
                                ; result in eax
    leave
    ret
```

In the example in Listing 4 is the pointer to the integer array loaded into the register `EDX` and the length of the array into the register `ECX`. This register is used in the instruction `LOOP` which controls the pass though the array. The register `ECX` is also used to access (address) elements of the array.

The sum of the array is computed in the register `EAX`. After the loop this value is sign extended into the register `EDX` and the average value is computed using instruction `IDIV`. The divisor is the length of array. After the division the result is stored in the register `EAX`.

Only registers which can be overwritten were used inside this function. Therefore it is not necessary to save and restore them.

Local Label

In function `average` is visible label `.back` starting with the dot. This type of labels are called "local labels". The validity of this label is only between two global labels. Whereas the global labels are necessary only to specify entry point of function, all labels inside function code can be local.

The next typical example can be the function for division of two integers which will return the result and also the remainder. The remainder will be passed by pointer to integer variable. The prototype in C language follows:

```
int division( int a, int b, int *remainder );
```

The implementation in Assembly language is visible in Listing 5.

Listing 5: Implementation of function `division`

```
    ; function division
division:
    enter 0,0
    mov eax, [ ebp + 8 ]        ; parameter t_a to eax
    cdq                         ; extension of eax to edx
    idiv dword [ ebp + 12 ]     ; eax /= t_b
                                ; result is in eax
                                ; remainder in edx
    mov ecx, [ ebp + 16 ]       ; t_remainder
    mov [ ecx ], edx            ; *t_remainder = edx
    leave
    ret
```

In the example in Listing 5 is visible that the use of the first two arguments and the return value is similar to the function `sum` in Listing 3. Moreover in this example is remainder returned via third argument. Its use is similar to passing array in function `average`. In stack is pointer to the variable where should be stored remainder. This pointer is loaded into the register `ECX` and the remainder is subsequently stored to this address.

Previous examples showed how to pass integer arguments to functions, how to pass array of integers and how to pass pointer to integer variable. Following examples will be focused on operations with strings.

The example in Listing 6 is the function which reverse content of string. At the begin of this function the standard function `strlen` is called to get the length of the string. The prototype in C language is:

```
char *strmirror( char *str );
```

The implementation in Assembly language is in Listing 6.

Listing 6: Implementation of function `strmirror`

```
    ; function strmirror
strmirror:
    enter 0,0
    push dword [ ebp + 8 ]      ; passing *t_str to strlen
    call strlen                 ; call strlen
    pop ecx                     ; clean stack
                                ; length of string in eax
    mov ecx, [ ebp + 8 ]        ; ptr. to first character
    mov edx, ecx
    add edx, eax
    dec edx                     ; ptr. to last character
.back:
    cmp ecx, edx                ; while ( ecx < edx )
    jae .end
    mov al, [ ecx ]             ; sel. of first and last char
    mov ah, [ edx ]
    mov [ ecx ], ah             ; store sel. chars back
    mov [ edx ], al
    inc ecx                     ; move to the right
    dec edx                     ; move to the left
    jmp .back
.end:
    mov eax, [ ebp + 8 ]
    leave
    ret
```

In the implementation in Listing 6 is visible that it is very important to use proper size of operands. All characters in string have size 8 bits (1 byte). Therefore it is necessary to use 8-bit registers `AL` and `AH`.

The shift of two indexing registers in opposite directions is also evident.

The classic example of integer to string conversion is in Listing 7. The prototype of function in C is:

```
char *int2str( int number, char *str );
```

Implementation of this function in Assembly language is in Listing 7.

Listing 7: Implementation of function `int2str`

```
    ; function int2str
int2str:
    enter 8,0
    mov eax, [ ebp + 8 ]          ; t_number
    mov ecx, [ ebp + 12 ]         ; t_str
    mov [ ebp - 4 ], ecx          ; part of t_str. for mirror
    mov [ ebp - 8 ], dword 10     ; l_base of number system
    cmp eax, 0                    ; branches for < > = 0
    jg .positive
    jl .negative
    mov [ ecx ], word '0'         ; add to end of t_str "0\0"
    jmp .ret                      ; all is done
.negative:
    mov [ ecx ], byte '-'         ; sign at beginning of t_str
    inc dword [ ebp - 4 ]         ; skip sign
    neg eax                       ; turn sign
.back:
    inc ecx                       ; t_str++
.positive:
    test eax, eax                 ; while ( eax )
    je .end
    mov edx, 0
    div dword [ ebp - 8 ]         ; eax /= l_base
    add dl, '0'                   ; remainder += '0'
    mov [ ecx ], dl               ; *t_str = dl
    jmp .back
.end:
    mov [ ecx ], byte 0           ; *t_str = 0
    push dword [ ebp - 4 ]        ; begin of str. for mirror
    call strmirror
    pop ecx
.ret:
    mov eax, [ ebp + 12 ]         ; return value is t_str
    leave
    ret
```

4.5 The Example of Using String Instructions

String instructions were described in section 3.5. It is necessary to set the register `ES` before string instructions are used. The direction flag `DF` is automatically set to 0, thus index registers will be automatically incremented.

The example in Listing 8 shows the use of instruction `SCAS` to compute length of string. This instruction will search the string terminator '\0'. The function prototype in C language is:

```
int strlength( char *str );
```

The implementation of function `strlength` is in Listing 8.

Listing 8: Implementation of function `strlength`

```
    ; function strlength
strlength:
    enter 0,0
    mov edi, [ ebp + 8 ]        ; t_str
    push ds
    pop es                      ; es = ds
    mov ecx, -1                 ; ecx = MAX
    mov al, 0                   ; searched character '\0'
    ; cld                       ; not necessary, DF is 0
    repne scasb                 ; searching
    inc ecx                     ; length without '\0'
    not ecx                     ; turn sign
    mov eax, ecx                ; string length
    leave
    ret
```

The second example of the string instruction use is code which will remove all spaces from string. In this code will be used pair of instructions `LODS` a `STOS`. The function prototype in C is:

```
char * strnospaces ( char *str );
```

The implementation of function `strnospaces` is in Listing 9.

Listing 9: Implementation of function `strnospace`

```
    ; function  strnospaces
strnospaces :
    enter 0,0
    push edi                        ; save  registers
    push esi
    mov edi, [ ebp + 8 ]            ; t_str
    mov esi, edi                    ; esi = edi
    push ds
    pop es                          ; es = ds
    ; cld                           ; not  necessary, DF is 0
.back:
    lodsb                           ; al = [ esi++ ]
    test al, al
    jz .end                         ; end of  string
    cmp al, ' '
    je .back                        ; skip  space
    stosb                           ; [ edi++ ] = al
    jmp .back
.end:
    stosb                           ; [ edi ] = '\0'
    mov eax, [ ebp + 8 ]            ; return  value
    pop esi                         ; restore  registers
    pop edi
    leave
    ret
```

5 AMD and Intel x86 Processors – 64-bit Mode

5.1 Registers

The 64-bit extension of 32-bit registers introduced at first AMD. AMD applied the same principle for extension as Intel did when extended 16-bit CPU 80286 to 32-bit CPU 80386DX. At that time Intel extended all 16-bit registers to 32-bit registers.

But AMD did not only extend size of registers, the number of registers was increased to twice. The whole overview of registers is visible in Figure 5.

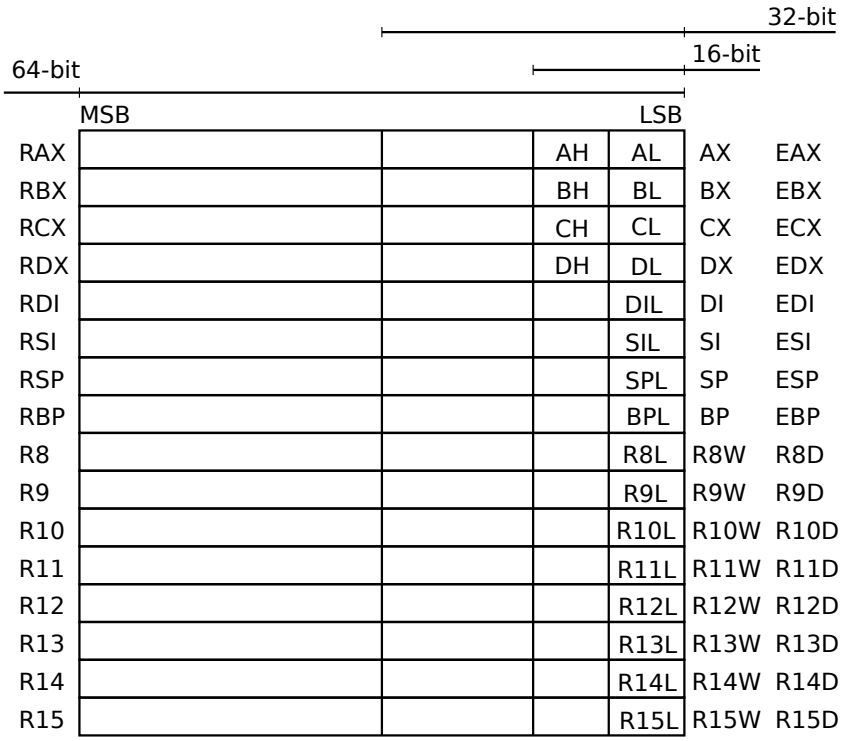

Figure 5: Register set of 64-bit processor

The entire original set of working registers was extended to 64-bit and names of all 64-bit registers were changed from Exx to Rxx. The new group of registers is named as registers R8-R15 and lower 8/16/32 bits is termed as RxL/RxW/RxD. It is also possible to access the lower 8-bit part of register SI, DI, SP, BP, which was not previously possible.

Other registers are unchanged. Of course, the EIP register was extended to 64-bit version RIP.

5.2 Addressing in 64-bit Mode

The addressing in 64-bit mode stayed practically the same as in 32-bit mode. Even though it is possible to still use 32-bit registers, it is necessary strictly to use 64-bit registers. The use of smaller registers is bad on principle.

The ABI standard describing the 64-bit programming interface is in attached document `abi-64.pdf`

6 64-bit Interfacing to C Language

6.1 Return Values

The principle of passing return values from function is almost identical in 32-bit and 64-bit mode:

- 8 bits - register AL,
- 16 bits - register AX,
- 32 bits - register EAX,
- 64 bits - register RAX,
- 128 bits - registers RAX-RDX
- float/double - register XMM0.

The small extension and change is only for float and double return values. In 64-bit mode is not longer used the FPU unit but the SSE unit is exclusively used.

6.2 Rules for Registers Usage

The 64-bit mode has also a few rules about registers usage inside functions. Some registers can be changed and some must be preserved. The most important set of rules can be summarized in a few points.

- Registers `RDI, RSI, RDX, RCX, R8 and R9` are used for parameters passing and can be changed in code of function.
- Registers `RAX, R10 and R11` can be also changed.
- Registers `RSP and RBP` are used for manipulation with stack. Their content must be preserved.
- Registers `RBX, R12, R13, R14 and R15` have to be preserved also.
- Direction flag `DF` must be set to 0 at the end of function.
- Registers FPU - `ST0 to ST7` a registers SSE - `XMM0 to XMM15` can be changed in function code.

6.3 Calling Function with Parameters

The combination of registers and stack is used for passing arguments into functions in 64-bit mode.

The first six integer and pointer values are passed by six registers (from left):

`RDI, RSI, RDX, RCX, R8 and R9.`

The first eight float and double arguments are passed using SSE registers:

`XMM0 to XMM7.`

The ABI standard contains one more important information for functions with variable number of arguments (, ...). These functions require to set number of argument passed by XMMx registers into register `AL`.

The rest of integer and float arguments are passed by stack in the same way as in 32-bit mode. The manipulation with stack and access to arguments and local variables was described in previous text. The one difference is that the size of data on stack have the twice size.

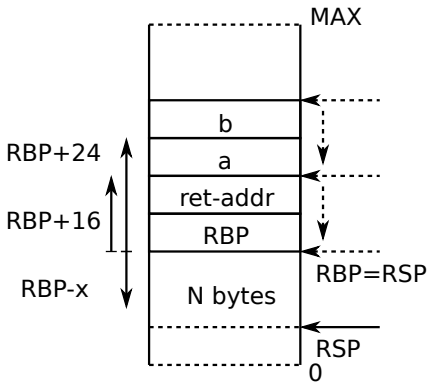

Figure 6: Stack in 64-bit mode

Figure 6 depicts the situation of case when parameters `a` and `b` are seventh and eight arguments of function. The access to local variables is the same.

6.4 Typical Examples of Functions in 64-bit Mode

In following text overwritten 32-bit functions from section 4 will be shown. These functions allow to compare differences between 32-bit and 64-bit mode. All of these functions are in attached archive `soj-fun64.tgz`.

The first example is again the function `sum`. The example can be extend to two cases. Function with parameter `int` and `long`.

```
int  sum_int( int a, int b );
long  sum_long( long a, long b );
```

The code of functions in Assembly language is in Listing 10.

Listing 10: Implementation of functions `sum` in 64-bit mode

```
    ; function sum
sum_int:
    enter 0,0
    mov eax, edi                ; parameter t_a
    add eax, esi                ; t_a += t_b
                                ; return value is in eax
    leave
    ret

sum_long:
    enter 0,0
    mov rax, rdi                ; parameter t_a
    add rax, rsi                ; t_a += t_b
                                ; return value is in rax
    leave
    ret
```

In example in Listing 10 is visible that parameters are passed via two registers in given (by ABI) order: `RDI` and `RSI`. The function do not use any local variables and all parameters are passed via registers. Therefore it is not necessary to manipulate with registers `ESP` and `EBP` using instructions `ENTER` and `LEAVE`. Arguments in function `sum_int` are 32-bit, thus result is computed in 32-bit register. The function `sum_long` has 64-bit type of argument and in the code the full size of registers is used.

The next example is manipulation with `int` array. The function will compute sum of all elements and than it will compute average value. The sum can be counted intermediately in 64-bit register to avoid overflow. The prototype in C language is:

```
int average( int *array, int N );
```

The first parameter is the pointer to `int` array and the second parameter is length of array. The code in Assembly language can be implemented as is visible in Listing 11.

Listing 11: Implementation of function `average` in 64-bit mode

```
    ; function  average_int_array
average_int_array :
    enter 0,0

    movsx rsi, esi              ; length of t_array
    mov rax, 0                  ; l_sum
    mov rcx, 0                  ; i = 0
.back:
    cmp rcx, rsi                ; i < t_N
    jge .endfor
    movsx rdx, dword [ rdi + rcx * 4 ]
    add rax, rdx                ; l_sum += t_array[ ecx ]
    inc rcx                     ; i++
    jmp .back
.endfor:
    cqo                         ; extension of rax to rdx
    movsx rcx, esi              ; t_N
    idiv rcx                    ; l_sum /= t_N
                                ; result is in rax
    leave
    ret
```

In example in Listing 11 is 32-bit length of array moved into register `RCX` with sign extension. Then the every element is extended into register `RDX` and then added to total sum. Before the division is the sum in register `RAX` sign extended into `RDX` and the it is divided by length of array.

The next illustrative example is function for division of two integers which will return the quotient and the reminder. The remainder will be returned as the third argument – pointer to integer. Function can be also implemented for type `int` and `long`.

```
int  division_int ( int a, int b, int *remainder );
long  division_long ( long a, long b, long *remainder );
```

The assembly code can be implemented as is visible in Listing 12.

Listing 12: Implementation of functions `division...` in 64-bit mode

```
    ; function division
division_int :
    enter 0,0
    mov rcx, rdx                ; save t_remainder
    mov eax, edi                ; parameter t_a to eax
    cdq                         ; sign extension of eax do edx
    idiv esi                    ; eax /= t_b
                                ; result is in eax
                                ; remainder is in edx
    mov [ rcx ], edx            ; *t_remainder = edx
    ret

division_long :
    mov rcx, rdx                ; save t_remainder
    mov rax, rdi                ; parameter t_a to eax
    cqo                         ; extension of rax to rdx
    idiv rsi                    ; rax /= t_b
                                ; result is in rax
                                ; remainder v rdx
    mov [ rcx ], rdx            ; *t_remainder = rdx
    leave
    ret
```

The both examples in Listing 12 are very similar, the difference is only the size of registers used for the computation.

To store a remainder it is necessary to save the value of the third argument `RDX`, which will be overwritten by the instruction `idiv`.

Previous examples were focused to passing integer values and pointers to integer arrays and values. Following examples will show manipulation with strings.

Next example is a mirror of string. At the begin of function is called standard function `strlen` to get length of string. Prototype of function in C language is:

```
char *strmirror( char *str );
```

The implementation in Assembly language is in Listing 13.

Listing 13: Implementation of function `strmirror` in 64-bit mode

```
    ; function strmirror
strmirror:
    enter 0,0
    push rdi                    ; save rdi
    call strlen                 ; call strlen
    pop rdi                     ; restore rdi
                                ; in rax is length of string
    mov rcx, rdi                ; ptr. to first character
    mov rdx, rcx
    add rdx, rax
    dec rdx                     ; ptr. to last character
.back:
    cmp rcx, rdx                ; while ( ecx < edx )
    jae .end
    mov al, [ rcx ]             ; sel. of first and last char
    mov ah, [ rdx ]
    mov [ rcx ], ah             ; store back sel. chars
    mov [ rdx ], al
    inc rcx                     ; move to the right
    dec rdx                     ; move to the left
    jmp .back
.end:
    mov rax, rdi                ; return t_str
    leave
    ret
```

The implementation in Listing 13 is visible that previous 32-bit version is very similar to new 64-bit implementation. One differences is addressing, pointers in 64-bit mode have size 64 bits. The second difference is function calling. In 32-bit mode argument was passed using instruction `PUSH/POP`. In 64-bit mode is the first argument of `strlen` prepared directly in register `RDI`.

Next example is classical problem from basics of algorithmization. It is conversion of integer to string. The prototype of function in C language follows:

```
char *int2str( long number, char *str );
```

The implementation of function is in Listing 14.

Listing 14: Implementation of function `int2str` in 64-bit mode

```
    ; function int2str
int2str:
    enter 0,0
    mov rax, rdi                ; t_number
    mov rcx, 10                 ; l_base of number system
    mov rdi, rsi                ; part of t_str. for mirror
    push rsi                    ; save t_str
    cmp rax, 0                  ; branches for < > = 0
    jg .positive
    jl .negative
    mov [ rsi ], word '0'       ; add to end of str "0\0"
    jmp .ret                    ; all is done
.negative:
    mov [ rsi ], byte '-'       ; sign at beginning of t_str
    inc rdi                     ; skip sign
    neg rax                     ; turn sign
.back:
    inc rsi                     ; t_str++
.positive:
    test rax, rax               ; while ( rax )
    je .end
    mov rdx, 0
    div rcx                     ; rax /= l_base
    add dl, '0'                 ; remainder += '0'
    mov [ rsi ], dl             ; *t_str = dl
    jmp .back
.end:
    mov [ rsi ], byte 0         ; *t_str = 0
                                ; rdi is t_str for mirror
    call strmirror
.ret:
    pop rax                     ; return value
    leave
    ret
```

6.5 The Example of Using String Instructions

The following code shows usage of string instruction to count length of string. The function prototype in C language follows:

```
long strlength( char *str );
```

The code in Assembly language is in Listing 15.

Listing 15: Implementation of function `strlength` in 64-bit mode

```
    ;long str_length( char *t_str );
str_length:
    enter 0,0
    mov rax, 0                  ; l_len = 0
.back:
    cmp byte [ rdi + rax ], 0   ; while ( t_str[ l_len ] != 0 )
    je .done
    inc rax                     ; l_len++
    jmp .back
.done                           ; return rax
    leave
    ret
```

The following function will remove all spaces from string. This function will have following prototype in C language:

```
char *strnospaces( char *str );
```

The implementation in Assembly language is in Listing 16.

Listing 16: Implementation of function `strnospaces` in 64-bit mode

```
    ; function strnospaces
strnospaces:
    enter 0,0
    mov rsi, rdi                ; rsi = rdi = t_str
    mov rdx, rdi                ; save rdi
    mov ax, ds
    mov es, ax                  ; es = ds
    ; cld                       ; not necessary, DF is 0
.back:
    lodsb                       ; al = [ rsi++ ]
    test al, al
    jz .end                     ; end of string
    cmp al, ' '
    je .back                    ; skip space
    stosb                       ; [ rdi++ ] = al
    jmp .back
.end:
```

```
stosb                           ; [ rdi ] = '\0'
mov rax, rdx                    ; return t_str
leave
ret
```

7 SSE

In 64-bit mode the computation with float point numbers is performed mainly on the SSE unit. It is also possible to use the older FPU unit, but the binary 64-bit interface for the C and Assembly language is designed directly for the SSE unit.

7.1 SSE Registers

The first version of SSE had only 8 registers with size of 64 bits. The extension of CPU from 32-bit into 64-bit version brought extension to 16 registers with the size of 128 bits. Registers are termed from `XMM0` to `XMM15` as is visible in Figure 7.

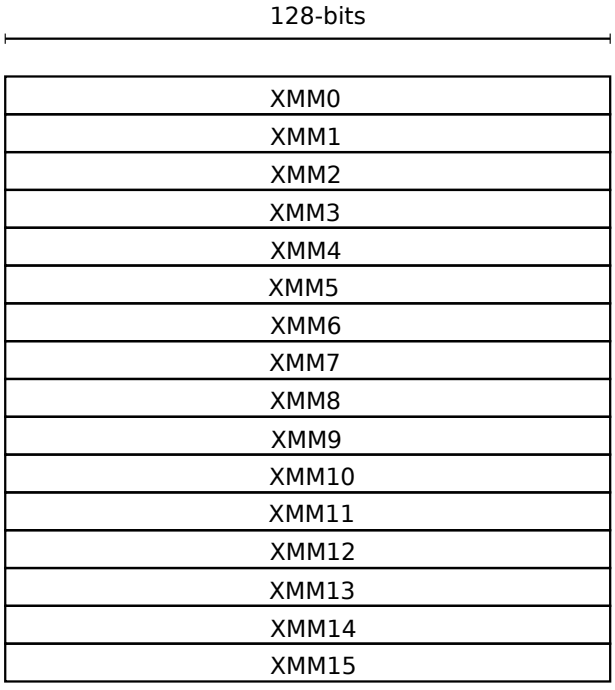

Figure 7: SSE registers

7.2 Content of Registers

Float point numbers can be stored in `XMMx` registers in a few different forms. The overview for clarity is depicted in Figure 8.

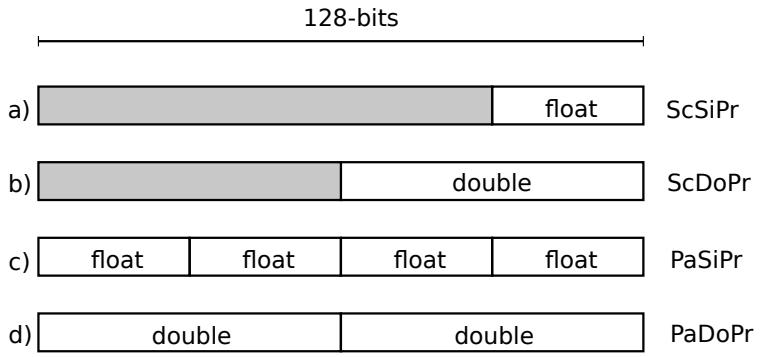

Figure 8: Possible forms of float point numbers in SSE registers

Float point numbers are stored in registers individually (scalar) or as a pair or four numbers. The computation can be performed with the single precision numbers (float) or with the double precision numbers (double). Overall the four possible forms for float point numbers are available:

- The form Scalar Single-Precision (ScSiPr) is in Figure 8 a),
- The form Scalar Double-Precision (ScDoPr) is in Figure 8 b),
- The form Packed Single-Precision (PaSiPr) is in Figure 8 c),
- The form Packed Double-Precision (PaDoPr) is in Figure 8 d).

In the following text will be used abbreviations introduced in Figure 8.

7.3 SSE Instructions

The instruction set of SSE unit has nowadays a few hundreds instructions. For the SSE introduction the only a few tens of them are necessary to be introduced. The reduced instruction set will allow realize the basic computation. Instructions will be herein again organized thematically, not alphabetically.

7.3.1 Moving Instructions

`MOVAPD dest, src`

This instruction moves pair of `double` numbers in the PaDoPr form from the operand `src` into operand `dest`.

The source and destination operand can be register `XMMx` or memory. It is not possible to use two memory operands. This instruction expects the alignment of memory operand to a multiple of 16 bytes.

`MOVUPD dest, src`

This instruction is similar to the instruction MOVAPD, but the alignment of a memory operand is not needed.

`MOVAPS dest, src`

This instruction moves four `float` numbers in the PaSiPr form from the operand `src` into operand `dest`.

The source and destination operand can be register `XMMx` or memory. It is not possible to use two memory operands. This instruction expects the alignment of memory operand to a multiple of 16 bytes.

`MOVUPS dest, src`

This instruction is similar to the instruction MOVAPS, but the alignment of a memory operand is not needed.

`MOVDQA dest, src`

This instruction moves 16 bytes from the operand `src` into operand `dest`.

This instruction expects the alignment of memory operand to a multiple of 16 bytes.

`MOVDQU dest, src`

This instruction is the same as the instruction MOVDQA, but the alignment of a memory operand is not needed.

`MOVSD dest, src`

This instruction moves one float point number `double` in the ScDoPr form from the operand `src` into operand `dest`.

The source and destination operand can be register `XMMx` or memory. It is not possible to use two memory operands.

`MOVSS dest, src`

This instruction moves one float point number `float` in the ScSiPr form from the operand `src` into operand `dest`.

The source and destination operand can be register `XMMx` or memory. It is not possible to use two memory operands.

7.3.2 Moving Instructions with the Reordering

`MOVDDUP dest, src`

This instruction converts one float point number `double` to PaDoPr form using duplication. For the clarity can be this behaviour explained by the following code:

```
dest[ 0 ] = dest[ 1 ] = src[ 0 ];
```

`MOVHPD/MOVLPD dest, src`

This instruction moves one float point number `double` from the source into the destination. The instruction `MOVHPD` moves the upper part and the instruction `MOVLPD` moves the lower part.

`SHUFPD dest, src, k8`

This instruction moves one `double` number from the source and one number from the destination into source. The constant `k8` defines by two lower bits selection of numbers from the source and destination. For the clarity can be this behaviour explained by the following code:

```
k8 = 0b000000i1i0;
dest[ 0 ] = dest[ i0 ]
dest[ 1 ] = src[ i1 ];
```

`MOVSHDUP/MOVSLDUP dest, src`

This instruction moves two `float` numbers from the source into destination using duplication.

The instruction `MOVSHDUP` selects from the source numbers at odd positions.

```
dest[ 0 ] = dest[ 1 ] = src[ 1 ];
dest[ 2 ] = dest[ 3 ] = src[ 3 ];
```

The instruction `MOVSLDUP` selects from the source numbers at even positions:

```
dest[ 0 ] = dest[ 1 ] = src[ 0 ];
dest[ 2 ] = dest[ 3 ] = src[ 2 ];
```

`MOVHPS/MOVLPS dest, src`

This instruction moves two `float` numbers from the source into destination. The instruction `MOVHPS` moves two upper numbers and the instruction `MOVLPS` moves two lower numbers.

`MOVHLPS/MOVHLPS dest, src`

This instruction moves two `float` numbers from the source into destination. The instruction `MOVHLPS` moves two upper numbers into lower part of the destination and the instruction `MOVLHPS` moves two lower numbers into upper part of the destination.

`SHUFPS dest, src, k8`

This instruction moves two `float` numbers from the source and two numbers from the destination into destination. The constant `k8` defines by low eight bits (four pairs) which numbers will be selected from the source and which from the destination. For the clarity can be this behaviour explained by the following code:

```
k8 = 0bi76i54i32i10;
dest[ 0 ] = dest[ i10 ];
dest[ 1 ] = dest[ i32 ];
dest[ 2 ] = src[ i54 ];
dest[ 3 ] = src[ i76 ];
```

7.3.3 Float Point Numbers Form Conversion

`CVTPD2PS dest, src`

This instruction moves and converts pair of `double` numbers from the PaDoPr form from the operand `src` into operand `dest`. The destination will contain pair of `float` numbers and two 0 in PaSiPr form.

The source and destination operand can be a register `XMMx` or memory. It is not possible to use two memory operands.

`CVTPS2PD dest, src`

This instruction moves and converts lower pair of `float` numbers from the PaSiPr form from the operand `src` into operand `dest`. The destination operand will contain the pair of `double` numbers in the PaDoPr form.

The source and destination operand can be a register `XMMx` or memory. It is not possible to use two memory operands.

`CVTSD2SS dest, src`

This instruction moves and converts the `double` number in the ScDoPr form from the operand `src` into operand `dest`. The destination operand will contain `float` number in the ScSiPr form.

The source and destination operand can be a register `XMMx` or memory. It is not possible to use two memory operands.

`CVTSS2SD dest, src`

This instruction moves and converts the `float` number in the ScSiPr form from the operand `src` into operand `dest`. The destination operand will contain `double` number in the ScDoPr form.

The source and destination operand can be a register `XMMx` or memory. It is not possible to use two memory operands.

`CVTSD2SI/CVTSS2SI dest, src`

This instruction moves and converts the `double` or the `float` number from the ScDoPr or the ScSiPr form from the source operand `src` into the destination `dest`. The destination operand will contains the 32-bit or 64-bit integer.

The source operand can be `XMMx` register or memory and the target operand must be 32-bit or 64-bit ALU register.

`CVTSI2SD/CVTSI2SS dest, src`

This instruction moves and converts the 32-bit or 64-bit integer from the operand `src` into destination operand `dest`. The destination operand will contain the `double` in the ScDoPr form or the `float` number in the ScSiPr form.

The source operand must be a 32-bit or 64-bit ALU register and the destination must be `XMMx` register.

7.3.4 Arithmetical Operations

All arithmetical instructions – addition, subtraction, multiplication and divison – are used in the same way. The differences is only the operand form.

`ADDPD/DIVPD/MULPD/SUBPD dest, src`

This instruction makes the required mathematical operation between the operand `dest` and `src` and the result is stored into destination operand. The both operands must be in the PaDoPr form.

The destination operand must be a `XMMx` register and the source operand can be `XMMx` register or the memory aligned to a multiple of 16 bytes.

`ADDPS/DIVPS/MULPS/SUBPS dest, src`

This instruction makes the required mathematical operation between the operand `dest` and `src` and the result is stored into destination operand. The both operands must be in the PaSiPr form.

The destination operand must be a `XMMx` register and the source operand can be `XMMx` register or the memory aligned to a multiple of 16 bytes.

`ADDSD/DIVSD/MULSD/SUBSD dest, src`

This instruction makes the required mathematical operation between the operand `dest` and `src` and the result is stored into destination operand. The both operands must be in the ScDoPr form.

The destination operand must be a `XMMx` register and the source operand can be `XMMx` register or the memory.

`ADDSS/DIVSS/MULSS/SUBSS dest, src`

This instruction makes the required mathematical operation between the operand `dest` and `src` and the result is stored into destination operand. The both operands must be in the ScSiPr form.

The destination operand must be a `XMMx` register and the source operand can be `XMMx` register or the memory.

`SQRTPS/SQRTSS/SQRTPD/SQRTSD dest, src`

This instruction computes square root. The operand can be in PaSiPr, ScSiPr, PaDoPr or ScDoPr form.

`RSQRTPS/RSQRTSS dest, src`

This instruction computes reciprocal value of square root. The operand can be in PaSiPr or ScSiPr form.

`RCPPS/RCPSS dest, src`

This instruction computes reciprocal value of number(s). The operand can be in PaSiPr or ScSiPr form.

7.3.5 Bitwise Instructions

The bitwise operation can be also performed between `XMMx` registers. All of this operation works independently on the data forms and all operations are performed for all bits. Thus the data form PaDoPr and PaSiPr are formal only.

All operations are designed for memory operands aligned to 16 bytes.

`ANDPD/ANDPS/ANDNPD/ANDNPS dest, src`

This instruction performs bitwise operation AND between `dest` and `src`. Result is stored in destination operand.

The instruction ANDNxx performs bit inversion of `src` before AND operation.

The destination operand must be a `XMMx` register and the source can be a register or memory aligned to 16 bytes.

`ORPD/ORPS dest, src`

This instruction performs bitwise operation OR between `dest` and `src`. Result is stored in destination operand.

The destination operand must be a `XMMx` register and the source can be a register or memory aligned to 16 bytes.

`XORPD/XORPS dest, src`

This instruction performs bitwise operation XOR between `dest` and `src`. Result is stored in destination operand.

The destination operand must be a `XMMx` register and the source can be a register or memory aligned to 16 bytes.

7.3.6 Comparison of numbers

`CMPccPD/CMPccPS/CMPccSD/CMPccSS dest, src, condition`

This instruction performs comparison of two parameters with the same form. The form can be PaDoPr, PaSiPr, ScDoPr or ScSiPr. Comparison is performed between operands `dest` and `src` and the third parameter `condition` defines type of condition. The result stored in the destination operand will be composed from 0, when condition is false, or all 1, when result is true.

The target operand must be a `XMMx` register and the source must be a register or memory.

The condition selection:

0 EQ - Equal,

1 LT - Less-than,

2 LE - Less-than or equal,

3 UNORD - Unordered (if either operand `dest` or `src` is a NAN),

4 NE - Not-equal,

5 NLT - Not-less-than,

6 NLE - Not-less-than-or-equal,

7 ORD - Ordered (neither operand `dest` nor `src` is a NAN).

`COMISD/COMISS dest, src`

Instructions COMISD a COMISS compares two scalar parameters`float` or `double` in form ScDoPr or ScSiPr. The operand `dest` must be a `XMMx` register and the operand `src` must be register or memory. The result is stored into `FLAGS` register in ALU and it can be evaluated directly by conditional instructions. This instruction set only three bits: ZF, PF and CF. Bits AF, OF and SF are cleared.

ZF,PF,CF = 111: Unordered.

ZF,PF,CF = 000: Greater-than.

ZF,PF,CF = 001: Less-than.

ZF,PF,CF = 100: Equal.

`MINPS/MINSS/MINPD/MINSD dest, src`
`MAXPS/MAXSS/MAXPD/MAXSD dest, src`

These instructions compares number(s) from the operand `src` and `dest` and store minimum/maximum number(s) into destination operand.

7.4 Typical SSE Examples

The first example is the addition of two `float` and `double` numbers. The prototypes in C language can be in following form:

```
float  add_float ( float a, float b );
double  add_double ( double a, double b );
```

The implementation in Assembly language is simple:

```
add_float :
      addss  xmm0, xmm1              ; a += b
      ret

add_double :
      addsd  xmm0, xmm1              ; a += b
      ret
```

In this example is visible that parameters with decimal point are passed directly by `XMMx` registers. Thus it is possible to directly use value in these registers. The return value is passed by the `XMM0` register and therefore it is possible implement the addition by single instruction. It is only necessary to select proper instruction for proper type of arguments.

The next example is computation of sphere volume. The volume is defined by following formula:

$V = 4/3 \cdot \pi \cdot r^3$.

The implementation in Assembly language using SSE instruction is simple. The function prototype in C language is:

```
double  volume_sphere ( double R );
```

The implementation follows:

```
volume_sphere :
    movsd  xmm1, xmm0              ; r
    mulsd  xmm0, xmm0              ; r*r
    mulsd  xmm0, xmm1              ; r*r*r
    mulsd  xmm0, [ pi ]            ; *pi
    mov  eax, 4
    cvtsi2sd  xmm1, eax
    mulsd  xmm0, xmm1              ; *4
    dec  eax
    cvtsi2sd  xmm1, eax
    divsd  xmm0, xmm1              ; /3
    ret
```

In this example it is necessary to have some global variable with π.

The next example demonstrates access to `float` elements in array. In this example will be selected maximum from array.

The function prototype in C language is following:

```
float find_max( float *array, int N );
```

The implementation in Assembly language:

```
find_max:
    movss xmm0, [ rdi ]          ; sel. first element as MAX
    movsx rcx, esi               ; N
    dec rcx                      ; skip first element
.back:
    comiss xmm0, [ rdi + rcx * 4 ] ; compare
    jae .skip
    movss xmm0, [ rdi + rcx * 4 ]  ; exchange MAX
.skip:
    loop .back
                                 ; resutl is in XMM0
    ret
```

The last SSE example shows the use of numbers in PaDoPr form. The aim is to compute average value of array with `double` numbers. In the loop the sum is computed by pairs of numbers, thus the length of loop is only half. At the end the sum is computed as sum of two subtotals.

The function prototype follows:

```
double  array_average ( double *array, int N );
```

Implementation in Assembly language:

```
array_average:
    xorpd  xmm0, xmm0              ; sum = 0
    xor  rdx, rdx                  ; inx = 0
    movsx  rcx, esi                ; N
    shr  rcx, 1                    ; N /= 2
    jnc  .nocf                     ; is N odd?
    movsd  xmm0, [ rdi ]           ; store odd element
    inc  rdx                       ; skip odd element
.nocf:
    xorpd  xmm1, xmm1              ; sum2 = 0,0
.back:
    movupd  xmm2, [ rdi + rdx * 8 ]
    addpd  xmm1, xmm2              ; sum2 += pair of numbers
    add  rdx, 2                    ; skip two numbers
    loop  .back                    ; while ( --rcx )

    addsd  xmm0, xmm1              ; sum += sum2
    shufpd  xmm1, xmm1, 1          ; exchange numbers in sum2
    addsd  xmm0, xmm1              ; sum += sum2
    cvtsi2sd  xmm1, esi
    divsd  xmm0, xmm1              ; sum /= N
    ret
```

8 References

1. System V Application Binary Interface, Intel386 Architecture Processor Supplement, Fourth Edition, 1997

   abi-32.pdf

2. Intel 64 and IA-32 Architectures, Software Developer's Manual, Volume 2: Instruction Set Reference, A-Z, Intel 2018

   intel-AZ.pdf

3. Michael Matz, Jan Hubička, Andreas Jaeger, Mark Michell, System V Application Binary Interface, AMD64 Architecture Processor Supplement, 2013

   abi-64.pdf

4. The NASM Development Team, NASM - The Netwide Assembler 0.98, 2003

   nasm-0.98.pdf

5. The NASM Development Team, NASM - The Netwide Assembler 2.11, 2012

   nasm-2.12.pdf

6. Raymond Filiatreault, Simply FPU, 2003

   FPU-Tutorial.zip